SRE 实践白皮书

v1.0.7

2026 年 3 月

SRE-Elite.com

版权声明

这项作品采用 CC BY-ND 4.0 许可进行授权。要查看此许可的副本，请访问 http://creativecommons.org/licenses/by-nd/4.0/

CC BY-ND 4.0 DEED

署名-禁止演绎 4.0 国际

您可以自由地:

共享 — 在任何媒介以任何形式复制、发行本作品 在任何用途下，甚至商业目的。

只要你遵守许可协议条款，许可人就无法收回你的这些权利。

惟须遵守下列条件:

署名 — 您必须给出 适当的署名 ，提供指向本许可协议的链接，同时 标明是否（对原始作品）作了修改 。您可以用任何合理的方式来署名，但是不得以任何方式暗示许可人为您或您的使用背书。

禁止演绎 — 如果您 再混合、转换，或者基于该作品创作 ，您不可以分发修改作品。

没有附加限制 — 您不得适用法律术语或者 技术措施 从而限制其他人做许可协议允许的事情。

声明:

您不必因为公共领域的作品要素而遵守许可协议，或者您的使用被可适用的 例外或限制 所允许。

不提供担保。许可协议可能不会给予您意图使用的所必需的所有许可。例如，其他权利比如 形象权、隐私权或人格权 可能限制您如何使用作品。

目录

修订记录 11

1.0.7 修订记录 11

1.0.6 修订记录 11

1.0.5 修订记录 11

1.0.4 修订记录 12

1.0.3 修订记录 12

1.0.2 修订记录 12

第一章 SRE 整体介绍 13

1.1 前言 13

1.2 SRE 发展历程 13

1.3 SRE 的目标 14

1.4 SRE 的组织架构 15

第二章 SRE 的基础（1.0.8 发布） 19

第三章 SRE 的职能 19

1 可靠性架构设计 19

1.1 应用韧性架构 19

1.1.1 分布式设计 19

1.1.2 解耦设计 20

1.1.3 冗余设计 20

1.1.4 熔断设计 20

1.1.5 限流设计 20

1.1.6 降级设计 21

1.1.7 可观测设计 21

1.2 基础设施保障 21

1.2.1 机房多活 21

1.2.2 网络容灾 21

1.3 数据灾备 22

1.3.1 数据备份 22

1.3.2 数据回滚 22

2 研发保障 22
2.1 研发保障体系设计 23
2.1.1 代码可靠性 23
2.1.2 代码仓库可靠性 30
2.1.3 构建可靠性 33
2.1.4 制品可靠性 36
2.2 研发保障工程体系设计 38
2.2.1 面向研发保障的持续集成流水线 38
2.2.2 面向研发保障的可观测设计 41
2.2.3 面向研发保障的操作调度操作平台 43
2.2.4 面向研发保障的 ITSM 平台 44
2.2.5 面向研发保障的容器平台 45
2.2.6 面向研发保障的编译加速平台 46
2.3 研发保障案例 47
2.3.1 腾讯游戏全球研发保障实践 47
2.3.2 某语音直播公司研发过程保障实践 99
3 入网控制 115
3.1 运行环境适配 115
3.1.1 运营环境设计 115
3.1.2 容器云适配 116
3.1.3 数据库存储适配 118
3.1.4 信创适配 119
3.2 运行环境交付 122
3.2.1 基础资源服务 122
3.2.2 可观测策略 122
3.2.3 自动化策略 124
3.3 测试策略 125
3.3.1 连通性验证 125
3.3.2 功能测试 126
3.3.3 性能压测 128

3.3.4 数据迁移 131
3.4 变更评审 131
3.4.1 稳定性架构设计评估 131
3.4.2 非功能性技术评估 133
3.4.3 变更保障准备工作评估 134
3.4.4 新系统或新业务上线保障评估 135
4 变更管理 136
4.1 发布管理与变更管理关系阐述 136
4.2 变更体系设计 138
4.2.1 变更体系设计原则 138
4.2.2 变更及发布流程设计 138
4.2.3 变更的工程体系设计 151
4.3 变更管理案例 167
4.3.1 B 站变更防控的设计与实践 167
4.3.2 携程云平台基础设施变更管理实践 183
4.3.3 某银行变更管理设计与实践 197
4.4 发布管理案例 212
4.4.1 中移互联网敏捷发布平台建设实践 212
4.4.2 某证券变更一体化平台建设实践 229
4.4.3 游戏 GitOps 发布管理实践 243
5 故障应急 248
5.1 故障体系 248
5.1.1 故障定义 248
5.1.2 故障等级 248
5.1.3 故障序列 248
5.1.4 关键业务场景 249
5.1.5 应用服务 SLI/SLO/SLA 250
5.2 故障应急体系 255
5.2.1 故障应急体系设计原则 255
5.2.2 故障应急流程设计 255

5.2.3 故障应急工程体系设计 272
5.3 故障应急案例 280
5.3.1 小米故障应急响应经验分享 280
5.3.2 中国联通数字化监控平台稳定性保障实践 299
5.3.3 腾讯全球化游戏故障管理实践 321
5.3.4 XX 银行应急管理一体化平台建设实践 351
5.3.5 美图故障管理体系搭建实践 362
5.3.6 B 站轻量级容灾演练体系构建 401
5.3.7 蚂蚁故障应急全流程体系构建及应用实践 427
5.3.8 xx 银行应急提升实践：深耕 1-5-10 工程 451
5.3.9 腾讯 IEG SRE 应急响应实践 476
5.3.10 小米米家故障应急保障体系实战：稳中求胜的构建之道 522
5.3.11 广发证券数智化重构故障管理：构建主动防御新体系 547
5.4 混沌工程与全链路压测 592
5.4.1 混沌工程与全链路压测概述 592
5.4.2 混沌工程与全链路压测体系设计 594
5.4.3 混沌工程与全链路压测关键流程 596
5.4.4 腾讯游戏 SRE 混沌工程及压测实践 598
5.4.5 携程全链路压测平台的建设与运营 638
5.4.6 小红书混合云容灾治理实践 676
5.4.7 蚂蚁大促场景下全链路压测体系构建与保障实践 725
5.4.8 腾讯游戏高可用保障体系实践 768
5.4.9 小米混沌工程与全链路压测实践 826
5.4.10 小红书基于流量地图的活动保障实践与 AI 探索 857
6 上线后持续优化工作 886
6.1 用户体验优化 886
6.1.1 基于用户端的直接用户体验优化 887
6.1.2 基于系统端的间接用户体验优化 887
6.2 运维琐事的日常管理及优化 889
6.2.1 运维琐事的介绍 889

6.2.2 运维琐事的质量管理 890
6.3.3 运维琐事的效率管理 891
6.3 业务全生命周期工具建设 892
6.3.1 研发期工具建设 892
6.3.2 上线期工具建设 893
6.4.3 运营期工具建设 894
6.4.4 下线期工具建设 894
6.4 运营成本分析及优化 895
6.4.1 运营成本分析及优化的必要性 895
6.4.2 运营成本实时监控 895
6.4.3 运营成本分析及优化的指标 896
6.4.4 运营成本的统计及分析方法 897
6.4.5 运营成本的优化方法 898
6.5.5 运营成本优化持续运营 900
6.5 持续改进 901
6.5.1 效率持续改进 901
6.5.2 质量持续改进 902
6.5.3 安全持续改进 904
6.5.4 人员能力持续提升 904
6.5.5 流程持续改进 905
7 平台工程 907
7.1 标准应用平台工程建设 907
7.1.1 应用元信息平台 908
7.1.2 统一资源供给 910
7.1.3 持续集成 911
7.1.4 持续部署 913
7.1.5 部署编排 915
7.1.6 可观测 917
7.1.7 成本（定价、用量、出账） 918
7.2 异构应用平台工程建设 920

7.2.1 总体设计 920
7.2.2 aPaaS 结构设计 921
7.2.3 iPaaS 结构设计 924
7.2.4 通用原子设计 926
7.2.5 SaaS 分级 929
7.2.6 服务管理 931
7.2.7 安全与审计 932
第四章 SRE 的进阶 935
1 SRE 团队的生存与持续发展 935
1.1 SRE 团队的生存法则 936
1.1.1 没有 SRE 岗位行不行 936
1.1.2 SRE 团队的预算从哪里来 937
1.1.3 SRE 上一级领导的视角 937
1.1.4 SRE 团队的生存法则总结 938
1.2 SRE 团队的阵地与产能 941
1.2.1 SRE 团队的阵地 941
1.2.2 SRE 团队的产能提升 942
1.2.3 SRE 团队的工程经济学思维 943
1.3 SRE 团队的货币化改革 943
1.3.1 SRE 团队的货币化方案 943
1.3.2 SRE 团队货币化的管理 945
1.3.3 SRE 团队的货币化工具 946
1.4 SRE 货币化后的持续发展 946
1.4.1 优化服务项的“商业”模式 947
1.4.2 SRE 内部的精细化管理 948
1.4.3 持续扩展货币化收入 950
2 SRE 团队的综合算力调度 951
2.1 基础资源管控 952
2.1.1 生命周期管理 952
2.1.2 属性及关系管理 955

2.1.3 成本管理 960
2.1.4 资源管控通道 964
2.1.5 资源节点管理 967
2.1.6 作业管理 969
2.2 公共基础设施 972
2.2.1 网关 973
2.2.2 权限管理 976
2.2.3 用户管理 977
2.2.4 登录管理 978
2.2.5 审计中心 979
2.3 流程及自动化基础设施 980
2.3.1 ITSM 980
2.3.2 标准运维 982
2.3.3 应用配置管理 984
2.4 算力调度基础设施 986
2.4.1 集群管理 987
2.4.2 运行时管理 994
2.4.3 应用治理 1011
2.5 算力市场与供给管理 1021
2.5.1 算力市场 1022
2.5.2 供给管理 1026
2.6 通用算力消费能力 1030
2.6.1 环境治理 1032
2.6.2 容量管理 1036
2.6.3 流量治理 1039
2.6.4 容灾规划 1046
2.7 专有领域算力消费扩展 1051
2.7.1 研发领域算力消费场景扩展 1052
2.7.2 低代码平台算力消费场景扩展 1060
2.7.3 金融证券领域算力消费场景扩展 1063
2.7.4 游戏领域算力消费场景扩展 1067

2.8 算力能效与安全控制 1071
2.8.1 算力能效控制 1071
2.8.2 算力安全控制 1075
3 SRE 团队的数据与 AI 场景（编写中） 1079
4 SRE 团队的基础监控与应用观测（编写中） 1079
附录 1079
1 参考文献 1079
2 术语 1079

修订记录

1.0.7 修订记录

本次新增约十三万字

- 新增第四章第 2 节《SRE 团队的综合算力调度》，新增基础资源管控、公共基础设施、流程及自动化基础设施、算力调度基础设施、算力市场与供给管理、通用算力消费能力、专有领域算力消费扩展、算力能效与安全控制等内容
- 新增第五章第 5.4 节《混沌工程与全链路压测》，包括概述、体系设计、关键流程等内容，并新增《腾讯游戏 SRE 混沌工程及压测实践》《携程全链路压测平台的建设与运营》《小红书混合云容灾治理实践》《蚂蚁大促场景下全链路压测体系构建与保障实践》《腾讯游戏高可用保障体系实践》《小米混沌工程与全链路压测实践》《小红书基于流量地图的活动保障实践与 AI 探索》共 7 个案例

1.0.6 修订记录

本次新增约六万字

- 新增了第四章“SRE 进阶”，供 SRE 管理者参考，相关模块包括了“SRE 团队的生存与持续发展”、“SRE 团队的综合算力调度”、“SRE 团队的数据与 AI 场景”、“SRE 团队的基础监控与应用观测”，其中本次发布“SRE 团队的生存与持续发展”，其他模块后续奉上
- 章节架构调整：将原第二章 SRE 组织架构并入第一章；新增第二章 SRE 的基础（编写中）
- 第三章第 5 节《故障应急》：结构进行了全面优化，新增“xx 银行”、“腾讯 IEG”、“小米米家”、“广发证券”等多个案例，并对原“美图”案例展望部分进行了更新

1.0.5 修订记录

本次新增约三万字

- 第三章第 5 节《故障应急》：结构进行了全面优化，新增 B 站和蚂蚁的两个案例
- 调整“应用服务 SLI/SLO/SLA”和“重大技术保障”章节至本章，并在后者新增《OPPO 春节业务保障》案例
- 大幅完善故障应急体系章节内容

1.0.4 修订记录

- 第三章第 2 节《研发保障》结构进行了优化，并增加了《某大型游戏全球研发保障实践》等合共 2 个案例，新增 2.7 万字
- 第三章第 5 节《故障应急》结构进行了优化，依据 2024 年 6 月 22 日北京小米站沙龙更新并增加了《小米故障应急响应经验分享》等合共 5 个案例，新增 4.5 万字

1.0.3 修订记录

- 第三章第 4 节《变更管理》依据 2024 年 4 月 13 日上海 B 站沙龙更新约 4 万字，包括 6 篇不同类型的企业案例

1.0.2 修订记录

- 增加了版权声明为 CC BY-ND 4.0
- 修正了目录没有 3.1.1 的问题
- 修改了页眉的时间点
- 修正了部分错别字

第一章 SRE 整体介绍

1.1 前言

2003 年，Google 启动了全新的“SRE 团队”，旨在通过软件工程方法提升系统可靠性。随着 SRE 理论与实践在 Google 日臻成熟，这一理念逐渐扩散到全球科技行业。自 SRE 引入中国以来，引发了众多企业的关注与实践探索，但各企业的实施路径和效果存在显著差异。

与此同时，中国互联网行业涌现出一批 SRE 倡导者，为社区发展做出了重要贡献：孙宇聪翻译出版了《SRE：Google 运维解密》，赵成在极客时间开设了《SRE 实战手册》课程，赵舜东在社区积极布道分享，等等不胜枚举。

2022 年，由赵成等人牵头，首批来自互联网、运营商、金融等领军企业的 SRE 团队负责人齐聚一堂，组建了 SRE 研讨社区。社区定期开展分享活动，共同探讨 SRE 在各企业的发展路径，交流实战经验，并汇总形成这份源自一线实践、详实且持续更新的《SRE 实践白皮书》。社区每年吸纳新成员，逐年更新白皮书内容，力求真实客观地呈现国内企业 SRE 团队的工作现状。

在长达两年的编写过程中，我们见证了不同企业对 SRE 的理解差异，并努力统一相似场景的定义；我们观察到不同企业对 SRE 职能的拓展，并将成功经验提炼为案例供业界参考；我们也记录了各 SRE 团队的真实演进，及时将变化更新至白皮书中。未来，我们将按季度更新各 SRE 团队在职能定位、组织架构、技术迭代等方面的最新实践。

2023 年，中国信息通信研究院（以下简称"信通院"）云计算与大数据研究所（以下简称"云大所"）稳定性保障实验室的专家加入了 SRE 研讨社区，深度参与社区交流，为《SRE 实践白皮书》的编写提供专业指导。

1.2 SRE 发展历程

SRE 运动在全球的发展经历了 20 年，下面是部分重要事件：

- 2003 年，Google 成立了第一个 SRE 团队；
- 2010 年，Facebook 拥有了一个 SRE 团队；

- 2014 年，USENIX 协会主办的首届 SREcon（网站可靠性工程会议）在美国举行，大会成为 SRE 专业人士交流经验和最佳实践的重要平台，标志着 SRE 作为一个独立且重要的专业领域在全球范围内的正式认可。
- 2016 年，前 Google SRE 孙宇聪翻译出版了首部中文专业书籍《SRE：Google 运维揭秘》，在国内引起了很大的反响，很多企业开始学习并成立自己的 SRE 团队；
- 2016 年，Netflix 成立了"核心 SRE 团队"。Uber 开始撰写有关其如何使用 SRE 的文章；
- 2016 年，蚂蚁集团在国内成立了第一支 SRE 团队，主要攻坚容灾架构，后续拓展到高可用、资金安全等多个业务风险领域；
- 2017 年，LinkedIn 开始宣传其" SRE 文化"；
- 2017 年，浙江移动正式组建应用 SRE 团队，开始收口 IT 系统的集成部署、应急保障、架构治理等工作职责，加速了传统企业的运维数字化的转型进程；
- 2018 年， 赵成在某次 SRE 的聚会上，拉起了 "聊聊 SRE"微信群， 国内 SRE 人才开始聚拢，SRE 社区初步成型，并逐步成为最具影响力 SRE 中文社区；
- 2021，阿里 CTO 线第一支横向 SRE 团队成立，隶属于技术风险与效能部，负责集团全局稳定性保障、资源成本等方面工作；
- 2022 年，腾讯在内部技术岗位设置中，新增了 SRE，标志着腾讯内部 SRE 体系的正式成立；
- 2023 年，信通院云大所稳定性保障实验室牵头编制《服务韧性工程（SRE）成熟度模型》标准，推动该领域深入研究与实践应用，并在稳定性保障实验室成立了专门的"SRE 工作组"。

1.3 SRE 的目标

Site Reliability Engineering（SRE）的主要目标是通过结合软件工程和系统运维的最佳实践，提高大规模分布式系统的可靠性、可用性、性能和效率。以下是部分 SRE 追求的核心目标：

- 可靠性：SRE 的首要目标是确保服务和系统的可靠性。这包括减少故障、提高系统的稳定性，以确保用户在任何时候都能够获得一致的高质量服务。

- 可扩展性：SRE 致力于设计和实施能够随着用户需求增长而扩展的系统。这涉及对系统的架构和资源进行优化，以便在不降低用户体验的情况下，适应实际工作负载持续不断的峰谷状态变化。
- 性能：SRE 关注系统的性能，旨在确保系统能够在合理时间内快速响应用户请求。这包括对系统瓶颈的持续监控和优化，以提高整体性能。
- 自动化：SRE 倡导自动化运维工作，以减少人为错误和提高效率。通过自动化，可以更快速地部署新功能、检测并响应故障，并合理地开展系统的升级和维护工作。
- 监控和告警：SRE 强调对系统的全面监控，以便及时发现并解决问题。通过设置有效的告警系统，可以在重大问题发生前迅速做出反应，从而减少对用户的影响。
- 故障恢复：SRE 强调迅速而有效地恢复服务，以最小化用户体验的中断。这包括制定完善的应急计划并持续演练。
- 企业实现 SRE 核心目标的过程并不相同，落地路径各异。不论 SRE 部门（团队）在企业中的存在形式和所处位置，SRE 相关实践工作存在于大量流程中。这些工作流程与研发、测试、运维、产品运营等团队紧密地融合在一起，所有参与团队都在上述共享的 SRE 目标上做着各自的贡献。

1.4 SRE 的组织架构

SRE 团队在组织中的存在意义主要是确保系统的可靠性和高效运行。通过引入 SRE 角色，组织可以更好地实现开发软件开发速率和系统稳定性之间的动态平衡，从而实现更高水平的可用性、性能和自动化。通常 SRE 团队在组织中使命如下：

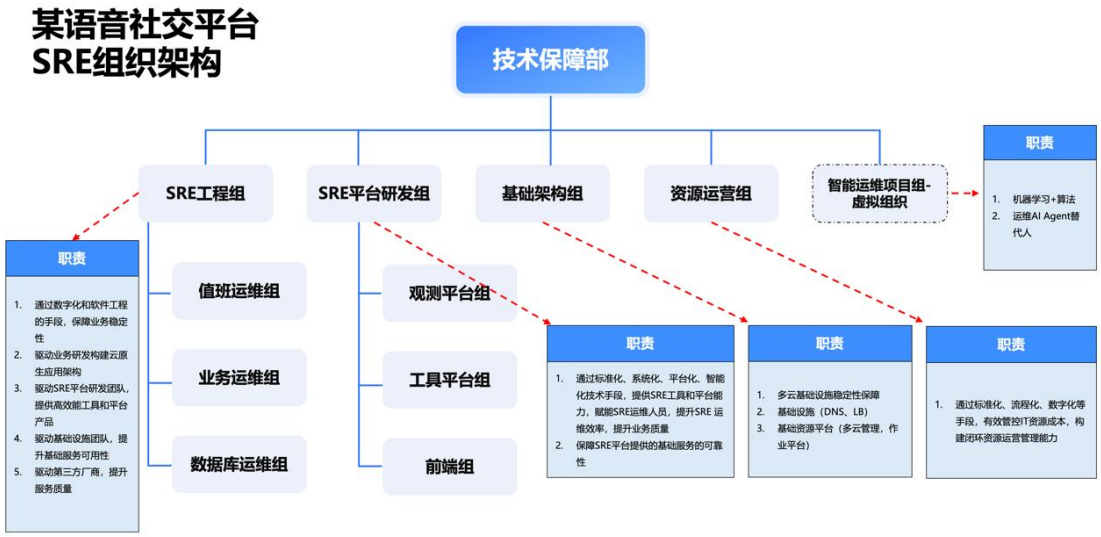
- 可靠性优先：SRE 团队致力于确保服务的高可用性和可靠性。他们关注系统的稳定性，采取工程化方法来减少故障和提高系统的稳定性。
- 自动化运维：SRE 团队推动自动化运维工作，以减少手动操作的错误和提高效率。通过自动化，可以更快速、可靠地进行部署、监控、故障检测和修复等操作。
- 质量保证：SRE 团队参与服务的全生命周期，包括设计、开发、部署和维护阶段，以确保系统在不同阶段都能保持高质量。

- 快速创新：SRE 通过标准化的可靠性基础设施（如自动化容灾、内建可观测能力）以及预防性工程实践，协同开发团队将运维负担转化为创新资源：一方面以可量化的风险护栏显著降低试错成本，另一方面借助自动化能力释放开发精力，从而系统性地加速高质量创新。

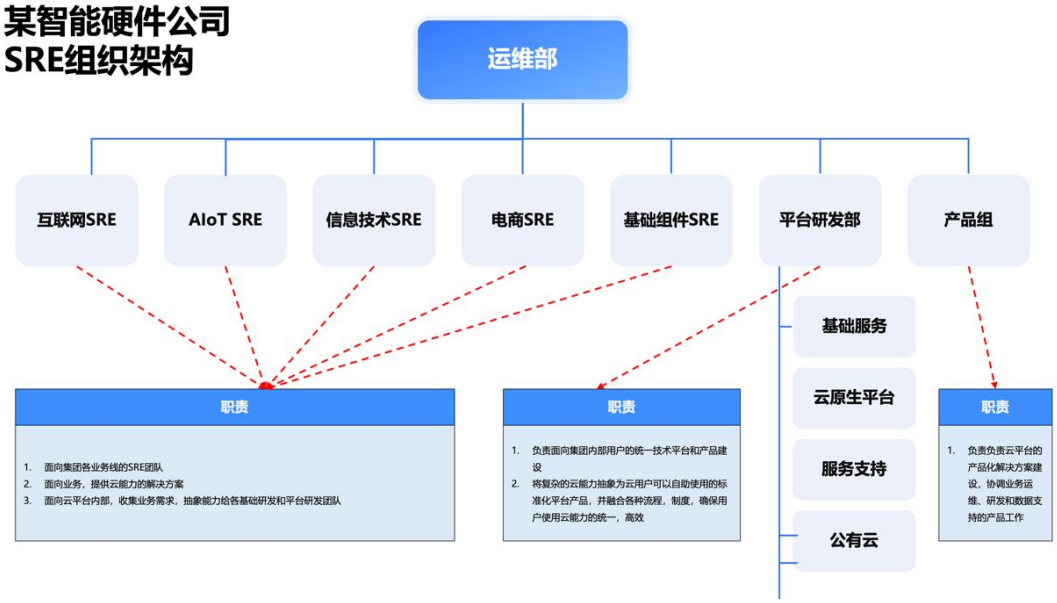
在组织架构中，SRE 团队的存在形式可以各不相同，这主要取决于组织的规模、业务需求和文化。以下是一些常见的 SRE 团队的存在形式：

- 中心化 SRE 团队：由一个专门的 SRE 团队负责支持整个组织的可靠性工作。这种模式有助于集中专业知识，确保在整个组织中实施一致的最佳实践。
- 嵌入式 SRE 团队：SRE 团队成员被嵌入到各个产品或服务团队中，与开发团队紧密合作。这种模式有助于更好地集成可靠性工作到产品开发的全过程中。
- 混合模式：一些组织采取混合模式，既有中心化的 SRE 团队，又在一些关键项目中嵌入 SRE 角色。这种方式能够兼顾专业化和贴近业务的优势。每种存在形式都有其优势和适用场景，关键在于根据组织的需求选择最合适的模式。不论哪种方式，SRE 的目标都是通过自动化和工程方法提高系统的可靠性和效率。

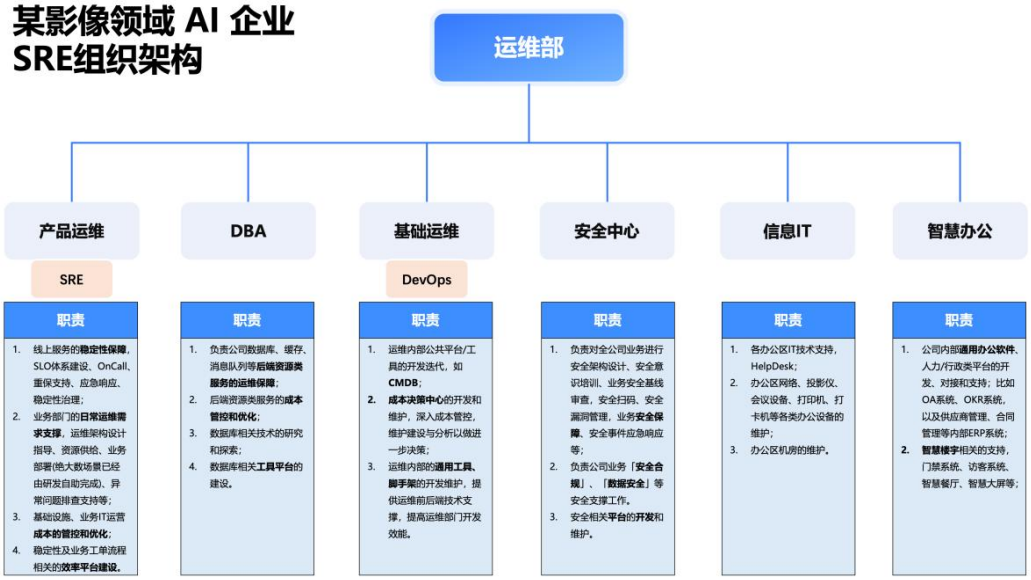
下面是国内某几家一线互联网 SRE 团队在组织架构中的设置模式。

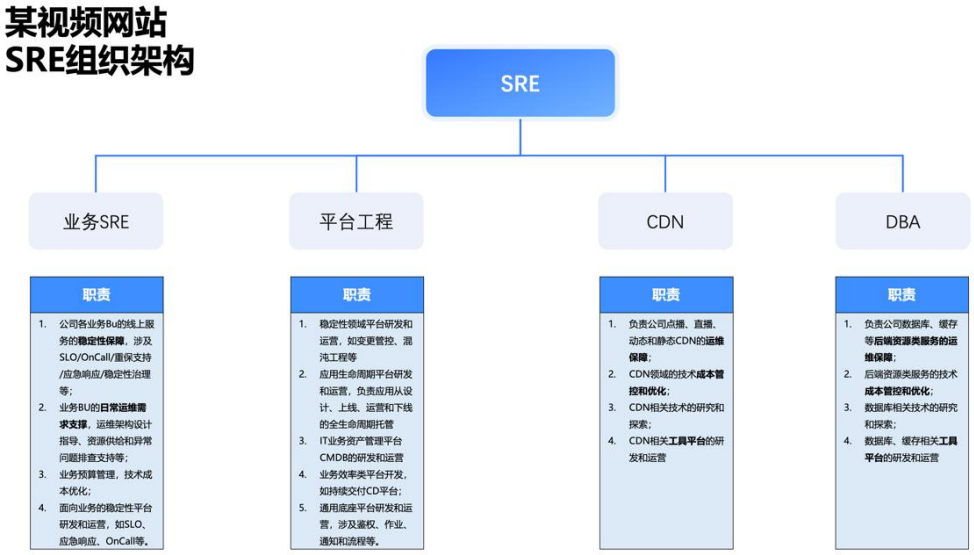


某智能硬件公司 SRE组织架构



某影像领域 AI 企业 SRE组织架构



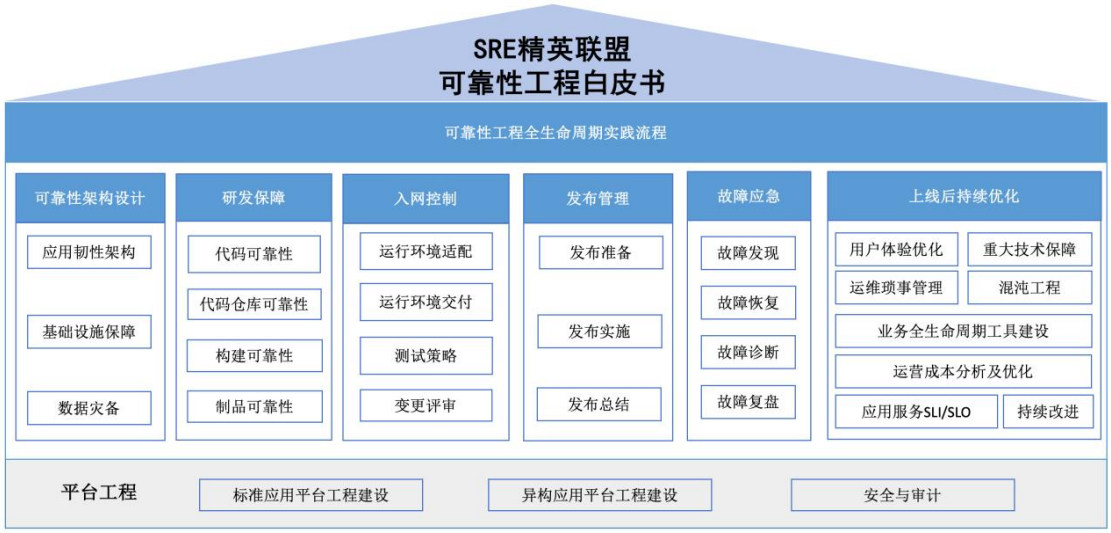


参考以上各个公司 SRE 团队在组织架构中的位置，通常 SRE 团队需要承担以下几项职责：监控、事故响应、事后回顾、测试与发布、容量规划、工具开发和可用性改进等。由于各个公司的业务形态的不同，SRE 团队在组织架构中也有不同的定位和名称，包括：SRE 产品运维、互联网 SRE 组、AIoT SRE 组、信息技术 SRE 组、业务 SRE 组等。

第二章 SRE 的基础（1.0.8 发布）

第三章 SRE 的职能

1 可靠性架构设计



可靠性架构设计是指在进行系统架构设计的过程中，根据系统的可靠性需求，采用分布式设计、解耦设计、冗余设计等高可靠性的架构设计方案，以提升系统的可靠性。

在进行可靠性架构设计的过程中，SRE 团队需要将应用架构设计流程完全融入其中，并与研发团队共同参与架构设计和评审工作。在系统设计阶段，应尽量消除可能出现的单点、依赖、容量等潜在风险，并提前为可能出现的系统架构风险做好应急准备。

1.1 应用韧性架构

1.1.1 分布式设计

在系统中，存在被划分为职责明确、粒度合适且易于管理的组件，这些组件（如计算资源、业务部分、数据等）都可以进行分布式的部署和运行。组件之间相互独立、互不干扰，通过分布式设计可以提高开发效率和可靠性。组件的拆分和分布可以通过复制、根据功能进行垂直拆分或根据用户访问模式水平拆分等形式。

在设计时应该充分考虑到组件间可能存在的相互干扰以及如何平衡不同组件之间的负载，并将系统所承受的压力进行均匀分配，以减轻压力对系统整体性能的不良影响。

1.1.2 解耦设计

在架构设计中，可以将各种逻辑功能划分为不同的服务模块，确保不同模块的故障对其他模块的影响是最小，从而最大限度地降低模块之间的耦合度。通过这种方式，可以将系统划分为多个相互独立的功能模块来实现。尤其值得注意的是，业务的主要逻辑与其他非核心模块是独立的，因此业务非核心模块的故障并不会对业务的核心功能产生负面影响。

1.1.3 冗余设计

为了确保资源有足够的安全余量，每个组件都需要有足够和合理的冗余实例，以确保单一组件实例的失效不会对业务的正常运行造成影响。对于不同类型的组件，我们需要明确地定义冗余量和冗余类型。在实际应用中，由于设备故障或者操作不当等原因导致服务器出现性能下降或崩溃现象时，系统会出现异常状态并产生大量信息。应用程序可能部署多个机房，当这些机房中有数据冗余时，一个位置的错误可以通过另一个位置的数据进行修正，确保整个系统的连续性和可靠性。为了提高系统可靠性，通常采用读-写分离的技术进行数据的冗余管理。读写分离是一种冗余的设计方式，缓存和数据库之间存在数据冗余，当缓存服务宕机时，可以从数据库回源到缓存。

1.1.4 熔断设计

熔断机制是应对雪崩效应的一种微服务链路保护机制，如果目标服务的调用速度较慢或超时次数较多，则此时会熔断该服务的调用。对于后续的调用请求，不再继续对目标服务进行调用，直接返回预期设置好的结果，可以快速释放资源。一般来说，熔断需要设置不同的恢复策略，如果目标服务条件改善，则恢复。

1.1.5 限流设计

限流是一种系统设计技术，用于控制访问应用程序或服务的流量，防止资源过载。常见的限流策略包括固定窗口、滑动日志、漏桶和令牌桶算法。这些方法可以帮助系统应对高流量，保持稳定性和可靠性。在实施时，通常需要结合其他系统保护措施，如队列、缓存、服务降级和熔断，以实现全面的流量控制和系统保护。

当流量被限制后，系统通常会采取以下措施之一：拒绝多余的请求、将请求排队等待处理、返回错误码（如 HTTP 429 Too Many Requests），或者提供一个降级的服务响应。这些措施可以缓解服务器压力。

1.1.6 降级设计

降级机制是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略地降级，以此缓解服务器资源的压力，释放服务器资源以保证核心任务的正常运行。从降级配置方式上，降级一般可以分为主动降级和自动降级。主动降级是提前配置，自动降级则是系统发生故障时，如超时或者频繁失败，自动降级。其中，自动降级可分为超时降级、失败次数降级、故障降级。

1.1.7 可观测设计

为了保证系统的透明性并迅速定位问题，采用可观测的设计方法变得尤为关键。可观测设计涵盖了日志记录、实时监控、追踪以及度量等多个方面，从而实现了系统状态和行为的可量化以及可分析性。在可观测的设计中，日志应当详细地记录所有的关键事件，监控系统需要能够实时捕获关键的性能指标，跟踪机制应具备跨服务请求的追踪能力，度量指标则应全方位地反映系统的健康状态。另外，健康检查机制需要自动地对系统组件状态进行评估，当出现异常指标时，告警机制会立即告知相关的工作人员。通过这些措施，我们可以清晰地观察到系统的运行状态，从而为后续的维护和优化工作奠定了稳固的基础。

1.2 基础设施保障

1.2.1 机房多活

系统所部署的机器或所在地须具备一定的冗余性。包括同机房多活、同城多活和异地多活等不同级别。将要建设的机房要求具有独立性，尤其是网络环境，机房之间通过专线来进行连接。

1.2.2 网络容灾

数据中心之间的互联网络是 DC 之间业务连接的重要载体，对存储灾备网络的时延要求较低、带宽较大、可靠性较高；业务灾备网络需要实现链路备份和快速的路由收敛。

1.3 数据灾备

1.3.1 数据备份

即对核心数据进行备份和恢复的能力。需对核心数据进行实时备份，并具备快速容灾切换的能力。需对备份恢复的能力进行周期性验证。

1.3.2 数据回滚

在系统出现异常情况下，迅速有效地恢复故障前数据状态，减少了故障给业务系统带来的冲击。回滚是否有效取决于回滚执行过程和回滚决策是否及时。

2 研发保障

通常说法是，SRE 处置的线上可靠性问题中，有 70%左右源自 CD 阶段（详见第三章第 4 节“变更管理”），15%左右源自研发阶段（例如代码的低质量、高维护成本等）。对于初创的 SRE 团队，应该将工作重心放在 CD 与 CO 阶段，做好救火工作，同时夯实 CD-CO 阶段的工程化，保障 SLA。在有阶段性成果之后，应该逐步向研发服务扩展进入 CI 领域，一方面尝试缩减那 15%的可靠性根因，更重要的是打通 CI-CD，以乙方服务心态保障业务团队的研发连续性，追求平台工程的研运一体化理念。

有些企业是由质量团队或者独立的效能团队承担这一部分职能，也有业务研发兼职的情况，SRE 团队应该本着以往运维团队的服务传统，将业务研发环境视为一个独立的线上业务，以服务心态尝试“接管”研发环境中代码仓库、CI 流水线、各类测试环境、代码分析扫描平台、制品仓库等研发基础设施的运维工作，并保障其可靠性，进而提高业务团队的生产效率，降低其出错概率。

有能力的 SRE 团队接下来可以依托自身的工程化能力将这些工具升级改进，与 SRE 前期已经建设的 CD-CO 平台整合，形成覆盖代码全生命周期的“研发运维运营”一体化的平台工程基础设施。这种发展路径借助运维 SRE 与业务开发团队的紧密关系减少了内耗，CI-CD-CO 全路径由 SRE 团队建设避免了轮子，还减少了例如 PE、DevOps 等小众的公司级岗位设置。

研发过程可靠性，指的是以 SRE 理论驱动研发连续性建设，提升研发管线的工业化水平，保障版本能够高质量迭代，从而持续保障业务的线上可靠性，同时兼顾了版本迭代效率；

SRE 主导的研发连续性保障，可以拆解为代码可靠性、代码仓库可靠性、构建可靠性和制品可靠性四个方面，每个阶段分别对其可靠性进行定义并提出相应的改善措施；

2.1 研发保障体系设计

2.1.1 代码可靠性

代码是基于一定需求实现，用于构建对应软件的文件集合。代码质量从基础上决定了软件的成败，是软件开发过程中不可忽视的一环。在软件版本快速迭代的今天，如何构建高质量的代码更显得至关重要。

代码可靠性的落地仅靠宣导或者文档还远远不够，需要建设完善的检查工具并量化效果，一般由平台 SRE 建设相关的工具，因此需要平台 SRE 需要深入了解影响代码可靠性的常见问题及提升措施，不断完善代码检查工具的能力；

2.1.1.1 代码缺陷

代码缺陷是指影响代码稳定运行的问题，或者未达到设计时的预期功能。缺陷产生的原因有多种，比如：软件的复杂性、编写的错误、需求歧义等。代码缺陷的及时发现与修复，对项目进度与工程质量至关重要。

代码缺陷检测，一般可采用静态分析法或动态分析法。

静态代码分析是指无需运行被测代码，通过词法分析、语法分析、控制流、数据流分析等技术对程序代码进行扫描，找出代码隐藏的错误和缺陷，如参数不匹配，有歧义的嵌套语句，错误的递归，非法计算，可能出现的空指针引用等等。

动态分析方法则一般应用于软件的测试运行阶段，在软件程序运行过程中，通过分析动态调试器中程序的状态、执行路径等信息来发现缺陷。

代码缺陷规避措施

代码缺陷种类较多，无法完全罗列，这里选取部分最为常见的缺陷并介绍对应的规避办法：

- 指针错误使用

指针的错误使用，一般要避免空指针、野指针，避免指针类型不匹配，不返回局部变量的指针。

- 内存非法访问

非法内存访问是指程序试图读取或写入未分配/受保护的内存，如数组越界等，这将会导致程序的不可控行为。因此，必须确保程序正确地分配和释放内存，避免缓冲区溢出等现象。也可使用静态分析工具检测代码中的内存非法访问问题。

- 变量未初始化

使用未初始化的变量，可能导致未知错误。一般来讲，变量需要在声明时赋予初始值。

- 资源泄漏

常见的资源泄漏包含 socket 泄漏，文件句柄泄漏，内存泄漏等。产生原因是未能正确释放已经分配的内存或其他资源，导致这些资源被长期占用。资源泄漏不仅会造成资源的浪费，系统性能下降，严重时超出系统限制会导致程序崩溃。

避免资源泄漏，在编程过程中，需要在资源使用完毕后进行资源的释放，如 socket/文件句柄的关闭，动态分配内存的释放。

- 竞争死锁

竞争死锁是指多个线程或进程持有资源，又互相竞争等待对方资源而导致死锁的情况。解决死锁问题，一般可采用超时使用机制、统一获取资源顺序和死锁检测机制来打破死锁产生的必要条件。

- 不当的 API 使用

不当的 API 使用，会导致程序异常。可通过仔细阅读 API 文档，了解 API 的使用方式，在使用 API 前进行充分测试的方法来规避。

效果评估

代码缺陷数作为代码质量指标之一，可以从数量、严重程度来归类。在度量层面，一般使用百行告警数来衡量代码缺陷指标。通过配置代码缺陷规则集，采用缺陷检测工具扫描，生成检测报告。

根据缺陷的严重程度分为严重告警（空指针、数组越界等），一般告警（变量未初始化等）和提示告警（如代码风格等）。设定各个告警等级的权重，统计代码行数，最终计算出百行代码缺陷告警数。

百行缺陷告警数=（严重告警 W1+一般告警 W2+提示告警 W3. /代码行数 100

（W1/W2/W3 为权重系数）

2.1.1.2 代码规范

代码规范主要是指是否遵守了团队或者业界的编码规范。代码规范主要涵盖：代码风格（如注释、空代码块、命名、格式化等），与异常处理等部分。

代码规范有助于提高代码可读性与可维护性，从而提升团队内开发效率。代码可读性帮助相关技术人员能够轻松阅读并理解代码意图与实现方式。代码从分支发起到主干的合并请求前，必须进行代码检查，这也是提前发现问题的方法之一。

代码规范提升措施

代码风格

良好的代码风格会帮助开发人员阅读和理解符合该风格的源代码，并且避免错误。此处所讲述的代码风格包括但不限于：命名规范，表达式与语句，缩进，对齐，注释，代码布局等。对于不同的编程语言，适用于不同的代码风格，对于同一项目或开发团队，应当使用统一的代码风格。

- 命名规范，命名要能够直观地表达本身的意图，同时具备可读与可搜索性，尽量遵循一些通用的写法。在此基础上命名长度应当尽量精短，避免触发代码行字符数限制规范；
- 缩进，一般采用 4 个空格缩进，而不使用 tab 键（特殊语言除外）
- 统一字符编码格式，通常采用 UTF-8 编码
- 单行字符数限制，长度一般不超过 120
- 行尾换行符，一般使用换行符 LF，禁止使用回车键 CR

异常处理

异常处理是为了防止一些未知错误产生而采取的措施。适当地使用异常处理能够提高程序的容错性。在处理异常方面，需要遵循：

- 只在可能出异常的块进行精准捕获处理；
- 捕获的异常必须处理或抛出给上层调用方；
- 异常处理效率较低，应避免使用异常做条件控制

效果评估

对于代码规范的效果评估，可以采用百行告警数来衡量。

百行告警数=（严重告警 W1+一般告警 W2+ 提示告警 W3）/代码行数 100

注：（W1/W2/W3 为权重系数）

百行告警数可以用来评估代码的质量和稳定性，较高的百行告警数可能意味着代码存在较多的缺陷和潜在问题，需要更多的测试和修复工作；

2.1.1.3 代码安全

安全性是指在为正常访问提供服务的同时，也能拒绝非法访问。同时不因为代码设计或实现的原因，导致信息泄露/非法侵入/系统崩溃等问题。

代码安全性提升措施

防止敏感信息泄露

敏感信息可分为系统敏感信息与应用敏感信息。系统敏感信息包含业务系统的基础环境信息，如系统版本、组件版本等；应用敏感信息包含用户信息和应用信息等，如用户TOKEN、密码、IP等。系统敏感信息泄露会为攻击者提供更多的攻击方法，应用敏感信息泄露危害则因泄漏信息内容而决定。

解决方法：

- 避免硬编码，禁止将密码等敏感信息写入代码，应该以配置或后台下发形式读取
- 处理异常时，避免将系统信息、DEBUG信息，或者敏感文件的路径输出到用户可见处

预防安全漏洞

代码安全漏洞，指编码过程中因不当的处理逻辑引发的安全风险；

常见的代码安全漏洞有：

- 脚本（SQL）注入，可通过减少拼接命令，对命令参数值进行过滤/校验避免
- XSS攻击，可以使用安全的JavaScript框架和组件，同时主动检测发现，转义输入等减少
- 越权访问，是由于权限设计错误，未授权用户获取甚至修改其他用户的信息。需要通过最小化原则的权限设计与审计来规避
- 通信安全，一般是由于未使用加密信道进行通信导致，可以通过使用加密/私有协议通信来避免

第三方组件安全

软件开发中不可避免地会引入依赖库，或者第三方 SDK。这些第三方组件作为系统的一部分，与原生代码并无本质区别，它们的安全性也同样影响整个系统。因此，需要在减少对第三方组件引入的基础上，加入相应的安全评估机制。

对于第三方组件的评估，我们主要从以下几个方面：

- 组件安全风险应在引入前/上线前/定期进行安全扫描
- 组件合规风险包括使用协议合规和监管数据合规
- 组件稳定性应当使用经过实际验证的 LTS 版本

效果评估

在衡量代码安全性方面，可以从敏感信息泄露、系统漏洞、第三方高危组件等几个方面来考量。可以通过代码扫描工具，扫描出已知系统漏洞与敏感信息，以及第三方高危组件的引入情况。

在得到敏感信息，安全漏洞个数，以及第三方高危组件个数后，可以制定代码安全性红线。一般来讲，敏感信息、安全漏洞是绝对不允许的。对于第三方高危组件，也要经过安全评估测试，其标准与本身代码相同。

(1) 对于敏感信息与安全漏洞，必须彻底清除

(2) 第三方高危组件，还可以采用 LTS 覆盖率可以用来衡量系统中使用的第三方高危组件稳定性。LTS 覆盖率=第三方高危组件 LTS 数/第三方高危组件数

2.1.1.4 代码圈复杂度

圈复杂度是一种代码复杂度的衡量标准，可以用来衡量一个模块流程判定结构的复杂程度。圈复杂度大说明程序代码的判断逻辑复杂，可用来表示对给定代码进行测试、维护或故障排除的难度，以及代码生成错误的可能性。同时，也可用来帮助开发人员确定是否需要对程序进行重构，以降低程序的复杂度，提高代码质量。

圈复杂度改善措施

降低函数圈复杂度主要通过对代码重构来进行，一般有以下几种方法：

(1) 将大函数拆分成多个小函数，每个小函数只负责单一功能

(2) 将条件判定提炼出来，成为独立函数

(3) 简化、合并条件表达式

(4) 优化循环结构，减少循环嵌套、使用更简单的循环结构等

效果评估

圈复杂度反映了代码的耦合度，圈复杂度越高的代码会有越多潜在的 BUG。

对于圈复杂度，可以从以下指标衡量：

(1) 单函数圈复杂度最大值小于等于 20

(2) 项目平均圈复杂度，一般不大于 4

项目平均圈复杂度=所有函数圈复杂度之和/所有函数个数

2.1.1.5 代码重复

代码重复指的是程序中存在相同或类似的代码段。在不同的位置或程序中出现相同的代码，会造成代码冗余和浪费。代码重复，不仅导致项目代码量的增加，影响程序的可读性和可维护性，增加代码的错误率和修改难度，也是设计不佳的一个标志。

代码重复的表现形式多种多样，常见形式有：

(1) 完全一样的代码

(2) 仅重命名标识符的代码

(3) 仅变量赋值不一样的代码

(4) 插入或删除语句的代码

(5) 重新排列语句的代码

代码重复改善措施

降低重复代码，是代码优化的重要方面之一，一般需要对相关功能进行重构。

常见的改善措施，主要是抽取公共代码、封装函数、使用继承和多态等

(1) 抽象出公共方法或函数，将重复的代码封装在一个函数或方法中

(2) 使用继承或接口，将共同代码放在父类或接口中，子类只实现自己的特定部分

(3) 使用设计模式，如工厂模式、模板方法模式等

(4) 利用现有的框架或库，避免自己重写

效果评估

代码重复的度量，可以使用代码重复率来表示，可以通过静态扫描工具得出。

代码重复率，指的是在一段代码中重复出现的代码段的比例。代码重复率越高，代码的可维护性和可读性就越差。

代码重复率=重复行数/代码总行数

在实际的工程中，一般建议：

(1) 单文件代码重复率最大值小于等于 10%

(2) 项目平均代码重复率小于等于 10%

2.1.1.6 代码注释与 API 文档

代码的注释与 API 文档编写是软件开发过程中非常重要的一部分，可以提高代码的可读性和可维护性。通过代码注释可以帮助阅读者快速理解代码的功能和实现方式。API 文档则是其他开发者了解使用软件的重要途径。开发者应该养成写注释和文档的好习惯，为自己和其他开发者节省开发时间。

代码注释与 API 文档提升措施

(1) 注释目的在于使阅读者能够快速掌握注释对象的使用方式与原理，良好的注释应包含注释对象的产生意图，设计考量与如何使用；注释一般应当包含文件注释，类/结构体注释，函数方法注释，变量注释以及适当的代码段注释；对于规范化命名的变量与简单函数方法，可以不进行注释。

(2) API 文档，应当描述各个类和方法的功能和使用方法，同时遵循行业和国际标准，具备兼容性和实时性。

效果评估

对于注释与 API 文档的考量，可以从 API 文档覆盖率，代码注释行密度来衡量。

注释行密度=注释行数/总行数 100*

此标准用于衡量百行代码中，所包含注释行数，一般认为低于 5 表示几乎没注释。

API 文档覆盖率=已覆盖 API 接口数量/总 API 接口数量

此标准用于衡量对外 API 文档的完善程度；一般来讲，至少需要达到 80%覆盖率，即覆盖大部分的 API 功能，忽略了一些不太重要或不常用的 API；同时也需要定期更新，使文档保持最新、全面、准确的状态。

2.1.1.7 代码质量红线

代码质量红线是指在开发过程中，开发团队所设定的一些规则和标准，用于确保代码质量达到一定的水平，也是衡量代码质量的一个综合考量。这些规则和标准通常是基于行业最佳实践和经验总结制定的，是团队开发的一种约束和保障。当代码质量超越红线时，就需要开发团队及时进行修正和优化，以确保代码质量和运行稳定性。

质量红线改善措施

质量红线的触发标准是基于每个质量指标的。提升每项指标质量有助于避免触发红线，也可以帮助开发团队提高软件开发的效率和质量，减少错误。

效果评估

质量红线一般包含以下几个指标：

(1) 代码缺陷

(2) 代码风格

(3) 代码安全性

(4) 圈复杂度

(5) 代码重复率

(6) 代码注释和文档

(7) 单元测试覆盖率

通过在代码合并、转测等场景下，对以上每个指标单独设定阈值，可以划定出不同场景下的质量红线，当某项指标触发质量红线时终止后续的 CI/CD 流程，并要求开发团队进行修复。

2.1.2 代码仓库可靠性

代码仓库就是存放源代码和资源的地方，亦称版本库、代码库，其核心功能是版本控制，记录一个或若干文件的变化，以便后续查看特定版本修订的情况;

代码仓库出现问题，对代码拉取、项目开发、编译构建等都会造成影响，所以代码仓库的可靠性是整体研发流程可用性的关键一环;

代码仓库的可靠性包括：仓库性能、仓库容灾、仓库安全和仓库可扩展性四个方面;

代码仓库的可靠性主要侧重网络优化、部署优化、配置优化、安全提升等工作，可由服务 SRE 和安全人员来承担相关能力的建设;

2.1.2.1 仓库性能

代码仓库的性能通常是指代码仓库对代码的存储、管理和处理时的速度和效率，包括代码提交和拉取的速度、分支合并的速度等，高性能的代码仓库，可以减少开发人员的等待时间，缩短产品交付周期;

代码仓库性能提升措施

(1) 控制代码仓库的大小：代码库的大小会直接影响仓库的性能，因为大型代码仓库需要更多的时间来处理和查找文件。因此，需要合理控制仓库大小，及时删除不需要使用的文件，并合理设置文件的保存周期;

(2) 合理设置代码仓库结构：如果代码仓库的结构合理，可以更快地查找和访问文件，从而提高性能，例如，某些版本管理软件，支持大文件单独存储在仓库之外，仓库中实际只存储一个很小的文本指针，可以将存储大文件的目录设置使用更适合的存储形式；

(3) 版本工具选型：不同的版本控制工具可能会对性能产生不同的影响。例如，Git 和 SVN、P4 的架构和设计理念的差异，在处理大文件的性能存在差异，需要根据业务资源文件的数量和大小、团队的多地分布特性等综合选择；

(4) 网络优化：如果多个人同时访问代码仓库，网络连接的速度也会影响性能，在评估代码仓库性能时，需要考虑网络连接的速度和质量，可在离用户就近的网络区域，部署边缘节点，缓存最近的版本，减少网络距离传输损耗，提高访问速率；

(5) 硬件升级：硬件也会影响代码仓库的性能，例如，使用较高 IO 性能的磁盘，可以提高代码仓库的读取速度；

(6) 集群化：通过集群化来部署代码仓库，可以让代码仓库支持更大规模团队的使用；

效果评估

一般采用下面 3 个指标来评估仓库的性能

(1) 文件下载速度：通常是指每秒传输的数据量，常见的单位有比特/秒（bps）、千比特/秒（Kbps）、兆比特/秒（Mbps）和千兆比特/秒（Gbps）等；

(2) 下载卡顿率：是指从仓库中下载时出现卡顿的频率或时间占比，通常使用百分比（%）来表示；

(3) 并发请求量：一般团队多人同时拉取或者提交代码可能会影响速度，通常使用 QPS 来表示系统每秒钟的请求量；

2.1.2.1 仓库容灾

代码仓库容灾是指代码仓库在经受自然灾害、设备故障、网络故障、人为错误等不可预测的问题后，通过备份、容错机制和恢复策略在最短时间内恢复到正常可用的状态。完备的代码仓库容灾机制，可以避免团队或公司核心代码资产遭受损失。

代码仓库容灾提升措施

(1) 数据备份：定期对代码仓库的数据进行备份，确保在数据丢失或损坏时能够及时恢复。

(2) 多地备份：将备份数据存储在多个地方，以防止单点故障。

(3) 容错机制：使用容错技术，如 RAID 等，以防止硬件故障导致数据丢失。或者将本地普通硬盘替换为云硬盘，云硬盘中的数据以多副本冗余方式存储，会避免数据的单点故障风险。

(4) 灾备恢复策略：制定灾备恢复策略，以便在发生灾难时能够及时恢复。

(5) 人员培训：对相关人员进行培训，提高应对灾难的能力和应变能力。

效果评估

一般使用以下几个指标来评估代码仓库容灾效果

(1) 恢复时间目标（RTO）：Recovery Time Objective，他是指故障发生时间到故障恢复时间，两个时间点之间的时间段称为 RTO；

(2) 恢复点目标（RPO）：Recovery Point Objective，是指系统恢复到怎样的程度。这种程度可以是上一周的备份数据，也可以是上一次的实时数据；

(3) 投入产出比（ROI）：Return of Investment，容灾系统的投入产出比，可以使用最高的性价比方案来达到容灾效果，为团队节省成本；

2.1.2.3 仓库安全

代码仓库的安全性是指代码仓库中存储的代码等数据受到保护的程度，以防止未经授权的访问、篡改、泄露和破坏。保护代码仓库的安全性包括但不限于访问控制、数据加密、代码审查、安全漏洞、操作审计、私有网络部署等。高安全性的代码仓库可以保护代码的机密性，完整性，避免因安全漏洞造成团队或者公司的损失和风险。

代码仓库安全性提升措施

(1) 访问控制：评估代码仓库中代码的访问控制机制，包括用户认证、授权、权限管理等，确保只有授权的用户能够访问仓库中的代码。

(2) 数据加密：评估仓库中存储的关键元数据或者敏感代码是否采用了合适的加密技术进行保护，以防止敏感信息泄露。

(3) 代码审查：评估代码审查机制，确保代码质量和安全性。详细可参考：4.2.1.3 代码安全。

(4) 安全漏洞：评估代码仓库中可能存在的安全漏洞，包括代码中的漏洞、第三方库中的漏洞等。

(5) 操作审计：评估代码仓库的操作审计，确保所有操作可追踪，以便及时发现和应对安全事件。

(6) 私有化部署：将代码仓库部署在私有网络中，例如，企业、学校的内部网络中，相比公网中的外部代码托管服务理论上会更安全；

效果评估

安全往往只有 0 和 1 的概念，要么安全，要么不安全，不存在中间状态，因此代码安全可靠性的效果评估可通过代码泄露等安全事件的发生与否来评估，同时通过审计能力来保障安全事件发生时具备可回溯能力；

(1) 代码泄漏次数：代码仓库代码泄露事件的次数；

(2) 漏洞数量：代码仓库中发现的漏洞数量；

(3) 访问控制：是否具备精细化权限控制的能力；

(4) 审计能力：是否具备良好的审计能力；

2.1.2.4 仓库可扩展性

代码仓库的可扩展性是指代码仓库在面对不断增长的代码量、用户数量、访问频率等变化时，能够保持高效的性能，同时可以方便地通过软硬件来扩展功能和架构，以满足未来的需求，代码仓库的可扩展性对一个快速发展中的团队尤其重要。

代码仓库可扩展性提升措施

(1) 分布式版本控制系统：使用支持分布式版本控制系统，比如 Git，可以支持代码仓库的架构可扩展性，可以将代码库分散在多个服务器上，从而实现横向可扩展；

(2) 集群化：使用集群化技术，如 Kubernetes 等容器编排系统，可以实现代码仓库的自动化部署和管理；

(3) 功能扩展：使用插件等能力，结合业务的个性化诉求，扩展版本控制系统的功能，满足团队的研发需求；

效果评估

代码仓库可扩展的能力可使用以下两个指标评估

(1) 架构可扩展性：代码仓库架构是否支持水平横向扩展。

(2) 功能可扩展性：是否可以通过插件扩展版本控制系统的功能。

2.1.3 构建可靠性

构建是指在构建机上把代码、资源文件等源文件编译打包成可执行的程序文件的过程；在当前的持续集成/持续交付的软件开发模式下，若构建出现问题，则新的软件版本无法快

速发布验证，软件质量就会受到影响，所以构建的可靠性对于软件服务的可靠性和迭代效率起到了重要作用；构建可靠性主要由构建效率和构建成功率两个方面组成；

构建可靠性提升涉及构建工具建设与规划、业务层改造优化、软硬协同等多个方面，一般需要由平台 SRE、业务开发、服务 SRE 多角色共同参与；

2.1.3.1 构建效率

构建效率即构建速度，取决于从构建启动到构建结束的耗时情况，构建耗时过长的话软件版本无法按时交付，对业务版本迭代效率产生了影响，构建可靠性就无从谈起。

构建效率提升措施

(1) 流程自动化

利用自动化构建工具或脚本把构建各个环节串联起来，减少各环节之间的等待时间；

(2) 并行化

通过把一些构建流程从串行改成并行，优化构建流程，提升构建速度。

(3) 增量构建

在构建机上执行一次构建时把构建过程中的一些临时文件、中间产物存起来作为构建缓存，等下一次构建时，由于一般情况下只有部分代码被修改了，那么没有被修改的代码就能免去构建环节，直接使用上次的构建缓存，这样就通过增量构建的方式减少了需要构建的内容，降低了构建耗时。

针对构建机首次构建时没有缓存的问题，可以搭建构建缓存共享服务器（例如 UE 引擎的 DDC 服务器），一台构建机的构建缓存会生成到缓存共享服务器上，供其他构建机使用。

(4) 分布式编译

相对单机有限的资源来讲，集群的力量无疑是强大的：一个人计算 100 道数学题，相比 100 个同样能力的人一人计算 1 道题，孰优孰劣不言而喻。分布式编译加速就是利用集群的资源，将单个节点的工作分配给一大批节点，然后再汇总结果。根据需要，资源数量可以近乎无限地扩充，不再受制于单机的物理架构；时间上，集群工作所需要的时间往往是原来的几分之一。

(5) 软硬协同

部分构建任务比如代码预处理、资源文件处理无法通过分布式编译加速分发到远端，只能在构建机本地处理，此时本地构建机性能就成为瓶颈，可以针对性地提升本地构建机的 CPU/内存/磁盘 IO 性能，再结合分布式编译系统，软硬协同提升构建速度。

(6) 多进程编译

有些编译软件默认只开启单进程编译，导致构建机硬件性能未得到充分利用，此时可以通过开启多进程编译来提升构建速度。

效果评估

(1) 基于基线：使用构建耗时超出基线比例来评估单次构建的效果，每次正常构建完成后，把本次构建耗时上报上去，持续若干天后，我们就能得到一段时期内的多次稳定构建耗时数据，把这些数据求一个平均值之后，我们就得到了这一段时期的构建耗时基线。当一次构建耗时超出基线很多时（比如超出基线 20%），这次构建就可能出现了性能问题，在得到构建耗时基线以后，我们可以将当前构建耗时与基线进行对比来作为 SLO，比如不超过基线 10%即为健康；

(2) 基于阈值：按不同的业务实际情况设置不同的阈值，例如设置构建耗时不大于 2H 为可靠性的衡量标准；

2.1.3.2 构建成功率

构建成功率指指定时间内，构建成功次数占构建总次数的比例。构建成功率低的话，需要多次构建才能产生可交付的版本，也是影响构建可靠性的一个关键因素；

构建成功率提升措施

(1) 保障构建环境可靠性

构建成功率受到构建环境影响，当构建机异常时（比如缺少某个依赖包，连不上代码仓库，磁盘故障等），构建就会失败。我们需要保证构建环境的可靠性，比如通过自动化方式批量部署构建机，避免手动部署时遗漏某些依赖包；尽量利用云上的高可靠性网络、计算和存储。

(2) PreBuild 预编译检查

通常的构建是拉取代码后再执行后面的编译流程，这就要求开发编写完代码后必须提交到代码仓库再启动构建。其实可以在提交代码到仓库之前就把问题暴露出来，问题越早发现，修复解决的成本越低；提交到代码库的代码质量越高，问题越少，团队协作起来就越顺畅，越高效。PreBuild 就是在提交代码到仓库之前，利用本地 PreBuild 工具进行本地代码检查或传输代码到远端进行预编译代码检查，从而尽早发现问题，实现质量左移。

效果评估

(1) 基于基线评估：每次构建完后将成功/失败状态进行上报，统计一天的构建成功率，持续若干天后，我们就能得到一段时期内的多天稳定构建成功率，把这些数据求一个平均

值之后，我们就得到了这一段时期的构建成功率基线。当某天的构建成功率超出基线很多时（比如超出基线 20%），这次构建就可能出现了性能问题。在得到构建成功率基线以后，我们可以将当前构建成功率与基线进行对比来作为 SLO，比如不低于基线 10%即为健康。

(2) 基于阈值评估：按业务需求设置固定的成功率阈值，例如 80%，可以取一个固定周期的所有构建进行统计分析并进行对比，例如以天/周/月等单位。

2.1.4 制品可靠性

制品为构建过程的产物，包括软件包、测试报告、应用配置文件等，最终提供给研发、测试等多个角色，用于下载并部署到 PC、console、移动端、服务器等多种不同的设备，从而完成发布和交付；

制品的可靠性分为制品下载可靠性、制品部署可靠性、制品安全可靠性三个方面；

制品可靠性提升涉及制品库工具建设与规划、安全、部署分发等多个方面，一般需要由平台 SRE、服务 SRE、安全人员共同参与；

2.1.4.1 制品下载可靠性

随着企业的快速发展，研发团队规模的扩大，为拓展全球市场在国内海外多地建立研发团队，制品的高效交付和管理也成为影响研发效率的关键，制品分发缓慢、下载困难等为代表的制品库不可用挑战，急需快速解决；

制品下载可用性提升措施

(1) 多地分发：根据研发的不同地域分布，按需设置制品的分发策略，实现构建完成后多地制品分发，从而达到提升制品速率的目的；

(2) P2P：制品的下载往往具备一定的峰值规律，例如早高峰会出现大量的集中下载，为提升并发下载的速率，可以使用 P2P 的下载策略，下载用户越多，速率越快；

(3) 专属工具：制品的下载使用专属下载工具，能够支持分片下载、断点续传、多线程、热点缓存等特性；

(4) 镜像源加速：建设距离用户更近用户的镜像源，来提升制品依赖的拉取速度，减少下载中断；

效果评估

(1) 下载成功率，下载成功率=1-（失败请求数/用户请求数），失败请求是指制品库返回的错误码为服务器内部错误码的请求（错误码>500.；但不包括触发频控导致的限流请求或者制品库升级、变更、停机而导致的失败请求。用户请求指的是 制品库服务器端接收

到的用户发送的请求，但不包括未经身份验证、鉴权失败或者欠费停服状态下的请求。用户端由于黑客攻击而对制品库的请求，或者由于配置了跨区域复制、生命周期规则而在后端异步执行的请求，均不视为有效请求或失败请求

(2) 下载速度，下载速度=包大小/下载耗时，下载速度和制品的存储区域分布及用户所在的区域有较大的关系，可针对不同地域、国家可设置差异化的可用性衡量指标；

2.1.4.2 制品部署可靠性

在制品下载完成后，还有一项重要的下游能力，即制品的自动化部署，使用户实现对制品产物即时体验的能力；

在大型业务研发过程中，多种研发角色手动安装制品产物非常耗时，在没有自动化部署的情况下执行诸如软件安装和升级会消耗大量研发人员的时间和精力，且对迭代的效率有较大的影响；

制品部署是指通过技术手段，打通构建流水线，将构建完成的制品主动推送到用户多种类型终端的过程，用自动化的方式对制品进行分发、安装、更新，让用户以较低的时间成本获取到构建产物，减少临时下载制品对迭代效率的影响，尤其对于具备超大制品的业务类型，制品部署的可靠性尤为重要；

制品部署可用性提升措施

(1) 多用途支持：为满足不同的测试验证目的，一个项目往往会设置多条构建流水线，例如用于逻辑、性能、开发调试等多种不同类型，制品部署能力需要支持到多种类型；

(2) 多平台支持：制品的分类按平台分有 iOS、安卓、PC、console 等，每种平台都有多种不同类型的 OS 版本和设备型号，碎片化程度高，预部署能力需要良好的机型兼容适配能力，能够支持多种不同类型的设备和平台；

效果评估

(1) 部署成功率，部署成功率=成功部署次数/总分发次数

(2) 部署人工耗时，一个完善的预分发方案应该尽量减少用户等待的时长，通过部署人工耗时指标来驱动减少人工参与；

2.1.4.3 制品安全可靠性

制品在持续构建过程中的包依赖，以及构建完成的产物（含 Docker 镜像、npm、helm、maven 等多种不同类型的格式），可能存在一些安全风险，导致制品不可靠；

制品安全性提升措施

(1) 漏洞扫描，制品需要经过漏洞、license信息的扫描与分析，并对漏洞和不合规的license告警并输出安全合规报告，漏洞库需要及时更新；

(2) 设置质量红线，禁止下载未经安全扫描或者未通过安全扫描的制品；

(3) 访问控制，针对不同角色设置差异化的权限，防止资源泄露，减少恶意盗取制品风险，权限以最小化为原则；

(4) 操作审计，提供制品库的操作审计功能，保证制品的下载使用等操作可追溯；

效果评估

(1) 漏洞扫描覆盖度：扫描能力能够覆盖足够多的制品库类型；

(2) 漏洞扫描速度：漏洞扫描的速度和及时性，能够在漏洞产生后，以最快的速度发现异常；

(3) 漏洞扫描准确性：漏洞库及时更新和维护，确保制品安全扫描结果的准确性；

(4) 访问控制：是否具备精细化权限控制的能力；

(5) 审计能力：是否具备良好的审计能力；

2.2 研发保障工程体系设计

2.2.1 面向研发保障的持续集成流水线

持续集成（CI）是现代软件开发中的关键实践，通过自动化的方式将代码的构建、测试、部署等环节有机结合，确保代码在每次提交后都能快速集成并验证，从而提高软件开发的效率和质量。面向研发保障的持续集成流水线设计，旨在通过一系列自动化工具和流程，保障工程编译，静态代码检查，测试用例运行和部署发布。

另外，通常情况下，平台需要提供以下关键组件能力：**流水线：**

通过可视化手段展示团队现行的研发流程，实现编译、测试、部署等环节的一体化管理，确保流程的顺畅与高效。

代码检查：

提供精细化的代码审查解决方案，从多个维度检测代码中的缺陷、安全漏洞以及规范性问题，确保产品质量得到充分保障。

代码库：

企业内部代码托管服务**凭证管理：**

为代码库、流水线等关键服务提供多样化的凭证与证书管理功能，增强系统的安全性。

环境管理：

可以将企业内部的开发编译机托管至流水线，统一管理**研发商店：**

由流水线插件和流水线模板构成，插件用于整合企业内部的多种第三方服务，模板则助力企业研发流程的标准化。

编译加速：

通过并行编译，编译缓存，硬件匹配优化的手段，显著提升构建任务的执行效率。

制品库：

通常基于分布式存储架构，具备扩展能力，其功能涵盖制品扫描、分发、晋级、代理、包管理等，同时提供多种依赖源仓库，如 generic（二进制文件）、maven、npm、pypi、oci、docker、helm、composer、nuget 等，以满足不同开发需求。

持续集成流水线的设计需要遵循以下几个关键原则：

- 自动化：尽可能将所有的构建、测试、部署步骤自动化，减少人为干预和错误。
- 快速反馈：确保每次代码提交后，开发者能够快速得到构建和测试结果，及时发现并修复问题。
- 可重复性：流水线的每一步都应该是可重复的，确保在不同环境下执行结果一致。
- 可扩展性：流水线设计应具备良好的扩展性，提供接口及相关的扩展机制，对接更多的插件，满足多样化的流水线能力需求。
- 可观测性：提供全面的监控和日志记录，便于问题定位和分析。

在流水线可视化设计上，尽量采用 Stage/Job/Task 三层结构展示，方便快速定位问题。

- Stage（阶段）
- 由多个 Jobs（作业）组成；
- 同一个 Stage 下的 Job 执行方式为并行，由于 Job 之间是相互独立的，某个 Job 失败后，其他的 Job 会被运行到完成；
- 一个 Job 失败，则该 Stage 失败。

- Job

可以运行在一个构建环境里，比如运行在 macOS；也可以作为不需要构建环境的普通任务调度编排。它有如下特性：

由多个 Tasks(插件)组成；

一个 Task 失败，则该 Job 失败，其余 Task 将不会运行；

Task

也被称为流水线插件，通常是一个单独的任务，如拉取 Git 仓库代码等。

Task 必须包含在 Job 内，同一个 Job 内的 Tasks 都是从上往下顺序执行（启用了高级流程控制的 Task 除外）。

2.2.2 面向研发保障的可观测设计

面向研发保障的可观测设计旨在通过全面的监控和分析，确保流水线的透明性和可控性。在流水线中，构建机承受着极大的工作负载压力，CPU 负载接近满负荷，同时占用大量内存和磁盘空间。此外，由于代码和配置问题，构建任务失败的可能性较高。通过可观测性工具，团队能够实时监控流水线状态，快速定位和解决问题。

要做好流水线的可观测性，要从以下几个维度进行考虑：**1. 构建机的监控 CPU 和内存使用率：**

监控构建机的 CPU 和内存使用情况，确保资源分配合理，防止资源耗尽导致构建失败。

GPU 使用率：

对使用 GPU 加速的构建任务，监控 GPU 的使用情况，包括利用率、温度和显存占用，以确保其高效运行。

磁盘空间：

定期检查磁盘使用情况，清理不必要的文件，确保有足够的空间进行构建。特别是游戏制品巨大**网络流量：**

监控网络流量，确保构建机与其他系统的通信顺畅，避免因网络问题导致的构建延迟。

2. 流水线监控**构建状态：**

实时监控每个构建任务的状态，包括成功、失败、进行中等，帮助快速识别问题。

任务队列：

监控流水线中的任务队列，优化任务调度，防止任务积压。

执行时间：

分析各个阶段的执行时间，识别性能瓶颈并进行优化。

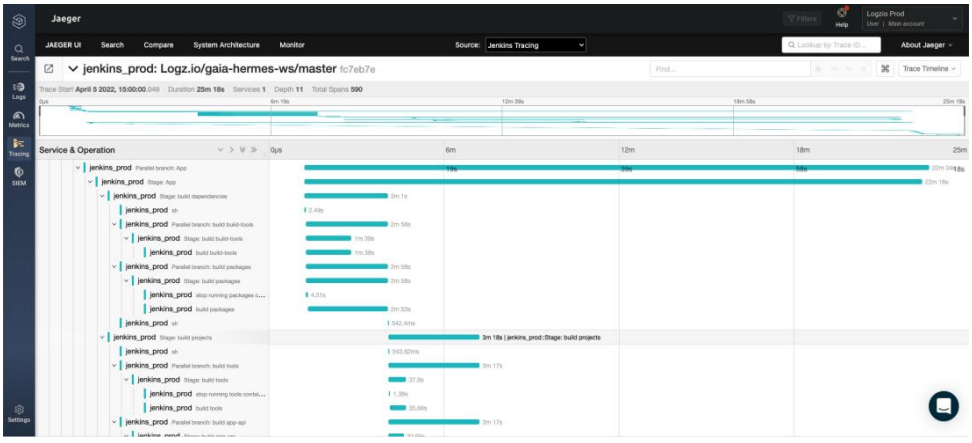
3. 日志监控

错误和警告：重点关注日志中的错误和警告信息，及时采取措施解决问题。

历史日志分析：通过分析历史日志，识别常见问题和趋势，改进构建流程。

4. Trace 跟踪

构建的过程基本上等同于一条 Trace， 通过对构建过程进行 OpenTelemetry 等链路跟踪协议后， 进行上传， 可以实现更细粒度的构建成功跟踪。如下图，通过上报到 Jaeger 后， 可以实现每个编译步骤的消耗时长的可视化。



5. 告警触达

该平台将指标、仪表盘以及告警信息推送至研发团队，使研发人员能够更全面地掌握持续集成环节中出现的问题，从而进一步释放系统可靠性工程师（SRE）的人力资源，并提高研发效率，降低沟通成本。

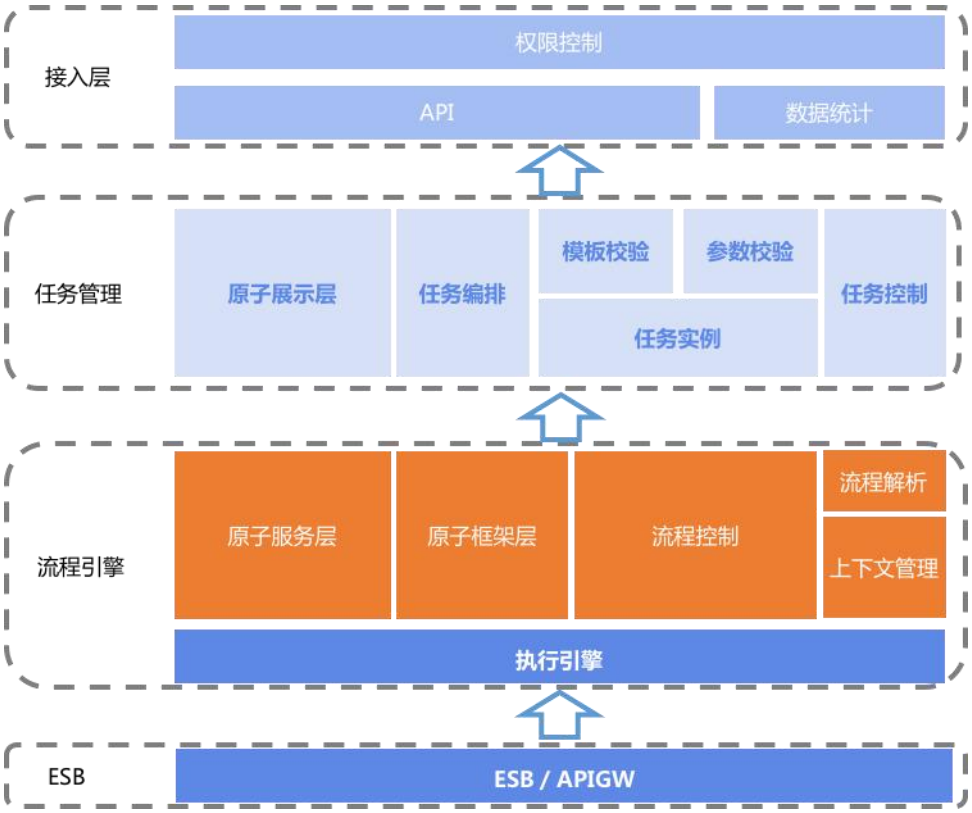
2.2.3 面向研发保障的操作调度操作平台

操作调度平台在研发保障过程中发挥了至关重要的作用，确保各项操作任务能够高效、准确地执行，并减少人为干预带来的风险。该平台主要负责以下几个关键工作领域：

1. CD 发布管理

在持续交付（Continuous Delivery, CD）过程中，操作调度平台承担着发布管理的核心职责。它不仅能确保从代码提交到生产环境的全过程自动化和高效运作，还能够通过智能调度来优化资源使用和任务执行的顺序。

自动化流程编排：操作调度平台通过自动化流程编排，将代码提交后的各项任务（如编译、测试、部署等）整合到一个无缝的流程中，并自动调度这些任务的执行顺序，以避免资源冲突或执行瓶颈。通过与 CI/CD 工具链的深度集成，平台能够在任务之间进行动态调度，确保每个流程步骤都在最佳时间点执行。



智能调度算法：平台利用智能调度算法，根据任务的优先级、依赖关系和资源可用性，动态调整任务的执行顺序和分配的计算资源。这种调度策略不仅能够优化资源利用率，还能提高部署效率，减少因资源不足或冲突导致的延迟。

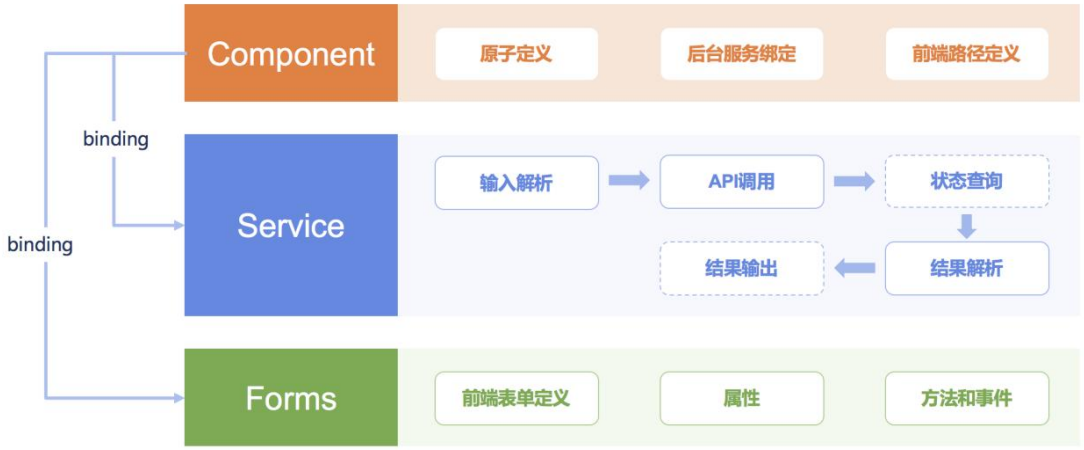
2. 批量化操作

批量操作是操作调度平台的另一重要功能，它支持对大规模操作任务的统一管理和调度，极大地提高了研发团队处理日常运维任务的效率。

统一任务调度：操作调度平台能够将大量重复性、标准化的操作任务（如批量更新配置、批量重启服务等）整合到一套自动化流程中，并通过统一的调度机制进行管理。

并行执行与任务分片：为了提高批量操作的效率，平台支持并行执行和任务分片技术。它能够将大规模任务分解为多个小任务，并分配到不同的计算节点并行执行，从而显著缩短任务的整体执行时间。此外，平台还具备任务依赖管理功能，确保各子任务按顺序正确执行。

动态资源分配：批量操作过程中，平台能够动态分配计算资源，根据任务的实时需求调整资源分配策略，以确保在高峰期也能保持高效执行。



2.2.4 面向研发保障的 ITSM 平台

IT 服务管理平台（IT Service Management, ITSM）在研发保障中起到了至关重要的作用，为研发团队提供了统一、标准化的管理工具和流程，确保研发过程中的各类服务请求、事件管理、问题解决、变更控制等工作能够高效、有序地进行。

2.2.5 面向研发保障的容器平台

容器平台为研发团队提供了一个高效、灵活、可扩展的环境，以便快速部署、测试、迭代应用。

为了有效支持研发团队的工作流程，容器平台必须具备一系列专门设计的功能，确保研发过程的高效和稳定。

自动化 CI/CD 管道集成：

容器平台与 CI/CD 工具链无缝集成，支持自动化构建、测试、部署。通过 Jenkins、GitLab CI 等工具，容器平台能够在代码提交后自动触发构建和部署任务，大大缩短了研发周期。

环境隔离与多租户支持：

通过命名空间和 RBAC（Role-Based Access Control）等机制，容器平台可以实现环境隔离，支持多个团队和项目的并行开发。这种隔离不仅限于资源使用，还包括网络和安全配置，确保不同团队之间的独立性和安全性。

观测性平台对接：

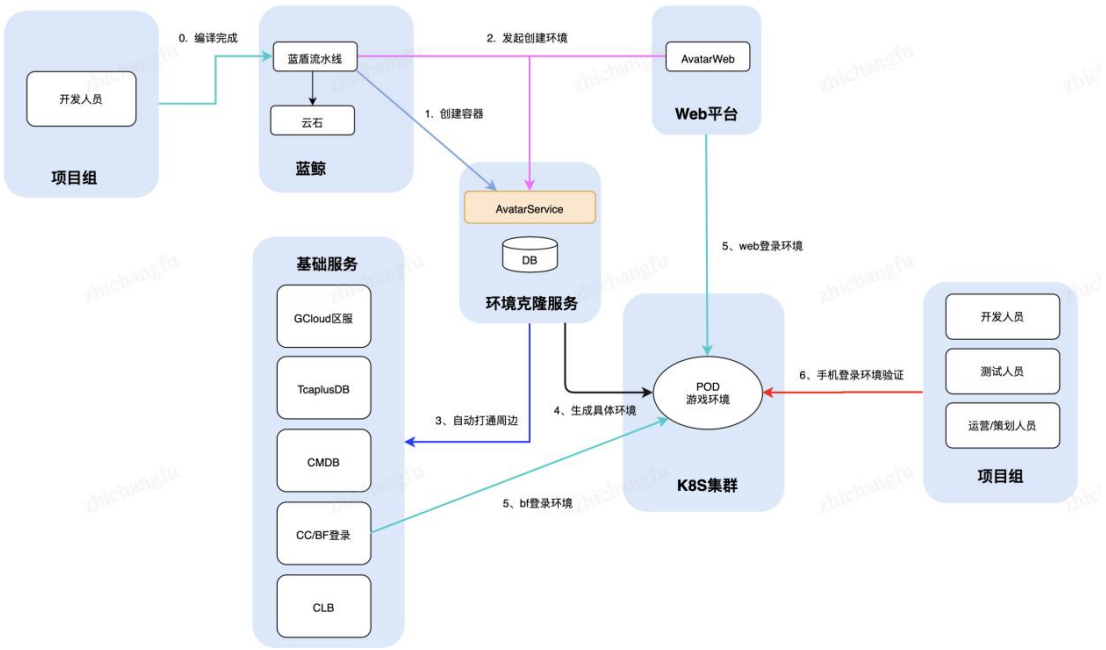
容器平台需要能与观测平台实现无缝的对接，将运行于容器的指标，日志，链路上报的观测平台中，并能实现监控告警。

敏捷发布与回滚机制：

容器平台支持蓝绿部署、金丝雀发布等敏捷发布策略，能够在不影响整体服务的前提下逐步发布新版本并监控其表现。一旦发现问题，平台可以快速回滚到之前的稳定版本，减少对业务的影响。

测试环境治理：

针对实际研发工作环节中，测试环境搭建流程过长，测试资源无法充分利用和测试设备日常维护人力成本高的问题，基于容器平台，以 k8s 工作负载的方式部署运行管理多个测试环境。具体架构如下

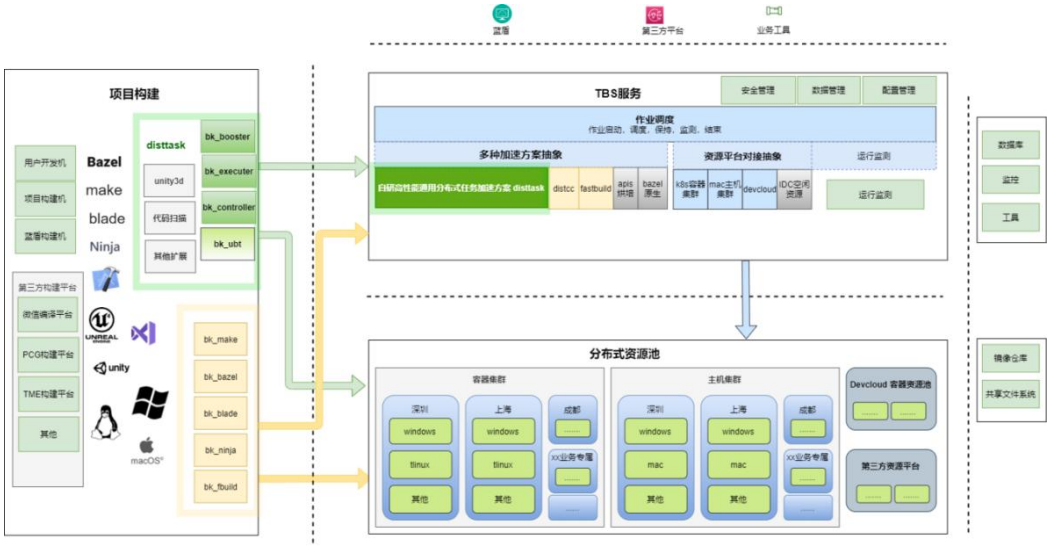


2.2.6 面向研发保障的编译加速平台

编译构建是项目开发和发布过程中的重要环节，同时也是非常耗时的环境，有些项目执行一次完整的构建需要几十分钟甚至几个小时，各种编译构建加速工具都能在一定程度上减小构建时长。

整体架构如下：

- 编译加速工具包运行于用户构建机，为构建机上的编译任务接入加速服务
- 分布式编译服务为集中式部署，负责接收编译加速工具包加速请求，调度加速资源，并负责加速任务过程全生命周期管理
- 分布式资源集群通过分布式编译服务引擎服务统一动态调度，直接为用户编译提供分布式加速



分布式加速底层通过 disttask 分布式任务引擎实现，disttask 各模块功能介绍如下：

- remoter worker 运行在分布式资源集群中，负责接收，执行和返回分布式任务
- local server 运行在构建机上，实现分布式任务底层基础功能，并可扩展不同应用场景的分布式任务实现
- disttask_executor 通用任务执行器，接管实际编译中的编译命令(如 gcc 命令，clang 命令)，是构建工具和分布式基础服务之间的桥梁
- 基于 disttask 提供的接口，根据实际场景需要，实现独立的构建工具

2.3 研发保障案例

2.3.1 腾讯游戏全球研发保障实践

SRE Elite 精选原因

这是一个完整的游戏行业研发保障案例。面对游戏研发中的复杂研发管线、大文件版本管理、冗长的构建过程和频繁的更新需求等挑战，SRE 团队通过稳定性保障、平台工具建设，以及与业务开发团队的有效分工，实现了高效的研发保障。

此案例覆盖了研发保障的多个关键模块，在代码可靠性，代码仓库可靠性、制品分发，以及构建加速等多个方面进行了优化，显著提升了代码提交和构建的成功率，并有效解决了代码库卡顿和文件分发效率低等问题。相关的优化内容非常详尽细节，具有很强的实践性，且大部分关键组件提供了开源的实现案例，非常值得参考。

（一）背景及设计原则

大型游戏研发对研发管线有独特的要求

伴随着腾讯游戏多元化的诉求日趋增长，越来越多的游戏开发团队开始尝试多地共研模式，这在工程版本控制、编译构建、制品共享等方面都提出了新的挑战。

游戏行业的研发具有以下独特特点和挑战：

1. 复杂且研发管线：用户提交的不仅是源代码，还可能包含体积达数百兆的单个素材文件，这些文件需要进行版本管理，而且还需要对原始素材文件为不同终端进行渲染，耗时巨大，出包过程可能长达 5～6 小时。而如果中途构建失败，第二天项目进度就会受到影响。

2. 版本库性能：版本库中包含大量大文件，整个代码库的体积可能超过上 PB，而且随着游戏生命周期的延续，代码库只会越来越庞大。以此未经优化前，用户在访问或修改版本库里头的资源时容易卡住，而游戏团队的规模可能达到上百人，人力成本极其高昂，如果卡住一个小时损失就会非常巨大。

3. 项目团队协作及代码质量问题：目前游戏的制作已经进入工业化生产的阶段，每个游戏项目可能都有几十人甚至上百人的参加，如何解决由于参与人数众多而带来的代码冲突以及代码质量不统一问题。

4. 业务制品构建耗时长，严重影响协同效率和敏捷开发。对于普通互联网应用安装包普遍在 500M 以内，单次构建在 30 分钟以内，游戏安装包和资源文件加起来一般可达 5G-20G 以上，客户端包单次构建耗时需要 2～5H 左右；

5. 激进的发布时间：游戏行业竞争极其激烈，游戏内部经常衍生出新的玩法和素材，用户期望极高，玩家口味变化极快，前期宣发费用极其高昂，项目必须如期上线才能减少前面投入的风险。

6. 频繁的更新：根据用户反馈，游戏需要频繁出包进行调整，并引入新玩法、新皮肤和新角色，

7. 制品分发效率及安全：现在的游戏动辄几十到上百 GB 的包，如何进行快速，稳定地分发，送到大量的测试及渠道用户。如何保障制品的安全。

8. 多方协作带来的安全问题：游戏行业已经从小作坊，演变成类似于好莱坞式工作室的分工协作模式，大量的资源需要通过外包的形式进行分包协作。由此带来安全问题非常突出，不少的竞争对手或者说相关的产业都会盯上我们的整体的研发流程，在游戏新版本

的研发过程中，一旦出现了资源的泄露，玩法的泄露，数值的泄露，有可能就会导致竞争对手或者黑产的狙击，很有可能导致版本需要重做，数百人月的工作量付诸东流。如牵涉到核心资产，如核心代码，核心引擎等，甚至会出现私服的情况，影响整个游戏的未来的生存。

举个例子，某日凌晨 1 时许，由于机房故障，部分流水线的构建机下线，我们本以为这么晚应该不会有用户进行构建，本决定第二天再修复。结果，凌晨 2 点和 3 点就有用户在公司论坛进行投诉，据了解，因为代码改动直接到了当天凌晨 1 点多，因此他们的构建时间只能选择在凌晨 2 点，才能满足版本在当天进行提测发版的进度要求。此事让我们更深刻理解到**研发管线的可靠性对业务的重要价值。**

总之，解决以上问题，做好研发保障，保障研发管线的高效安全运行，价值巨大，挑战巨大。

SRE 团队负责研发保障是一个合适的选择

研发阶段的可靠性是指，以 SRE 理论驱动研发的可用性建设，提升研发管线的工业化水平，保障版本能够按正常周期迭代，从而实现高质量持续交付有效价值的目标**从软件工程的角度：**

- SRE 的稳定性保障工作，本质上是软件工程的一部分，也就是运维左移，从源头保障软件的质量**从 SRE 专业能力的角度：**
- 利用 SRE 团队既有平台能力又有服务能力的特点，从平台和服务两个地方切入，建立 SRE 团队主导的工具平台和服务体系，让研发团队在平台和服务上需求都得到最大化的满足。
- 服务上，SRE 团队做研发服务，可以复用现有的运维支持体系，与研发相比，SRE 有高度的业务可用性意识和能力，能很好地保障业务研发工具链的可用性，防止因为工具链的不可用或性能低下，影响业务团队研发效率。
- 工具上，SRE 团队可以复用，开发，并扩展在 CO /CD 领域的工具去进行研发保障，例如，利用可观测的平台，对研发工具链的健康状态以及使用体验进行观测，度量研发工具链上下游健康状态，故障时快速定位根因，同时提供成熟的可观测落地方案，标准化实施，降低建设成本**从组织分工上的角度：**
- SRE 团队负责研发保障，业务开发负责产品功能开发，对于企业来说，可以发挥最大的投入产出比。SRE 可以更体系化和规模化进行全面保障。业务开发的

本质逻辑是更加聚焦快速开发迭代版本，完成产品功能开发。理想状态下，业务开发无需关注研发平台上下游所有组件的稳定性。

- 从职业发展来说，SRE 工程师在 SRE 组织中，有明确的成长路径和岗位通道，且通过从 CO/CD 扩展到 CI 域，扩展了职业发展方向，并且与相关能力设置在业务开发团队相比，更具有稳定性和成长性。

简单来说，SRE 团队是既有服务意识，又有平台及产品的建设能力，从软件工程的角度整体保障角度，完成了业务研发不擅长但是有迫切需求的工作。因此，我们认为由 SRE 工程师去从事研发保障的工作，是一个高 ROI，合乎组织利益及各方干系人利益，能提升研发效率，可持续的解决方案。

本案例最后总结部分，会有我们业务研发同学对 SRE 从事专业度的认可和反馈。

在此基础上，SRE 切入研发保障，需要注意以下三点：

稳定高效：

需要 SRE 团队在服务上有韧性，和研发团队建立信任。

低成本定制：

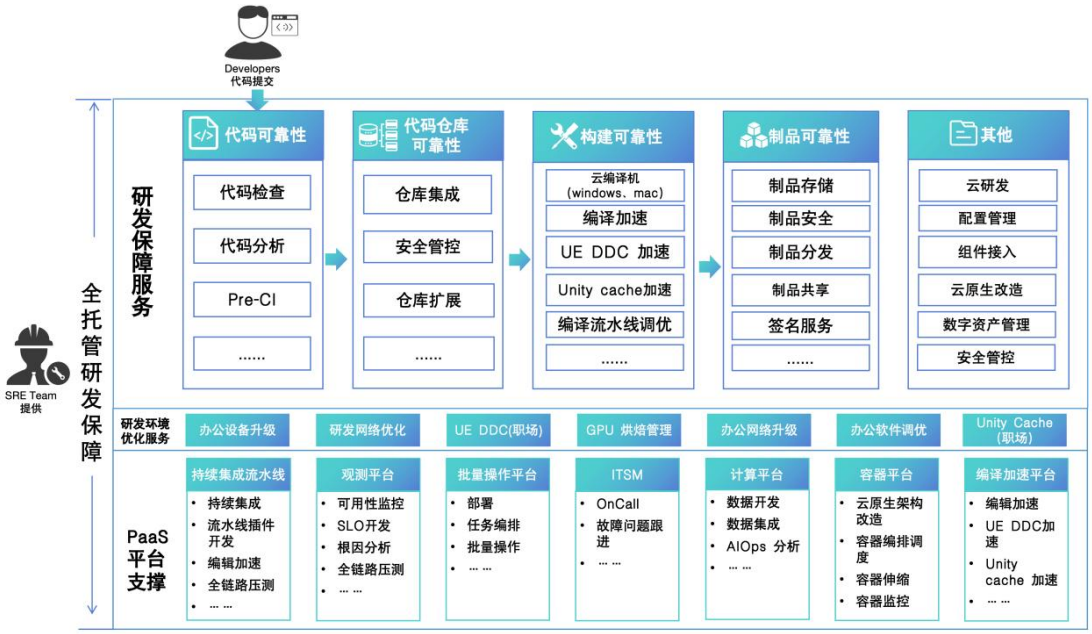
SRE 团队能够低成本地为每一个业务定制研发流水线方案，定制个性化需求，例如兼容多种代码库等这就要求我们的方案必须是 PaaS 模式，也就是平台工程的效果。

安全可靠：

一方面是研发团队对代码安全的要求，例如外包开发团队的代码安全。我们通过离岸云研发模式解决另一方是研发环境的安全性，避免代码泄露事件发生。

平台能力和服务能力方面，做到稳定高效，低成本定制，安全可靠，让用户对 SRE 团队提供的研发服务产生依赖，用户用得越深入，与 SRE 团队的绑定就越紧密，SRE 团队的研发服务就会可持续，从而建立长期服务关系。

（二）体系设计及关键流程



SRE 团队提供的全托管研发保障体系，涵盖从代码提交、代码仓库管理、构建流程到制品分发的全过程。通过代码分析、仓库安全管理、编译加速、制品共享等多项服务，确保了研发过程的高效性和安全性。大部分情况下，研发只需要往平台提交代码后，后继的环节将不需要进行干预，即可高效获取到相应的制品。

图中还包括由 SRE 团队研发并提供的支持的 PaaS 平台，SRE 团队利用相关的平台能力，提供持续集成、观测、批量操作、ITS 管理、计算和容器平台等功能，进一步优化研发环境。

整体体系强调 SRE 团队在提升研发效率、保障流程稳定性和安全性中的关键作用。

备注：案例中可能会提及各个平台的简称，此处进行统一进行备注说明：

- 蓝盾：蓝鲸持续集成平台的简称，开源地址

https://github.com/TencentBlueKing/bk-ci

- 标准运维：蓝鲸智云标准运维的简称，开源地址

https://github.com/TencentBlueKing/bk-sops

- ITSM：蓝鲸流程服务的简称，开源地址

https://github.com/TencentBlueKing/bk-itsm

- 容器平台：蓝鲸容器管理平台简称，开源地址

https://github.com/TencentBlueKing/bk-bcs

- BKTurbo：蓝鲸编译加速平台简称，开源地址

https://github.com/TencentBlueKing/bk-turbo

- CodeCC 代码检查：CodeCC 插件(腾讯代码分析插件），开源地址 https://github.com/TencentBlueKing/ci-CodeCCCheckAtom
- 制品库：蓝鲸制品库简称，开源地址 https://github.com/TencentBlueKing/bk-repo

(1) 代码可靠性研发保障实践

某游戏客户端通过质量红线保障各阶段代码质量

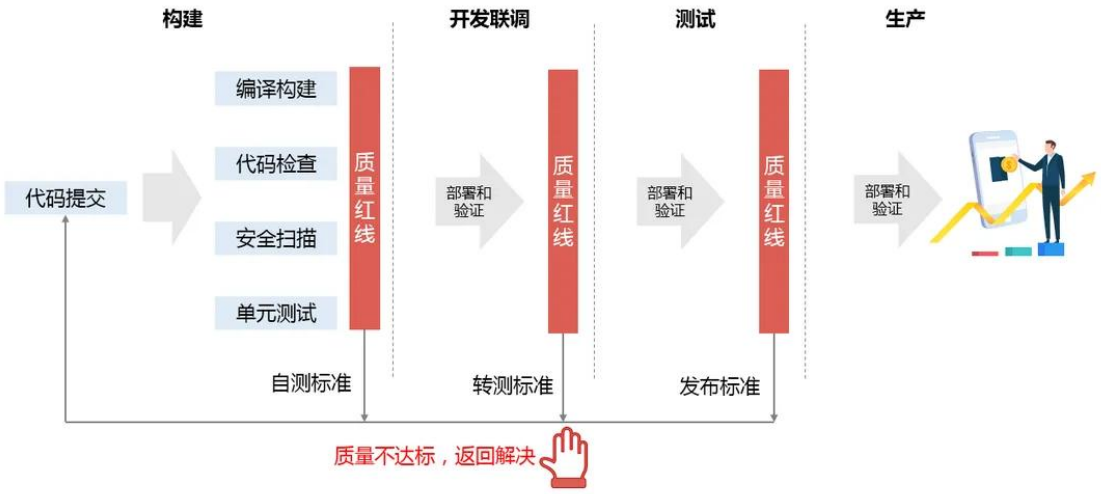
在某游戏客户端业务研发过程中，我们经常会有类似的困惑：

- 确立了团队的编码规范，但还是有不符合规范的代码被合入；
- 拥有自己的转测试标准，但是只能人工跟进，不能落实到自动化流程中；
- 在版本的后期，Bug 数居高不下。但很多问题是可以在前期通过代码检查和单元测试发现的。

为了解决上述问题，借鉴丰田精益生产的思想，腾讯打造了质量红线 Gate 服务。

质量红线的思想起源于丰田精益生产的立即暂停系统（stop-the-line Andon），又称为安灯（Andon）系统。当车间生产流水线上的员工遇到麻烦，立即拉一下信号灯，班组长就会立即跑过来帮助解决，其生产流水线也会停止直到问题解决。这样能够尽早暴露问题，解决问题，而不是把问题流到生产汽车的后续步骤。

解决方案



质量红线是指通过设置质量标准，控制流水线的行为，使得每一阶段的出口质量都必须符合质量标准的一种服务。

设计思想

设计思想如下

1、独立成质量红线服务，基于关键点控制质量

从用户使用场景出发，将开发、部署测试环境、部署生产环境等相关的插件（共 10 个）选出作为关键控制点。这些控制点插件在流水线中可以非常灵活地编排，但只要质量红线设置了标准，就必须质量达标控制点才能执行通过。

2、指标灵活可扩展

设计了统一的指标定义规范和指标灵活可扩展的机制。主要包括系统插件指标（主要是 CodeCC 代码检查）、脚本任务指标、研发商店插件指标；

3、支持为代码合入设立红线

支持设置质量要求，并通过流水线 Git/Github 事件触发机制与工蜂/Github 串联起来。以工蜂为例，当用户在工蜂发起 MR 时，会通过 Merge Request hook 触发流水线执行。流水线执行拉取代码、代码检查、单元测试等操作。若不满足质量要求，质量红线就会让流水线失败，并将失败的结果回写到 Code，作为 MR 的辅助决策。

4、与流水线融合

质量红线被设计为一个独立服务后，虽然有集中管理的便利，但用户对其感知度不高。只有用户正确配置了红线之后，才能在流水线看到，较难被用户发现。且用户创建红线时各项信息只能一项项填写，有时不知道怎么填比较好。为了用户更好地使用它，我们在设计上将红线和流水线服务进行了融合。

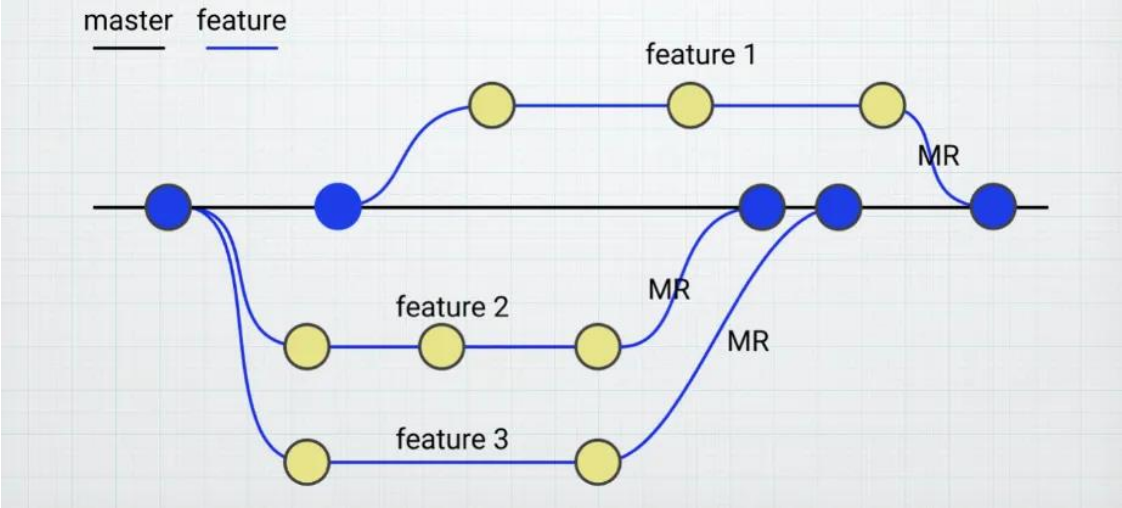
关键流程包括：

1 确立团队 Git 工作流

Git 相比 SVN，其优势不仅在于是一个分布式版本控制系统，而且在于其强大的分支管理能力。其中 Git 工作流，主要是需要定义好分支策略。

在项目初期，团队只有 2~3 人。此时往往不需要额外分支，一个 master 打天下。但是随着团队逐步壮大，大家发现代码冲突越来越严重，同时由于缺少代码检视（Code Review），部分质量较差的代码和无关内容也时不时被提交上去。

为了解决这个问题，“分支开发、主干发布”模式呼之欲出。团队从主干上拉出分支，并在分支上开发软件新功能或修复缺陷。当某个分支（或多个分支）上的功能开发完成后要对外发布版本时，通过 MR（Merge Request）合入主干。在 MR 时，进行 CR 代码检视。



2 确定团队质量标准

为了保证合入 master 的代码质量，团队需要设立自己的代码质量标准，例如编码规范、单元测试等。但是单单依靠人肉代码检视 CR，很难保证新合入的代码都符合规范。而工具自动化执行，能够做到客观准确、持续检查。因此我们可以将质量标准，放到腾讯 DevOps 平台提供的一系列工具之中。

以腾讯代码检查平台为例，可以支持五个维度的代码质量标准。像缺陷、安全和代码规范，都能通过规则配置来进行具体调整，以便适应不同团队的要求。

缺陷：BKCheck 工具可检查出空指针、内存泄漏、数据越界、并发缺陷等问题；

安全：敏感信息工具可以检测密码泄露、内部 IP 泄露等问题；

代码规范： 九款工具可以检测出逻辑、变量、代码风格、最佳实践等代码规范问题；

圈复杂度：可以检查出过于复杂的函数。函数复杂度越高，存在缺陷的风险越大；

重复率：可以检查项目中复制粘贴代码片段等问题，避免产生大量冗余代码。

以腾讯 DevOps 平台上的 Bash/Batch Script（脚本任务）为例，支持用户自由编写自己的脚本。例如可以支持单元测试，将单元测试用例执行失败数作为团队要求，也能支持获取 Tapd（腾讯内部需求管理/缺陷平台）遗留缺陷数作为质量要求。

3 配置流水线和质量红线

在确定了这些质量标准之后，如何将其与 Git 工作流无缝衔接起来呢？

3.1 创建一条 MR 触发执行的流水线

首先，我们需要配置一条 DevOps 流水线，它能在发起 MR 时自动执行，拉取相应代码进行分析。

添加 Git 事件触发的插件，并将事件类型设置为 Merge Request Hook。当工蜂项目中新增 MR 时会触发该流水线。

添加拉取 Git 代码的插件，并将分支名称设置为 ${hookSourceBranch}。${hookSourceBranch}是一个流水线变量，指代源分支。例如从 feature1 分支向 master 合并代码，那么 feature1 就是源分支，master 是目标分支。因为源分支在 MR 中可能会变化，有时是 feature1，有时是 feature2，有时是 feature3，因此使用流水线变量更加灵活方便。

添加 CodeCC 代码检查插件，选择语言和规则集。例如可以选择 IEG 缺陷检查、腾讯代码规范，还有啄木鸟安全检查、重复率和圈复杂度等。

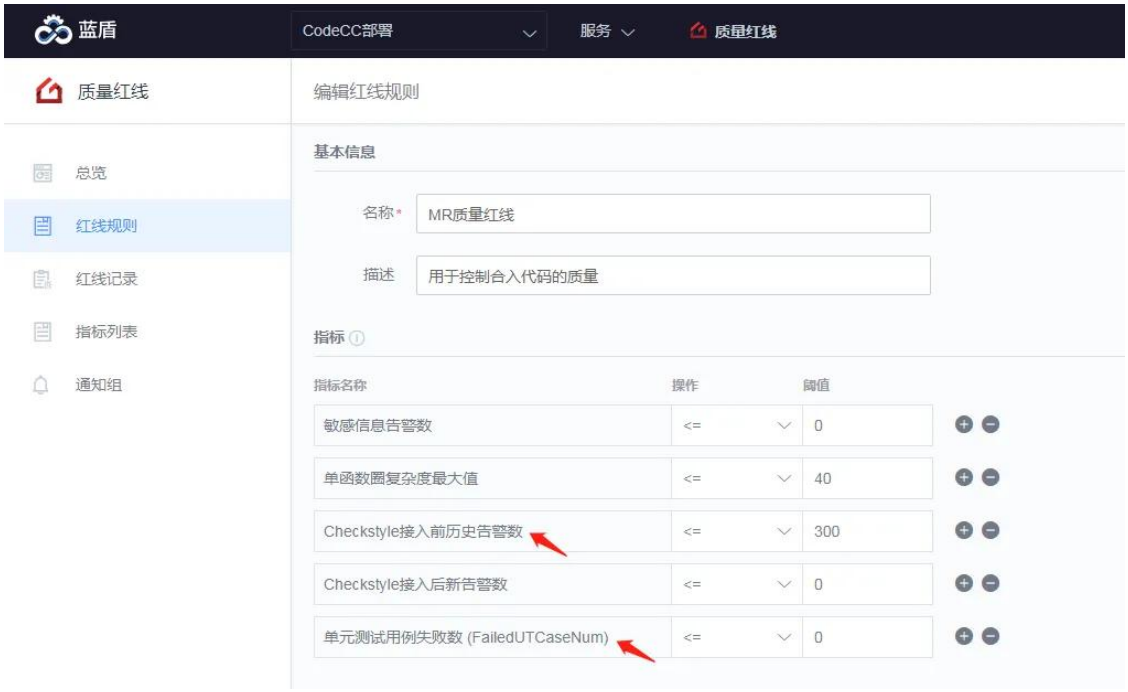
3.2 配置质量红线

接着我们需要给这条流水线配置一条质量红线。通过质量红线，我们可以将团队代码质量标准固化到流程之中。以某 Java 项目为例：

指标 设置如下图。

对于 CheckStyle，一般接入后首次扫描会产生不少告警，全部修复清零较为困难，因此这里设置的指标为“Checkstyle 接入前历史告警数<=300，Checkstyle 接入后新告警数<=0”，保证新增代码都是符合代码规范的。

单元测试用例失败数 (FailedUTCaseNum)是一个自定义的指标。用户也可以在指标列表中自定义其他指标，在 Bash 脚本任务中通过 setGateValue 函数将其上报给 DevOps 平台的后台，用于质量红线的判断。



4、Git 平台结果展示

当我们在工蜂发起 MR 检查时，此时会触发流水线执行。流水线会自动拉取发起 MR 分支（源分支）的代码，进行单元测试和代码检查。

如上图所示，由于敏感信息告警数没有符合质量红线的要求，导致流水线执行失败。此时，DevOps 平台会与 Code 平台联动，将结果同步到 Code 平台。

执行结果第一部分是流水线整体执行结果。它会在工蜂 MR 页面的顶部显示，包括流水线名称、时间、状态和详情链接，点击“详情”，可以跳转到 DevOps 平台流水线详情页面。通过这个流水线整体执行结果，评审人能够快速判断出待合入代码（源分支）是否符合了团队质量要求。

5、处理告警

为了使得代码能够符合团队的质量要求，保证代码能够被合入 master，我们需要修复和处理质量红线拦截指标的告警。

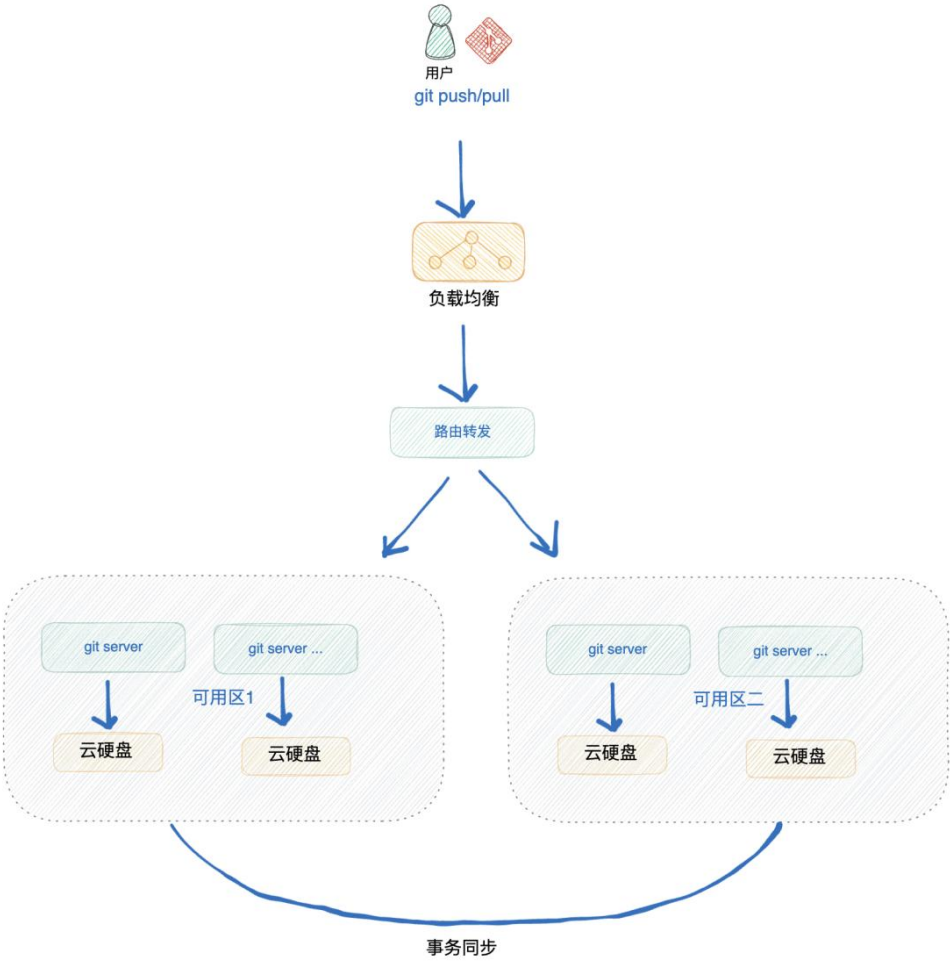
(2) 代码仓库可靠性研发保障实践

研发工具链平台众多、监控建设程度参差不齐，无法度量整个研发工具链的健康状态（运行是否正常、体验好不好，故障时快速定位根因），需要一套统一的研发基础设施可观测体系和配套的保障方案，为业务提供稳定可靠的研发工具链服务。

以下是一个缺少代码仓库可靠性保障的案例，用户频繁反馈代码仓库卡顿。

Git 代码仓库可靠性案例 部署架构设计

以下是某公司的代码仓库部署方案。



仓库容灾

如上图的部署架构图，在多个可用区部署 Git 服务器，根据路由转发将数据多写到多个可用区。

在多个可用区间保持事务同步，确保数据的一致性。当其中一个可用区出现网络故障时，另一个可用区仍然可以独立提供读写服务。

大文件加速技术方案 架构设计

Git 本身不善于处理大型二进制文件，因为这些大文件通常会导致性能问题、仓库体积的急剧膨胀。Git LFS（Large File Storage）是一个 Git 管理大文件的扩展，Git 版本仓库仅需维护大文件指针，大文件本身存储在文件服务器，通过这种方式可以在 Git 仓库中高效管理和存储大文件。

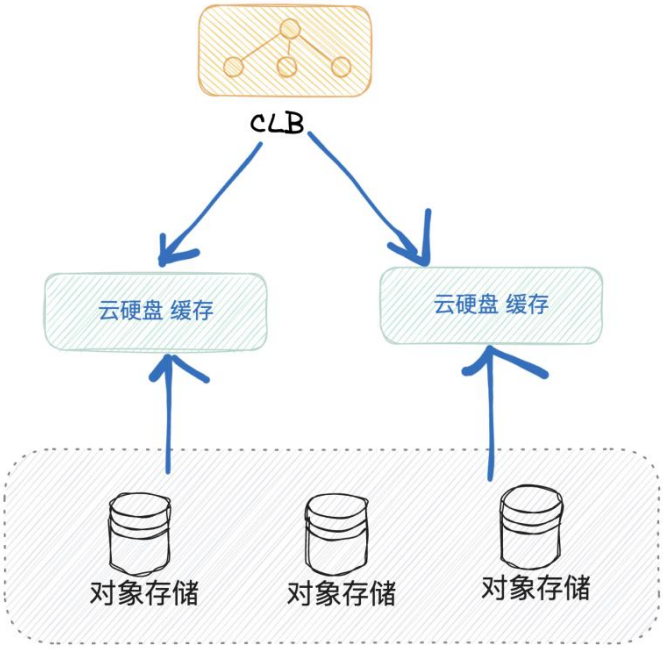
某项目 Git 代码仓库 LFS 加速案例：

改造前：

- 项目前期使用 云文件存储（Cloud File Storage，CFS) 作为 LFS 的文件服务器，随着业务发展遇到存储瓶颈和 CFS 带宽瓶颈，导致文件下载性能急剧下降。

改造后：

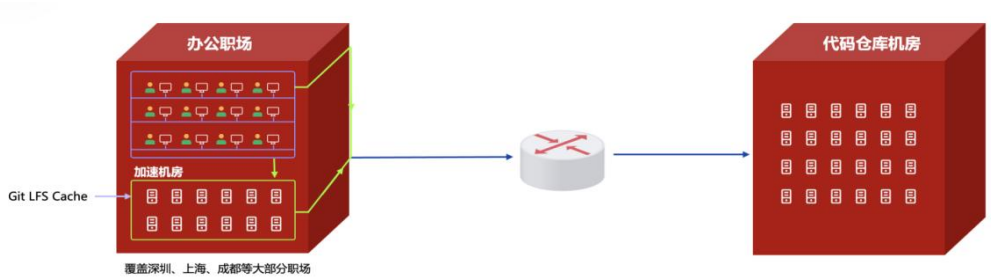
- 使用对象存储 COS 作为 LFS 文件服务器，可以实现单个存储桶容量在 PB 级别，通过创建更多的桶对存储容量进行平行扩展，同时可以复用 COS 的异地容灾能力，实现 LFS 文件服务器异地多活。
- 在 LFS 之上提供使用 高性能云硬盘（Cloud Block Storage，CBS）作为缓存节点，基于一致性 HASH 将相同文件请求到相同的缓存节点，进一步提升请求速度。



边缘加速

文件的下载速度和物理距离有强相关性，尤其 Git 仓库中的大文件，Cache 加速的原理是在办公职场的就近位置建设加速节点，从而提升 Git 大文件的下载速度。

在某公司的实际案例来看，相对于从直接从代码仓库拉取，平均速度提升一倍以上。



可观测工程

通过建设 Git 服务的可观测工程，我们可以度量和保障 Git 的服务质量，以下是某公司 Git 服务可观测工程的案例。

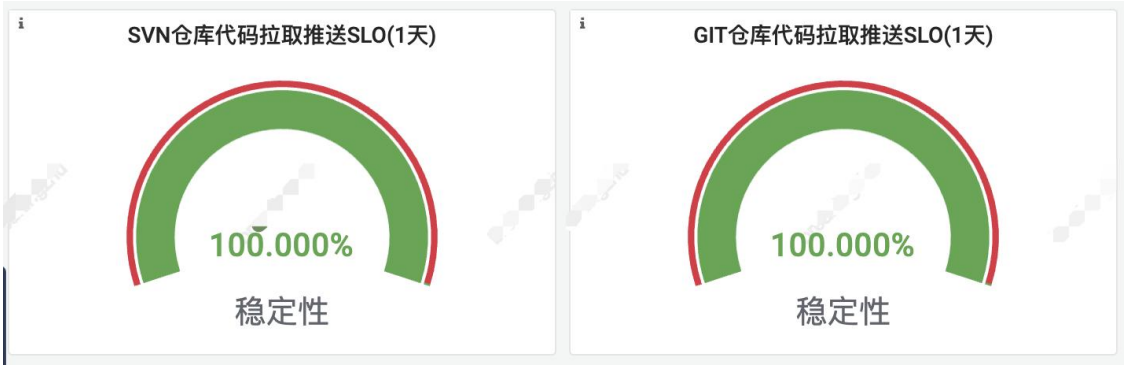
基于 6.7 应用服务 SLI/SLO 章节的实践方案，建设研发保障领域 Git 的可观测工程。

1. SLO 整体服务水平

计算公式：

服务可用性 =（1 - 不可用的总时长 / 总服务时长）* 100%

服务不可用：指标计算结果低于目标，则该类别提供的服务不可用

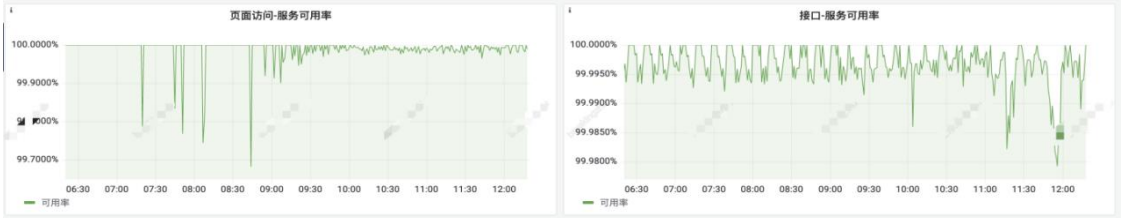


2. 对应 SLI 指标

a) 用户拉取和推送代码服务可用率，预期>99%

b) 页面访问服务可用率，预期>99%

计入不可用需要满足的条件：可用率<99%且请求量>1000



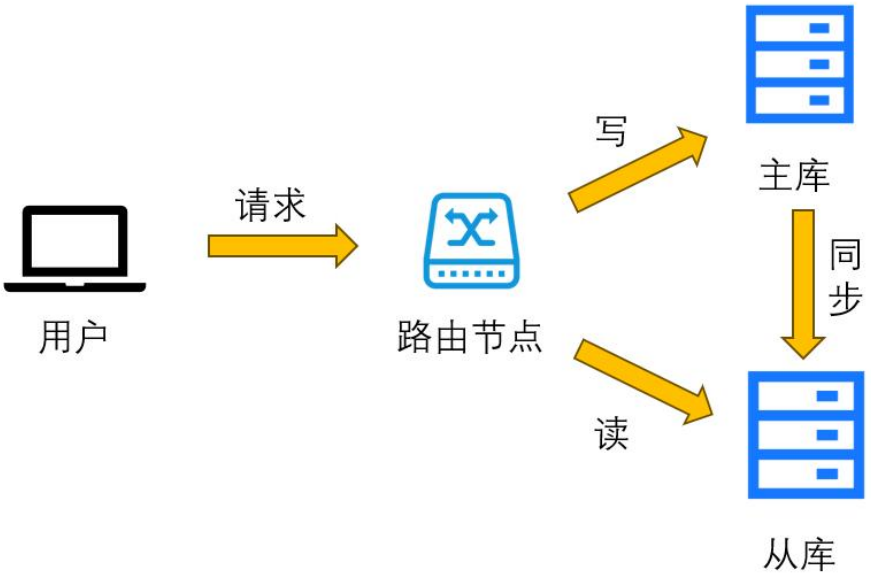
SVN 代码仓库可靠性案例 SVN 仓库容灾——主从架构

SVN 选择成熟的主从架构来进行容灾。

- 主服务器（Master）：主服务器负责存储和管理主要的版本库。所有的提交操作都首先发生在主服务器上。
- 从服务器（Slave）：从服务器是主服务器的一个副本，它复制了主服务器上的所有数据。从服务器的主要作用是备份主服务器的数据，以及在主服务器出现故障时提供访问服务。
- 同步过程：为了保持主服务器和从服务器之间的数据一致性，需要定期进行同步操作。同步过程可以通过 SVN 的 svnsync 工具来实现。同步操作会将主服务器上的最新更改复制到从服务器上。
- 故障切换：当主服务器出现故障时，可以将从服务器提升为主服务器，继续提供服务。

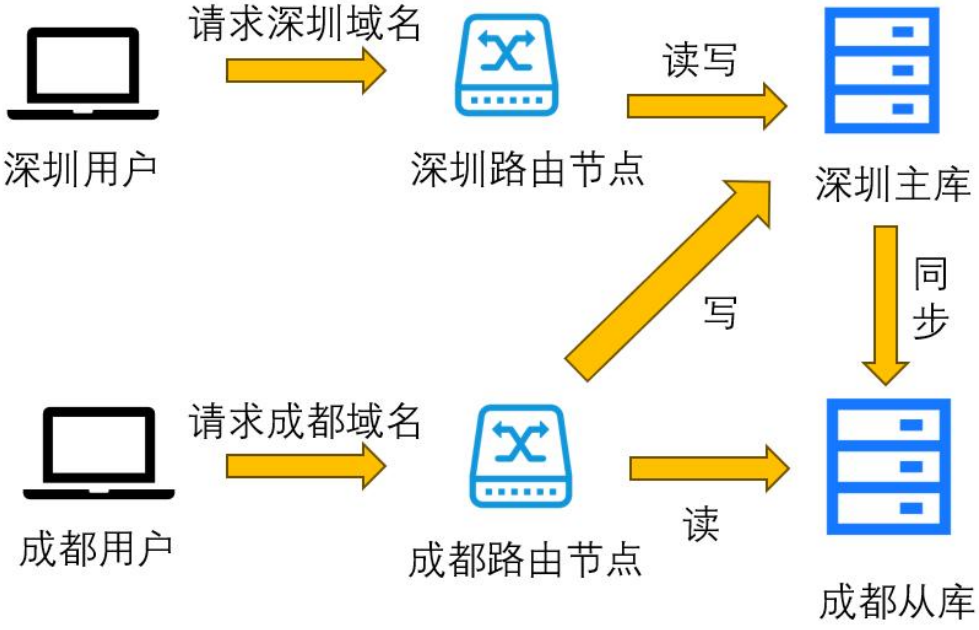
SVN 仓库性能 **1. 读写分离**

保证主从一致性后，可以在 Server 前面添加路由节点，将用户请求转发到从库，实现读写分离。



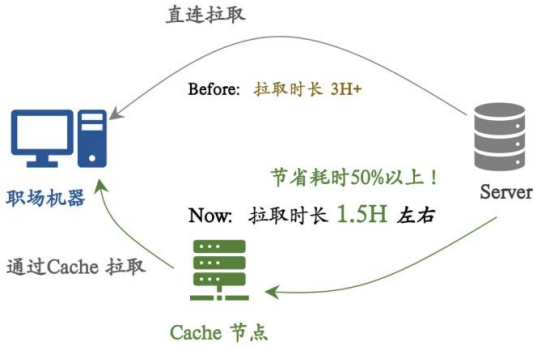
2. 就近读取

对于跨地域集群的仓库，现在支持一个集群部署多个跨地区的节点，集群内自动同步。用户通过访问当地的域名，实现就近访问和跨地区协作。



3. 边缘加速

边缘加速服务简单来说就是通过在职场办公区域楼层或者大厦，就近部署一个 Cache 节点，当有内容下载的时候，会自动地缓存到 Cache 节点。后续有其他同职场的同事再次进行拉取的时候，将会优先从 Cache 节点进行拉取，这样一来，不仅能够享受局域网的下载速度，也能分担多人下载时服务端的压力，让服务更稳定，同时下载速度也会更快。



4. 存储冷热分级

随着整体业务不断稳步前进，SVN 仓库的数量不断增加，其所需存储计算资源也随之上涨。在降本增效的大环境下，成本飞涨、资源利用率低等问题显露无遗。对此，我们针对所有仓库进行冷热分级，确保在不影响用户正常使用的前提下，尽可能地压榨服务器性能，提升资源利用率，节约运营成本。

整体方案

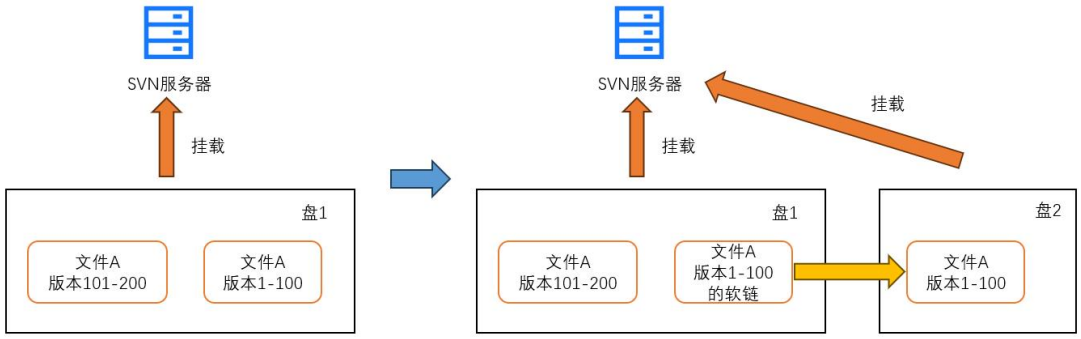
数据采集：对仓库数据使用情况进行多维度采集

热度分析：分析仓库数据热度情况，进行标记

冷数据迁移：对标记的冷数据迁移到低成本存储集群

SVN 仓库可扩展性——版本分片

一些项目经过多年发展，仓库容量已经快达到单块盘存储容量上限了，我们采取了称之为“版本分片存储”的方式，挂载多块盘+软链来解决这个问题。



如上图所示，我们通过在服务器上新挂载盘 2，将老的版本文件迁移到新挂载的盘 2 上，并在原来的盘 1 上创建软链，软链指向新挂载盘 2 的文件夹路径。

当用户访问 SVN 仓库时，还是访问盘 1 的文件，大部分时间都是访问较新的版本文件，如果访问到旧的版本文件时，还能通过软链访问到迁移到盘 2 的文件。

SVN 仓库安全

源代码对企业的重要性不言而喻，因此加强源代码保护至关重要。可以从源代码本身、内部企业人员权限和做好监控审计等三方面着手，进一步加强企业重要源代码的安全性。

一、源码分级

对源码进行分级，确保和明确重要源码的保护措施。

企业内部源码具有优先层级，明确哪些核心代码需要被保护。对于保密等级高的代码：

1、 授权范围：仅仓库的项目成员有权限访问。

2、 禁止通过任意方式突破代码仓库原有的授权范围，包含不仅限于 fork、push 到其他仓库等。

3、 禁止在无权限访问的范围保存、发布、讨论、传播。

4、 禁止下载或保存源代码到非开发环境机器，例如本地办公机、个人电脑、U 盘等存储媒介以及云平台（如各类网盘、网络笔记等）等。

二、精细化控制

精细化访问控制，对于员工的权限进行限制。

为不同角色分配访问权限应遵守最小授权原则，根据场景设置必要权限；

针对转岗或职责调整员工应及时收回相关项目代码仓库权限并清理本地保存代码；

对于非项目参与同学不应该给予项目访问权限，特殊情况应该邮件申请报备；

任何人如果需要调用其他工程的代码，在不必要的情况，不提供源代码，应采用 API 调用等方式；

三、监控和安全审计

配套建设代码仓库安全审计和监控系统

尝试越权访问时触发告警

大批量异常拉取代码时触发告警

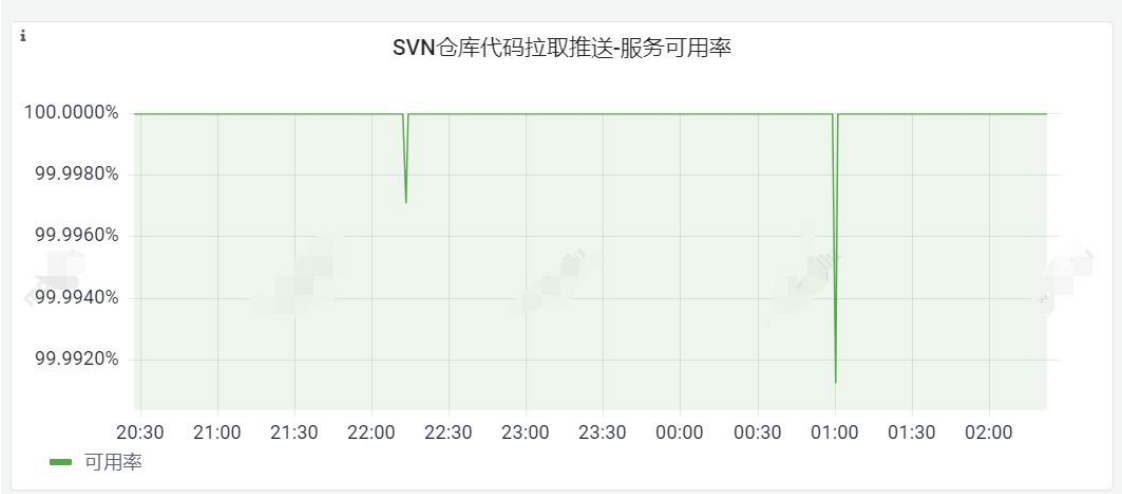
代码拉取、提交记录长期可追溯

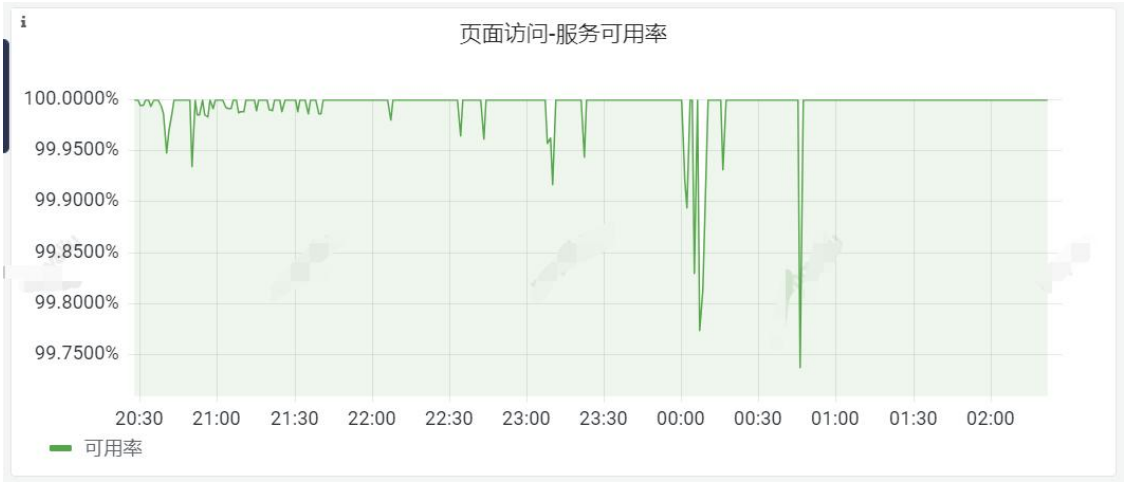
SVN 仓库可观测

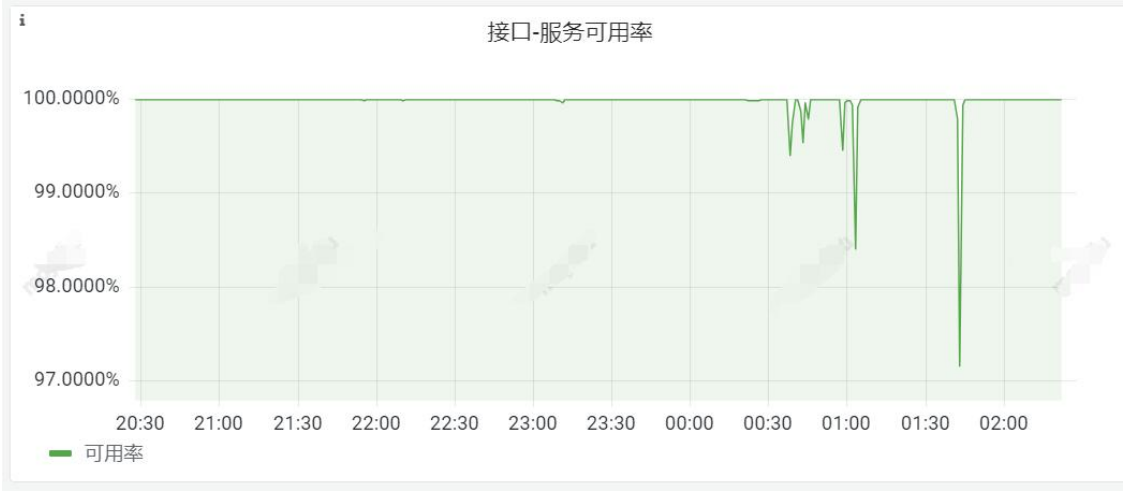
1. SVN 仓库 SLI 定义

代码仓库提供 代码拉取推送、页面访问、接口三种服务类别，按照下表定义考核目标

类别	指标	计算方式	目标
SVN 仓库代码拉取推送	服务可用率	（用户拉取和推送代码成功请求数 / 用户拉取推送代码总请求数）* 100%	>99%
页面访问	服务可用率	（用户请求成功数 / 用户访问总请求数）*100%	>99%
接口	服务可用率	（接口请求成功总数 / 请求总数）* 100%	>95%







2. 可用性 SLO

- 服务不可用：指标计算结果低于目标，则该类别提供的服务不可用
- 服务可用性 =（1 - 不可用的总时长 / 总服务时长）* 100%

P4 仓库可靠性案例

P4（Perforce）仓库介绍

Perforce，简称 P4，是一款功能强大的集中式版本控制系统，广泛应用于软件开发、游戏开发、芯片设计和数字资产管理等领域。它提供了版本控制、工作空间管理、变更处理和分支模型等功能，支持跨平台操作，并且能够处理大型文件和二进制文件。国内使用 P4 的公司有阿里巴巴、腾讯、华为、字节跳动、长安汽车等，海外使用 P4 的公司有育碧、甲骨文、思科、英伟达、三星等。

注：

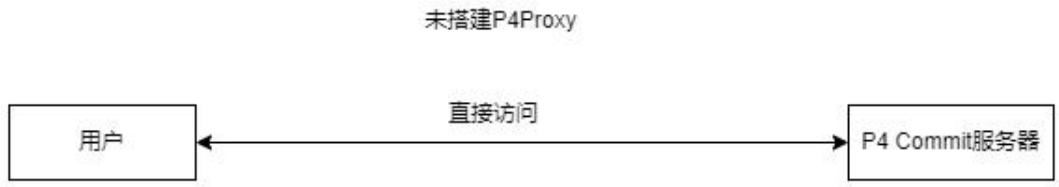
Perforce 是 Perforce Software, Inc.的商标和产品，使用 Perforce 需要向 Perforce Software, Inc.购买产品使用授权。

P4 仓库性能

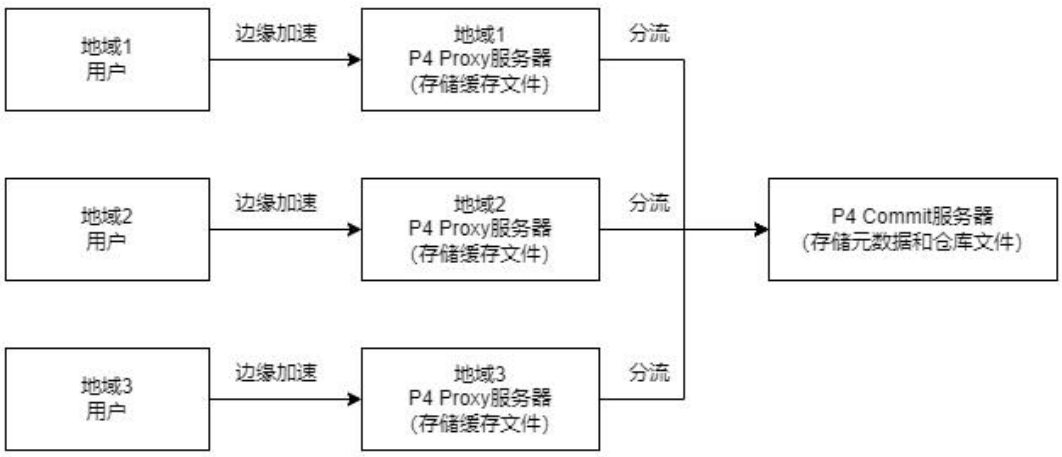
- 架构优化-Proxy

由于 P4 Commit 服务器（主服务器，存储所有仓库和提交记录）默认是单点服务的模式，当用户量很大的时候，就会遇到网卡和磁盘 IO 瓶颈，导致拉取很慢。此时可以搭建

Proxy 作为缓存服务，用户访问 Proxy，Proxy 发现本机上有缓存文件，就会直接返回给用户，而不会再去请求 commit 服务器，这样就减轻了 commit 服务器的压力。如果 Proxy 发现本地没有缓存文件，就会请求 commit 服务器，把文件拉下来存在本地作为缓存，并把文件传给用户。整个机制与 CDN 类似。



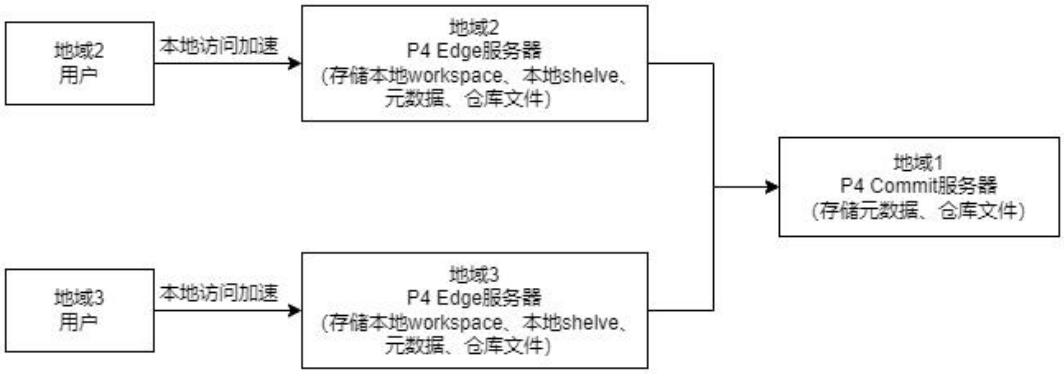
可以部署多台 Proxy，分流用户请求，提升并发承载能力，另外 Proxy 部署需要尽量靠近用户地域，起到边缘加速作用。



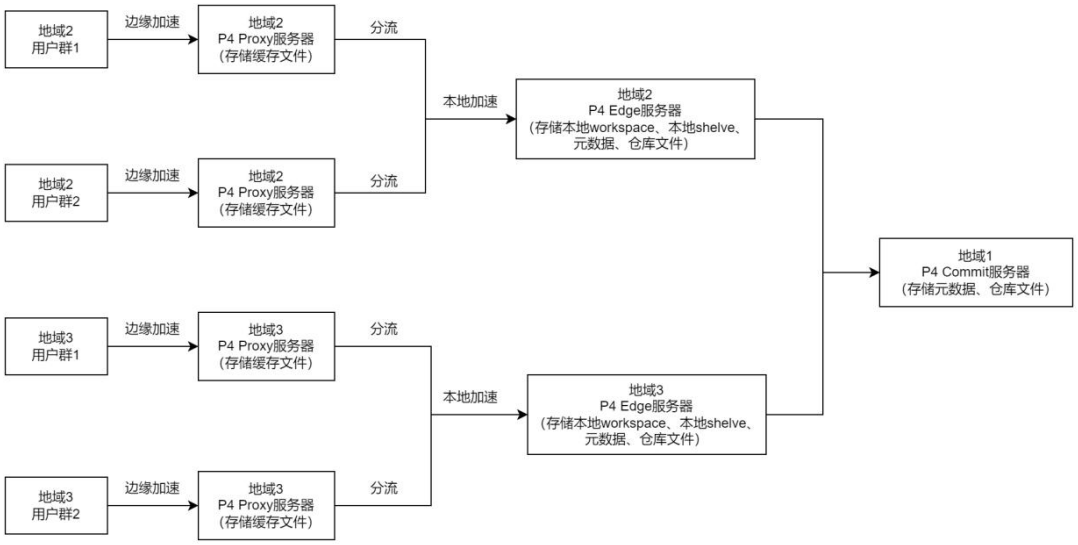
Proxy 预拉取：正常情况下，只有当用户拉取了代码，proxy 上才会有缓存，这样就导致一个新文件创建后用户首次拉取的时候是没有缓存的，会回源到 commit 给 commit 造成压力，我们可以在服务器上定时用程序来模拟用户拉取 P4 文件，生成缓存，这样真正用户访问的时候就没有回源压力了。

- **架构优化-Edge**

Proxy 在远距离（跨城、跨国）大批量小文件传输时表现不佳，因为仍需要访问 Commit 去执行大量的 metadb 数据读写操作，此时可以在用户本地部署 edge 来进行优化。Edge 把 workspace 和 shelve 数据放到了本地，这样拉取时只需要读写本地 Edge 的 metadb，大幅提升效率。



Edge 还可以搭配 proxy 使用，用户先访问 Proxy，Proxy 连接 Edge 进行缓存。



3.存储优化

p4d（commit/replica/edge）的数据文件主要分为 3 类：DB 文件（db.*）、事务日志（journal）、仓库文件（depot），三类文件有不同的要求。

- DB 文件（db.*）
- 内容：版本控制与管理用数据表
- 特点： 超低延迟高带宽的随机读写
- 设备选择： 高性能 SSD
- 事务日志（journal）
- 内容：事务流水日志，可搭配 checkpoint 用来恢复数据
- 特点：可持久地顺序写入，对顺序写入性能有要求
- 设备选择：高性能 SSD 或 HDD
- 仓库文件（depot）
- 特点：高空间占用，对延迟要求不高
- 设备选择：大容量高带宽设备，最好能支持扩容，例如大容量 SSD、HDD 或 NFS

部署时建议把 DB 文件、事务日志文件和仓库文件分 3 块盘部署，以达到最佳性能。

Proxy 上的数据以缓存数据为主，对读写性能要求高，可以使用多块本地 SSD 盘组建 Raid0 阵列来使用。

四、多地访问优化

Commit 部署在用户最多的城市，CommitStandby 需要和 Commit 同城不同机房。

Proxy、Edge 需要在用户所在的城市就近部署 **P4 仓库容灾**

数据备份方式

热备份：

集群中的 replica 节点或 standby 节点

冷备份：

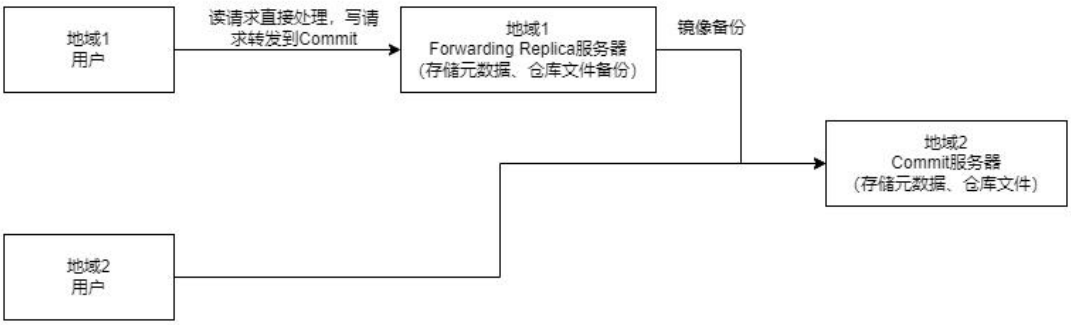
DB: 历史 checkpoint + 历史 journal

仓库：rsync 等文件拷贝机制

热备：镜像（replica）节点备份内容

- Replica 节点包含主节点的完整元数据 + 仓库数据镜像
- 可以帮助主节点分担处理读请求
- 灾备：随时可以切换为主节点

- 根据灾备机制完整性，分为 replica 和 standby
- 根据接受的请求类型，分为 forwarding 和 read-only
- 实际类型：forwarding-replica、read-only replica、forwarding-standby、(read-only) standby



热备：Standby 节点备份内容

standby 基本特性与对应的 replica 一致，具备对应 replica 的所有功能，standby 可以看作改进型的 replica。相同点：

- 热备节点
- 完整的数据同步备份
- 分担部分请求处理
- standby 提升了灾备场景的能力，不同点：
- 更完善的 journal 同步机制，避免数据丢失或集群状态不一致
- failover 机制，实现自动主备切换
- Mandatory 模式下同步确认机制降低集群性能

节点类型	读请求	写请求	故障切换	主节点类型	改进的同步机制
Forwarding Replica	本地处理	转发给主节点	手动	Commit	否
Read-Only Replica	本地处理	拒绝	手动	Commit、Edge	否
Forwarding Standby	本地处理	转发给主节点	自动（failover）	Commit	是
(Read-Only) Standby	本地处理	拒绝	自动（failover）	Commit、Edge	是

DB 数据冷备份

冷备对象：最近一个 checkpoint 和 checkpoint 以来的 journal

其他 checkpoint 和 journal 可以用来支持回滚等操作

方法：

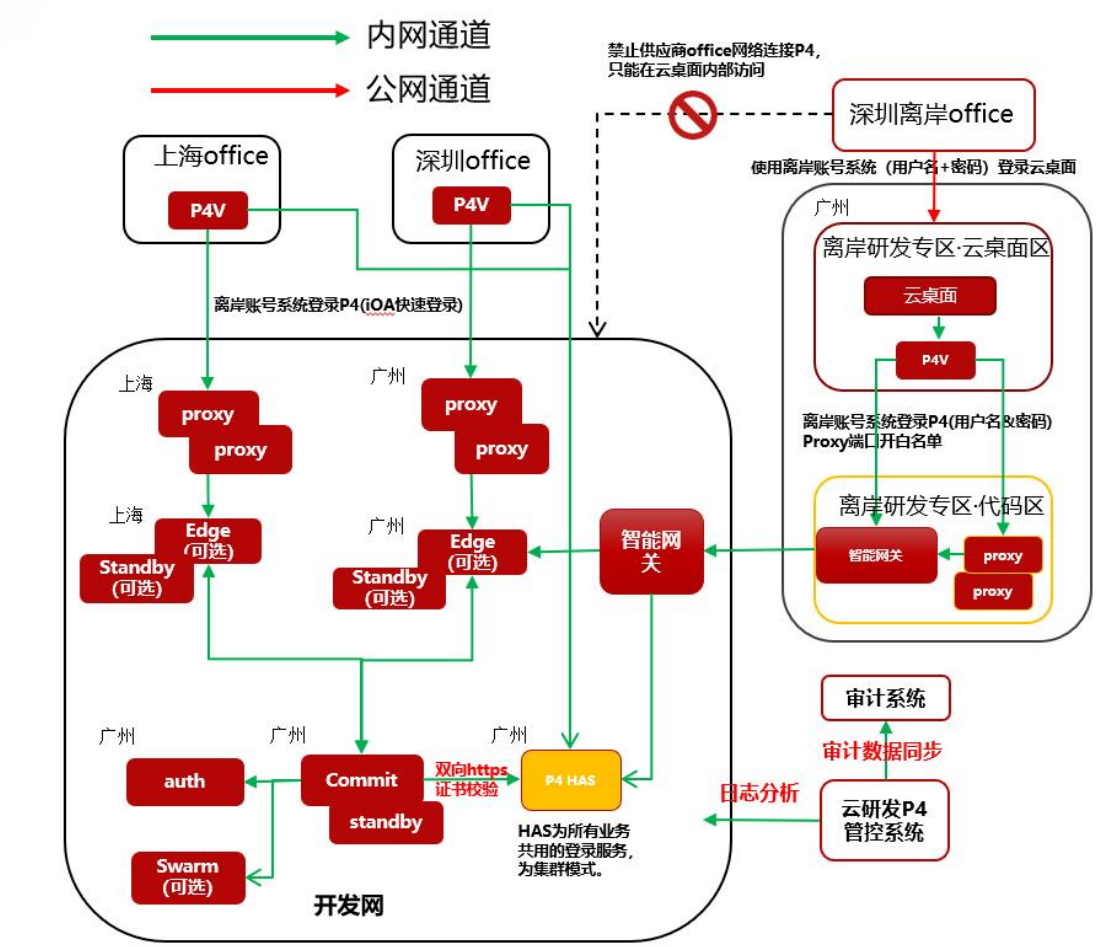
- Journal 和 DB 放在不同的存储设备上

- 定期跑 checkpoint 和 journal rotation
- 历史 checkpoint 和 journal 进行可靠存储
- 可靠存储包括：
- 使用可靠的存储设备（Raid1 磁盘阵列，云存储，备份系统等方案）
- 多副本机制 **P4 仓库安全**

我们制定了一些安全加固规范：

- 公网连接必须配置 SSL 加密
- 公网访问需要通过安全组对来源 IP 进行限制
- P4 密码安全级别必须设置成 3 以上：p4 configure set security=3
- 禁止自动创建用户：p4 configure set dm.user.noautocreate=2
- 强制新用户修改密码：p4 configure set dm.user.resetpassword=1
- 启用监控：p4 configure set monitor=25（这里至少要是 2，建议用 25 来开启锁 DB 进程监控）
- 开启审计日志：使用-A 参数开启审计日志，例如 p4d -A /data/perforce/auditlog
- P4 需要用非 root 账号部署，禁止监听高危端口
- 权限表里默认的所有用户写权限要去掉（write user* *//...）**P4 仓库总体架构**

为了满足内部员工和外部供应商共同使用 P4 的需求，基于性能、容灾、安全三方面的优化方案，我们设计了如下 P4 集群架构。内部员工直接访问开发网的 P4 集群，供应商通过云桌面在离岸研发专区（专门为供应商设立的网络专区）访问 proxy，请求通过智能网关转发到开发网 P4 集群，以此来达到内部员工和外部供应商在安全、可靠、高效的情况下使用 P4 协同办公的能力。



SRE 可靠性提升措施：

1.使用统一账号系统，提供统一鉴权；

2.元数据（含权限表、用户表、trigger 等）自动备份，一键恢复；

3.集群化部署，主备容灾，使用云盘避免机器故障导致数据丢失；

4.代码拉取、提交、删除、变更记录接入公司审计系统；

5.人员离职、组织架构变更，邮件提醒权限删除；

6.禁止供应商 office 网络连接 P4，只能在云桌面内部访问；

7.p4auth 服务器单独架设，commit 异常，切换 replica；

P4 版本要求：

P4 HAS 要求 p4 版本>2019.2，若 P4 版本无法升级到 2019.2 以上可以不用 HAS 而是用公司内部 LDAP 鉴权。

P4 仓库扩展性 1. trigger

P4 的 trigger 机制可以在用户执行特定命令时触发对应的程序，从而扩展仓库功能。以下列举一个 P4 登录鉴权的 trigger 案例。

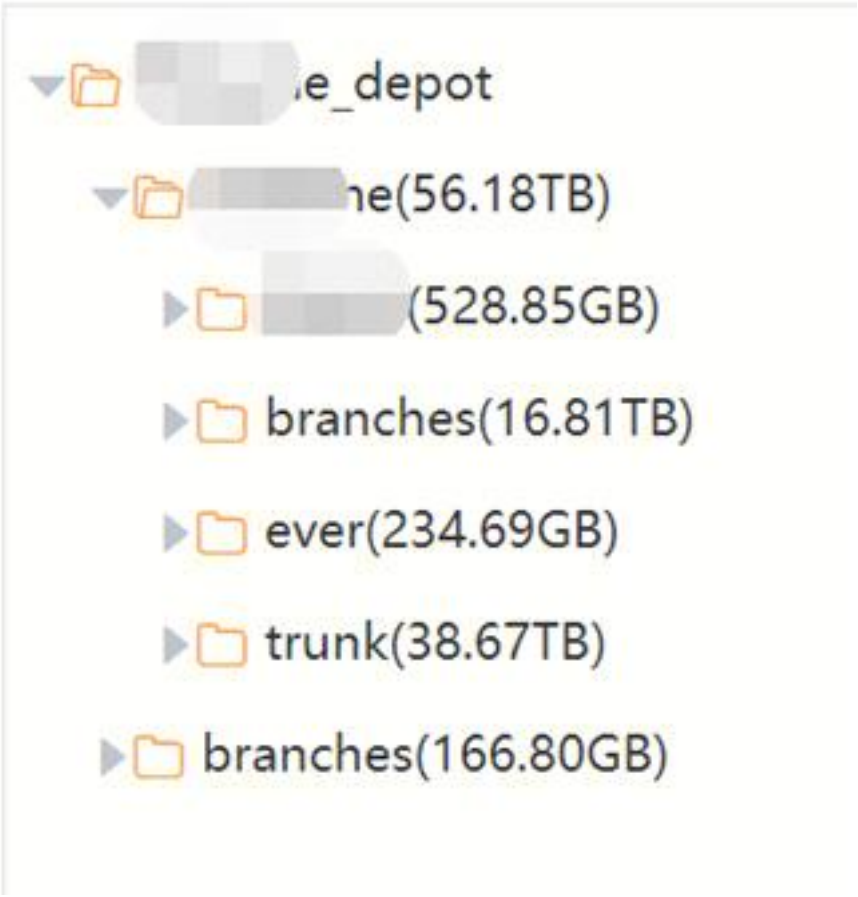
Triggers:

ldapAuth auth-check auth "/data/script/p4auth_ad-no_null ldap.domainname.com 389 %user%"

这个 trigger 的类型为 auth_check，会在用户登录时触发，调用/data/script/p4auth_ad-no_null 这个程序，把用户登录用户名和密码传给 ldap.domainname.com 这个 ldap 服务器的 389 端口，校验通过用户即可成功登录 P4 服务器。

2. 容量管理

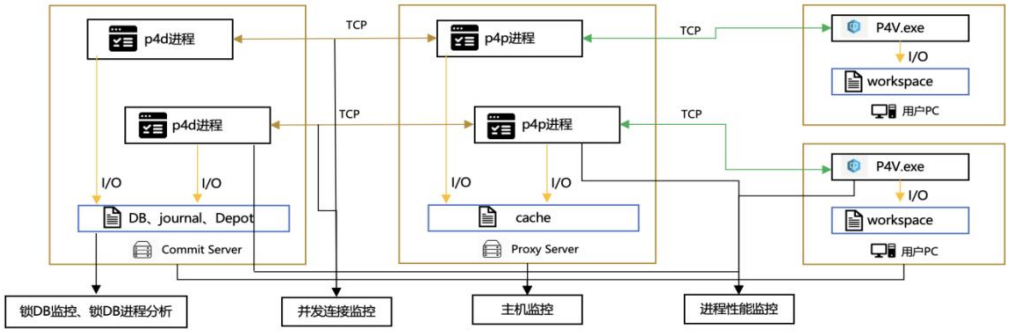
由于 P4 上经常会存储很多大的二进制文件，硬盘空间容易满。我们建设了容量管理系统，利用 P4 SDK 分析出每个目录、每个文件的所有历史版本占用的存储空间总大小。

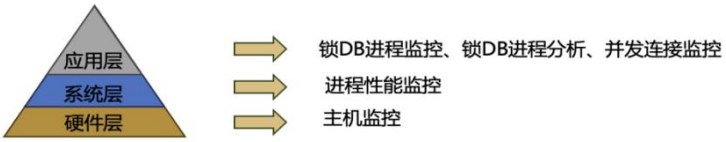


容量管理系统可以基于 P4 的 filetype +S 属性，配置每个文件在服务器上保存的历史版本个数，自动清理超出设置范围的历史版本文件。

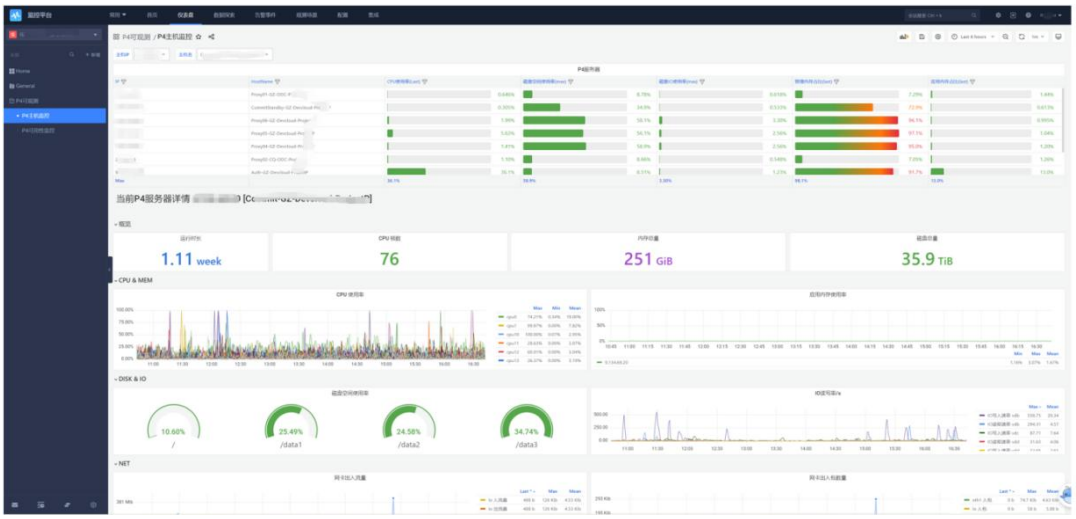
P4 仓库可观测性 1. 监控建设

基于用户访问 P4 的进程网络、存储调用链，我们建设了主机监控、进程性能监控、并发连接监控、锁 DB 监控、锁 DB 进程分析等监控能力。





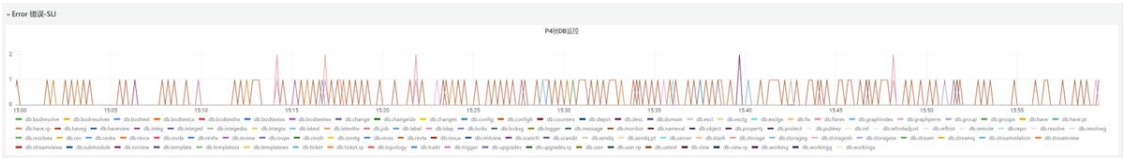
主机监控：监控 P4 集群各主机 CPU、内存、磁盘、网络等使用情况



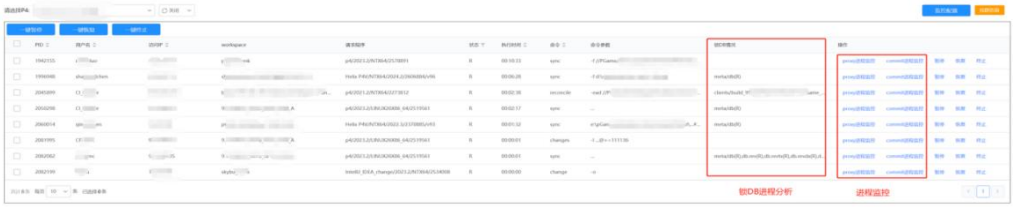
进程监控：监控 p4 连接进程（p4d、p4p）的 CPU、内存、IO 等使用情况

P4 并发连接数监控：监控 P4 的并发连接数情况

锁 DB 监控：监控各个 P4 DB 表的锁 DB 情况（不锁为 0，读锁为 1、写锁为 2.



锁 DB 进程分析：分析出锁 DB 进程以及该进程具体锁了哪些 P4 DB 表，并且能一键批量暂停、恢复、终止异常 P4 连接。



2. 可用性 SLO

首先设定 SLI

指标	目标
P4 并发连接数	<前四周的周峰均值*150%
P4 锁 DB 持续时间	<5 分钟

前四周的周峰均值定义：（前四周每周的并发连接数峰值的和）÷ 4

配置 SLI 监控项：

- 服务不可用：指标的值未达成目标，则该类别提供的服务不可用
- 服务可用性 =（1 - 不可用的总时长 / 总服务时长）*100%

根据监控告警时长计算出服务不可用总时长，由此计算出服务可用性，仪表盘如下

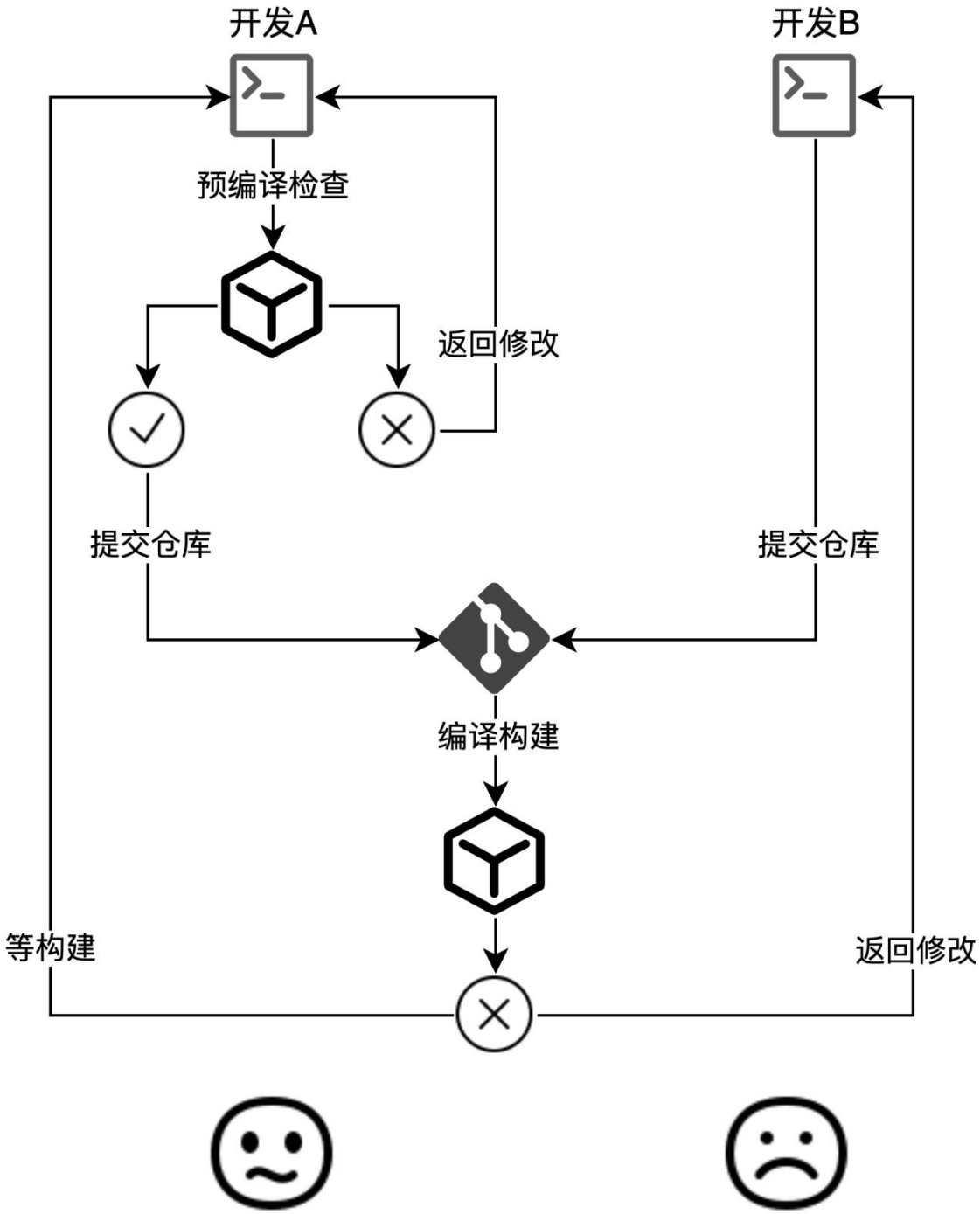
(3) 使用 Prebuild 提升构建效率的实践

这是一个典型的通过预编译解决代码合并容易出错，提升构建成功率和效率的案例，适用于超过 10 人以上且高频做代码提交合并的开发团队。

该案例描述了一个超过 50 人的大型开发团队，对于如何提升编译速度做了详细的方案拆解，在本地构建资源有限的情况下，通过利用云计算资源方案，还能大幅提升构建效率的问题，体现出互联网特点。

背景及设计原则

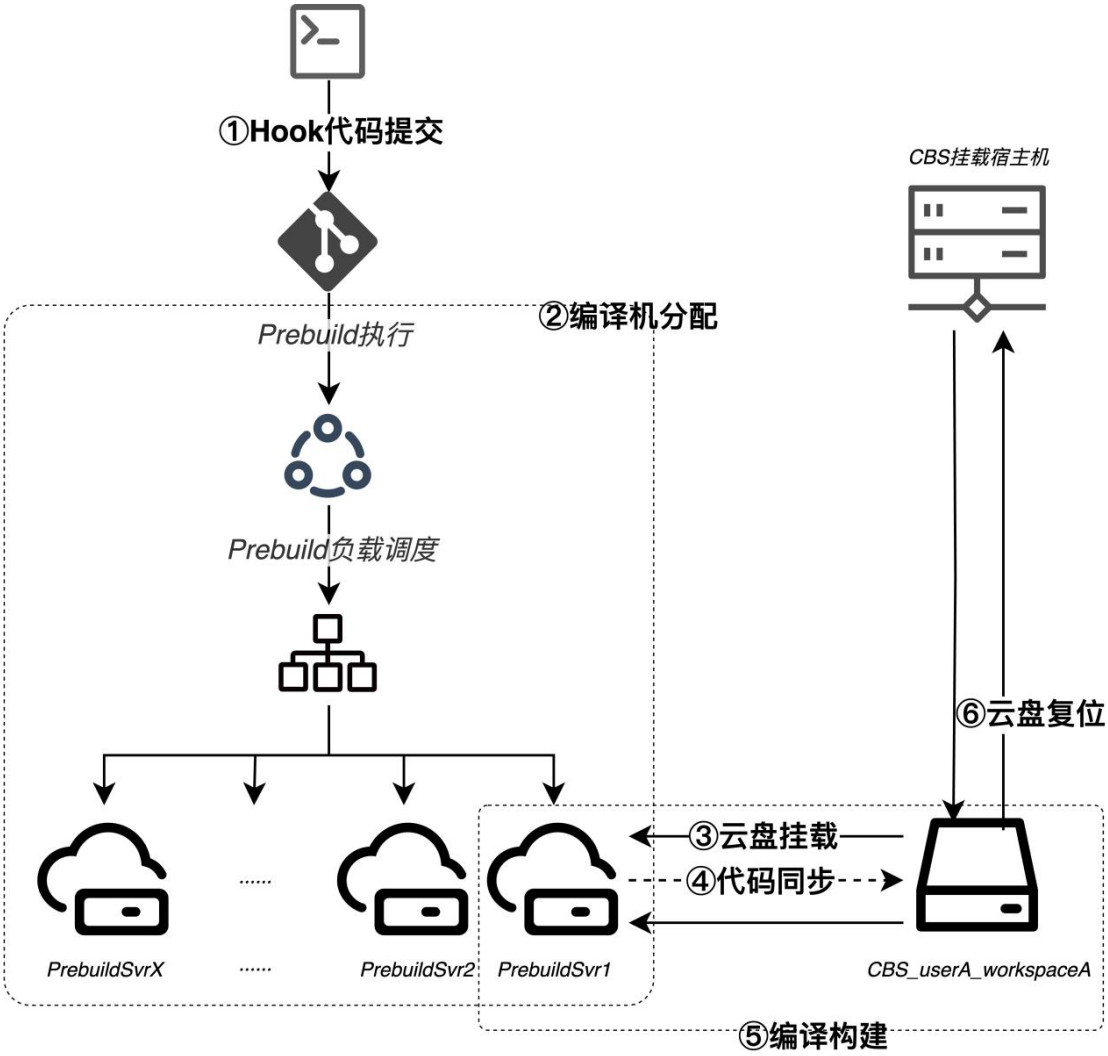
某项目的开发团队人数规模超过 50+，多个开发在合版时，出现合并失败的情况比较多见，特别是在人工执行检查时容易遗漏，并且引起本地编译卡顿。结果影响整体的版本出包和迭代效率，造成整个项目经常熬夜等构建完成，造成很大非必要的加班，拖累整个团队出包交付效率。



图：原始构建出包方案影响不同开发间协作构建出包效率

解决方案：

通过使用 Prebuild 方案，把本机需要编译的资源同步到云上，利用云的强大计算资源能力，快速构建后，将编译好的文件同步到本地盘，提高编译速度。



图：Prebuild 实现流程图

设计思路 and 关键流程

1. Hook 代码提交：

代码提交：开发人员提交代码到代码仓库时，触发 Hook 机制。

2. 编译机分配：

(1) 负载调度：预编译服务器（prebuildSvr）根据负载情况分配编译任务。

(2) 预编译服务器：包括 prebuildSvr1、prebuildSvr2、prebuildSvr3 等。

3. 云盘挂载：

(1) 存储初始化：根据初始化生成的存储映射表，将 CBS 个人工作空间目录 CBS_userA_workspaceA 临时移动到 prebuildSvr1 上。如果无映射，则临时分配并将映射存入数据库。

4. 代码同步：

(1) 同步代码：userA 用户本地的 workspaceA 代码差量同步到 cbs 个人目录 CBS_userA_workspaceA。

5. 编译构建：

(1) 蓝盾流水线：在 preSvr1 启动蓝盾流水线进行编译构建（preSvr1 也可以接入 tbs 算力）。

6. 通过办公机器人通知&云盘复位：

(1) 通知用户：编译完成后，通过办公机器人将编译结果通知用户。

(2) 云盘复位：然后将 CBS 目录 CBS_userA_workspaceA 从 prebuildSvrA 摘除并回 cbs 的宿主机 A 进行挂载。

主要亮点

1.存储计算分离：成本节约，无需一人一机；数百 G workspace 秒级同步，避免原模式分配新编译机的数百 G 文件同步

2.Hook 机制避免泄露：支持 Git/P4/SVN 仓库；支持按文件类型触发（例如 lua 无需触发）

3.不影响开发体验：prebuildSvrA 不阻塞提交；不消耗本机算力；构建异常会通过办公机器人通知失败详情实践效果

通过实施 preBuild 方案，可以显著提升代码提交和构建的可靠性，减少构建失败的情况，提高整体开发和构建效率，并通过自动化机制优化开发流程。主要收益有：

1.保障编译正确性：保障所有开发人员提交代码仓库的代码都经过了本机编译正确性检查

2.构建成功率提升：代码合流后，构建成功率提升 30%以上

(4) 出包加速构建可靠性研发保障实践

中大型游戏的研发周期一般来说都是比较漫长的，在和项目组沟通调研的过程中，发现在构建这一环节中，游戏包体的构建出包漫长导致很多的研发/测试流程会有效率上的阻塞情况。“构建出包慢”这种情况，几乎每个项目都会或多或少遇到这样的问题，尤其是在使用比较广泛的 UE 引擎构建出包过程中。针对以上的问题，腾讯推出了 InGame 出包加速服务，提供 UE 引擎下 Cook 阶段加速服务

在同项目组深度调研沟通过程中，我们深挖了基于 UE 引擎开发的游戏项目，从构建出包再到真机安装验证整体流程中的各个环节，即 Build - Cook - Stage - Package&Deploy 与 Install/Test (注：Install/Test 不属于 UE 引擎出包环节)。

Build：

代码编译（耗时占比 20%）**Cook：**

资源烘焙（耗时占比 70% 注：中型项目 5+Hr/ 特大型项目 20+Hr）**Stage：**

拷贝缓存（耗时占比 5%）**Package & Deploy：**

打包 & 发布（耗时占比 5%）

在 UE（Unreal Engine - 虚幻引擎）项目中，“Cook”（资源烘焙）是一个至关重要的环节。它涉及将项目中的各类资源（如纹理、模型、音频等）经过精心处理和优化，转换成目标平台（如 PC、游戏机、移动设备等）能够顺利识别和使用的格式。这个过程可以比作烹饪食物，将原材料经过一系列加工处理，最终变成可以直接食用的美食。通过“Cook”，游戏资源得以优化，减少了内存占用，同时确保了游戏能够在不同的平台上顺畅运行，从而扩大了游戏的受众范围。

然而 Cook 也是游戏构建出包全流程中最耗时的环节，且业内目前还没有比较好的解决方案，UE 官方直到 5.3 版本才有一个单机多进程的 Cook 方案；同时，其他阶段特别是代码编译 Build 阶段业界已有比较好的解决方案（如，商业化方案 Incredibuild，开源方案 FastBuild 等），Stage/Package 阶段总体耗时少，本身也只是简单的文件归档迁移等操作，优化的空间和价值并不大。因此，Cook 阶段的加速成为了游戏研发出包效率提升过程中的关键突破点。

Cook 阶段出包加速方案设计思路

为了解决 Cook 阶段出包加速的诉求，我们基于 Cook 阶段耗时进行了具体分析；**Init：**

初始化阶段，主要是设置 Cook 过程的基本参数和配置，准备开始处理资源 **Load/Save：**

加载阶段，Cook 会加载项目中的资源文件，如纹理、模型、音频等。

保存阶段，处理后的资源会被保存为可被目标平台使用的格式。

Mesh/Anim：

Mesh 阶段，Cook 会对模型资源进行处理，包括顶点数据的优化、网格的简化等。

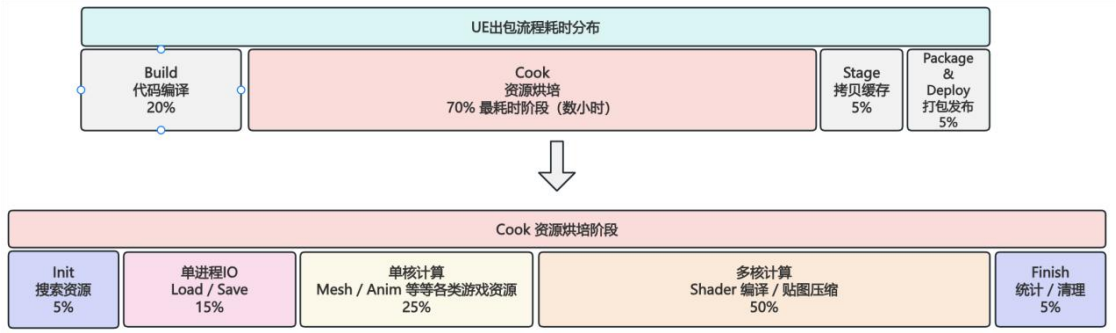
Anim 阶段，对动画资源进行处理，包括动画数据的压缩、优化等。

Shader：

Shader 阶段，Cook 会对着色器资源进行处理，包括着色器的编译、优化等。

Finish：

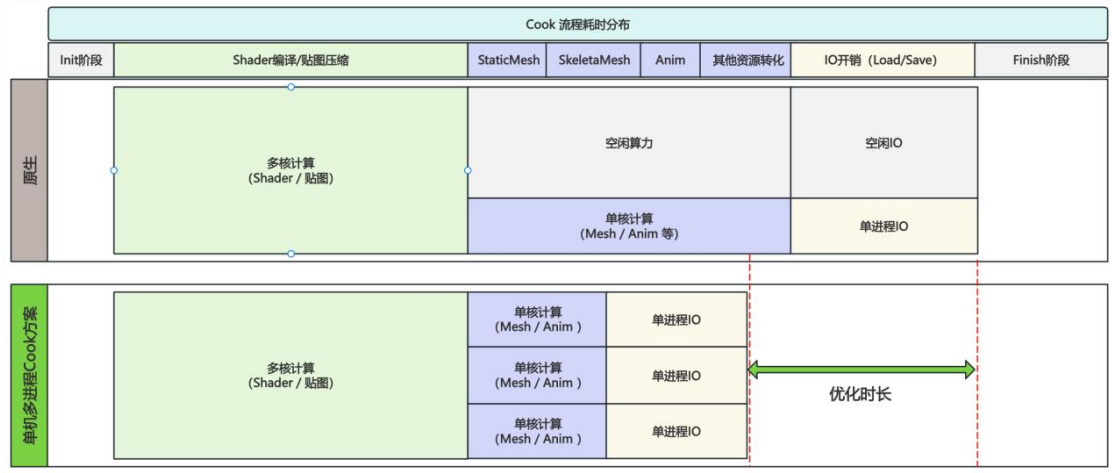
完成阶段，Cook 会检查所有资源是否已经成功处理，并生成最终的构建产物。



从上图可以看到，除了启动的初始化和结束的统计清理等工作，Cook 阶段最核心的工作是在单个 UE 进程下的 IO 和资源计算，包括单核计算和多核计算在内的一系列的资源计算。针对不同的场景，我们进行了多个优化方案的落地。

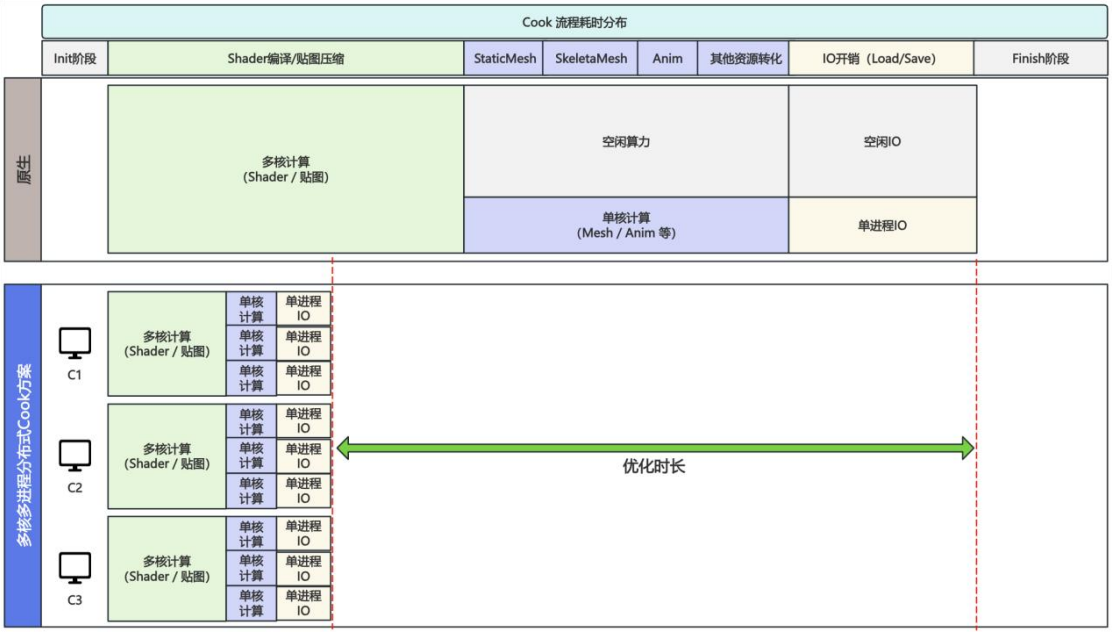
优化点 1：单机多进程 Cook 方案

针对单进程 IO/单核计算场景，通过并行处理充分利用了单机冗余算力资源，同时实现了 IO 瓶颈的突破；充分利用单机硬件资源进行计算提速；



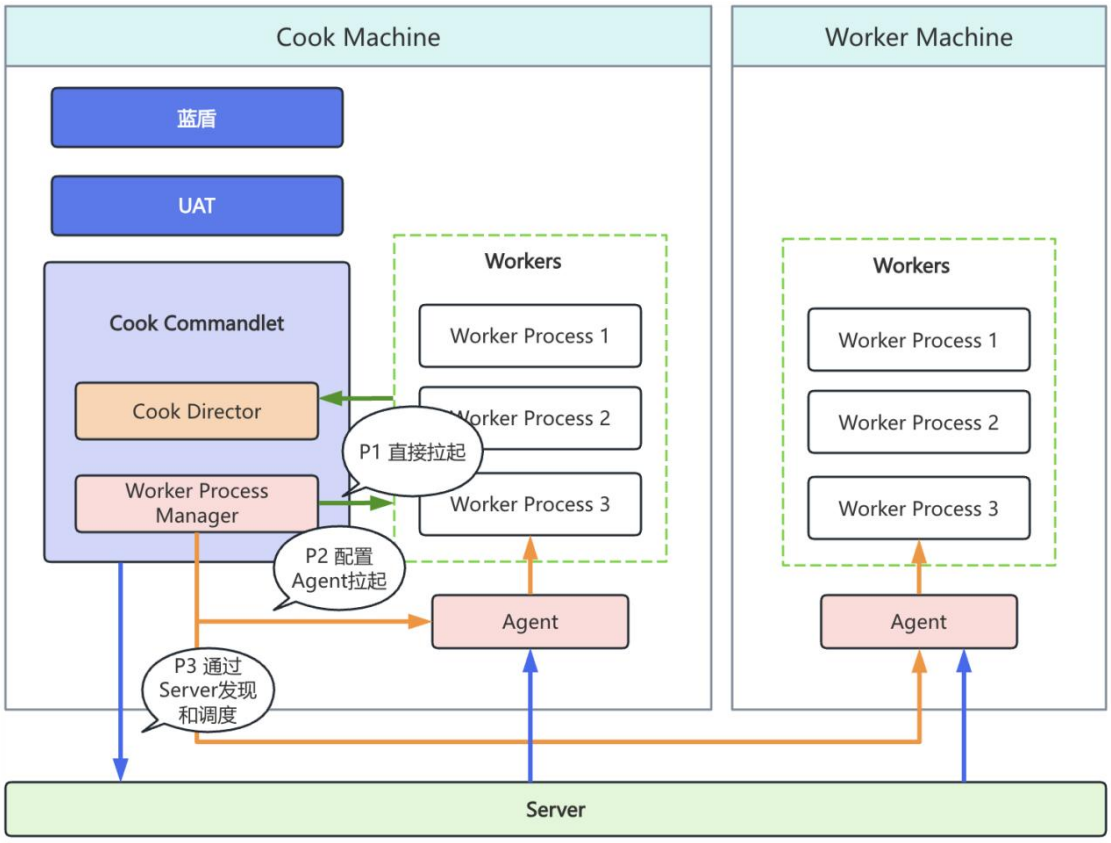
优化点 2：多机多进程分布式 Cook 方案

针对多核计算场景，通过物理机算力的分摊可以多倍扩展算力效能；实现了算力的极大拓展，提升 cook 阶段加速上限。



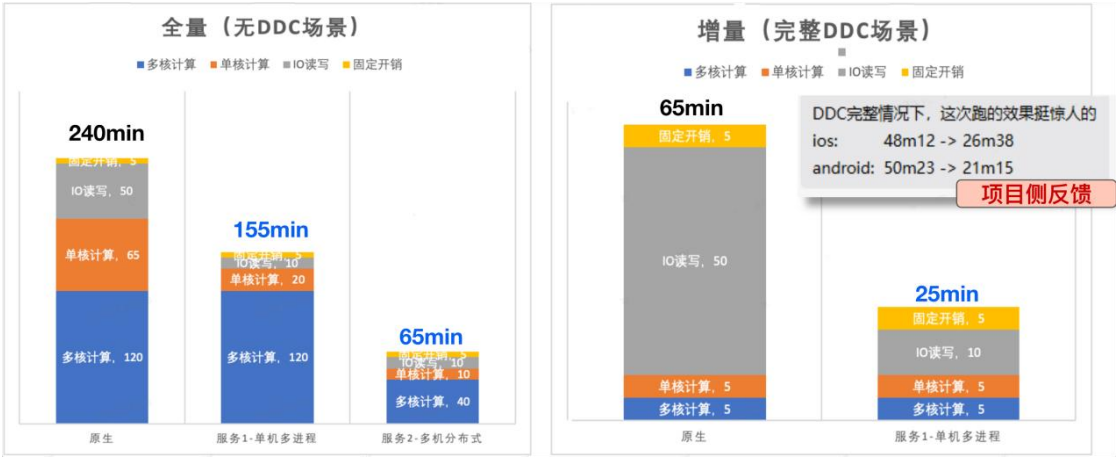
优化点 3：Agent 智能调度分布式 Cook 方案

在整个 Cook 阶段，通过对 Agent 的灵活调度以及一系列算法的优化，可以对构建机的空闲算力进行合理调度，将 Cook 阶段耗时进一步压缩。其核心是通过对于 Agent 的灵活调度，对于多层级的进程进行管理，同时实现物理机负载均衡；



实践效果

当以上优化点都落实到项目侧（大中型项目，Cooksize 100GB+， Packages 15w+），在全量与增量加速场景下平均提效加速 3-4 倍，使得总体出包的速度大大地降低，为腾讯游戏研发效能提升做出了极大贡献。



(5) 制品晋级研发可靠性保障实践

为了应对行业的快速变化，企业的版本更新频率不断加快，制品的版本质量对生产环境的可靠性至关重要。通过建立一套制品晋级流程，可以有效地把控外发版本的质量。

一般情况下，制品的等级通常分为 Alpha、Beta 和 Release 三个级别，每个版本通过相应的测试后，才能晋级到下一个级别。

以“主干开发、分支发布”的研发模式为例，外发版本的特性和 Bug 修复在主干开发阶段经历 Alpha（开发自测通过）和 Beta（测试人员测试通过）两个阶段。当代码合入到外发分支时，将进行功能测试和制品检查，以确保制品达到外发版本的 Release 等级要求。

制品晋级主要包含构建检查、制品一致性校验、功能测试和关键指标巡检等 4 个考察维度。

1. 构建检查

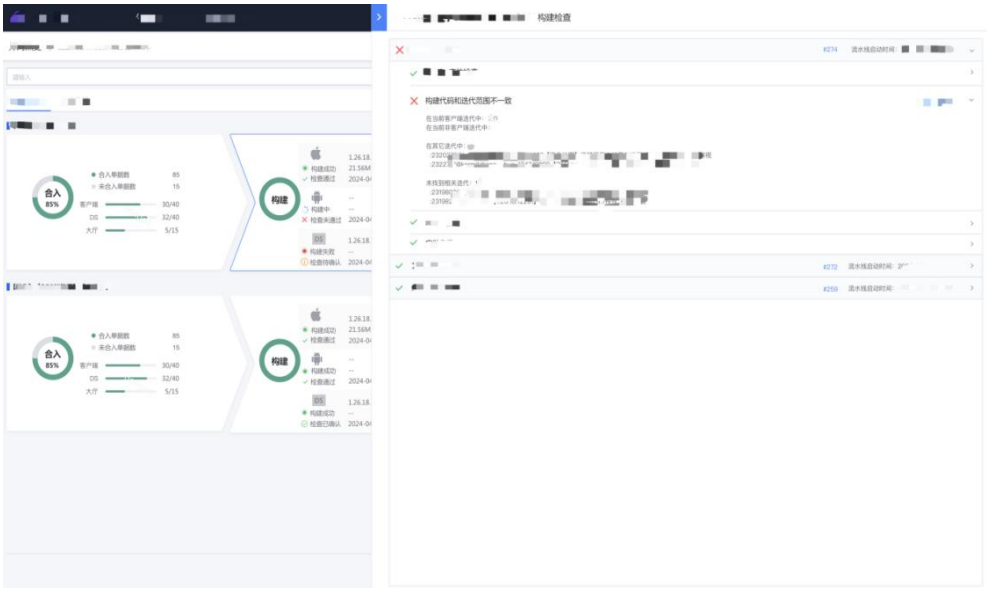
在构建过程中需检查构建原材料（即代码）是否符合预期。根据检查项类型，可以分为“致命检查项”和“二次确认检查项”，构建通过的标准是所有检查项全部通过或经过问题澄清。

(1) 致命检查项

当致命检查项不通过，必须终止构建流程，并排查问题直到解决。

以下是某业务致命检查项不通过案例：

在构建过程中发现构建原材料（即代码分支）和研发管理工具中规划的迭代范围不匹配，构建原材料中存在未关联迭代需求或 Bug 的代码。



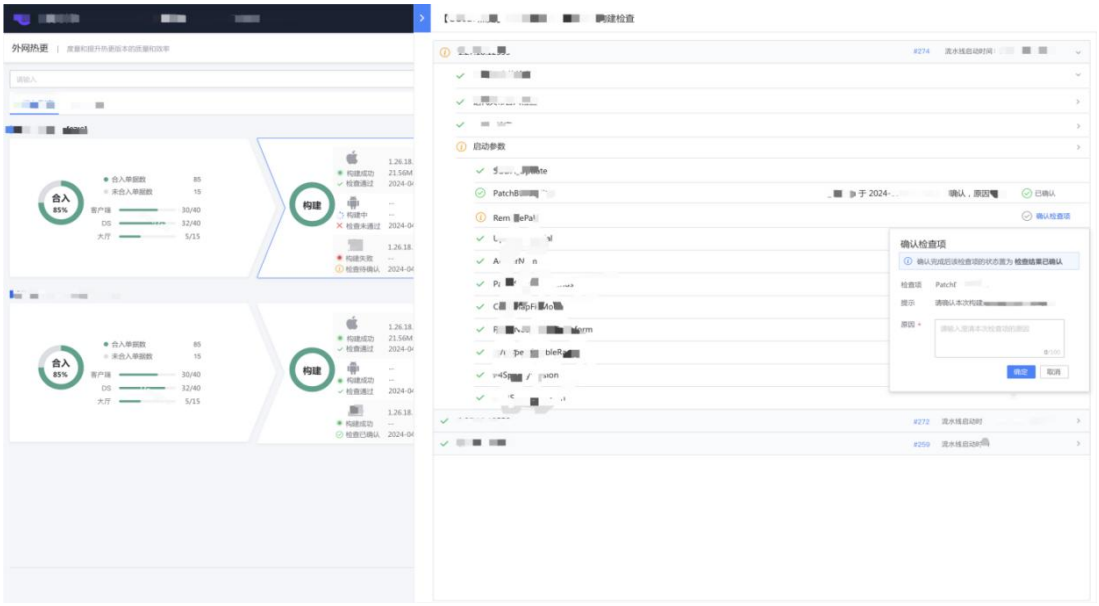
(2) 二次确认检查项

二次确认检查项包含非致命检查项和非明确的构建检查逻辑检查。

对于非致命检查项，构建检查人员根据实际情况，可选择继续维持“不通过”状态，或置为“已确认”。对于非明确的检查项，可以选择将其置为“已确认”或重新构建。

以下是某业务的实践案例：

在构建过程中存在构建参数与默认值不一致的场景，构建检查人员根据实际情况澄清，或当不符合预期时重新构建。



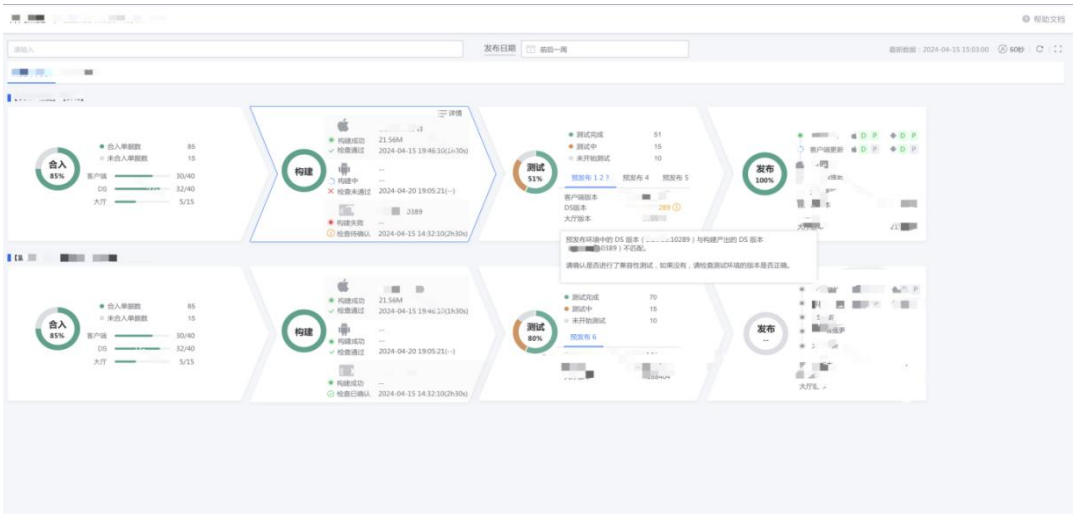
2. 制品一致性校验

当版本更新频繁时，可能会出现版本未正确部署到测试环境，而测试人员反馈测试通过的情况。这种情况下，测试人员测试的版本和构建制品不一致，导致测试未通过的版本被发布到生产环境，引发用户投诉。

因此，需要校验构建制品和测试环境中运行版本的一致性，以确保测试人员测试的版本与构建制品一致。

以下某 Top 业务的实践案例：

当测试环境的版本与构建制品版本不一致时，系统给出提示，指引相关人员检查并跟进。



3. 功能测试

功能测试主要包括 API 自动化测试、UI 测试等，相关内容在 3.3.2 章节已做详细介绍，此处不再赘述。

4. 关键指标巡检

构建的制品正确部署到测试环境后，需要经过功能测试。然而，功能性测试在特定时间段内完成，测试通过不意味着制品达到了外发标准。因此，需建立一套外发制品关键指标巡检流程，涵盖版本质量、服务器性能、关键路径用户体验等关键指标，以确保制品达到外发的标准。

1) 版本质量

检查客户端和服务端的崩溃数据，如程序崩溃次数、程序 Coredump 次数等。

2) 服务器性能

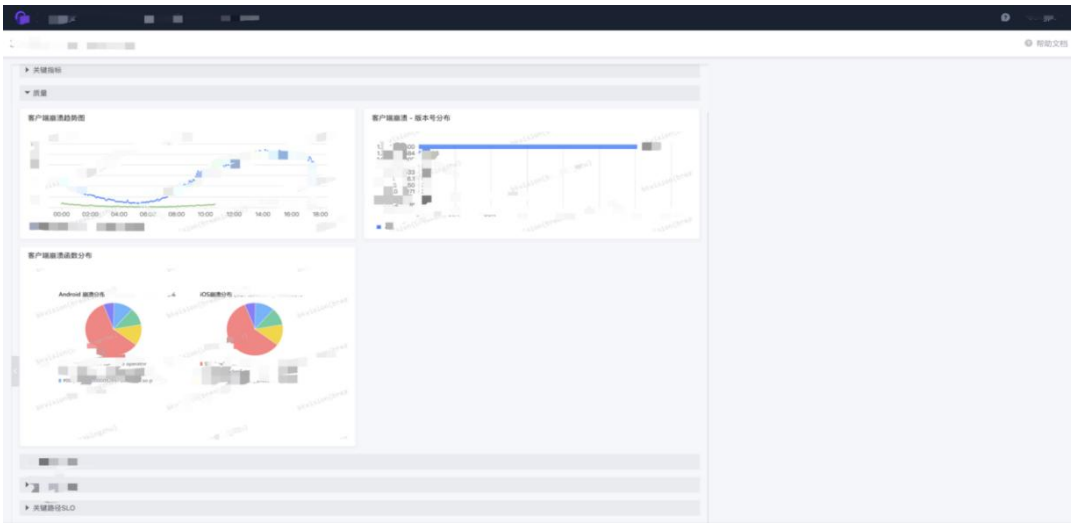
对比当前版本相对于上一个版本在单机性能维度的表现，评估单机承载能力是否下降。

3) 业务关键路径场景指标

观察业务关键路径指标是否符合预期，如注册、更新、登录、匹配、对局和支付流程的关键指标。

以下某业务的实践案例：

计划外发的制品通过了功能测试，但在版本质量维度发现客户端偶现崩溃、服务端 Coredump 情况，找到问题原因后，重新构建制品并走制品晋级流程。



当制品在分支发布的环节通过了构建检查、制品一致性校验、功能测试、关键指标巡检后，制品等级从 Beta 晋级为符合发布标准的 Release 级别，最终 SRE 将制品可靠地发布到生产环境。

(6) 制品安全扫描和制品全球分发可靠性保障

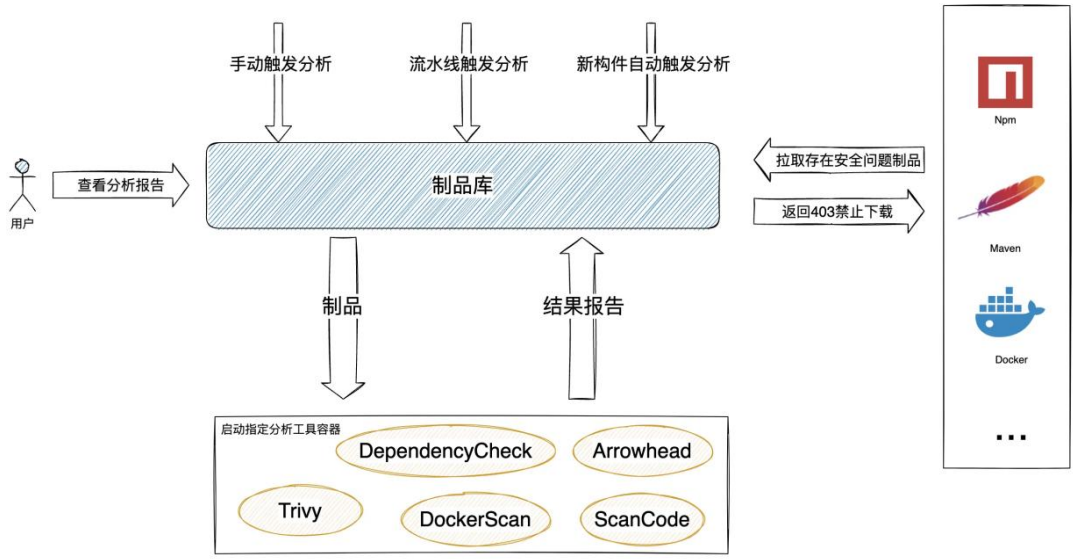
制品如何安全、有效、快捷部署，是指通过技术手段，打通构建流水线，将构建完成的制品主动推送到用户多种类型终端的过程，用自动化的方式对制品进行安全扫描、分发、安装、更新，让用户以较低的时间成本获取到构建产物，减少临时下载制品对迭代效率的影响，尤其对于具备超大制品、全球发行的业务类型，研发保障制品库分发的可靠性尤为重要。

制品库安全扫描研发保障可靠性实践

业务相关制品中往往包含了大量第三方组件，但是这些组件是否安全，使用的许可证是否允许商用等问题存在着很多不确定性。对海量制品进行分析，帮助用户发现制品存在的安全漏洞、敏感信息、第三方依赖许可证等信息

具体实现架构上，制品库管理员在将分析工具注册到制品分析平台后，用户即可指定要分析的制品创建分析任务，分析结束后会通过聊天机器人通知用户分析报告的地址。从

分析任务的创建，到加载文件执行分析，再到分析结束后保存报告，整个流程都是在制品库集群内部完成，很大程度上避免了制品传输到外部带来的带宽压力与数据安全风险。



制品分析平台的设计与实现

分析任务的来源有三种，手动触发、流水线触发、新构件上传后自动触发。触发后就会根据任务指定的分析工具与搜索规则，开始搜索需要分析的制品，为每个制品创建子任务，提交到任务队列中等待执行。

同时针对制品安全扫码研发保障上，平台会根据现网的制品扫描速度监控数据，在制品库管理后台为不同的分析工具分别设置 MaxTime/MB，然后根据这个值为制品计算出超时时间。在创建子任务时也会设置一个最小允许执行时间，避免一些小文件计算出的超时时间过短导致分析工具启动后还没初始化好任务就超时的问题。

以下是某 Top 业务在研发制品保障的具体实践：

制品分析管理服务通过 Micrometer 将当前的任务执行情况数据导出，并对接到蓝鲸监控平台。业务侧可以通过监控可以查看当前的任务执行、扫描速率等信息，针对性地做一些优化工作。

1.业务快速通过流水线插件配置制品扫描

在“归档构件”插件中，可以配置使用制品库安全扫码的功能

a. 流水线添加归档构件或构建并归档 Docker 镜像插件，勾选启用制品扫描

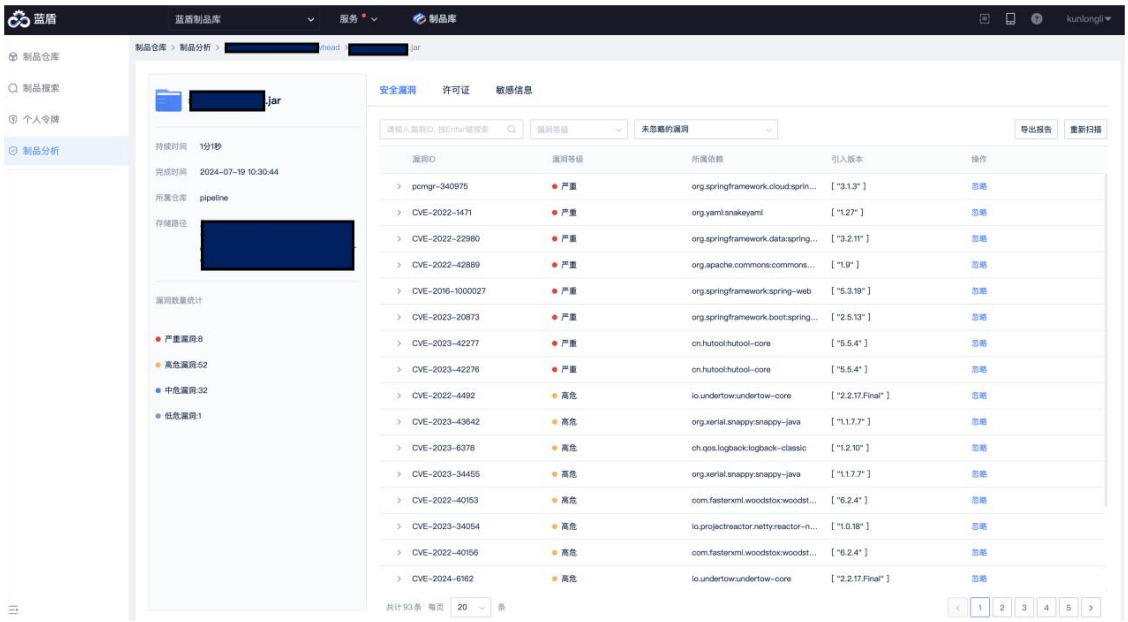
b. 选择同步或异步扫描，同步扫描会阻塞流水线，可以结合质量红线使用，异步扫描会在制品上传完成后继续执行流水线

c. 扫描结束后可以通过聊天机器人通知构建触发人

2. 业务快速查看报告

插件里面可以开启扫描开关，开启后归档完成会触发扫描通过聊天机器人发送报告

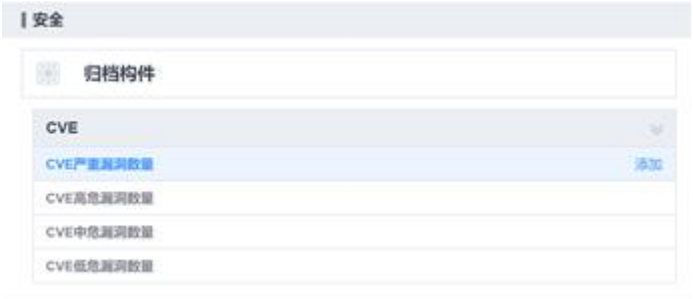
通过具体链接页面可以查看报告，在报告页面，即可查看扫描方案已扫描的制品列表的安全情况，展示的风险等级为制品扫描出的漏洞最高风险等级。从而协助研发人员针对制品问题快速进行优化确认，以免在后续外发系统后，出现更大的安全事故。



3. 结合质量红线

同步制品库扫描可以结合质量红线来使用，对于安全扫描有问题的制品，可以通过质量红线来优化制品质量

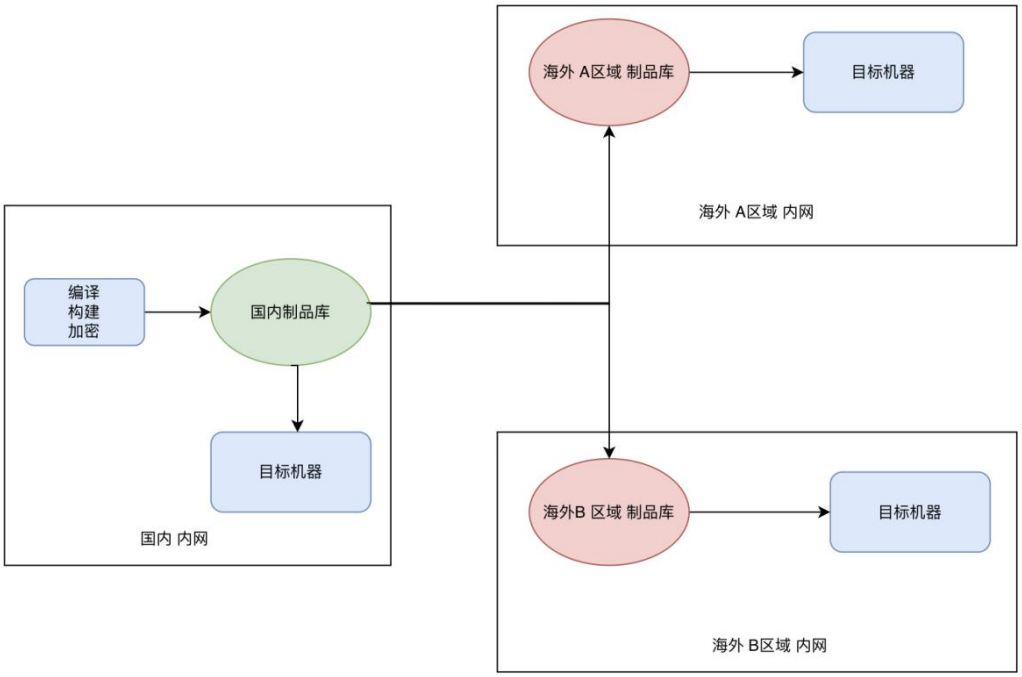
具体步骤，可以在归档构建插件配置页点击立即配置。配置页面点击立即添加。选择单个指标，找到归档构件相关指标添加即可



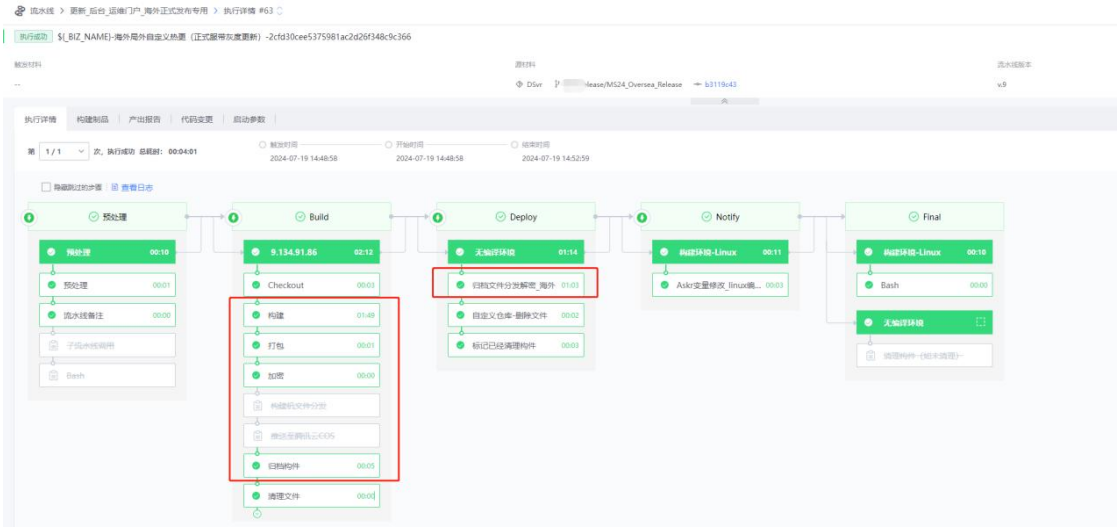
制品部署研发保障可靠性实践

在产品出海的潮流下，业务全球统一发行已经是大势所趋，如何保障版本的一致性部署是在全球化统一发行过程中避不开的难点，以下是腾讯游戏全球发行中版本的一致性部署的解决方案：

- 在国内编译机上从代码仓库(P4/Git/SVN)拉取最新的版本进行编译和构建，并对构建包进行加密后上传到国内的制品库；
- 海外制品库是国内制品库的镜像，在完整性方面有着强一致的校验；
- 国内版本部署时从国内的制品库进行拉取；海外的版本部署从各个区域对应的制品库进行就近拉取，国内和海外共用新的构建包，保障了版本的一致性；



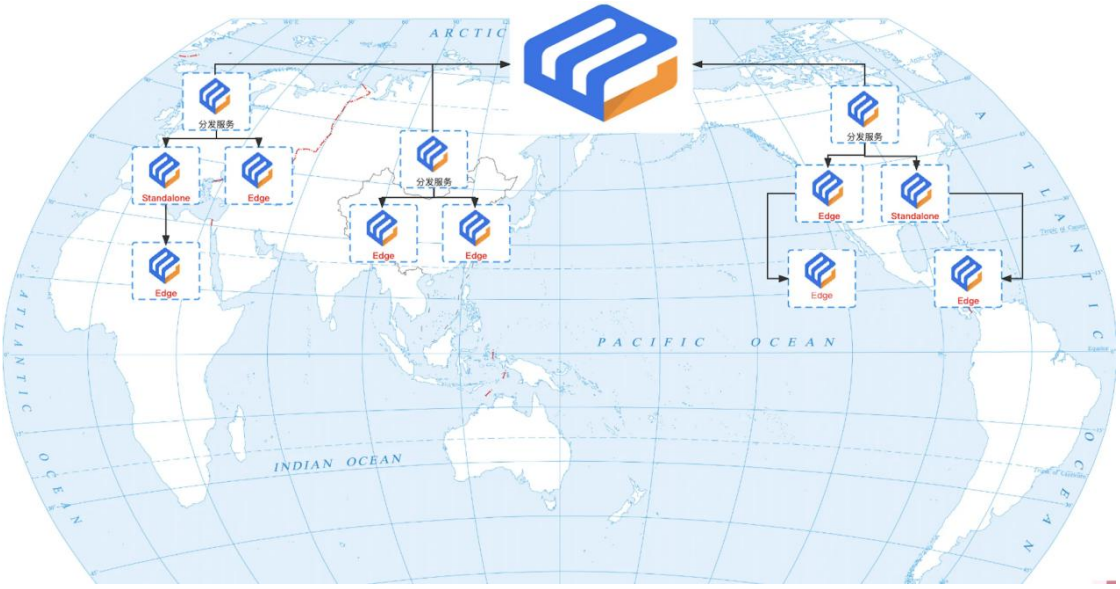
以下是某业务在研发制品分发保障中，平台流程的编排步骤如下：



制品库为了支持用户多地域、多节点使用制品库的场景，实现了制品分发功能，不仅支持多种部署模式以及集群间同步功能，而且还支持将制品分发到异构集群（第三方仓库）。

目前制品分发功能针对同构集群节点已实现 Generic、Maven、Docker、npm 等制品类型的分发；针对异构集群现已实现 Docker、Helm 类型制品的分发。制品分发服务已在多

个环境中进行部署，主要使用场景包含将制品分发到海外不同部署环境中、将构建并上传到制品库的镜像分发到第三方的镜像仓库中。

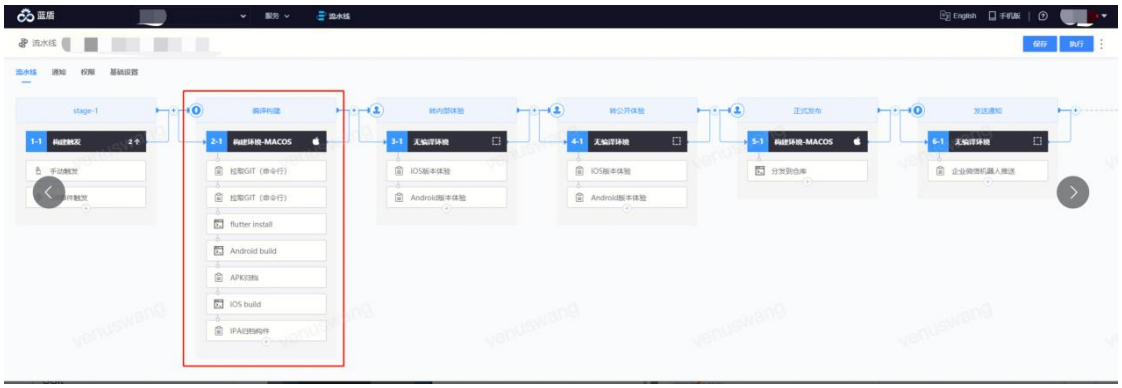


通过 Micrometer 采集各种指标（请求数量、请求频率、线程池正在执行线程数、等待任务数量等），并对接到蓝鲸观测平台。通过查看配置的指标面板，可以查看当前任务执行速率，等待任务个数等；同时还可以查看整个服务的性能状况。

同时，根据研发保障报表，可以查看整个任务全流程中的各种数据，例如筛选出异常任务数据。每次任务执行都会有详细的记录，如任务起止时间、任务调度时间、任务对应制品实际开始传输起止时间、任务执行成功率、传输速率、失败任务数、失败任务详情、执行时长等数、任务失败原因等。依托于任务记录可对每次分发任务进行回溯或者当发生分发失败时可以快速定位问题并解决；同时依托于详情数据可以进行任务对账。

3.下载可靠性研发保障实践快速安装

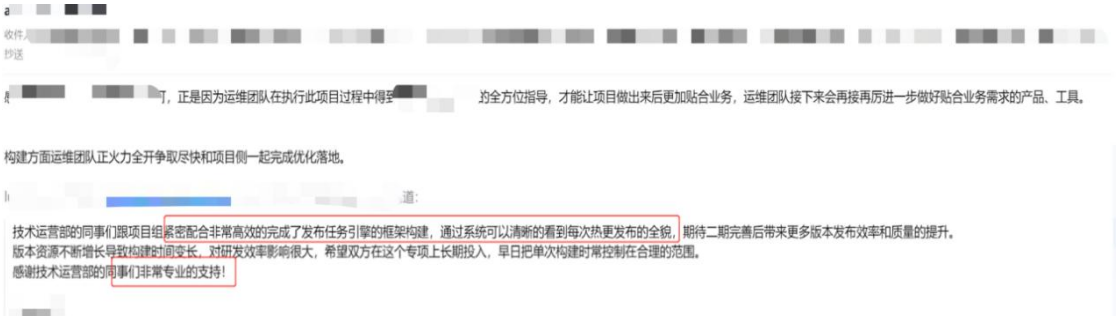
业务可以将流水线构建出来的 IPA/APK 包发布到蓝盾手机应用，用户可以直接通过蓝盾 APP 直接快捷安装新的构建包，便于产品、测试团队验证产品新特性，发现产品潜在缺陷，让产品在内部验证流程上得到闭环，加快版本的迭代完善，并且配套着权限控制，让测试产品能够更安全的对内测试和体验。



通过平台的制品库或者蓝盾 APP，业务就可以查看最近构建的新客户端包，产品和测试团队可以直接扫码安装，加快了产品的安装和验收进程，验收过程有问题，可以及时进行代码修复和重新编译构建，以此进行功能的快速迭代和 bug 的快速验收修复。

（三）总结及展望

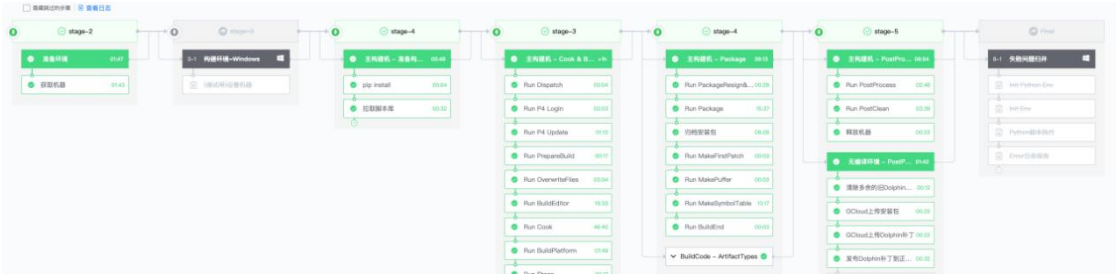
在我们实践中，当 SRE 从代码可靠性、代码仓库可靠性、构建可靠性和制品可靠性的角度切入，解决了业务研发迫切的问题，专业度得到了业务研发的好评，以下选取部分的内部客户反馈。



研发管线优化

编译加速，全量从 6 个小时降低到 3.5 小时左右

客观可量化数据上，有以下的整体业务收益：



步骤	优化前	优化措施	优化后
编译机部署	1、蓝盾 Agent日均掉线告警数 100+ 2、高强度构建 日均磁盘满告警数 30+ 3、170+构建机项目组自行维护，50多人都可操作，编译环境（开发、插件、Agent、命令、系统配置）千差万别 4、Win/Macos编译过程频繁死机	1、标准化管理 2、监控调优 3、系统参数调优	1、蓝盾 Agent日均掉线告警 0-1次（稳定性提升 90%+） 2、管控内机器日均磁盘满告警0次（完全解决） 3、构建机维护由运维全面接管，且全部实施标准化管理（完全解决） 4、异常实时监控，机器可用率99.9%以上
拉取代码（P4代码拉取）	1、P4代码拉取慢，不到100Mbps，问题定位慢，解决效率低	1、Proxy、Commit配置升级 2、启用云研发方案 3、开启并行拉取 4、员工硬件升级 5、实时告警	1、速度提升至500Mbps以上，云研发场景最高到4Gbps 2、异常响应15min以内，大部分问题5min解决
资源拉取	1、UE DDC访问延迟高，下载慢5MB/s 2、Untiy CacheServer下载非常慢2MB/s	1、本地部署、就近部署	1、UE延迟从 30~50ms，降低至 1ms，下载速度提升至25MB/s，提速5倍 2、并发下载已到 20MB/s，提速10倍
代码编译	1、编译构建慢，耗时超过预期	1、TBS多构建机并行能力、pump加速能力、云端加速能力 2、定制高配构建机	1、通过TBS多构建机并行能力、pump加速能力、云端加速能力，编译耗时降低48% 2、全量编译2h缩减至40min
客户端打包	1、签名耗时10min	1、本地签名	1、签名耗时降低至1min，签名效率提升90%
整体	1、核心流水线执行成功率不足 60%	ALL	核心流水线执行成功率提升至 90%+（提升 30%+）

- 业务收益-核心流水线执行成功率从不足 60%提升至 90%，成功率提升达 50%。
- 业务收益-通过代码可靠性保障，质量红线拦截比率 16%，代码缺陷数从 311 个下降为 0，代码安全漏洞数从 37 个下降为 0，包含了圈复杂度、代码规范等多维度的代码评分提升了 40%
- 业务收益-数据资产管理 P4，速度普遍提升 4 倍+，卡顿率降低 70%（几乎为 0.，优化下载拉取链路，下载速度由 100Mbps 到 800Mbps
- 业务收益-通过优化代码仓库边缘加速，代码仓库拉取速度提升 100%，多人同时拉取代码仓库成功率为 99.99%；代码仓库每日卡顿次数从 10+次，下降到 0 次
- 业务收益-通过接入 Prebuild 方案，业务编译构建成功率提升：代码合流后，构建成功率提升 30%以上
- 业务收益-业务日构建综合提效 50% 以上。针对构建机软件流程优化，优化前，业务通过定制构建机，无法用镜像方式进行套件部署，N 台部署 10+构建套件需要 2*N 小时以上。SRE 介入后，通过批量自动化，安装 10+款软件，N 台部署 10+构建套件人工操作环节仅需 1 分钟
- 业务收益-通过蓝鲸持续集成平台中，BKTurbo 集群分布式编译，游戏业务构建出包效率提升 30%～70%；云渲染时间下降 98%
- 业务收益-通过制品可靠性保障，业务制品安全扫描接入覆盖度提升 100%，制品分发到全球，1GB 文件，保证在 5min 内完成分发，同时达到 99.9% 的分发成功率；制品分享效率上，以头部业务为例，通过制品构建后快速应用安装，安装后占用存储空间约 10G 的游戏，节省单测试人力 30+分钟/次，对于 200 人的测试

团队，单次即可节省 100+ 小时的人力消耗，从而保证业务每日出包每日即可完成快速测试的敏捷开发流程持续顺畅进行。

2.3.2 某语音直播公司研发过程保障实践

SRE Elite 精选原因

此案例展示了某语音直播公司在现代化软件架构下的研发保障实践。面对微服务、容器化和服务网格等新技术带来的挑战，该公司构建了全面的研发保障体系，涵盖快速发布、稳定性保障、代码可靠性和服务运行等多个关键模块。其中，采用服务网格进行环境隔离和金丝雀发布，属行业内的创新实践，体现了深入的云原生应用。此外，通过 IDE 插件对接环境进行调试，大幅提升了问题排查的效率。此案例实践性强、创新性高，具有广泛的借鉴意义。

（一）背景及设计原则

以人工智能、大数据、云计算、微服务、云原生等为代表的新型软件架构不断发展，这不仅重塑了整个软件产业的技术生态，还深刻地改变了软件技术的技术架构和商业应用，推动了软件设计、开发、生产、应用等各个环节的发展模式变革。在这种新形势下，软件的应用化发展呈现出数据化、平台化、智能化、云化等特点。

某语音直播公司在现代化的软件架构思想指导下，正是围绕微服务、容器化、服务网格 等现代化技术构建了内部的软件体系。现代化的软件架构思维给传统的研发过程保障带来了全方位的新挑战；同时直播场景在稳定性和软件架构方面与传统软件架构存在差异。

在现代化软件架构下研发具有以下新特点和挑战：

1.基于敏捷开发思想的快速发布与构建

在敏捷开发思想的引导下，团队致力于快速迭代和交付。每天的构建和发布次数达到百次级别。

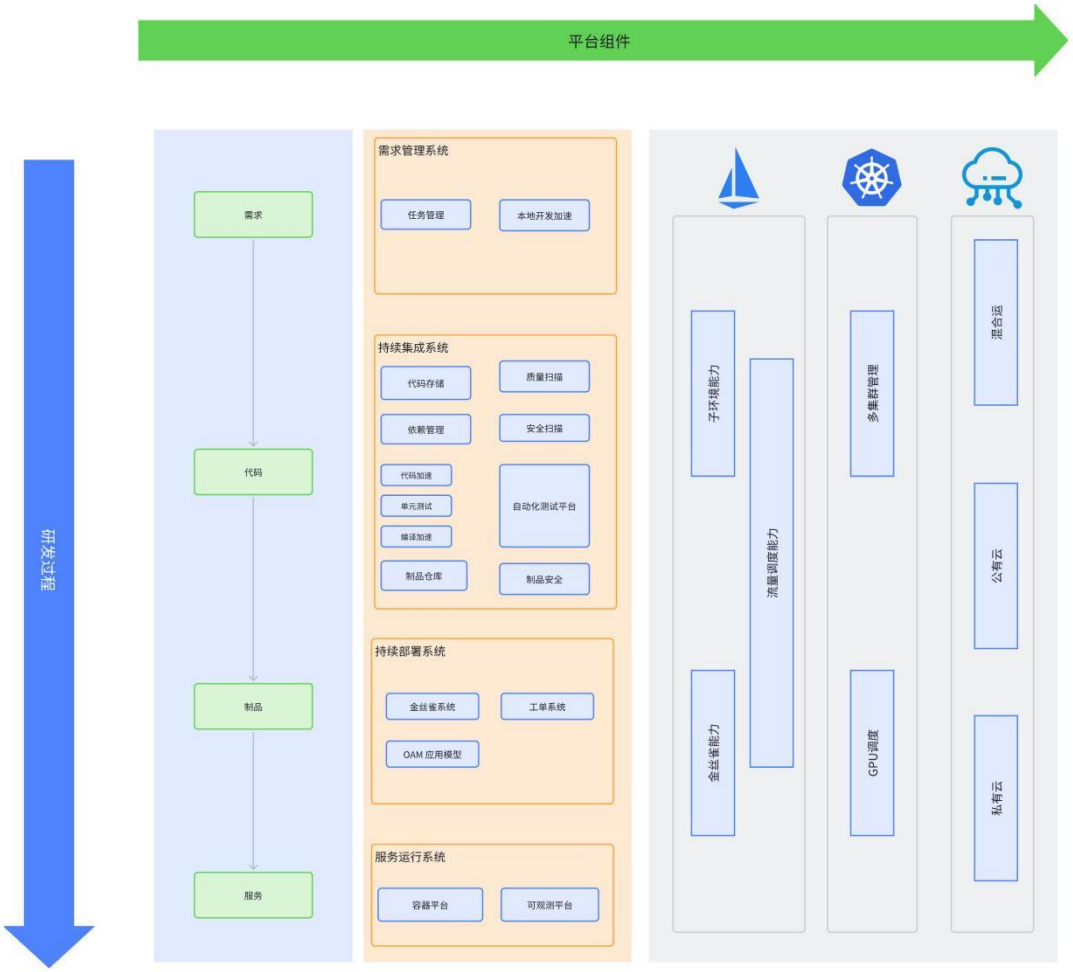
在这种情况下，如何确保构建的快速和顺畅，以及发布过程的平稳和稳定，成为一个巨大的挑战。

2.极高的稳定性要求

在如此快速的迭代情况下，如何保障业务发布的稳定性，尤其是直播场景对稳定性的要求更高。在出现问题时如何快速回滚恢复，以及如何避免影响大量用户成为保障稳定性的关键一环。

（二）体系设计及关键流程

体系设计



为了应对以上挑战，我们构建了四个相关系统来应对。

需求管理系统

为实践 BizDevOps 特将业务团队纳入研发过程，通过需求/任务管理能够及时反馈业务诉求的进度和流向。包含

①任务管理服务，针对需求的状态、自动流转，时效统计等；

②本地开发，能够实现研发本地 IDE 编写代码，自动关联需求，并能一键部署在云端集群本地 Debug。

持续集成系统

为使研发养成良好的代码持续集成习惯，通过流水线引擎自动将代码存储管理、单元测试、自动化测试、质量扫描、安全扫描等手段与工具结合起来。通过质量阈的有效代码提交，将进行自动化编译，最终生成制品。

持续部署系统

针对服务部署所关联的配置和交付过程进行统筹管理。包含 OAM 模型管理、工单流程、金丝雀发布等子系统。

服务运行系统

主要包括

①容器平台，针对服务运行时资源、路由资源、流量控制资源进行管理；

②可观测平台，对部署和应用在集群里的资源进行运行时数据采集、告警等稳定性保障。

(1) 基于 OverlayFS 的拉取源码加速实践

背景：流水线中的任务都需要基于代码执行，公司内部也有很多大仓库（数据量 3-6G），同时流水线本身也是容器化运行的，如果每次都全量拉取代码，消耗时间就会很久很久。所以需要一套机制来解决基于云原生环境的仓库代码缓存方案。

需求分析：

- 基于大仓库的场景，需要做冷启动（预先 clone），同时需要适应代码提交做增量更新。
- 基于流水线的场景，我们需要提供针对 repo、branch、commit 提供代码，同时考虑到流水线的制品晋级和失败重试的情况，需要最大程度复用代码的存储。
- 基于容器化的场景，流水线任务都是在 Pod 里面运行，需要代码以文件系统的方式在 Pod 里面可以访问。

基于 OverlayFS+NFS 的 CSI 方案设计

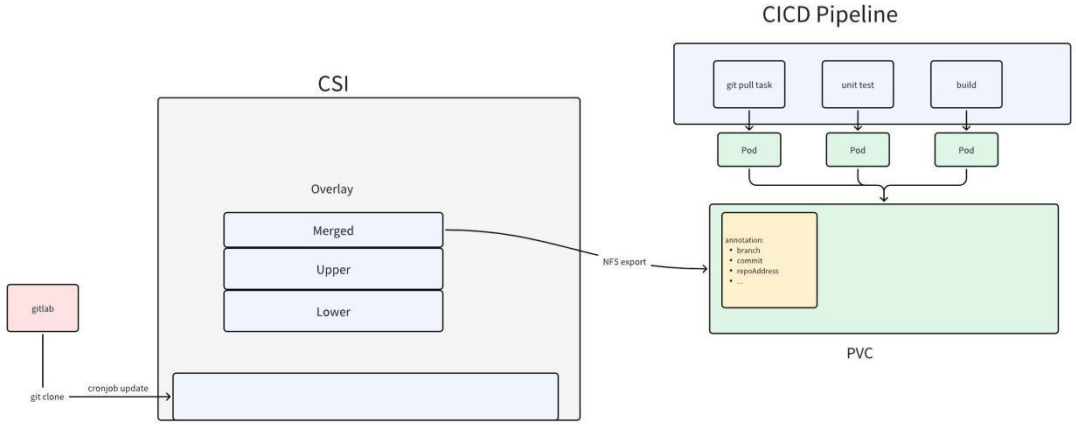
基于流水线的场景考虑，我们需要尽可能复用已拉取到存储介质里（PVC）的代码仓库，所以 OverlayFS 技术是适用于这个场景的。

通过定期将代码仓库中的代码全量制作快照数据，作为 OverlayFS 的 lowerdir，流水线任务拉取代码仓库的时候，只需要拉取少量变更代码（和上次快照之间的差异），可减少 90% 以上的代码拉取量，从而实现加速效果。通过 NFS 协议将 OverlayFS merged 之后的文件作为 PVC 的存储介质提供给流水线任务运行的容器。

K8s 技术选型

基于上面的结论，最简单的方式就是流水线任务 Pod 绑定一个 PVC，这个 PVC 对应于需要的代码文件。

在 CSI 二次开发实现侧，由于 K8S 内置了 NFS 挂载，所以我们的挂载协议基于 NFS 实现成本最低，只需要自己实现 Provisioner，PVC 声明 Provisioner 对应的 StorageClass 即可提供代码文件数据。



(2) 基于持续集成的代码可靠性实践

背景

在公司内部，随着敏捷开发和持续集成/持续交付（CI/CD）理念的广泛应用，开发团队需要频繁地进行代码提交和合并，团队也建立了每日集成的机制。这种高频次的代码集成如果缺乏有效的质量保障机制，可能会导致以下严重后果：

- 代码质量下降：未经严格检查的代码可能包含逻辑错误、代码风格不一致等问题，影响代码的可维护性和可读性。
- 合并冲突增加：频繁的代码提交和合并如果没有质量检查，容易引发更多的合并冲突，增加开发和维护成本。
- 安全漏洞：缺乏代码安全性扫描的代码可能包含潜在的安全漏洞，给系统带来安全风险。
- 测试压力增大：未经质量检查的代码流入测试环境，会增加测试人员的工作量，导致测试周期延长，影响项目进度。
- 发布风险增加：质量不过关的代码进入生产环境，可能导致系统不稳定，甚至引发严重的生产事故。

因此，为了确保代码的可靠性和稳定性，我们需要建立一套严格的质量红线机制，确保每一次代码提交和合并都经过严格的质量检查，只有通过质量检查的代码才能进入测试和生产环境。

保障机制 质量卡点任务

为了保障代码的可靠性，我们在 CI/CD 系统中集成了多种质量检查任务组件，具体如下：

- Sonar 扫描：集成 SonarQube 进行代码质量扫描，检测代码中的潜在问题，如代码异味、漏洞和技术债务等。用于确保代码的可维护性和可读性，减少技术债务。
- 单元测试：在代码提交和合并时，自动运行单元测试，确保代码的功能正确性。
- API 自动化测试：在代码提交和合并时，自动运行 API 测试，确保接口的功能正确性和稳定性。
- 作用：验证系统各接口的正确性，确保系统的整体功能稳定。
- 代码风格扫描：集成代码风格检查工具（如 ESLint、Pylint 等），确保代码风格的一致性。
- 代码安全性扫描：集成安全扫描工具（如 Snyk、OWASP 等），检测代码中的安全漏洞。

质量门禁设置

结合业务现状，有部分老服务代码问题较多，业务又在快速迭代，如果一刀切所有问题都修复才能上线，需要投入人力去修复老服务，投入产出比不高。因此，在 CI/CD 系统上，我们开发了质量门禁设置功能，允许研发管理者，在项目维度，将服务分组，不同分组适用不同的质量门禁，自定义以下质量指标：异味，bug，漏洞，单元测试覆盖率，来适配业务自身的情况，平衡代码质量和研发效率。

Sonar 扫描质量门禁常用分组如下：

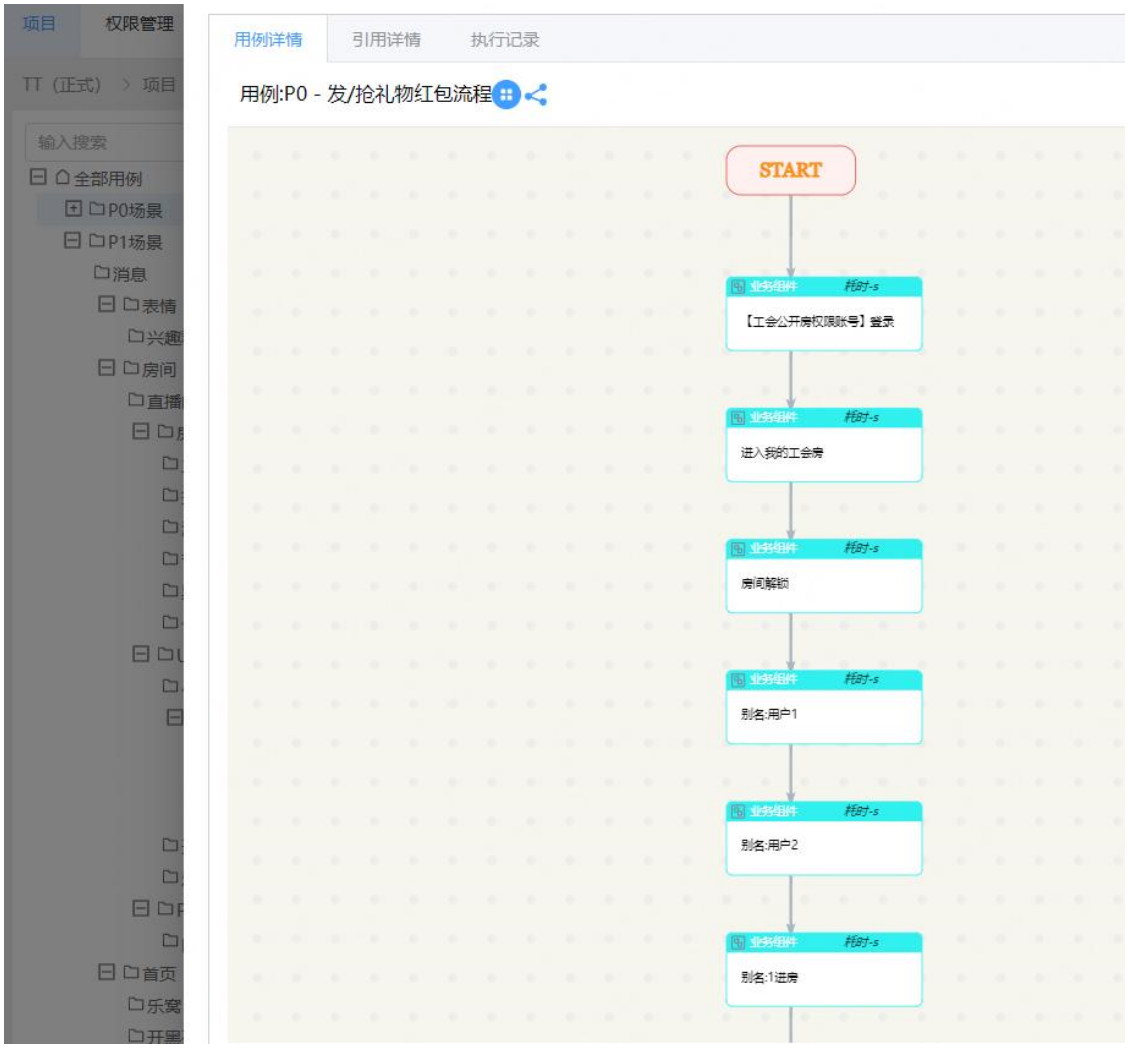
- 老旧服务分组-质量阈值：bug：1，漏洞：1，异味：99，单测覆盖率：0.0%；也就是允许异味的存在，以及单测覆盖率为 0，但不允许 bug 和漏洞不修复
- 标准服务分组-质量阈值：bug：1，漏洞：1，异味：1，单测覆盖率：30.0%；不允许 bug、漏洞、异味不修复，不允许单测覆盖率小于 30%
- UI 界面示例

API 自动化测试

- 接口用例编写

在质量保障要求上，需要服务 100%通过所有的 P0-API 测试用例，不允许失败，需要说明的是，业务侧认定为 P0 级别的用例，都是 APP 最核心的流程，例如：登录，进房，上麦，送礼，等。

这些用例是测试人员通过自研的测试平台上编写，将业务流程，利用测试数据，按顺序调用不同接口来实现，编写界面 UI 示例如下：



若干这些用例组成的用例集合100%通过，保证了单个服务的逻辑变更，对于APP主流程不会造成阻塞，依然符合预期，减少了测试人员回归测试的压力。

- 接口测试可靠性保障

同一个迭代往往涉及多个需求，多个服务的代码编写，如果所有正在开发调试的服务，都部署到了同一个测试环境中，在API自动化测试中发现有用例未通过，无法快速定位到是哪个服务的修改引起的，为了解决这个问题，在API自动化测试中，我们也使用到了后文中提到的“子环境”概念。

在执行API自动化测试之前，需将涉及的服务变更部署至独立的子环境。所有依赖的上下游接口服务请求将定向至稳定的基线环境。此举确保了在API测试执行期间，唯一发生变化的是当前服务。因此，若测试结果中出现未通过的用例，可以明确判定为当前服务变更所致，从而便于研发人员迅速定位并修复问题。

实践效果

sonar扫描，单元测试，代码风格扫描的结果，会在流水线中汇总体现，在整体质量卡点任务通过后，研发人员会收到飞书通知，提醒他本次质量卡点是否通过，也可以主动在流水线中查看，UI示例如下：

飞书通知

流水线

API 自动化测试的结果同样也有飞书通知，以及流水线中展示。

红线机制

结合业务研发流程，在 CI/CD 系统上，我们设置了一系列的红线机制，具体应用在以下场景**识别到有问题的代码 Push：**

在研发人员 push 代码时，自动触发流水线运行。如果有质量检查任务不通过，将会提醒研发人员提交有问题，请检查并修复，从而减少后续合并和发布的风险。

阻止不合格的 Merge Request：

在研发人员发起 Merge Request 时，自动触发流水线运行。如果有质量检查任务不通过，将会阻止研发人员完成 MR，直到问题被解决。这样确保只有通过质量检查的代码才能合并到主干，减少合并冲突和代码质量问题。

发布质量红线机制：

在运行发布上线的流水线时，如果本次编译无法通过质量检查，将无法进入提测阶段和上线阶段，建立了质量红线的机制，类似于安灯系统，确保流入提测和生产环境的代码质量，减少测试压力和发布风险。

通过以上手段和机制，我们建立了一套完整的代码可靠性保障体系，确保每一次代码提交、合并和发布都经过严格的质量检查，保障代码的可靠性和稳定性，减少测试压力，提高开发效率。

下面我们将谈及 Service Mesh（服务网格）的研发过程保障实现。

(3) 基于服务网格的研发过程保障实践

我司落地 Istio 四年以上，最初一年里流量管理能力仅仅只是应用在网关处通过 VirtualService 对指定 host 和 path 进行规则路由。随着 Istio 的稳定和功能加强，我司也在进一步探索 Service Mesh 在解决实际业务诉求方面的各种可能。

结合不同的应用场景与 Istio 的特定功能，分别在部署环境、金丝雀发布、开发过程三个方向取得了实质性的突破。

基于子环境的多环境稳定性保障实践 背景

各个业务团队出于对数据和网络的隔离，拥有众多的部署环境。包括：开发环境、集成环境、测试环境、灰度环境和生产环境，不同环境具有明显的差异。

- **开发环境：**

由于频繁变更，存在不稳定的代码、服务覆盖等问题，从而导致环境不稳定;

- **测试环境：**

直接影响 QA 环节，引入较繁琐的人工审批流程，以确保测试环境稳定;

- **跨部门服务因为依赖**

，会进一步放大了环境不稳定的问题。

为保障各部署环境的稳定，特引入了基准与子环境的概念。

- **基准：**

由相对稳定的服务/组件构建的可提供完整业务服务的系统，同一部署环境内只允许一套基准环境。默认标记名为“base”

- **子环境：**

由至少拥有一个服务/组件构成，借助基准环境上游服务/组件，对有特定标记流量提供完整业务服务。一个部署环境可拥有多个子环境，同一服务/组件可存在于多个子环境，不同子环境具有标识唯一。

实现方案 资源部署：

单个 Namespace 下实现对单个应用多个版本的共存部署，在原有 Labels 的基础之上，对于子环境的资源（Deployment、Pod、ConfigMap、HPA）满足以下规则：

所有资源的命名.metadata.name 以子环境名字作为后缀，即{app}-{senv}，同时修改关联，包含：

Deployment 的.spec.template.spec.volumes[].configMap.name

HPA 的.spec.scaleTargetRef.Name

所有资源的 releaselabel 调整为以子环境名字作为后缀，即{app}-{senv}，同时修 Deployment 的.spec.selector.matchLabels 中的 release（基准环境的 deployment 必须保持原有的值不变：{app}）

所有资源，新增 senvLabel 用于标记环境，取值为子环境名字{senv}（基准环境的值为 base）

Deployment

YAML 自动换行 复制

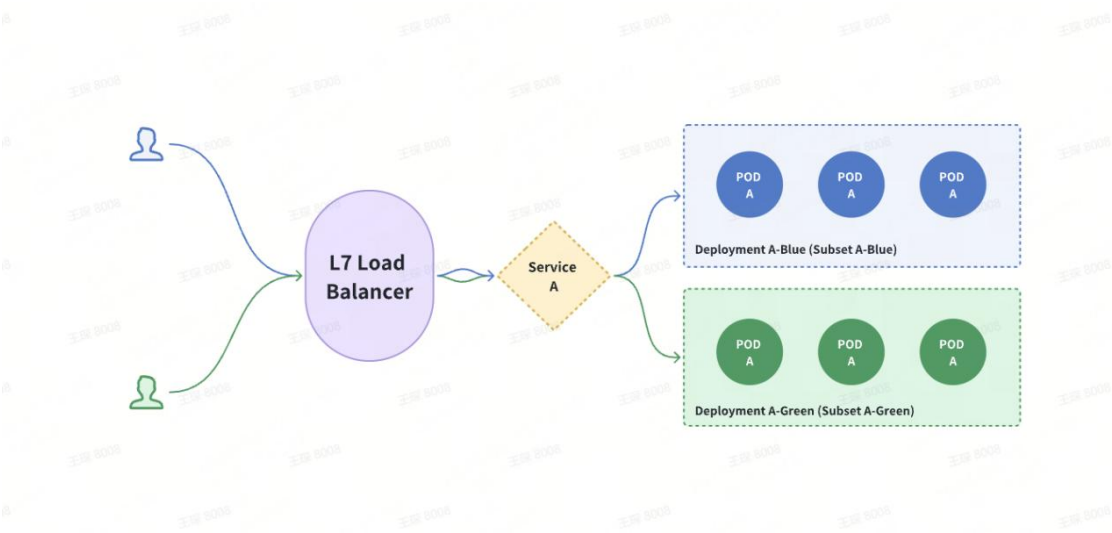
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: greeter
    release: greeter-canary
    senv: canary
  name: greeter-canary
spec:
  selector:
    matchLabels:
      app: greeter
      release: greeter-canary
  template:
    metadata:
      labels:
        app: greeter
        release: greeter-canary
        senv: canary
    spec:
      containers: ...
      // ...
      volumes:
      - configMap:
          defaultMode: 420
          name: greeter-canary
        name: config
```

ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app: greeter
    release: greeter-canary
    senv: canary
  name: greeter-canary
data:
  greeter.yaml: ""
```

路由管理：

通过单 Service+Istio Subset 的方式，实现流量分别到基准 or 子环境。



规则如下：

- DestinationRule：根据 senv 进行 subset 划分
- VirtualService：根据规则将流量路由到不同 subset 中

Destination Rule

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: greeter
spec:
  host: greeter.default.svc.cluster.local
  subsets:
  - name: primary
    labels:
      senv: base
  - name: canary
    labels:
      senv: canary
```

Virtual Service (Delegator)

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: greeter-delegator-80
spec:
  http:
    - name: canary
      match:
      - headers:
          x-qw-trafic-mark:
            exact: "canary"
      route:
      - destination:
          host: greeter.default.svc.cluster.local
          subset: canary
    - name: default
      route:
        - destination:
            host: greeter.default.svc.cluster.local
            subset: primary
```

基于流量染色的金丝雀发布实践 背景

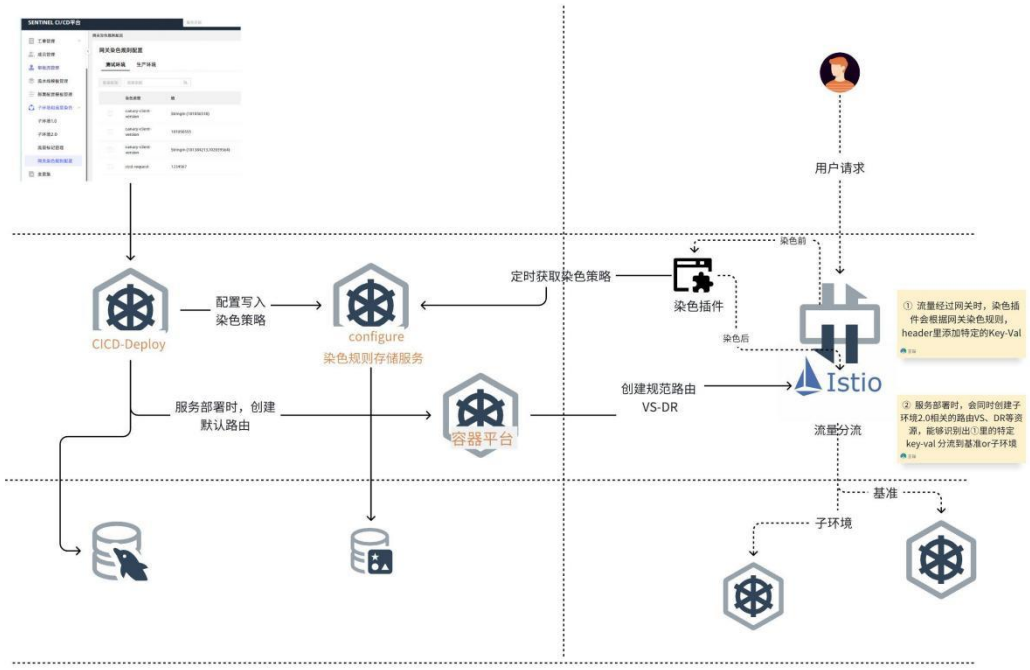
金丝雀发布（灰度发布）作为常见的稳定性保障手段，是指在生产环境正式发布之前，放入小比例的流量并逐步提升，旨在实现流量验证，避免程序缺陷引发大范围影响。以下是金丝雀发布常见的概念。

- 金丝雀环境：是在生产部署环境预创建的一个特别标记的“子环境”，其环境名字和流量标识均为 canary。
- 流量标记：HTTP/GRPC 请求头中携带的指定头（Key 为 X-Qw-Traffic-Mark）的值。
- 流量染色：于请求链路某一环节的中间件（网关、网关插件等），为请求设置特定流量标记的动作。

- 染色规则：用于网关插件中间件针对满足条件的流量，染色成对应的流量标记，所建立起来的一种关联关系。

实现方案

要实现金丝雀发布，在实操上是通过，配置特定的染色规则（用户设备号、用户 ID、App 版本号等）以流量标记的方式，附加到同学呢协议上，并由 istio 识别后，根据标记将部分小比例流量分流至相应的环境。



以下是相关实现金丝雀发布过程中，

控制面组件在整个方案中的作用:

- CICD 平台，Deploy 子服务，对 子环境 基本信息、流量标记、染色规则进行管理。
- 染色规则服务(Configure)，对网关染色规则进行实际的存储管理，作为染色插件的数据提供方。
- 容器平台，服务部署时管理具体的路由规则实例（VS、DR）。

数据面组件在整个方案中的作用：

- **Istio 网关：**

作作为用户流量的入口，Istio 网关处理和管理进入服务网格的外部流量。它支持 HTTP、HTTPS、gRPC、TCP 和 UDP 协议，提供灵活的流量控制和路由能力。

- **染色插件：**

纯自研的 WasmPlugin，通过 Istio Gateway 被加载。首先读取 Configure 服务的染色规则，然后根据规则对经过网关的流量进行“染色”，即，在 HTTP Header 中增加特定的 Key-Val。其中 Key 固定为 X-Qw-Traffic-Mark，Val 为染色规则绑定的流量标记。

- Istio（Virtual Service、Destination Rule）通过识别染色后的流量标记，采用 VS 或 DR 规则分流到具体环境的 POD。

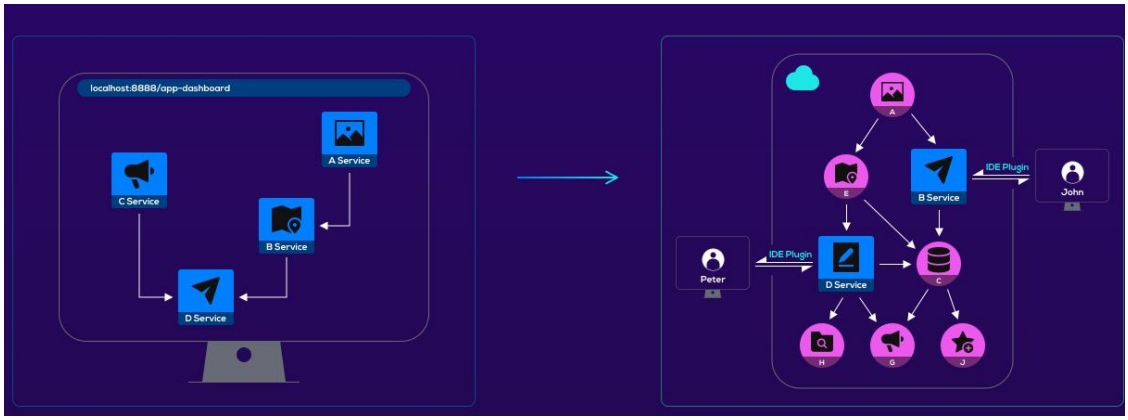
通过上述组件的协同工作，系统实现了高效、安全的金丝雀发布流程。

基于子环境的开发过程加速实践 背景

在研发工序中，在需求-->开发-->联调-->测试-->发布整个流中，最频繁的动作就是开发-->联调的过程。特别在微服务、多人协作开发的模式下，往往会出现开发人员有多个版本需要在集群环境中调试的情况。而且在调试过程中也必须通过流水线发布才能发布到集群中，流水线的耗时等待更加加剧了这个过程。

实现方案

如果想要加速这个过程，最好的方式为：让开发人员拥有在本地 IDE 开发调试一样的方式，对集群中的服务进行调试。这样就减少了流水线的发布等待过程；借助子环境的隔离能力可以做到服务版本的隔离，解决多人服务争抢的问题；借助子环境流量染色的能力，实现服务间的功能联调。方案前后对比如下所示：



既能降低开发人员的学习成本（与本地 IDE 调试一致），又能加速研发过程。

我们通过自主研发 Vscode、Jetbrains 插件的方式，为开发人员赋予了这样的能力。

IDE 插件的运行流程如下：

i.当开发人员需要进行远端调试时，会克隆当前服务在集群中的工作负载，并为其打上开发人员的标识（子环境、用户），使其作为服务的一个“子环境”在集群中运行；

ii.借助复用“CSI”的代码加速功能，将开发人员本地的代码增量同步到工作负载中；

iii.启动服务，并针对相应语言启动一个调试服务器（例如 golang 对应 dlv）；

iv.通过端口转发，将本地 IDE 的调试客户端连接到集群的调试服务器上；

v.调试结束后，回收集群中对应的资源。

如此一来，便实现了开发人员在本地远程调试服务的功能。

（三）总结及展望

目前，公司内部已有超过 95% 的业务实现了容器化，且 80% 的业务运行在服务网格中，覆盖了所有核心业务。基于 Kubernetes 和服务网格等云原生基础设施，公司构建了一套从应用研发到发布上线的全流程保障体系，将效率和稳定性视为生命线。

面向未来，随着大模型与 AI 应用的兴起，研发过程保障迎来了新的机遇与挑战。例如，使用大模型辅助编码、进行代码审查（Code Review）以及问题排查等，大模型在研发领域展现出巨大潜力。我们将重点思考如何基于大模型构建一个能够保障各场景应用研发过程的体系，以进一步提升研发效率和质量。

3 入网控制

3.1 运行环境适配

3.1.1 运营环境设计

产品在经过了核心概念的提炼，通过对市场分析、确立用户定位、关注用户体验和产品风险等方面后，建立项目筹备小组，在完成了开发产品原型，验证技术风险，及通过相关评审确定进入量产阶段以完成产品的全部内容的开发。此时，将进入产品面向用户的试运营阶段，根据新产品评价体系每月对产品运营数据指标进行评测和汇报；确认产品质量是否已达到面向外部用户的标准；经过相关评审后，项目组可根据实际情况启动正式运营。

通过运营环境的规划与设计，可以根据过往的运营经验形成可运营规范，帮助项目组和研发规避相同的风险，同时结合运营策略进行推演正式运营后的情况，以此来提前发现潜在的风险和问题，便于和运营、研发团队进行评估和优化排期。同时，对于规模化测试所需人力、用户体验、成本预估、设备及机房选型、网络带宽资源及周边组件的容量，可用性等进行提前的评估和筹备。鉴于越来越多的组织采用多云环境，我们在规划过程中特别注重对公有云、私有云和混合云的供应商进行优劣势分析，同时充分考虑运营环境与云服务的耦合程度，做出恰当的选择和权衡。

从组织结构上来说，可以成立一个专家组，主要的工作项是横向提炼各业务在运营阶段中的各种共性问题、解决方案的沉淀和刷新，确立可运营规范。同时，在业务的关键节点参与技术评审。

对业务支撑团队，在进行运营环境规划与设计时，主要可分为如下几个步骤：

架构。业务决定量产并分配到支撑团队后，支撑团队需获取《业务架构说明书》《部署文档》，运营节点（建议近三个月为宜）等必要的业务相关文档从而了解业务架构、各模块功能和通信逻辑、容灾及技术指标等信息。以传统模式部署还是云原生模式部署，在这个环节会确定下来。

评审。运维专家及支撑团队组织项目组、研发团队进行可运营规范、评审标准、版本交付规范等的宣导;深入就业务架构，运营目标涉及的技术问题进行专项沟通，以此来对业务有个整体了解，梳理风险及问题。

验证。通过对业务测试环境的搭建，掌握业务搭建方法，并分析可自动化和改善的点，此阶段非必须建设自动化，因为业务架构成熟度和交付结构可能会经常变动。同时根据对外测试的的目标选定部署的机房、机型及合适的网络运营商等环境规划。

对齐。正式运营阶段筹备前根据节点时间进行倒推梳理整体支撑的检查列表，明确工作项、完成时间、负责人和细节部分，并与干系人逐一明确对齐，共同完善。和研发、运营共同确定业务稳定性目标 SLO 及相关的指标 SLI，以及围绕用户体验所需的场景和分析方式，并做好相关的数据埋点、呈现和监控。对于基线数据提前做好规划和收集。

迭代。配合业务压测，了解性能瓶颈，并重点进行方案的拟定和保障实施。根据运营中的问题刷新风险问题列表，并设计解决方案持续跟进，并在下次产品迭代更新后进行实际效果验证。

3.1.2 容器云适配

随着云原生技术的不断发展，容器以及 Kubernetes 等技术的长足发展给企业数字化转型带来了便利。容器技术将软件运行环境打包成一个“集装箱”，方便在不同环节进行传递；而 Kubernetes 则将容器的调度和部署标准化，让开发运维人员不再关注资源层面的调度和容灾。容器云适配则是通过云平台、Kubernetes 和 Docker 等云原生技术帮助企业降低成本，加快业务迭代，对企业数字化转型提供强有力的技术支撑。

通过容器云适配可以简化了服务部署等方面的运维管理的复杂度，让业务团队更加专注于自身核心业务逻辑的开发，从而实现快速的业务迭代。同时通过虚拟化、资源池化以及弹性调度等技术，增加资源交互弹性，帮助业务团队不断降低资源成本，因此 SRE 应推动业务容器云的适配。

改造筹备。筹备相关的内容主要目的是梳理各模块之间的逻辑以及交互方式，便于针对容器环境进行适应性评估，明确业务架构与容器环境之间的差距，便于业务进行架构调

整。例如按照业务进程本身是否缓存数据，可分为有状态模块和无状态模块，无状态模块。同时还需要关注进程的服务发现机制，如果业务自己有成熟的服务注册机制，例如业务自行设计的服务注册中心，那么在容器化的过程中，优先会有部分模块会落入 Kubernetes 集群中。

网络评估。一般情况下，Kubernetes 的集群内与集群外是两套独立的网络，一旦容器化后落入 Kubernetes 集群中，内网访问时网络链路就会发生变化，也就是 overlay 网络。在 overlay 方案中，我们可以认为集群中的所有容器，默认都可以通过 IP 地址相互直连。集群内可以主动访问集群外，但是集群外看到的来源 IP 是集群节点的 IP 地址，并不知道容器 IP 地址。即使知道容器 IP 地址，集群外也不能直连容器 IP 地址基于这些条件，业务需要评估各模块访问关系，确认落入集群中后是否需要进行针对性适配。

工作负载评估。工作负载评估的基础原则是如何基于业务无损、不停机更新的情况下，可以选用哪种工作负载进行容器管理。对于各类更新方式：滚动更新、蓝绿发布、金丝雀发布，业务落地容器化必须至少选择一种方式进行管理，这是保障我们利用好容器特性的基础。常用的工作负载是 Deployment 和 StatefulSet，是 Kubernetes 原生提供的分别解决微服务部署，有状态服务部署的方案。建议优先评估是否可以采用原生工作负载进行部署。

弹性伸缩评估。弹性伸缩是云计算中一种常用的方法。通过设置伸缩规则来自动增加/缩减业务资源。HPA 是 Kubernetes 基于弹性伸缩进行的规则设计，Kubernetes 对 Workload 的资源使用（CPU，Mem）、自定义 Metric 指标进行负载评估：当负载超过设定高水位时，例如 CPU 平均负载超过 60%，就扩容 Pod 实例以降低负载到设定水位之下。当负载低于设定低水位时，例如 CPU 平均负载低于 20%，裁剪 Pod 实例以释放资源，降低成本。要实现水平扩缩，业务需要在业务逻辑与架构上实现优雅退出、负载均衡。优雅退出：业务容器负载降低时，HPA 会针对模块实例进行缩容操作，业务容器必须要优雅退出，保障业务逻辑顺利完成退出动作。负载均衡：当负载增加时，HPA 会直接增加容器数量。无状态服务的处理方式较为简单，可以通过前置的负载均衡器有效均衡负载；有状态服务则需要业务动态感知实例/容量变化，合理分配玩家至新增服务实例。

容器云适配后的交付和管理。容器云适配后建议使用 GitOps 来进行持续交付。GitOps 是由 Alexis Richardson 在 2017 年首次提出的，该模型主张将 Git 作为“单一事实来源”来管理和同步开发和生产环境，而不是使用传统的基础设施管理和部署方式。GitOps 的核心思想是将应用系统的声明性基础架构和应用程序定义存放在 Git 版本库中。将 Git 作为交付流水线的核心，每个开发人员都可以通过提交拉取请求（Pull Request）并使用 Git 来加速

和简化 Kubernetes 的应用程序部署和运维任务。每个环境都应该有一个代码仓库，用于定义给定集群的期望状态，然后 GitOps operator 会持续监控特定分支（通常是 master 分支），并在探测到 Git 发生变更后，将此次变更传递到集群，并更新 Kubernetes etcd 中的状态。通过使用像 Git 这样的简单熟悉工具，开发人员可以更高效地将注意力集中在创建新功能而不是运维相关任务上。

3.1.3 数据库存储适配

为确保项目数据库运行的稳定性、可靠性、可扩展性、安全性以及高性能，SRE 应该对数据库存储进行适配或优化，确保可以为应用程序提供高质量、高性能、稳定性强的数据存储和访问服务。主要包含以下几个方面：

选择合适的数据库类型。根据项目需求、项目架构设计和数据模型等，选择合适的数据库类型，如果数据具有固定的结构和关系，那么关系型数据库（如 MySQL、PostgreSQL、Oracle 等）可能是更好的选择。如果数据具有动态的结构，那么非关系型数据库（如 MongoDB、Cassandra、Redis 等）是更推荐的选择。除数据模型外，在评估数据库类型的时候还需要考虑项目的数据规模、业务场景、可扩展性、性能、一致性、可用性、成本等因素，综合考虑数据库选型。

数据库部署和配置。选择合适的服务器部署数据库服务，并进行最佳的数据库参数配置，如调整内存、连接数、缓存等参数，以满足项目所需的数据库运行的性能和资源需求。

数据库高可用和扩展。根据项目需求，实现数据库的高可用和扩展，如搭建主从复制、分布式设计、读写分离等。

数据库监控和告警。实施数据库监控，收集数据库关键性能指标（如 QPS、慢查询、磁盘空间等），并设置合适的告警阈值，以便在出现问题时及时发现并处理。

数据库备份和恢复。制定数据库备份策略，定期备份数据，确保数据安全。同时，要熟悉并掌握数据库恢复流程，以便在发生数据丢失或损坏时能够快速恢复。

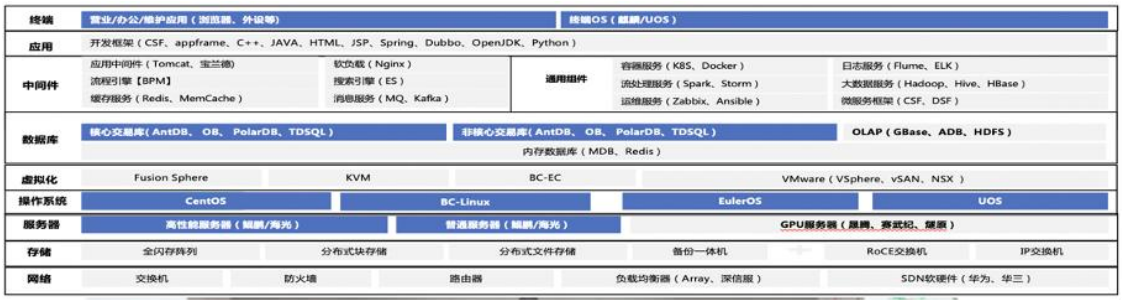
数据库性能优化。分析数据库性能瓶颈，优化查询语句、索引、表结构等，提高数据库性能。

数据库安全。保障数据库安全，如设置合理的权限、防止 SQL 注入攻击、加密敏感数据等。

3.1.4 信创适配

在政策推动下，自主可控成为当下热点。各大企业均在进行自主可控替代，加大信创布局。信创产品相较于传统商业产品，信创产品的技术成熟度、生态成熟度存在着客观差距。对于 SRE 来说，针对含有信创产品的运行环境适配，需要建立以下工作流程：总体目标制定、信创产品选型、产品验证测试、推动适配改造、业务测试、割接上线、运行保障。

总体目标制定。根据发展趋势，企业在整体 IT 投入中，明确对信创产品的采购比例。从而制定企业自主可控的总体战略目标，即年度含信创替代的项目计划，以及项目中进行信创替代的比例和对象，绘制全局信创替换后的架构图，重点标注端到端自主可控列表，包含硬件、数据库存储、容器平台及组件、终端等。每一次的信创项目完成后，均可以更新端到端的全局信创架构图以及具体的信创产品应用清单。



信创端到端全局示意图

信创产品选型。确定项目的信创替代目标后，具体在哪一层进行信创产品的替代也随之明确。根据项目的生产环境及业务场景，对替换对象进行选型。以数据库存储、硬件、中间件这些为例，SRE 有以下维度的选型策略。

1. 数据库存储

在信创过程中，SRE 参与各类信创项目的开展，为确保生产运维的可维护性，SRE 需要在组件选型适配上重点关注。项目需要通过不同的业务场景完成数据库选型，从联机事务处理过程、后台批量事务处理过程、联机分析等业务场景构建出业务主要特征，根据操作特性、数据量、实时性、一致性和 SQL 依赖等主要特征来适配业务场景。

2. 硬件（服务器、操作系统）

在操作系统选型上需要从管理清晰度、公司战略、使用成本和可维护性为出发点，综合考虑开源和商业的使用版本。

3. 中间件

在中间件选型上要根据业务开发语言的适配程度、改造难度、可维护性等综合考虑，更多地了解实例基础配置的差异性，以及各类中间件产品的特性，维护过程中的复杂性等确认选型产品。

产品验证测试。针对各个产品，搭建同步于生产规模的产品测试环境，主要包含计算环境以及网络环境。对产品自身的相关项目进行测试，以体现产品的能力。测试完后，输出详细总结报告，报告中应包含每一个测试项的结果指标、现存 bug 和原有的非信创产品的横向指标比对。

以数据库产品测试为例，会进行数据类型及语法测试、数据库性能测试（包含数据查询、更新、删除、聚合、多表关联等场景）、业务 SQL 测试、高可用测试、数据备份和恢复测试、可维护性测试（包含错误检测和提示、日志信息、在线升级等）。完成所有产品测试后，把测试过程中发现的问题和风险整合到产品测试报告，以供选型决策参考。

推动适配改造。确定好信创替代产品后，根据信创产品的特性，需要已有应用架构做关联适配，主要包含两类，一类是集成部署架构的适配改造，如一些国产化的操作系统，部署配置应按照其系统路径进行调整；另一类是应用的代码适配改造，确保兼容信创产品后不影响系统稳定性。以下列了分布信创的适配改造场景：

1. 针对硬件（服务器、操作系统类）

进行编译适配：针对语言编译类型选择不同方案适配 CPU 架构；通过容器封装或者构建相似环境实现操作系统适配；

2. 针对数据库

进行数据库解耦及开发框架升级：将与数据库多样化交互收敛到中间层，降低 90%换库导致的应用改造工作量；将分布式事务控制在应用层；禁用存储过程和数据库链，禁止面向特定特性进行编码，实现面向对象编程。

3. 针对终端

进行代码适配：业务 SQL 类、函数类调整，CSS 布局，渲染适配调整；弹窗异常、回调事件不触发、按钮无效、无法投屏等适配。

业务测试。在应用架构根据信创产品进行适配改造后，进入业务测试阶段，首先在测试环境上，使用测试数据进行功能、性能测试，并进行切换演练。确认过程无误后，在真实环境搭建的创新集群上进行测试。

1. 功能测试

第一轮采用生产核心的业务场景或回归测试用例，对信创集群进行接口调用。梳理报错场景进行分析改进。

2. 性能测试

使用自动化压测平台，对信创集群进行压测，根据新集群和生产集群的比例，对比压测后的响应结果，评估集群的性能情况。

3. 非功能测试及切换演练

针对评审中因引入信创产品后，可能引起的非功能性，比如由于数据库中间件的新增，需要重点测试批量数据操作的性能，由于开发框架的调整，需要重点测试故障转移、接口超时设置生效、服务熔断能力等。

此外，由于信创集群的引入，在割接中当晚会使用流量倒换的模式，把部分流量切换到信创集群/平面。所以需重点测试切换能力是否可以快速生效。

割接上线。通常情况下，为确保生产业务的稳定性，涉及信创产品替代的项目，采用先单平面改造上线，通过流量控制，把一部分生产流量引导信创平面进行验证，并行运行稳定后，再进行全量替换的方案。

割接当晚首选不停服发布，首先把部分流量引入到信平面，进行功能、性能及其他非功能性测试验证。验证无误后，把新老平面并行运行，并进行相应的测试验证。重点测试平面间切换开关的演练，如果次日业务量高峰时新平面出现问题，可随时启动切换开关，把流量全部引导到原有的老平面。

运行保障。针对信创项目割接，建设应急处理的三板斧，切流、重启、扩容，首先应保证割接次日系统异常时可通过切流快速应急恢复，其次信创平面的容器云平面，可实现自动重启。此外，还应具备弹性扩缩容或人工接入时的一键扩容能力。确保生产业务系统的稳定性。

在可观测维度，重点关注将信创领域的关键指标整合进统一的运维监控体系中。鉴于信创涉及从端到端的技术栈改造，其磨合过程可能较为漫长，因此，我们可以考虑采用eBPF技术来实现对网络、系统和应用的全链路监测，从而为系统故障诊断和持续稳定运行提供坚实的支撑。

3.2 运行环境交付

3.2.1 基础资源服务

围绕稳定性系统下，SRE 需要提供一个可恢复性、可扩展性的基础资源服务。基础资源包括：服务器、存储、网络、数据库、中间件等。SRE 应推动通过服务化方式交付基础资源。主要需要完善如下方面的内容：

推动可恢复性的基础设施环境建设。可恢复性的基础设施环境是指信息系统依赖的硬件基础设施环境、应用平台、中间件、数据库等资源能够有效地从故障中恢复。为了达到可恢复性，这些基础设施在技术选型、构建、运行保障中，需持续考虑到各种风险，包括自然灾害、技术灾难、请求激增、人为错误或外部攻击等，并采取预防性、冗余性、高可用架构等以减少潜在的损失。它还具有在发生灾难后快速恢复的能力，这包括备份、冗余、快速恢复服务和关键信息系统的保护。可恢复性的基础设施环境还强调在设计和规划阶段就考虑到恢复能力，而不仅仅是灾害应对和恢复阶段。

以云的方式交付基础设施环境。SRE 应推动混合云的平台服务能力建设，以支持按需选择一个能够满足需求的云服务。SRE 根据应用程序的要求、用户的数量、数据流量等因素，在云平台的服务目录，在线选择资源池的 CPU 、内存、存储、网络、数据库、中间件、平台应用服务等资源，并提交相关申请。SRE 可以在线查看已分配的资源和运行的应用程序，并可以通过管理终端对应用程序进行管理和维护，例如，启动、停止、重启、删除等操作。当不再需要某项资源时，可在线释放该资源，并将其返回到资源池中。同时，SRE 应在云环境部署应用程序，包括安装和配置应用程序、配置相关的网络、安全设置，以及在部署完成后，对云环境进行管理和监控，以确保其正常运行和安全性。

量化评估基础资源环境。SRE 应建立量化基础资源服务的稳定性与成本管理指标，包括平台性能（资源服务的平均响应时间、吞吐量、处理能力等）、可靠性（资源服务的可用性、容错性、故障率等指标）、可扩展性（资源服务的系统扩展性、负载均衡能力、资源管理能力等指标）、安全性（数据安全性、访问控制、漏洞管理等指标）等。成本指标，以及资源成本管理（资源的利用率、调用次数等指标）。

3.2.2 可观测策略

可观测主要是为了应对 IT 运行环境与技术架构复杂性，重点解决 SRE 在故障发现、故障诊断与故障恢复环节的应急过程管理。要求 SRE 需要从原来只负责可用性被动保障的

角色跳出来，站在白盒角度看系统运行状况，剖析系统层面的运行信息。SRE 在推动可观测试策略时，需建立以下工作流程：

制定可观测标准化规范。为有效实施可观测性策略，需推动监控、日志、链路追踪等相关技术规范的制定，确保信息系统满足这些技术要求。规范应聚焦于增强主动监控性能指标数据上报能力，优化日志的标准化和可读性，实施数据埋点以建立链路追踪，并提供必要的基础设施服务支持，如系统监控数据上报和日志采集分析。技术规范应适用于新建和现有系统的持续优化，并明确配套的工作流程，确保可观测性工作贯穿软件的需求、设计、开发、测试等各环节。此外，组织内需要加强规范宣讲，推动系统架构师和研发工程师等人员认识到可观测是系统建设的必要条件。

建立可观测工作小组。可观测在生产保障中应用，但具体工作涉及产品、研发、测试、运维多个团队的协作，SRE 需根据可观测规范建立以系统或业务为单位的可观测工作小组，以提升协同效率。SRE 是可观测工作小组的牵头人，负责推动具体系统可观测能力的持续建设。因此，SRE 需要清楚的知道系统的业务运行状态、应用服务状态、批处理状态、性能管理、容量水位、依赖环境等黄金指标，明确相关黄金指标的数据计算逻辑，推动相关功能的建设与验收。同时，需要提前参与到系统设计阶段，梳理指标异常监控策略，以及系统依赖的应用平台、系统软件、基础设施、上游系统的监控策略，以及如何通过应用日志辅助故障定位。另外，需将系统的上下游、交易请求链路、资源依赖等关系作为软件交付物之一，系统能够为感知链路节点的健康状况提供数据。

融入需求设计评审阶段。SRE 在系统的非功能性设计评审阶段，应参与可观测方案的制定，提出软件可观测需求，并明确具体的监控、日志、链路的数据埋点，以及工具支持的需求。

跟踪可观测的技术实现。在入网控制阶段，SRE 应构建完善的数据采集、加工与处理平台，涵盖日志管理（如 ELK）、性能指标监控（如 Prometheus、Zabbix）、链路追踪（如 Zipkin、Apache Skywalking），持续性能剖析（如 Pyroscope），eBPF 无埋点观测等能力。

为了处理和分析不同工具产生的可观测数据，我们建立了一个统一的数据关联框架，该框架包括数据收集器、处理器、存储系统以及查询和可视化工具。这些组件共同作用，实现了多种观测数据的关联融合，并支持多层次的数据下钻和追踪，构建起了一个可观测的拓扑结构。

此外，还需建立一套全面的数据质量监控机制，覆盖数据的生成、收集、加工和使用各个环节，以确保数据的准确性，可靠性及实时性。

验收可观测策略交付。SRE 在入网控制阶段应验收可观测相关数据埋点、可观测工具、生产环境配置等工作，确保系统上线后，配套的可观测工作机制能够顺利开展。

3.2.3 自动化策略

SRE 应尽可能的推动自动化一切的工作期望。自动化策略是将事件驱动思维模式融入到运维的方方面面，可以从思维、技术两个角度发力。思维角度，即 SRE 组织从一线操作、二线运维、管理岗位，要对重复性、操作性的工作琐事有天然的排斥感，并想方设法用软件方式代替手工操作。技术角度，一是从 SRE 工具层面建立以原子脚本、编排任务、任务调度的自动化操作能力；例如，通过编写脚本来自动化日常运维任务，并使用如 Terraform 等工具来实现基础设施即代码（Infrastructure as Code，IaC），确保环境的一致性和可重用性。二是将 SRE 手工操作标准化，并将标准的运维操作场景化，基于场景将自动化操作与工作机制相结合；这可以通过编写标准操作手册（SOPs）并将其转化为可执行的代码。三是 SRE 工作前移，推动应用系统自身自愈或无人值守的可靠性设计。

推动定时任务集中管理策略。为了解决定时任务的作业调度可靠执行问题，在没有集中作业调度策略前，SRE 需要各显神通，采用类似 crontab 、Windows 定时作业，以及基于软件系统程序的定时作业等解决方案。由于多个定时作业执行状态可能会相互影响，分散式的定时作业管理容易引发风险、效率、管理等问题，比如：遗漏某些重要的业务调度步骤引发账务风险，手工任务周期设置有误导致任务漏做或重复做风险，调度批次异常无法及时发现的风险，人工重复执行不可重复操作作业任务，重复建设调度管理工具扩大成本等。因此，SRE 应推动集中式的作业调度自动化管理，例如部署 Kubernetes CronJobs 等工具，将规律性的任务固化由机器执行，支持多种目标端对象的任务执行、支持统一采控的远程任务执行、支持低代码的流程编排、支持快速应对作业异常处置等能力，解决系统稳定性保障涉及的生产力、安全控制等问题。

推动软件交付的自动化策略。软件交付是 IT 的关键价值创造，持续交付是提升软件交付速度的一个工程实践。持续交付的软件发布方式提倡全自动化，即围绕软件程序的部署，编排发布各环节的自动化操作。发布流水线是自动化策略落地的关键技术，流水线将一个软件发布环节串联起来，让软件交付过程中不同的角色可以透明地看到整个过程。同时，通过线上化各环节的执行步骤能够量化持续交付的水平，比如自动化测试覆盖率、缺陷数

量、每天构建次数、发布平均时长等，量化数据能让团队清晰地看到低效环节并进行改进。对于不同的应用系统，SRE 应建立针对性的自动化发布策略，比如针对敏态应用，SRE 还要推动蓝绿、灰度/金丝雀、滚动、红黑的自动化部署策略等。

推动故障自愈策略。自动化的故障自愈是指通过自动化的方式，在系统出现故障时快速、准确地恢复到正常状态，减少人工干预的必要性，提高系统的稳定性和可靠性。SRE 在推动系统的故障自愈时，可以从应用程序应对异常时自愈的自动化，包括应用程序健壮性涉及的降级、熔断等，以及在应用程序以外配置的自动化自愈策略。从技术实现角度，主要需要解决自动化故障检测策略涉及的预设的故障检测规则和算法，系统可以自动检测故障，并根据预先设置的动作自动修复。

推动应急预案自动化策略。SRE 首先需推动应急预案的线上化，线上化应急预案有助于将标准化动作自动化，比如针对主机、应用服务、容器等最小运行对象单元的应急。线上化应急预案后，可推动预案的策略管理，支持多种策略组合、策略发布与组装、场景编排、通用场景等。接下来，SRE 应推动预案策略与自动化工具、接口和流程执行，关联场景绑定。

3.3 测试策略

3.3.1 连通性验证

通常在项目集成部署完成后，SRE 会联合开发团队对部署的环境先做连通性测试，目的是保证各个系统或模块之间的数据传输正常、请求和响应的格式正确，并且确保接口在不同条件下的稳定性和可靠性。主要可以分为以下四个步骤：确定测试目标、构建测试请求、单调用连通性测试、压测连通性测试、记录和报告测试结果

1. 确定测试目标

根据部署的架构图，确定要测试的系统范围以及调用通路，列出各个系统模块要测试的接口清单，包括系统模块名称、URL、IP 地址、端口号等信息，以及被测试接口的功能、参数、请求方法（如 GET、POST 等）和预期响应格式（如 JSON、XML）等。

2. 构建测试请求

根据接口的特点和需求选择合适的测试工具。一般选取最核心的业务功能创建连通性的测试请求，包括请求的 URL 或 IP 地址、请求方法、请求头、参数等。

3. 单调用连通性测试

利用选定的核心业务功能测试请求，对各系统模块接口发起单笔调用，检查返回的数据格式、数据内容是否满足预期结果。用此方法遍历各个系统模块之间的边界连通性。

4. 压测连通性测试

完成基本的单笔调用连通性测试后，为进一步评估接口的性能。利用压测工具，对核心业务场景用力发起生产100%的压力测试（目前实际使用过程中，有流量回放能力的公司，直接取核心业务的生产流量即可）。检查成功率，对错误响应进行核实，明确是功能参数问题还是性能问题，并予以改正。

5. 记录和报告测试结果

记录测试过程中的关键信息和结果，并生成测试报告。测试报告应包括测试目的、测试环境、测试脚本、测试结果、问题和建议等内容。

3.3.2 功能测试

SRE功能测试主要指在开发已完成交付测试，提交可交付代码版本的基础上，正式接入生产环境进行的测试，其功能测试主要包括版本功能测试（针对当次版本变更功能的测试，验证可用性、准确性）和回归测试（针对系统现有核心功能进行回归验证，确保非当次版本变更内容的准确性）。整体功能测试可以分为以下几个步骤：测试流程和计划制定、生产验证测试执行、当晚缺陷处理及次日故障跟踪、测试故障分析及评估、测试用例持续设计和维护。

1. 测试流程和计划制定

制定上线功能测试管理流程，贯穿测试过程中缺陷的提出、处理、复测、结束等各个阶段，按照工作任务规范职责，形成测试计划清单，及时高效地关键节点进行监控，从而保证上线验证测试的有效性，以提高系统上线的整体质量。

2. 生产验证测试执行

第一阶段：测试数据准备；

第二阶段：测试执行。将自动化用例通过平台一键式发起执行，在确认所有自动化计划发起完成之后，开始执行手工用用例。上线当晚上测试组根据上线范围进行自动化测试，如遇到异常情况影响测试进度，及时通知客户并协调相关人员处理。测试组严格执行上线测试流程，明确红线时间，完善联动升级和通报机制，确保上线测试过程顺畅。

3. 当晚缺陷处理及次日故障跟踪

生产当晚问题处理：主要分为四步，对于生产实时出现的问题，及时上报上线或变更管理员，协调相关资源协作排查问题，同时实时通报问题现象、进展、解决方法和最终处理结果。

测试报告编写：功能测试组编写系统验收测试报告，测试报告内容包括缺陷发现时间，测试进度，缺陷的准确描述，以及后续缺陷修复情况等。

测试进度通报：功能测试组对生产验证测试报告进行通报，若当晚未发现缺陷，则发送60分钟核心用例测试阶段性通报；若当晚发现缺陷，则发送问题发现通报并且整点时要发送问题进度通报。

测试缺陷处理：当测试发现缺陷，先做专业性的判断，是否为真缺陷还是业务理解问题，或者是电脑、手机、网络等客观因素导致的缺陷，若为真实缺陷，联合开发团队共同解决当晚的测试缺陷。

测试任务结束：如果当天上线依然存在遗留问题或故障，需要交接给次日保障人员持续跟踪、解决和通报。同时，生产上线当晚测试内容进行总结，内容主要包含准发布验收测试问题分析、生产验证问题分析、生产次日相关故障分析、生产验证测试分析，并输出上线生产验证情况总结。

1. 测试故障分析及评估

对生产上线当晚测试发现的问题或故障进行还原，并将故障现象和测试抓包数据反馈给保障值班人员。通过保障值班人员处理之后，将故障进行记录，包括故障现象、解决方式、故障分析等信息。以每周为维度统计上线次日故障，罗列出SRE相关故障总数，划分出重大故障、重要故障、一般故障的个数，进而细分未覆盖故障数与已覆盖故障数目，分析得出故障未覆盖原因以及问题归属系统，评估可进行优化覆盖用例数目。

经生产验收核心业务回归测试后，系统或平台未发现任何重大故障。这意味着系统在回归测试过程中成功通过了核心业务功能的验证，并且没有发现对系统关键功能或数据造成重大影响的故障。减少了重要故障的风险。重要故障指的是那些可能导致系统崩溃、数据丢失、功能不可用或对业务流程产生重大负面影响的故障。然而，需要注意的是，虽然没有发现重大故障，但仍可能存在一些较小的问题或不太常见的故障情况，因此仍需对系统进行持续的监控和改进。

2. 测试用例持续设计和维护

用例库基于业务量、投诉量、故障标准和企业考核标准将业务分成四个星级，星级越高，业务越重要。并定义二星级以上为核心业务，在生产回归测试中予以覆盖。对用户有

独立入口操作的页面定位为业务场景，作为用例建设的基础。通过调研用户使用系统的习惯，录制前台 UI 操作用例。并通过对核心参数进行多枚举值覆盖的方式，建设多路径覆盖的测试用例，对 UI 用例进行补充。每一轮生产验证测试复盘完后沉淀的改进方案和补充用例，都会纳入测试用例库。

自动化用例维护均在上线前 1 至 2 天内完成，可以确保系统上线前的最后一轮检查和修复。自动化用例维护涉及对已有的自动化测试脚本进行更新和修复。当系统发生变更或更新时，现有的自动化用例可能会因为页面结构或功能变动而失败。因此，在上线前，测试团队会仔细检查自动化用例的稳定性和有效性，并根据需要修复脚本中的问题。演练平台创建计划演练发起周期性测试，提高测试效率和准确率。

3.3.3 性能压测

性能测试包括压力测试与负载测试。全链接压力测试通过超负荷的负载条件来测试系统的稳定性和可靠性，以确定系统在极限负载下是否能够正常工作。负载测试则模拟用户访问系统，检测系统在不同负载下的响应时间和资源使用情况通过。在进行全链接压力测试时，定义性能指标和测试场景，设计合适的测试用例，并可使用 Jmeter 工具执行测试。通过监控系统的性能指标和记录测试结果，可以分析系统的性能瓶颈并提供优化建议。总结而言，性能测试是一种评估系统性能和稳定性的测试方法。通过模拟真实场景的用户行为和系统负载，它可以评估系统在不同负载下的性能指标，并发现性能问题和瓶颈。

1. 全链路压测工作流程制定

制定上线压力测试管理流程，贯穿测试过程中方案设计、缺陷的提出、处理、复测、结束、上线后的复盘改进等各个阶段，按照任务分类明确职责，且各任务对应到具体方案及工作清单，保证生产压测管理流程可以切实得到执行，从而保证生产压测的有效性，以提高系统上线的整体质量。

2. 全链路压测方案设计

针对某次项目压测，根据不容的业务场景，设计压测方案，主要确定实施范围和实施方法。

(1) 确定实施范围和压测指标

按照生产实际业务大类，可以分为查询类和受理类，因模拟受理类业务容易产生脏数据，工作量成本和代价较大（受理占比较大的一些场景可以考虑模拟受理类）。通常情况下我们的压测实施范围首先定位为查询类业务。针对生产系统如此多的查询类场景，既要

能够压测出生产瓶颈，同时又能够将影响面降到最低，较好的方法是选取 TOP 查询类的场景作为压测范围。压力测试无论是生产环境还是测试环境，都需执行相关的指标来衡量压测的效果，也便于后续统计评估同一个接口在各月的性能趋势。

(2) 确定实施方法

集中化场景压测：指单纯的查询类生产业务压测，跟真实生产业务场景存在一定的差异，为了更真实地拟合生产实际，考虑通过对单集群的压力测试来反映真实的生产场景。

混合性场景压测：针对爆发式增长的特殊业务场景，比如“充值送话费”活动，“充值”等，他们主要的特征是在某个特定时间段集中迸发而导致业务受理的瞬时高峰，因此，我们需要模拟混合业务场景对系统进行压测，测试系统的性能短板，并指导资源的最佳配置。

3. 全链路压测执行

(1) 执行生产验证测试必要条件

各系统都连接好，可以测试全流程的业务；生产环境测试结果能够真实反映系统运行状况；生产环境能测试一些入网环境无法测试的场景；生产验证测试能够发现一些由于环境差异导致的软件缺陷。

(2) 测试执行的工作项

其一是测试案例执行，其二是执行结果汇总，指第三方测试组对测试案例执行结果进行汇总；处理缺陷即生产测试验证支撑组修复测试过程中缺陷，提交测试用例让执行组进行回归测试；其四是测试管理工具的管理与维护；最后是协调与管控，对执行过程中遇到的问题协调多方进行处理与解决。

(3) 生产验证测试执行策略

生产验证测试执行就是根据测试案例编写阶段编写的生产环境测试用例来执行，不过在执行测试时有几个地方需要注意：仔细检查软件生产环境是否搭建成功；注意测试用例中的特殊说明；注意全面执行测试用例，每条用例至少执行一遍。执行测试用例时，要详细记录软件系统的实际输入输出，仔细对比实际输入和测试用例中的期望输入是否一致，不要放过一些偶然现象。

(4) 实时监控

对于生产压测主要监控内容有：云平台监控、日志监控、应用监控、CSF 接口监控、网络监控、服务治理监控、主机监控等等。

4. 生产当晚问题处理以及次日故障跟踪

对于生产实时出现的问题，及时上报上线或变更管理员，协调相关资源协作排查问题，同时实时通报问题现象、进展、解决方法和最终处理结果。

测试报告编写：测试组编写系统验收测试报告，测试报告内容包括测试结果，缺陷修复情况等。

测试报告核实：测试组对生产验证测试报告进行核实，主要核对测试执行状况、测试结果、缺陷修复情况。

测试报告发布：入网测试报告核实通过后，测试管理组组长发布入网测试报告。

生产准出确认：根据测试计划和测试报告分析各项指标作为评判依据，如案例覆盖率、入网测试通过率、关键业务测试通过率、端到端测试通过率等，测试结果是否满足准出条件。

最后是生产次日故障跟踪，主要是用于确定压测当晚的实施效果是否存在不足或遗漏，跟踪人员通过次日跟踪也能够更加熟悉生产，对保障生产稳定性以及全链路压测和生产环境的紧密型都有极好效果。

5. 压测总结报告分析与评估

对测试结果进行分析，根据测试目的和目标给出测试结论。通过对接口流量的业务成功率、系统成功率和生产耗时等重要指标进行详细的分析，这些采集到的数据被视为本次上线压测接口的基线，用于评估系统在压力下的表现。在性能回归测试的压测过程中，性能测试组成员会将实际的接口耗时、业务和系统成功率与基线进行比较。这样的比较能够清楚地显示出接口性能的升降幅度，进而判断出系统可能存在的性能瓶颈和潜在问题。

同时，性能测试组成员也会积极抛出问题并持续追踪这些问题。在追踪的过程中，他们会对各个问题进行深入的分析，并提出切实可行的建议和优化方案。这些建议和优化方案的目的是为了解决相应的问题，并最大程度地提升系统的性能和稳定性。通过持续的压测和性能优化工作，性能测试组成员能够不断改进系统的性能，并确保系统在高负载情况下仍能够正常运行。为业务的顺利进行提供了坚实的基础，也保证了用户能够获得更好的体验和服务质量。

6. 全链路压测自动化与维护

基于开源的 Jmeter 工具做定制开发，形成全链路压测自动化工具。其架构是控制台、压力生成器、分析器，主要作用是配置测试场景，通知代理器进行数据初始化或清理，搜集测试过程中被测系统各个环节的性能数据，并根据要求模拟一定数量的虚拟用户对被测系统发送业务请求，实现对被测系统的压力测试，其中一个虚拟用户对应一个业务并发；

将会发送测试的过程及结果数据信息给控制台进行采集汇总。最终分析并展示测试结果，同时对测试结果进行保存。

每月例行对生产全链路压测结果、测试账号、测试数据和测试场景用例进行检查，对场景变化的用例进行及时维护，对出现问题的测试账号、测试数据等及时地修复，以确保生产压测的准确性和安全性，确保各种数据可持续、可继承、可追溯。

3.3.4 数据迁移

1. 数据完整性和准确性测试

由 DBA 采用数据库稽核工具，支持同构/异构数据库间的数据稽核比对，包括 count 稽核和表全字段稽核，通过监控可以快速定位两端的差异，保证两端数据库同步数据的完整性和准确性；并且支持割接后增量数据回传，保障割接后发生故障能够快速回切恢复。

2. 可用性测试

由 SRE 进行迁移可用性测试，目的是确保数据在迁移后能够正确地被使用。测试所有核心及非核心的功能，并确保它们达到预期的结果。另外，在迁移完成后的一段时间内（如每日早晨的开门测），针对之前已经测试过的功能进行再次回归测试，以确保在迁移后的初期没有引入新的错误。

3. 性能测试

由 SRE 进行迁移性能测试，测试系统的在迁移后的性能是否与之前相同。同全链路性能测试的方案，按步骤进行验证。

3.4 变更评审

变更评审主要是为了降低系统变更投产带来的风险，并让变更如期交付业务。变更评审中，不同的 SRE 组织会在系统交付生命周期的不同阶段建立相对应的变更评审机制，比如项目立项环节的可用性评审、技术或部署架构可用性评审，设计阶段的非功能性、可运维性评审，上线环节的 CAB 评审等。

3.4.1 稳定性架构设计评估

SRE 组织为了推动稳定性架构管理，建议在整个技术线层面建立跨团队的技术架构管理组织，负责制定稳定性架构管理规范、技术组件规范，以及相应的技术管理与技术评审

流程。SRE 应重点从高可用、故障恢复、可扩展性、数据完整性、部署环境角度推进相关评估工作。

高可用是运维管理的一条底线保障要求，运维主要工作是消灭单点风险，提升系统韧性，比如数据库中提到的主备、主从、分布式，数据中心的两地三中心、分布式多活，以及将一个应用系统同一个服务组件部署在多个数据中心机房、不同物理机的多个虚拟机上、为应用的负载均衡提供网络硬件或软件负载均衡器、提供具备高可用架构的消息中间件等 PaaS 云服务等。为了更好地推进评估工作，SRE 需要提前提供架构高可用的规范，制定通用组件、信息系统架构高可用参考模式，将高可用要求更早地落地在系统设计过程中。

故障恢复可借鉴最佳实践、具体信息系统的特点等，制定相应的故障恢复能力要求。在应用系统层面，需关注应用拆分、服务或系统交互解耦、服务无状态、减少总线节点服务依赖、增加异步访问机制、多层次的缓存、数据库优化、限流与削峰机制、基础设计快速扩容等。在基础设施层面，具备可恢复性的基础设施环境必须能够高效地从自然灾害或人为灾难中迅速恢复。这意味着需全面考虑各类风险，涵盖自然灾害、技术故障、人为失误及网络恶意攻击等，并采取相应的预防措施以最大限度地降低潜在损失。具体措施至少应包括数据备份、系统冗余、快速恢复服务以及对关键信息系统的严密保护。

技术架构的可扩展性是指在不影响现有系统功能和性能的前提下，系统能够扩展或增强其功能的能力。通常涉及对系统设计、架构和模块化的考虑，以便于未来的扩展和升级。这包括将系统拆分为多个独立的子系统或模块，每个模块负责特定的功能和业务逻辑，降低整个系统的复杂度与模块间的耦合度，提高扩展性；支持横向和纵向的扩展能力，通过增加服务器数量或提高每个服务器的处理能力，来提升整个系统的处理能力，或通过升级单个服务器或组件的硬件来提高其处理能力；系统支持弹性伸缩，即根据系统的负载情况自动调整计算和存储资源，以实现系统的动态扩展和缩减；减少对总线节点服务的依赖，将多节点组成的逻辑交互改为端对端访问方式，以减少交易影响因素，避免单节点性能问题影响其他应用系统；增加异步访问机制，因为同步机制在性能出现问题时，会在短时间内耗尽最大连接数，即使这个最大并发数是正常情况下的 10 倍。将同步连接改为异步通信或引入消息队列都是解决上述问题的方案；支持多层次的缓存，可以从前端、应用内部、数据库等层面建立缓存。

数据完整性是运维保障的底线要求，持久化数据的生命周期通常会比系统与硬件的生命周期长很多，很多新系统上线或架构调整都考虑数据迁移工作。同时，还要关注一些复杂性数据处理，比如：批次、清算、对账等操作，这些操作极易受数据问题影响，运维侧

需要关注数据处理的异常中断原因定位、哪些环节是可以应急中断、中断后是否支持多次重试、与第三方系统约定数据不一致时以哪方为基准等应急处置机制。

选择合适的部署环境需要考虑多种因素，包括应用程序的特性、性能要求、可扩展性、可靠性、安全性、服务提供商的支持和成本效益等。不同的因素将对部署环境的选择产生重要影响，比如某些应用程序可能需要大量的内存和计算资源，有些需要选择能够提供高吞吐量和高数据处理能力的平台服务，有些需要选择云服务使应用程序能够随着需求的变化而扩展，有些需要选择高度稳定与安全性的基础设施环境。

3.4.2 非功能性技术评估

运维的非功能性设计是主动应对可运维性问题的切入点，直接决定系统在生产环境的成本与收益，甚至决定系统生命周期的长短。以下罗列运维侧需要推动的非功能设计。

系统运行状况可观测。云原生提出可观测的监控指标、日志、链路三要素同样适用于传统以主机为代表的技术架构。运维是在一个黑盒子的成品上进行监控、日志、链路完善，像 NPM、APM、BPM 等是运维侧发起的一些解决方案。从非功能设计看可观测，需要运维前移，推动必要的监控、日志、链路相关的研发规范，提升主动上报监控性能指标数据的能力，优化日志的可读性，并提供必要的基础设施服务支持，比如支持系统监控数据上报、日志采集分析等。

故障隔离与服务降级。故障隔离和服务降级的目的是以牺牲部分业务功能或者牺牲部分客户业务为代价，保障更关键的业务或客户群体服务质量，是防止连锁性故障蔓延的方法。在设计中，运维侧需要从系统或业务角度，梳理应用系统所调用的各个服务组件，对各个服务组件出现故障时的假设，及应对措施。

性能评估。性能容量管理主要基于响应时间（系统，功能，或服务组件完成一次外部请求处理所需的时间）、吞吐率（在指定时间内能够处理的最大请求量）、负载（服务组件当前负荷情况）、效率（性能除以资源，比如 QPS=并发量/平均响应时间）、可扩展性（垂直或水平扩容的能力）等黄金指标开展。性能问题对稳定性的挑战不仅仅是单组件不可用，更大挑战是某个组件性能问题不断扩散到别的组件，导致大规模的故障。性能问题极易引发复杂的异常问题，SRE 需要加强相应的技术评估。

终端版本向下兼容。移动化后终端版本的管理越来越重要，在架构上一方面需要尽量保证升级后的版本要向下兼容正在流通的低版本的可用性；另一方面要对流通的版本进行收敛管理，支持多种在线更新机制。

基于基础平台运行。无论是基础设施平台，还是PaaS层的应用平台，或持续交付工具链，系统均需要尽量与公司现有基础平台对接，避免引入新的技术栈。

可配置而非硬编码。在应急时，发现有些“参数”硬编码在程序中，无法快速调整，需要推动配置管理。另外，IP的DNS域名改造也是可配置的一个方向，减少人工在后台修改。

日志规范化。日志分析是认识应用系统的关键手段，SRE可关注以下几点：一是“存储”，因为行业或企业内部有保存日志数据的要求，需要一个成本可控、可横向扩容、支持海量日志数据的管理平台；二是“查询”，日志是感知线上系统运行状况，了解代码级问题的一个窗口，在出现业务问题、故障应急等场景下，可以利用日志查询问题，需要建立实时的数据采集、高效的日志检索能力；三是“监控”，基于日志分析关键字、正则检索、模式匹配等方式，能够提供监控能力；四是“分析”，对日志的非结构化数据进行加工处理，生成结构化数据，用于运行数据分析，实现异常发现、故障定位、性能分析、容量评估等分析工作。

测试方案完备。在测试方案评审中，SRE应重点围绕性能、容量、压力、稳定性架构等测试方案的评审。评审的角度包括，测试用例的描述是否清晰，测试用例的执行结果是否符合要求，测试结论涉及遗留问题的应对措施等。

3.4.3 变更保障准备工作评估

变更带来系统稳定性风险。从生产变更故障引发率看，来自变更的故障遥遥领先其他因素引发的故障，变更后通常是运维组织最为繁忙的时段。不管是大的软件基线，还是小到一个功能迭代，甚至一个参数或配置的调整，也可能引发一个重大故障。变更带来的风险点很多，有设计上的程序缺陷类的问题，也有管理上的版本管理的问题，也有操作层面的执行问题，也可能是协作层面上下游系统沟通不畅引发的相互影响的问题等，解决变更带来的风险问题是一个极为复杂的系统性工作。SRE可考虑从以下工作中加强变更保障的评估。

监控和告警是否就绪。建立完善的监控覆盖面，包括基础设施、平台软件、数据库、中间件、操作系统、性能等技术指标监控，与特定系统相关的业务功能、客户体验指标监控，以及与系统上线后重要业务功能首笔出现的监控。

运行健康检查方案是否就绪。需建立完善的监控和预警体系是保障系统可运维性的重要手段，能够实时监控系统的各项指标，及时发现问题并进行预警，以便及时处理问题。

应急预案是否就绪。故障应急工具是否就绪：系统出现故障时，需要能够快速恢复并进行处理，避免故障扩大化。

技术运营工作是否就绪。系统上线后，需要进行全面的运营管理工作，包括用户服务、首笔业务发生的跟踪、异常数据或逻辑问题的处理、安全管理等方面的内容，以确保系统能够稳定、高效地运行。

文档是否完善。根据组织软件交付的协同机制，明确新系统上线的文档交付清单，比如：项目与需求、技术架构图、测试结论（含压力测试）、安装部署（与环境相关）、应用/业务监控、首笔功能监控及保障、重要功能清单、重要配置与参数、运行效能评估等文档。

知识库是否就绪。建立完善的知识库是保证系统可运维性的重要基础，需要对系统的各项功能和操作进行详细的描述，并建立知识库，以便在需要时能够快速查找相关信息。

3.4.4 新系统或新业务上线保障评估

新系统或新业务上线是从 0 到 1 的过程，具体的工作通常包括：上线准备工作、制定技术方案、评估测试管控、上线文档准备、风险评估、安全保障措施、运维监控准备、上线过程协同、上线发布、系统验收、上线试运行等。

新系统或新业务上线给企业带来机会与挑战。一方面，作为项目重要里程碑，新系统将为企业业务发展或运营管理助力，通常业务需求方会投入大量精力在上线后的运营推广工作，以期望更好地给业务及客户带来价值；另一方面，新系统上线带来众多的不确定性因素，需要对不确性因素进行管理。

不确定性包括：新业务流程的合规性、数据安全、隐私安全等问题；新业务对现有上下游系统在业务及性能层面的影响；新系统在设计上是否满足业务活动的性能、容量要求；配套的稳定性相关的监控、日志、应急、数据备份等非功能性需求是否就绪；功能已知缺陷的影响是否评估；发布上线、环境部署、下线方案是否就绪；对系统业务运行情况的观测能力等

新系统或新业务自身的业务风险可能会对存量业务流程产生影响。在加强风险评估工作上，SRE 应加强以下风险的评估：

新系统业务带来的风险防范、监管合规、数据安全、隐私安全等问题；

新系统业务流程可用性、终端体验、数据准确性等风险；

新系统业务对现有上下游系统在容量、性能方面的影响，以及上下游配合改造对原有业务带来的影响；

新系统资源、架构、重要功能是否满足业务期望，以及接下来业务活动的性能、容量要求；

系统压力测试、容量评估方案、功能遗留缺陷等是否评估到位等；

另外，SRE 还应主动推动以下工作：

标准化先行，建立业务与技术层面的合规、风险、隐私、安全等方面的管理要求，并辅助相关技术检测手段。

业务系统非功能性需求的建设，比如系统回退、版本切换、灰度发布、终端体验感知等配套功能的实现。

业务系统技术运营需求的建设，比如对首笔业务感知、业务流水监测、关联系统的技术运营监测等。

系统性能与容量管理，包括相关评估指标、基线容量设计、指标数据加工、容量评估分析报告、压力测试等工具建设。

同时，针对可能突增的业务模式，SRE 需要建立限流、削峰机制。架构优化上，需要前端系统做交易并发控制开关，必要时进行前端限流、削峰，以及后端服务降级，通过一些前端交互设计减少客户的体验影响，重点保障系统的核心服务。

SRE 通过聚焦新系统或新业务上线阶段，做好运维工作左移，提前做好资源交付能力建设提高系统上线速度，利用运维平台能力建设帮助业务研发专注业务逻辑，提前参与到架构及非功能性需求的研发与验收，能够让系统上线后融入平台化管理模式。

4 变更管理

根据业界经验及 Google 多本 SRE 书籍的论述，约 70% 左右故障是由变更引起的。然而，业务发展中的变更是不可避免的，因此，通过变更管理来控制变更的风险，尽可能降低由变更导致的故障率和影响范围，是提升系统稳定性的一条可持续且高 ROI 的路径，也是每个 SRE 团队面临的关键课题。

4.1 发布管理与变更管理关系阐述

发布是软件工程术语，变更是系统工程术语，在各自领域，这两个概念常常相互包含对方。SRE 关注的核心问题是业务应用的稳定性，SRE 所涉及的“发布”大多特指代码变更，

而“变更”的范围则包含了硬件变更（网络设备、主机、存储等一切 IaaS 硬件）、系统软件变更（各类 OS 等）、应用软件变更（与 SRE 相关）、安全设施（机房配套设施及安全防护设备）等一切对 IT 系统调整的动作。因此本白皮书将“发布”定义为“变更”的一部分，特指版本发布。

发布管理是指整个软件开发周期中，软件从开发阶段转移到生产环境的过程。这个过程包括规划、调度、控制以及测试软件版本的构建、部署和交付。发布管理的重点是确保软件的新版本能够顺利、有效地发布，同时减少对现有系统运行的影响。在 SRE 领域，发布管理通常强调自动化部署流程，以减少人为错误，提高发布速度和可靠性。

变更管理则更加宽泛，涵盖了所有对生产环境可能造成影响的活动，包括软件更新、运行环境、配置修改等。变更管理的目标是确保所有变更都是经过计划、测试和批准的，从而最小化任何负面影响，确保系统的稳定性和可靠性。变更管理流程通常包括变更的申请、评估、批准和监控。

关于两者的关系：

发布管理是变更管理的一个子集：在 SRE 框架下，所有的软件发布都应视为变更，因此需要遵循变更管理的流程和原则。这意味着每一个软件发布都需要经过严格的审查和批准过程，以确保它不会对生产环境造成不可预见的负面影响。

相互支持的目标：尽管两者的焦点不同，但发布管理和变更管理的最终目标都是提高服务的稳定性和用户满意度。通过精细的发布管理，可以有效地推出新功能和修复，同时通过严格的变更管理可以确保这些发布不会导致服务中断或性能下降。

共享的实践和工具：在实际操作中，SRE 团队会使用各种工具和实践来支持这两个过程，例如版本控制系统、自动化测试、持续集成和持续部署（CI/CD），以及监控和警报系统。这些工具和实践不仅支持快速安全的代码发布，也帮助团队管理复杂的变更。

发布（Release），指将经过测试的、验证正确的软件版本（包括但不限于客户端、服务端程序，相关配置文件、数据等）导入到目的变更地点的行为。通过规范有效的发布管理流程，可以使发布实施更快，成本更优，风险更小，保障业务的稳定性和可用性。

发布管理（Release Management）的目标是确保所有线上生产环境的版本发布行为具备可控性和可追溯性，具体包括但不限于以下几个方面：

(1) 合理定义发布计划，以确保对业务影响最小化，并降低风险。

(2) 确保发布到生产环境的版本统一，并经过必要的版本质量保证过程。

(3) 确保包含安装、测试、验证和回退等发布过程中的各个环节操作的规范性。

(4) 确保发布相关的利益干系人（如 SRE/研发/测试/产品/运营团队和用户等）的行动一致性。

在国内业界的实践中，不同职责的团队所肩负的变更任务会不一样，例如，基础架构部门，云平台部门 SRE 更多的是进行基础架构层面的变更管理，而业务 SRE 更多负责发布管理，所以以下的内容侧重点，差异性也会因此而出现。

4.2 变更体系设计

4.2.1 变更体系设计原则

SRE 关注的核心是业务应用系统的稳定性，SRE 描述任何问题都是从软件工程的视角出发，围绕业务系统做出某方面调整改造，或者开发了某些管理系统。Google SRE 针对变更管理强调自动化操作的技术实践，包括渐进式发布、过程和结果的反馈/检测、回退等。

4.1 中明确了在 SRE 视角下变更包含发布，发布特指代码变更，以及发布与变更在工程层面应该共享实践及工具，因此 SRE 变更体系设计的原则包括两点：严谨的变更流程以及通用的工程化体系。

4.2.2 变更及发布流程设计

4.2.2.1 变更发布准备

由于变更/发布涉及到产品、开发、测试和运营等多个职能团队的参与，因此在正式实施操作之前，必须进行一系列充分的事前准备工作。作为线上环境的第一责任人，SRE 在整个发布过程中担负着“总协调”的角色。在发布前，SRE 需要进行以下关键任务：

- 审核变更发布产品功能和服务的可靠性。
- 评估变更发布的风险，并制定相应的应对策略。
- 选择合适的变更发布方式和方案，以最大程度降低影响。
- 协调各职能团队完成相应的准备工作，确保协同合作。
- 主导制定具有可操作性的发布流程清单，确保流程的顺畅执行。
- 预先制定针对各种可能的变更发布异常的快速执行应急预案，以确保对突发情况的迅速响应。

(1) 变更风险评估

变更风险评估是指在进行软件发布或系统变更前，对可能存在的风险进行评估和分析，以便在变更过程中及时采取措施来降低风险，确保变更的稳定性和可靠性。在业务全生命周期中，产品希望在用户侧得到快速验证和反馈，这就要求每次的功能迭代都能快速发布到线上环境，而 SRE 围绕业务的性能和稳定性，更加关注线上环境的持续稳定运行，变更带来改变，改变带来风险，变更与稳定在某种程度上存在冲突，因此，针对线上环境的每一次变更/发布（包括常规发布和紧急发布），引入风险评估环节变得尤为重要。由于变更概念较为宽泛，为使读者深入理解后续环节，本章节后续内容将以变更中最为重要的版本发布（代码变更）为例，介绍风险评估之后的环节。

1. 版本发布容量风险评估

通常情况下，新功能的发布往往会带来临时用户访问量的急剧增长。因此，SRE 在执行发布之前需要做好容量风险的评估工作。为了做到这一点，一方面可以依据发布前压力测试的数据，结合运营数据模型（包括发布过程中和之后的流量及增速预测），提前准备足够的发布容量冗余，并尽量避开在业务高峰期进行发布；另一方面，可以选择将首次发布限制在某一单独区域的用户，通过该区域的容量增长数据的分析，建立后续大规模发布时的信心和预期。

2. 版本发布依赖风险评估

发布依赖风险评估主要包括基础设施、程序引用、下游依赖以及上游调用几个关键方面。在基础设施方面，涵盖计算、存储、网络等要素，需要确保相关基础设施负责人积极参与发布过程，进行全面的风险评估和制定相应预案。在程序引用方面，包含第三方类库、代码、数据等，必须避免偶发性故障、程序 Bug、系统错误或安全问题对发布正确执行造成影响。对于下游依赖，应设定合理的超时时间，提前有效预估和沟通下游流量，防止对下游系统造成过度负担，确保强依赖变成弱依赖，同时高优先级应用不得依赖低优先级应用。至于上游调用，需要明确各调用方的身份，根据不同优先级制定合适的分配、限流和熔断策略，以确保整体系统的稳定性和可用性。

3. 版本发布异常风险评估

并非每次发布都能按计划如期完成，作为 SRE，我们需在发布前进行细致的异常风险评估，主要涉及发布延时、发布故障以及发布回退（包括程序、配置、数据回退）这三个方面，以应对潜在的不确定性。发布延时指的是由于停机发布实际所造成的停机时长超出原计划时长，或者由于热更新不停机发布新功能用户无法使用，从而对业务可用性产生不

利影响。发布故障指的是在发布过程中人为操作不当或不可预知原因而引发的故障，例如文件、配置发布出错，主机、网络设备故障等。至于发布回退，指的是在发布过程中发生无法正常推进的情况，导致线上环境的用户数据受到污染，必须将业务的程序、配置和数据回退到发布前的某一时刻。为确保系统的稳定性，我们必须在发布前充分评估这些异常风险，以准备好应对可能的挑战。

4. 其他风险

在新功能上线初期，面临着业务竞争、无法控制的用户行为等因素引发的诸多风险，其中包括大量的 DDoS 攻击、客户端滥用行为等。这种情况可能导致服务过载，甚至新功能的崩溃等问题。因此，SRE 在新功能发布上线之前需要通过有效的压力测试来规划业务应对这些挑战。最佳的做法是为新服务设计具备“功能开关”的机制，并灵活运用灰度和阶段性发布的方法。

通过详尽的压力测试，可以模拟和评估在真实环境中可能发生的各种情景，从而有效预测系统在高负载和攻击情况下的表现。同时，引入“功能开关”机制使得在遇到问题时可以迅速关闭或切换功能，以减轻系统负担。此外，采用灰度发布和阶段性发布的策略，可以逐步将新功能引入生产环境，降低潜在问题对整个系统的冲击，使问题的范围得到有效控制。

通过这些规划和策略，SRE 能够更好地保障新功能的上线稳定性，降低因各类风险而导致的服务中断和性能问题的风险。

效果评估：

SRE 理念倡导拥抱风险，对于发布服务来讲，不可能做到 0 风险，发布的可靠性并不是一味地追求越高越好，因为极高的可靠性需要付出巨大的成本，与实际的回报不成正比。在错误预算（Error Budget）的允许范围内，根据当次发布服务的具体情况，评估风险容忍度，并预留处理风险突发情况的时间。此外，在尽可能减少计划外停机时间（unplanned downtime）的基础上，寻找质量、效率、成本和安全之间的平衡点，是一种有效的发布风险评估方法。

(2) 发布方式确定

发布方式是软件开发和维护过程中采用的一系列策略和方法，旨在确保应用程序的性能、可靠性和可用性。SRE（Site Reliability Engineering）发布方式的核心目标是最小化应用程序停机时间和风险，并确保新版本应用程序经过充分测试和验证，具备良好的性能和可靠性。

典型的SRE发布方式涵盖蓝绿部署、金丝雀发布、滚动发布和灰度发布。此外，在发布过程中，还存在着多种具体的部署方式，如整包部署、增量部署和容器化部署，它们主要区别在于所采用的技术实现方式。

零风险发布是SRE不断追求和实践的目标，为此有多种发布方式可供选择，以满足不同的发布需求。在选择SRE发布方式时，主要考虑以下几个方面：

1. 基于应用程序的复杂性

在面对高复杂度的应用程序时，可能需要采用蓝绿部署或滚动发布等发布策略，以确保整体应用程序的连续可用性。蓝绿部署允许同时运行两个版本的应用程序，其中一个版本是当前稳定版本，而另一个版本则是待验证的新版本。经过充分的测试和验证后，可以逐步将流量切换至新版本，直至完全替代旧版本。相较之下，滚动发布则以逐步更新应用程序不同部分的方式来保障整个应用程序的可用性。

2. 基于可靠性需求

如果应用程序对可靠性要求很高，可能需要使用蓝绿部署、金丝雀发布或灰度发布等方式，以确保新版本应用程序的性能和可靠性得到充分测试和验证。金丝雀发布可以逐步增加新版本应用程序的流量，以测试其性能和可靠性。如果出现问题，可以立即回滚到旧版本。灰度发布可以将新版本应用程序的流量引导到一小部分用户，以测试其性能和可靠性。如果出现问题，可以立即回滚到旧版本。

当应用程序对可靠性的要求极高时，常常需要考虑采用蓝绿部署、金丝雀发布或灰度发布等发布策略，以确保新版本应用程序的性能和可靠性是经过充分测试和验证的。金丝雀发布允许逐步增加新版本应用程序的流量，从而评估其性能和可靠性。在发现问题的情况下，能够迅速回滚至旧版本，确保系统稳定性。另一方面，灰度发布则能够将新版本应用程序的流量引导至一小部分用户，以进行性能和可靠性测试。同样，在发现问题时，也能够立即回滚至旧版本，以最大程度地降低潜在影响。

3. 基于SRE团队能力和经验

在团队经验不足或缺乏自动化工具支持的情况下，推荐选择更为简单的发布方式，例如滚动发布。滚动发布以逐步更新应用程序不同部分的方式确保整个应用程序持续可用。相较而言，当团队拥有丰富经验和自动化工具支持时，可以考虑采用更为复杂的发布策略，如蓝绿部署、金丝雀发布或灰度发布。这些高级发布方式能够更精细地控制新版本的推出，从而最大程度地保障系统的可用性和稳定性。

4. 基于时间压力

在面临时间压力时，选择合适的发布方式至关重要，以确保应用程序更新的及时性不以牺牲质量为代价。滚动发布和灰度发布是在此类情况下推荐的策略，因为它们能够提供快速且连续的部署能力。

滚动发布允许按计划逐步替换应用程序的旧版本，这种逐步更新的方法有助于维持应用程序的持续可用性，同时允许在必要时快速回滚。

灰度发布则通过将新版本引入给一小部分用户，以实现对新版本性能和可靠性的快速测试。这不仅加快了反馈获取的过程，还降低了潜在风险，因为任何问题都可以在影响扩大之前迅速被识别和解决。

选择这些策略有助于团队在紧迫的时间框架内有效地推进项目，同时通过及时的反馈和验证确保新版本的性能和可靠性。这种灵活性在处理紧急部署或快速迭代时尤为重要，确保了即使在时间压力下，也能保持应用程序的稳定性和用户满意度。

效果评估：

SRE 发布方式的评估标准主要涵盖故障率、可重复性、回滚能力、时间效率以及用户满意度等多个方面。通过对这些关键指标的综合考量，我们能够全面评估不同发布方式的优劣，并选择最为适宜的发布策略来有效地管理应用程序的更新和维护。

这一系统化的评估方法有助于深入理解各种发布方式在实践中的表现，从而为决策者提供明智的选择。通过对 SRE 发布方式的精准度量，我们能够更加精细地调整发布流程，以满足应用程序更新的需求，同时最大程度地确保系统的稳定性和性能。

(3) 发布协调

发布协调是 SRE 团队在软件发布全过程中协调各个团队、统筹推进各发布环节，以确保发布过程的顺利进行。每次业务上线发布都牵涉到产品、开发、测试、运营、市场等多个职能团队，其成功依赖于这些团队的紧密合作。由于 SRE 团队在日常工作中积累了丰富的产品知识、技术架构能力，并建立了良好的跨团队关系，使其成为最适宜协调各团队的角色。在发布前、发布中和发布后，SRE 以全局视角统筹整个流程，引导团队构建可靠、可扩展、稳定且性能卓越的产品。

(1) 制定详细的发布计划，与开发团队和其他相关团队协商，明确发布时间、版本号等信息。借助功能发布上线日期，逆向规划代码提交截止时间、测试打包、发布前准备和发布停机时长等排期，并与所有相关团队成员确认方案和排期，输出发布电子流和工单。

(2) 审核发布产品的新功能和内部服务，确保它们与原有业务的可靠性标准一致，并提供提升可靠性的具体建议。对于发布的业务模块，通过 SRE 技术手段检测和预防可能增加停机时间或妨碍性能目标的单点故障，制定高可用性和快速灾难恢复方案。

(3) 在发布过程中作为多个团队之间的联系人，考虑当前发布的影响范围。可以迅速组建一个虚拟团队，联系相关职能人员，包括但不限于产品、运营、开发、测试人员，通过线上线下会议的沟通方式，持续检查发布版本所涉及的各团队工作进展。

(4) 跟进发布所需任务的进度，负责处理发布过程中的所有技术相关问题，包括但不限于构建发布自服务模型的平台工程，提升自动化程度，解决由资源不足引发的容量问题，以及通过服务降级处理过载的技术预案等。

(5) 作为发布全流程的“守门人”，决定所有发布项是否都是“安全的”，确保发布过程进入可控状态，降低发布失败率，缩短发布时间周期。

效果评估：

SRE 团队通过与其他团队（如开发、测试团队等）的协调合作，以确保发布过程的顺利高效，并最大程度地减少对用户的影响。对其效果进行评估可从以下几个方面进行衡量：

发布效率：通过协调发布流程，降低了团队间的沟通成本，缩短了相互等待的时间，提升了发布效率。

发布稳定性：SRE 团队与其他团队协作，始终将业务稳定性视为首要衡量指标，确保发布版本在生产环境中稳定运行，减少故障和停机时间。

发布质量：通过协调发布流程，SRE 团队能够保证发布版本的质量，涵盖代码、文档和测试等多个方面的质量保证。

用户满意度：通过协调发布工作，最大程度地减少对用户的负面影响，包括减少停机时间、降低用户投诉数量，提升用户体验，从而提高用户的满意度水平。

(4) 发布执行清单

发布执行清单是指在正式操作之前由业务 SRE 团队制定的详细清单，其中包含了软件或系统发布的所有步骤和操作。其主要目的在于确保发布操作流程的顺利和成功。该清单通常由业务 SRE 团队编写，并在发布过程中按照既定的操作方案进行实施。完善且准确的发布执行清单能够有效保障发布过程中避免由于人为操作导致的发布步骤缺失或执行不准确而引发的系统中断。

(1) 发布执行清单包含但不限于以下要素：

a. 各步骤的具体执行内容及预估耗时。详细的执行命令能够降低发布过程中人员操作失误的可能性，而预估耗时则有助于SRE发现执行步骤是否存在异常，并评估发布总耗时。

b. 执行成功确认手段。清晰地执行成功确认手段有助于在发布过程中尽早发现系统中断，并增强发布过程中的信心，例如，通过观测请求量指标是否恢复到7天前的环比水平。

c. 操作回滚步骤。清单必须包括回滚步骤，以便应对发布过程中可能出现的问题和故障。

d. 测试与验证步骤，至少要包含新功能的测试与验证，以及用户关键链路的基础功能的测试。确保该次发布新增加的功能和关键链路的验证符合预期

e. 发布操作人员协同，对于跨团队或多实施人员的发布，需要相关的实施人员要做到知会和协同，尤其是对于有前后顺序的步骤，需要前置步骤实施完毕并且结果符合预期后才能进行后置步骤的实施。

(2) 发布执行清单按执行顺序分为发布前、发布中、发布后三个阶段。发布前的检查清单应关注于检测评估发布前置环境是否符合发布条件，发布中的执行检查清单则包括具体的发布变更操作，发布后的执行检查清单主要用于验证此次发布是否成功。

(3) 利用工具化模板将发布流程清单线上录入，通过可视化的图形界面进行任务流程编排和执行。采用插件对接与开发的方式，实现与发布操作所需工具和平台的调用，从而实现发布流程清单的跨系统调度自动化。这样可以减少发布过程中的人为切换等待时间，并降低人工操作的失误。

(4) 发布清单应在团队中共享，以避免人员单点问题。在发布过程中，应当在发布清单中记录发布执行起止时间、实施内容、实施人员等关键信息，作为后续发布流程回顾及优化的依据。

效果评估：

SRE发布执行清单是一个旨在规范发布过程的详细清单，其目标在于确保发布过程的高效性和流畅性，以最小化故障和停机时间。对SRE发布清单的效果评估可以从以下几个方面考虑：

发布速度：SRE发布清单提供了一个标准化的发布流程，有助于团队更迅速地将新功能和问题修复版本发布到生产环境中，从而显著提高整体发布效率。

发布稳定性：SRE发布清单中涵盖了各种验证步骤，可在发布前进行充分的测试和验证，以确保发布的版本在生产环境中运行稳定。这一方法大幅减少了故障和停机时间，从而增强了系统的整体稳定性。

发布质量：SRE发布清单中包括回归测试、性能测试等步骤，确保发布版本的高质量。通过这些措施，不仅提高了系统的整体质量，还提升了用户的满意度。

(5) 应急预案制定

发布应急预案是一项用于指导发布过程中处理紧急情况和发布事故的战略性计划。通常由专业的业务SRE团队编写并在发布过程中备用，旨在确保发布的系统在任何情况下都能保持安全和稳定。应急预案在发布工程中扮演着业务连续性管理的关键角色。即便在充分准备和严格执行的情况下，每次发布变更仍然存在不可预知或无法评估的风险，SRE通常通过应急预案来应对这些挑战。在系统非预期中断的情况下，SRE可能面临巨大的压力，并且在短时间内难以全面了解业务系统的上下游状态。这种情况下，临时决策的恢复方案可能因未全面考虑而引入新问题。因此，为避免引发更严重的问题，SRE有必要在发布前充分准备应急预案。

制定发布应急预案的整体原则应遵循以下几个步骤：

(1) 明确应急预案范围和目标：详细列出此次发布应急预案的覆盖范围和应对目标，包括需要关注的模块、业务指标、观测手段、预案启动条件、实施方案以及实施后的预期表现。这样有助于协助SRE在发布过程中做出准确的判断和应急操作。

(2) 发布过程和完成后的监测：在发布过程和完成后，SRE应通过监测指标来判断是否发生了非预期状况，并确定是否需要启动应急预案，以提前发现系统中断。

(3) 与发布协调步骤的联动：当出现超出预期的情况时，应快速组建故障团队，制定团队人员清单和分工计划，以缩短系统中断的恢复时间。

SRE发布的应急预案制定步骤如下：

(4) 确定应急预案的范围和目标：明确此次发布应急预案的覆盖范围和应对目标，包括哪些紧急情况需要应对，以及应急预案的主要目标是什么。同时，对可能发生的紧急情况进行分类和分级，制定相应的应急响应计划，包括行动步骤、通信流程、资源调配等。

(5) 组建应急响应团队的成员和职责：确定应急响应团队的成员和职责，包括开发团队、测试团队、运维团队等。建立紧急联系方式，包括电话、邮件、即时通讯等，以确保在紧急情况下能够及时联系。同时，需要定期组织团队审查和更新应急预案，以保持其有效性和适应性。

(6) 制定详细的应急预案内容：包含但不限于：

a. 预发布测试：在正式发布之前进行预发布测试，确保版本的稳定性和兼容性。

b. 限流措施预案：在版本发布期间采取限流措施，以避免系统过载和崩溃。

c. 监控和报警预案：加强监控和报警机制，及时检测和响应系统异常和故障。

d. 回滚计划预案：如果版本发布出现问题，立即启动回滚计划，将系统恢复到之前的稳定状态。

(7) 应急预案演练：根据应急预案制定应急演练计划，并进行演练，评估应急预案的有效性和实用性。

效果评估：

效果评估方面，可以从以下几个方面来衡量：

(8) 有效性：应急预案是否能够有效地应对发布事故，包括是否能够快速响应、准确识别问题、及时采取措施、有效恢复等。

(9) 可操作性：应急预案是否易于操作和使用，包括是否能够提供清晰的指导和说明、是否能够让发布操作人员快速掌握和操作、是否能够减少人为错误等。

(10) 可维护性：应急预案是否易于维护和更新，包括是否能够及时更新和修订、是否能够保持最新的技术和流程、是否能够在下一次的发布中继续复用等。

4.2.2.2 版本发布实施

(1) 发布前确认

发布准备是指在正式进行发布操作之前所需要操作的一系列步骤，以确保发布流程的顺利和成功进行。在执行正式发布之前，SRE（Site Reliability Engineering）工程师需执行如下工作，但不限于：环境备份、制品预分发、告警屏蔽以及各项准备工作的最后确认。

(1) 发布流程确认。需要在发布前与相关干系人对齐整体发布内容，需要明确各项发布的时间及执行内容。

(2) 风险评估。需要对发布内容进行全面评估，以确定发布对现有系统的安全性、机器性能、环境依赖等是否会产生影响，通过风险评估可以避免在变更过程中出现无法回滚、安全漏洞等问题。

(3) 环境备份。在发布前，必须对系统的数据、配置以及业务程序进行备份，以确保在出现问题时能够快速恢复。数据备份范围包括但不限于业务数据库、文件系统等，而配置备份则包含业务配置文件、证书文件等。选择适当的备份方式（手动或自动备份），以确保备份的及时性和完整性。

(4) 制品预分发。制品预分发是指在发布前将需要更新的文件提前分发到目标服务器，以便在发布时能够快速更新。这一步骤有效缩短了发布时间，降低了发布过程中出现问题

的风险。完成预分发后，需要进行验证，确保预分发的制品能够正确地更新到目标服务器上。

(5) 监控告警屏蔽。为防止发布停机时引发不必要的告警，SRE 在前置工作中需要屏蔽部分监控告警指标，以防止告警风暴的发生。设置合理的屏蔽时长，并记录屏蔽监控指标的详细信息，包括指标名称、屏蔽时间范围、屏蔽原因等。同时，需要避免屏蔽敏感指标或对系统安全性产生负面影响，以确保非发布模块的稳定性和安全性。

(6) 各项准备工作的最后确认。作为发布前准备工作的把关人，业务 SRE 需确认测试是否充分、发布环境是否准备就绪、备份和恢复是否有效、风险评估和应急预案是否充足、通知是否及时准确、发布执行清单是否完整清晰等相关事项。

(7) 效果评估。发布前置工作的效果可从以下几个方面考虑：

发布成功率：衡量发布前置工作效果的重要指标之一是发布成功率。较高的发布成功率表示前置工作效果较好。

发布耗时：另一个衡量效果的指标是发布时间。若前置工作充分，发布时间应缩短，提高发布效率。

问题处理速度：若出现问题，评估前置工作效果的一个指标是问题处理速度。充分的前置工作应该能够加速问题解决，减少系统故障时间。

用户反馈：用户反馈是评估前置工作效果的另一个关键指标。不良的用户反馈表明前置工作未达到预期效果，需要进一步改进。

(2) 发布执行

发布执行是 SRE 团队负责执行一系列发布操作，包括停服、更新、起服等，以将新功能成功发布到线上正式环境并对用户开放。为确保每次业务新特性稳定上线，SRE 团队需制定可靠的发布执行规范和发布意外防范机制，以确保各类对象（如二进制程序、配置文件、数据库及依赖环境）以可重现、自动化的方式发布到业务生产环境。

发布执行的主要步骤包括：

部署：执行部署脚本，将软件部署到指定目录。

测试：在部署完成后进行功能、性能和安全等测试，以确保新版本正常运行。

监控：通过监控告警手段监测发布后运行状态，及时发现并解决问题，保障系统稳定性和可靠性。

回滚：如果出现问题，按照已制定的应急预案和操作流程，及时回滚到上一个版本。

发布执行过程中还应包括以下内容，包括但不限于：

自服务模型：SRE 发布工程师通过开发工具、制定最佳实践，使整个发布执行过程自动化，不再需要 SRE 工程师干预。研发团队或其他第三方人员可以自主掌握和执行发布操作，构建发布执行的自服务模型。

发布操作的简单化：用户可见的功能应尽快发布上线，以减少每个版本之间的变更。发布执行时应按小批次进行，易于理解每次发布对系统的影响。通过逐步改变系统，同时考虑每次变更对系统的改善和退化，寻找最佳方案。

发布操作的标准化：统一运维操作入口，降低复杂度，提高可管理性。避免手工误操作，防范悲剧发生，将原本复杂易错的手工操作实现标准化运维，使运维操作更加可靠。同时，减少由运维人员手工误操作所带来的业务风险。

发布执行的稳定性与灵活性：平衡稳定性和灵活性，通过构建通用发布流程、实践和工具，提高发布执行的可靠性。最小化发布执行流程和工具对开发人员灵活性的影响。

最小化意外复杂度：SRE 团队需不断努力消除发布执行过程中的必要复杂度，发现并提出抗议，以减少引入的意外复杂度。

效果评估：

对发布执行效果进行准确性、成功率和效率三个方面的评估：

(1) 准确性：操作是否按照计划和目标进行，是否准确无误地完成了发布操作。

(2) 成功率：发布执行的成功率是否高，是否达到预期目标，是否影响了业务的稳定性和可用性。

(3) 效率：发布执行的效率是否高，是否在规定时间或提前完成任务。

(3) 发布验证

SRE 发布验证是指在软件发布后对已部署的软件进行验证和测试，以确保软件能够正常运行，并满足预期的功能和性能要求。发布验证贯穿于 SRE 的整个发布过程中，而不是仅仅存在于部署后。根据不同的发布计划，存在不同的验证流程，其中包括但不限于以下几个方面：

1. 稳定性验证

稳定性指系统要素在外界影响下表现出的某种稳定状态。对于应用程序而言，稳定性是保障系统能够满足 SLA（服务水平协议）所要求的服务等级协议的关键。在应用程序发布过程中，应持续关注关键的：基础指标、应用指标及业务指标。

基础指标：所依赖的系统和基础设施层面的指标

例如：系统负载、内存容量、网卡流量

应用指标：能够反馈业务依赖的接口服务是否可用的指标

例如：请求速率、请求时延、请求成功率

业务指标：能直观地反馈业务或者系统功能是否可用的指标

例如：在线人数、订单量、评论量

确保应用程序能够以一个安全的状态持续运行。如果观察到系统整体处于非稳定状态，应及时采取相应的应急措施，以将发布的影响降到最小。

2. 确定性验证

应用程序的发布通常旨在修复缺陷或提供新特性，因此发布的结果应该是确定的。在发布过程中，对应用程序进行持续观察，只有在一个或多个周期内应用程序的运行状态符合预期，功能性或缺陷性的改动能够生效时，才能将发布视为有效。一旦发现应用程序在发布过程中出现了非预期的情况，应及时记录并评估是否需要继续发布，以规避风险。

3. 依赖性验证

依赖性指系统中不同组件之间的相互依赖关系。一个复杂的系统通常由多个组件组成，系统中某个组件的变动可能对整体系统产生影响。在发布过程中，应关注与发布程序存在依赖关系的其他组件的运行情况，以确保应用程序的改动不会对其他组件造成预期外的影响。最好在发布前对依赖组件的运行情况进行评估，尤其是对性能或时延要求较高的组件，以避免对业务造成雪崩效应。

4. 效果评估

SRE 发布验证的效果可以从以下几个方面进行评估：

发布验证的平均时间：衡量发布验证的平均时间，即发布验证开始到完成的时间。

发布后的稳定性：衡量发布后系统的稳定性和可靠性。

发布后的性能：衡量发布后系统的性能，包括响应时间、吞吐量、并发量等。

发布后的用户满意度：衡量发布后用户的满意度，包括用户的反馈和投诉等。

通过对这些标准的综合评估，可以优化发布验证的流程和方法，提高系统的稳定性和可靠性。

(4) 应急预案实施

SRE 发布应急预案实施是指在软件系统发布过程中，对于突发紧急情况，SRE 团队采取一系列紧急措施以确保发布过程的稳定性和安全性。以下是 SRE 发布应急预案实施的详细步骤：

(1) 紧急通知：在出现紧急情况时，SRE 团队应立即通知相关人员，包括但不限于开发团队、测试团队和运维团队，以便共同制定应急方案。

(2) 应急方案选择：SRE 团队根据事先制定的应急方案，选择最适合情况的解决方案。这包括确定问题的性质和影响、采取的紧急措施、修复时间等关键因素。

(3) 紧急回滚：在紧急情况下，如果发布过程出现问题，SRE 团队应立即执行回滚操作，将系统还原到之前的稳定版本，以最小化对用户的影响。

(4) 问题排查和解决：SRE 团队应立即进行问题排查，采取必要的措施来解决问题，以确保发布过程的稳定性和安全性。

(5) 事后总结：在问题解决后，SRE 团队应进行全面的事后总结。这包括对应急过程的分析，问题原因的详细剖析，以及提出改进措施，以便在下次发布中更有效地执行。

通过 SRE 发布应急预案实施，团队能够迅速、有效地应对紧急情况，保障发布过程的稳定性和安全性，最终提升用户满意度。

效果评估：

发布应急预案实施的效果可以从以下几个方面进行评估：

(6) 预案执行情况： 应急预案是否能够按照预定计划执行，是否能够应对突发情况做出及时调整。

(7) 故障恢复时间：应急预案是否能够快速、有效地解决故障，缩短故障恢复时间，减少业务影响。

(8) 用户投诉数量。应急预案实施后，用户投诉数量是否下降，是否能够有效地保障用户的服务体验。

(9) 实施效益。应急预案是否在合理的成本下，能够限制发布故障的影响范围，最大限度地减少业务损失。

通过对这些方面的全面评估，团队可以更好地了解 SRE 发布应急预案实施的实际效果，为未来的发布过程提供有针对性的改进建议。

4.2.2.3 发布总结

SRE 发布总结是指在软件系统发布后，SRE 团队进行的一系列总结和评估活动，旨在深入挖掘问题、总结宝贵经验，并提炼出切实可行的改进措施，以优化未来的发布过程。SRE 发布总结通常包括以下详尽步骤：

(1) 数据收集：收集涵盖发布过程各个方面的数据，其中包括但不限于发布时间、故障和停机时间、用户反馈等多维度信息。

(2) 数据分析：对所收集的数据进行深入分析，以明确识别发布过程中存在的问题和瓶颈，确保全面了解系统性能和用户体验。

(3) 经验总结：精炼并深度总结发布过程中所获得的经验和教训，包括成功实践的要点以及需要改进的方面，为未来的发布活动提供宝贵指导。

(4) 改进措施提出：根据对经验和教训的深入总结，明确提出一系列切实可行的改进措施，以不断提高发布过程的质量和效率。

(5) 改进实施：将提炼出的改进措施有机地融入到发布计划中，并在下一次发布活动中有序实施，以确保团队在实践中持续演进。

通过 SRE 发布总结，团队得以持续优化发布流程，提升发布效能和质量，有力保障业务的稳定性和可用性，最终为提高用户满意度奠定坚实基础。

4.2.3 变更的工程体系设计

为了减少因变更引发的故障，SRE 团队应通过软件工程体系设计一套通用的，可持续性的变更系统。

4.2.3.1 面向变更管理的 ITSM 设计

ITSM 全称（Information Technology Service Management）,面向变更管理的 ITSM 工单设计要以“变更流程管理，变更风险评估，变更窗口管理，变更自动化，变更知识库”作为核心的功能，从而确保更严谨、有效地实施变更操作。

(1)变更流程管理

ITSM 系统需提供变更流程管理的基础能力，以承载变更管理流程规范落地，其中包括：

- 变更定义：支持变更流程负责人对变更管理的业务表单和工作流进行自定义。
- 变更申请：支持变更申请人填写变更工单需求并提交变更申请。
- 变更审核：支持变更复核人对变更内容进行复核及修订。
- 变更审批：支持对变更申请进行审批，包括单人审批、多人会签审批、多级审批等。
- 变更实施：支持对变更实施的结果进行记录。

(2)变更风险评估

对变更的风险进行评估是变更管理中的重要一环，不同的变更风险级别会对应不同的变更管理策略。因此，ITSM 工具应该能辅助提升变更风险识别的准确度，减少因风险误判而引发的故障，包括：

- 变更风险级别定义：支持风险级别自定义。
- 变更风险评估规则：支持风险影响因子和风险计算规则的定义。
- 变更影响面分析：支持与 CMDB 进行联动，获取变更直接影响或间接影响的资源对象，以分析本次变更可能影响的关联资源。
- 变更资源保障分析：支持与存储资源系统，云资源系统等进行联动，以分析本次变更所需要的基本的硬件资源和人力资源。
- 变更冲突检测：支持与人员工作日程联动，提供可视化的变更事件“日历日程”，结合变更日程分析可能存在的冲突。

(3)变更窗口管理

变更窗口的定义是变更风险管控的重要手段，在影响较小的时间段实施变更，即使变更过程中出现异常也能将损失降低到最小。ITSM 工具需提供变更窗口维护和合规性检测的能力，包括：

变更窗口维护：支持对可执行变更的时间段进行定义，以供变更申请时进行选择。

变更窗口合规性检测：支持将变更实施的时间计划与变更窗口的定义进行联动校验，以确保不会出现实际变更实施在变更时间窗口之外的情况。

(4)变更自动化

企业业务的敏捷、研发的敏捷都会导致运维变更的频率增加，仅靠人工已难以满足变更效率的要求，不管是基础架构的变更还是应用版本的发布变更，都已经在寻求更高效的自动化执行工具。因此，ITSM 的变更管理能够与自动化工具便捷的集成也相当重要，复杂变更逐步抽象为多个低风险的标准变更，并将标准变更实现端到端的自动化。为实现与自动化的联动，ITSM 的设计应具备：

- 第三方集成能力：支持如 API、脚本等系统集成方式，以获取自动化执行前所需的基础数据，或者合适的时机调用自动化动作。
- 自动化流程节点：ITSM 的流程编排需支持自动化节点，以便于管理流和工程流之间衔接的编排。

(5)变更知识库

知识库可以将过去的变更经验和知识进行整理和归类，在面临类似的变更需求时，可以用来学习培训，提供辅助决策，从而降低错误风险，减少重复工作，提高工作效率。

知识库一定要进行分类整理，变更知识库要把“变更类型、变更原因、变更范围”作为主要的分类维度，再配合“关键字检索”的功能，就可以让变更知识库发挥重要的价值。

- 变更类型：硬件变更、软件变更、配置变更、流程变更等
- 变更原因：故障修复、性能优化、功能增强、合规需求等
- 变更范围：单个系统内、多个系统等

4.2.3.2 面向变更管理的 CMDB 设计

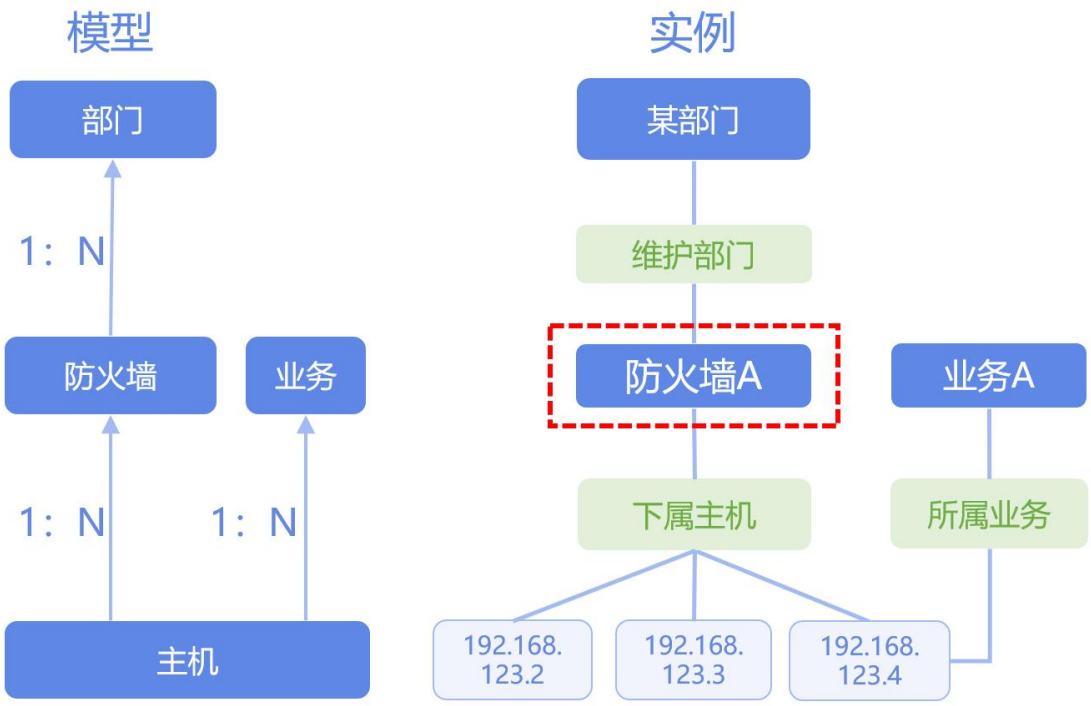
CMDB（Configuration Management Database，配置管理数据库）是一个存储 IT 基础设施中所有配置项（CI，Configuration Item）信息的数据库，包括硬件、软件、网络设备、服务等。

CMDB 不仅存储配置项的属性信息，还存储配置项之间的关系。在变更实施过程中，CMDB 可以通过“配置模型、配置实例、场景联动”等能力，识别准确的“变更信息”，支持这些流程的运转、发挥配置信息的价值，帮助实施人员执行变更任务。

(1)配置模型定义与管理

1. 模型管理：根据元数据之间的关联关系，将变更操作常用的一组数据定义为一个模型，比如主机模型，包括主机的名称，操作系统，CPU，内存，磁盘，MAC 地址，CPU 架构，内网 IP，外网 IP，IPv4，IPv6，云主机实例 ID，云主机状态，创建时间，创建人，固资编号，设备 SN，主要维护人，备份维护人，所在省份，所属运营商，质保年限，SLA 级别，寻址方式等。按照这个方式，根据变更场景定义“业务模型，项目模型，交换机模型，负载均衡模型，防火墙模型”等。需要提供一个把这些元数据建立连接的“模型管理”的功能。

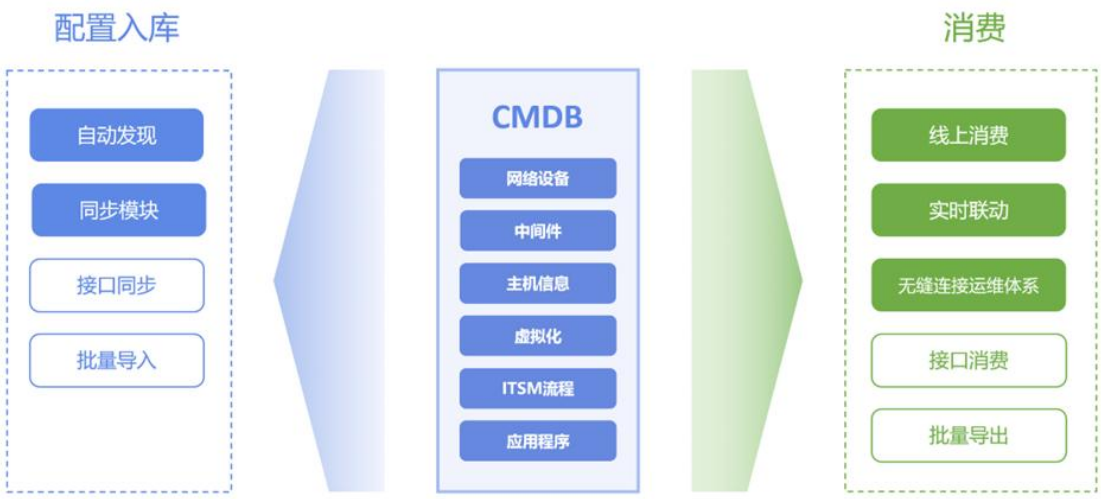
2. 模型关系：变更的配置数据之间，天然存在关系。使用“模型关系”定义模型之间是如何关联。要提供可视化的拖拽页面，通过关联关键字，在不同模型之间建立一对一，一对多的约束关系，所有实例支持无差异关联。要提供关联关系的创建，删除。为了确保模型的基础功能，部分内置关联不可以删除和修改。



(2)配置元数据入库与消费

变更实施前，需要将与变更相关的配置项元数据录入 CMDB，包括网络设备、中间件、主机信息、虚拟化、流程信息、应用程序、维护人员、系统环境等，要支持批量录入、更新、导入等功能，支持通过 API、SNMP、日志分析等模式的自动发现入库元数据。变更发起后，周边系统可以通过订阅和推送的方式获取消费数据，不仅要做到消费数据的实时联动，还应该做到无缝连接运维体系，从而通过变更管理，实现 CMDB 配置数据的全生命周期管理。

- 资源入库：CMDB 的数据新增入库应支持与变更管理中的资源申请、应用投产等变更场景进行关联，在相应流程结束后，数据可自动新增录入 CMDB。
- 资源变更：CMDB 的数据变更应支持与变更管理中的资源变更（如：主机扩容、设备迁移）、应用版本变更等变更场景进行关联，在相应流程结束后，数据可自动更新 CMDB。
- 资源退库：CMDB 的数据变更应支持与变更管理中的资源退库（如：主机下线、设备报废）、应用下线等变更场景进行关联，在相应流程结束后，数据可自动更新 CMDB。



支持更高效地变更管理流程是 CMDB 重要消费场景之一。变更管理流程主要是对 CMDB 中的数据信息进行查询消费，其中包括：

- 配置项的基础信息查询：变更流程的发起环节，需要从 CMDB 获取变更对象相关的配置信息，以识别变更对象是否准确。因此，CMDB 工具的设计应考虑提供实例数据的信息查询接口。
- 配置项的拓扑结构查询：变更流程的评估环节，需要从 CMDB 获取变更对象的关联信息，通过支撑、依赖、上下游等关联关系来识别变更的影响面，以辅助评估变更风险。因此，CMDB 工具的设计应考虑提供基于特定对象实例的拓扑结构信息查询接口。
- 配置项的管理信息查询：变更流程的审批环节，需要通知对应的运维负责人。尤其针对重大变更，需要组织变更的干系人一同进行审批。此时，需要从 CMDB 获取变更对象的组织、负责人等信息，以实现变更管理者进行准确的通知和会议组织。因此，CMDB 的配置项模型设计应考虑管理信息字段的设计和维护。

(3)配置实例管理

1. 支持配置实例的创建、更新和删除等能力，信息更新时，通过空值判断，长度校验，正则表达式等，确保数据准确。

2. 在变更发起时，要对实例进行准确的查询，支持全局查询，多维度联合查询，管理实例查询等。

3. 为了避免变更越权操作，所有的变更操作要按照角色，业务，实例，层级等多维度控制。

4. 提供操作审计的功能，对变更操作可以追溯，降低变更风险。

(4)CMDB 与变更场景的联动

CMDB 的重要性上升，是因为不仅仅需要支撑变更管理流程更高效地运转，同时还需支撑后续的自动化变更执行。而自动化场景中，如何支撑好应用发布层面的变更是比较复杂的，CMDB 的模型设计需要重点考虑如何支撑好应用发布自动化的场景。

CMDB 应用层的模型设计主要采用以业务为中心的设计理念，我们称之为“业务拓扑”：

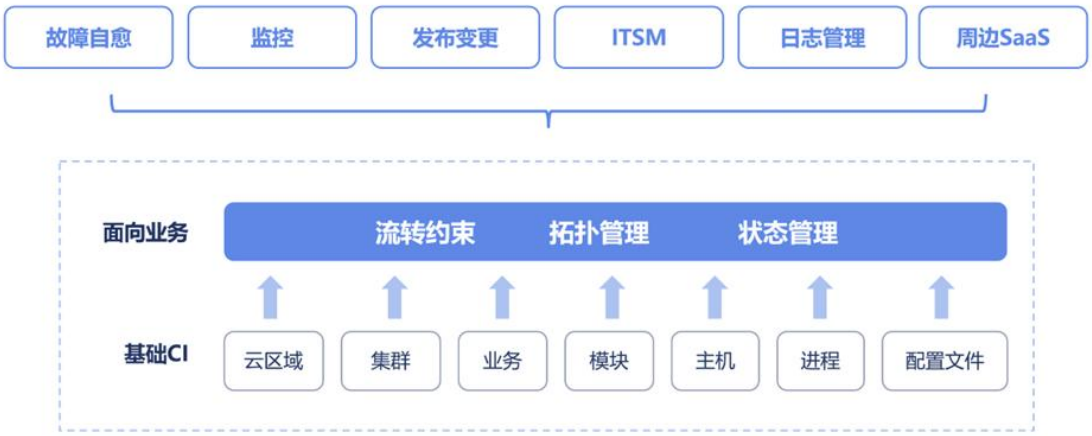
业务拓扑是对业务应用系统所提供的业务功能进行领域的划分，纵向来看，业务拓扑描述的是一个应用系统的划分规则，其最小颗粒度为单独对外提供服务能力的模块。

第一，模块为业务拓扑末端节点。模块在功能上对外提供独立的服务能力，在应用系统中为最小的组成部分，往下，与基础架构资源直接关联（中间件，主机，调用的数据库等）。

第二，业务拓扑一般不要超过 3 层。超过三层的业务拓扑太深，容易出现管理上的混乱，不利于场景的消费和管理。

第三，支持多环境管理。在应用从需求到发布的阶段中，需要在不同环境中进行进阶，开发环境，构建环境，测试环境，预发布环境，发布环境等。环境影响的不仅是基础架构资源的划分，还是影响应用配置的变动。不管是在 DevOps 还是 SRE 理论体系中，应用环境的管理都强调了其重要性。

第四，支持多集群管理。这里的集群指的是分布式的集群。每个集群都对外提供相同的业务服务，集群中包含着一个或多个模块。集群一般在互联网应用架构下出现得比较多，如：广东集群，广西集群。



4.2.3.3 面向变更管理的作业平台设计

变更是一个高频操作，作业平台（Job Platform）的目标是提供一个统一的软件工程体系，能够将变更中使用的“脚本执行，文件传输”等操作规范化，标准化，无需远程登录目标主机，就可以完成变更操作。通过这种设计，可以不断沉淀优秀的变更经验，为变更操作的自动化，与周边系统的联动，提供有力的功能保障。

(1)作业通道能力

设计一个先进的作业平台，首先在底层需要有一个能面向海量、异构、跨区域的 IT 基础架构的高性能、高可用、多功能化的作业通道能力。这些能力包括以下方面：

1. 任务服务。可以目标管控节点上的远程作业任务执行能力，如 Bash、Perl、Bat、PowerShel、Python、SQL 等。

2. 文件服务。提供文件快速传输能力，实现在不同的文件大小和节点规模下，动态调整组网策略以此达到最佳的文件传输效率，为业务提供高效稳定的底层文件传输服务。

3. 代理（Agent）。可以安装在业务需要管控的实体机、虚拟机或者容器里面，是任务执行和文件传输的实际执行者。代理还需要具备灵活的、可自定义的插件扩展功能，以支持通过 snmp、ipmi 等各类协议对各类 IT 设备或接口进行命令操作或数据获取。

4. 平台服务（Proxy 集群）。由于被管控的设备往往分散在各个网络区域，作业通道还需要支持在各个物理或虚拟网络区域部署高可用的 Proxy 集群，以实现高性能、高可用的跨区统一管控能力。

(2)变更脚本执行与变更文件分发

脚本执行和文件分发是作业平台最基本能力，工程化落地需要考虑以下几个方面：

1. 快速执行脚本能力。快速执行脚本提供的使用场景是针对需要快速执行的一次性任务，并且可能会反复调整脚本逻辑来执行得到用户想知道的结果，类似于在终端跳板机Console执行的效果。脚本内容应支持手工录入和脚本库内容引用，并且支持各类的脚本语言；脚本需要支持传参，并且支持敏感参数的屏蔽；支持系统执行账号的指定或选择，并可支持执行超时等设置。

2. 快速文件分发能力。快速分发文件的目的也是让需要进行一次性传输文件的用户，可以快速地执行或调试并得到结果。除了基本的文件源地址与源目录、目标地址与目标目录的配置外，也应该支持传参、超时、执行账号选择等配置，还需要考虑分发的上传/下载限速以避免对生产网络造成业务影响。

3. 实例对象管理。作业平台是SRE体系下的一个能力模块，作业平台不应该维护一套独立的IT对象配置信息，而应该与CMDB进行对接，脚本执行与文件分发中的实例均应该来自CMDB而不是在作业平台中单独维护。

4. 脚本库管理。脚本库是作业平台的重要资产，最常见的分类有两种，以IT对象类别以及场景类别，IT对象包括从IaaS的硬件、网络、OS、虚拟化、容器等，到PaaS层的中间件、数据库、消息队列、缓存等，再到SaaS层的各类Web服务器，应用等。场景则是基于生命周期考虑的从资源交付、安全合规、巡检，到日常变更相关的启停、扩缩容、迁移、应急自动化、灾备切换、下线等操作场景。当然，脚本库的建设也需要考虑参数、版本、账号、调试等能力，这里不再一一赘述。

5. 文件管理。从场景上看文件分发相关的场景包括软件安装、软件更新、配置文件更新、操作系统补丁下发等；从类型上看文件可简单分为二进制文件和可编辑的文档类配置文件。这两类文件有参数、版本、分类等通用的管理诉求，可编辑的配置文件还需要具备在线编辑、生产比对、KV提取与发布的管理功能。

(3)变更作业模板与执行方案

通过流程编排能力，将运维操作场景中涉及的多个脚本执行或文件分发步骤组合成一个作业模板，这个作业模板尽可能把场景相关的共性逻辑都包含进去，然后再根据实际使用场景衍生出相应的执行方案，那么作业模板和执行方案的关系即为"一对多"。

1. 作业模板。作业模板需要具备基础信息、全局变量、作业步骤、作业编排四个基本能力。基础信息主要是描述作业模板名称、场景标签。全局变量需要支持字符串、命名空间、实例、加密字符串等类型。作业步骤需要支持执行脚本、文件分发、人工确认三种类型。作业编排则是可以将这些步骤进行串并行组合最终形成一个作业模板。

2. 执行方案。作业模板创建完后，即可以从模板中创建出一个或多个根据场景需求定制的“执行方案”；每个执行方案可以从模板中勾选自己所需的步骤，修改全局变量的变量值。可以根据变更范围设置执行方案的策略和机制。

滚动策略：按照百分比（10%，按每 10%台一批直至结束）或者具体的数值（10，按每 10 台一批直至结束）设置。

滚动机制：如执行失败则暂停、忽略失败，自动滚动下一批、不自动，每批次都人工确认。

3. 作业调度。作业执行方案可支持定时、立即开始、周期等多种调度能力。

4. 执行控制。作业执行过程中，根据业务需求需要提供相关的开始、暂停/恢复、终止/强制终止、人工确认等功能，并实时展示相关的运行状态。

(4)变更操作安全(权限与审计)

1)高危语句拦截。作业平台应提供可自定义的高危语句自动检测与拦截功能，以避免高危变更命令操作对生产环境造成故障。

2. 权限管理。作业平台作为企业统一作业变更操作底座平台，需要具备足够细颗粒度、多维度、灵活适配各类组织架构的权限设计，以保证安全的变更操作。从工程化落地的最佳实践讲，需要从整体 SRE 工程体系上进行权限中心的设计，作业平台的权限申请、审批、控制将由权限中心进行集中控制管理。

3. 作业审计。作业平台应提供所有通过页面执行、定时执行和 API 调用等渠道触发的任务历史查看记录以便于进行审计或在故障诊断中分析 IT 变更对故障的影响。

4.2.3.4 面向变更管理的微服务容器编排设计

随着业务发展，需要不断地对软件系统进行更新和优化，一些快速迭代的业务场景逐步使用微服务的架构，就需要设计针对微服务架构的变更管理工程体系。

(1)微服务容器集群管理

1)统一集群管理，纳管不同种类的 K8S 集群

在容器管理平台纳管不同来源的 K8S 集群，在业务持续建设过程中，在不同地域会综合考虑云厂商的能力、价格，以及业务特性，选择不同云厂商的容器服务。比如国内使用 TKE 和 ACK 容器服务，在海外使用 AWS 和 GCP 的容器服务。但业务是一个团队在维护，为了避免每个业务运维方都单独对接不同厂商的容器服务，额外增加适配和管理的成本。

2)支持不同的集群管理模式

容器管理平台要能够支持不同的集群管理模式，来应对不同业务场景需求。

容器管理平台需要支持独立集群模式，以便特殊业务应用能够独享该集群资源，并提供对等的权限控制；对于自建独立集群需要提供配套的集群自动化运维方案；对于托管独立集群需要对接第三方管理平台。

容器管理平台需要支持共享集群模式，在大型单个集群内提供基于 Namespace 级别或者 vCluster 级别的资源隔离能力，以应对业务在无需关心集群及节点资源情况下使用容器技术进行发布的场景。

容器管理平台需要支持联邦集群模式。大型业务往往在分布式处理能力和容灾能力上有多地多中心部署的需求，容器管理平台需要提供联邦集群管理能力，便于业务像向单集群发布应用一样向多集群发布应用。

3)集群容量管理

容器管理平台在不同集群管理模式下，提供一致的容量管理方案。集群容量管理，能够实时查看集群容量，能够手动或者自动对集群进行扩缩容操作来满足容量设计要求。

在集群容量管理自动工作时，应该具备按照一定的可配置策略，通过 CA 从节点池或者资源池获取节点，并自动加入对应集群，达成容量设计目标。

4)统一对外暴露的 API

通过建设统一的容器管理平台，对外暴露统一的集群管理 API 接口，对业务方屏蔽底层的 K8s 集群版本和厂商差异。同一个资源在不同的 K8s 版本中 APIVersion 是不一样的，为了规避发布业务方做持续适配，或者让第三方业务系统消费时单独适配，容器管理平台需要适配差异，对外提供统一的 API 服务。

5)表单化的集群资源管理能力

在不同集群模式下，对 K8s 集群提供表单化管理能力，能够对 权限范围内的 Node、Cluster、NodePool 等资源进行统一管理。能够实时查看权限范围内的集群状态、资源用量、CPU 利用率、装箱率、节点列表等基础信息；能够通过页面对权限范围内的集群进行增删改查。

(2)微服务应用部署及变更

1)管理微服务应用部署及变更配置和流程

以表单模式和 YAML 模式构建容器应用部署配置，包括但不限于编排 Deployment、StatefulSet、Service、ConfigMap 等资源对象。表单编排模式，是将 YAML 文件中的重要变量和常用变量抽取出来，做成让用户选择的表单，降低运维工程师使用门槛。

表单模式的局限性在于，不是所有的容器配置都会支持在表单中支持，因此也需要支持以原 YAML 文件方式去编排容器应用，支持用户导入 YAML 文件，从 K8s 直接同步 YAML 文件等方式。

不论是表单模式还是直接编辑 YAML 文件，均应提供不同部署及变更配置的版本管理能力，并支持不同版本间的比对、升级、回退。

微服务应用的发布变更应该支持通过任务模板，创建多个步骤/节点执行微服务发布任务。发布任务模板中可以编排容器发布节点，不同容器发布节点支持串行发布、并行发布等方式。针对微服务场景，每次版本发布要灵活发布业务系统中的不同服务，因此节点需要支持灵活的开关，满足微服务特性要求。

2)支持多样化的应用部署及变更方式

不同业务形态下，应用的特性有很大差异。容器管理平台应该具备多样化的编排调度能力，来支持不同类型应用的部署和变更。

支持无状态应用的部署和变更。支持通过 K8s 原生 Deployment 方式或者其他扩展 CRD 的方式，支持无状态应用的部署和变更，并提供过程可视化和结果的实时反馈。

对有状态服务的支持。有状态应用往往是业务中的核心服务或者特殊服务，在业务系统中往往具有重要的价值，容器管理平台应当通过扩展 CRD 方式对有状态服务提供更贴近真实场景的部署和变更能力。这些能力应该包括但不限于以下方面：

a)容器原地更新能力；

b)支持变更前置或后置处理；

c)支持按照一定的步骤或者顺序完成部署或变更；

d)支持部署或变更交互式过程控制；

不管是无状态服务还是有状态服务，容器管理平台应该能够使用上述能力，依据用户设定的方式来动态调整应用实例个数来达成应用容量设定。当满足扩容条件时，自动拉起新的服务实例；满足缩容条件时，能够按照一定策略驱逐容器，并缩容容器实例，以达成应用容量设定目标，并兼顾减少碎片和装箱率要求。

微服务业务部署和变更，在灰度策略上有更灵活的选择。容器管理平台应该实现这些灰度策略，并管理灰度过程，以确保业务部署和变更的顺利。这些灰度策略，包括但不限于以下种类：滚动升级、蓝绿发布、金丝雀发布。在进行灰度变更时，能通过南北向流量调配来控制灰度过程。

3)容器应用状态管理

容器管理平台提供页面，展示 Namespace、Workload、Pod、Container 级别的状态数据，使得用户能够以直观方式实时查看应用运行状况。

除此之外，K8s 提供的 Service，以及其他自定义控制器，可能提供横向维度的映射关系，在应用状态管理页面，也需要对这类映射关系进行展示。

在实际 SRE 工作中，往往同时涵盖传统物理机、VM 和容器，为便于统一管理，在 CMDB 中需要维护容器应用拓扑关系，以及容器与母机的映射关系。在 CMDB 中的容器应用拓扑关系，可以精确到 Pod 级别，以便于精确维护不同资源间的关系。

(3)微服务容器的可观测

微服务容器在变更时的，支持查看容器服务相关的事件、日志、监控。会调用 API 实时拉取事件和日志，采集监控数据，查看容器资源对应的详细监控数据。

查看容器发布状态：会调用 K8s 接口查询容器资源的状态，比如 deployment 是否处于 running 状态，副本数是否符合预期

查看容器发布事件：会调用 K8s 接口查询容器资源的事件，pod 调度到哪个节点，镜像拉取成功，pod 启动成功等事件记录

查看容器发布日志：容器变更之后，运行中会产生容器日志，查看容器标准输出日志中是否存在较多 error 字段

查看容器发布监控：容器变更之后查看容器的 CPU、内存、网络等指标监控是否正常。

4.2.3.5 面向变更管理的跨系统调度编排设计

完整的变更过程，往往会涉及对各个系统进行变更执行，例如变更前对监控告警策略的调整，变更过程完成对应用的更新和网络策略的变更，变更后进行全面的业务验证等。有些变更还需要考虑多环境、多应用、多实例、多机房、多可用区的灰度变更流程设计，此时需要不仅是作业平台，还需要提供跨系统调度编排能力来实现变更全过程的自动化。

(1)通用调度引擎

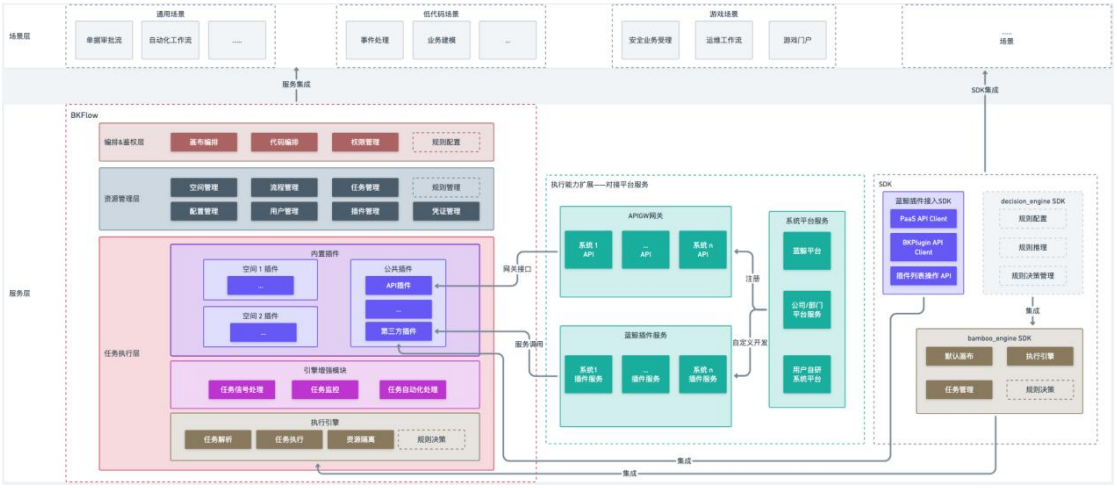
通用调度引擎是整个跨系统调度编排设计的核心部分，它负责管理和调度不同系统之间的数据和业务流程，即变更的流程执行能力。在变更流程执行过程中，也要支持决策引擎和 FEEL 表达式解析器，让变更的操作流具备自动化的逻辑判断能力。

- 流程引擎：可以解析，执行，调度由用户创建的流程任务，并提供如启动，暂停，继续，撤销，跳过，停止，强制失败，重试和重入等灵活的控制能力，支持

立即、定时和周期性启动的调度方式，支持并行、子流程等进阶特性，并可通过水平扩展来进一步提升任务的并发处理能力。

- 决策引擎：基于 Python 的 DMN(Decision Model Notation) 库，使用 FEEL(Friendly Enough Expression Language) 作为描述语言，可以作为决策引擎，可用于流程引擎中分支网关条件表达式的解析，解决实际业务场景中的决策问题。

流程调度引擎在设计的时候，可以分为任务执行层，资源管理层，编排和鉴权层。



通用调度引擎需要具备并发处理能力、容错能力和可扩展性，以适应不同类型变更操作，如脚本执行、API 调用、基于通用网络协议的变更操作等。

(2)编排能力

编排能力是将变更过程中各个系统的数据和业务流程进行有序组织和协调的能力。

支持通过“可视化画布”的方式，拖拽变更流程中需要的节点，在变更流程中添加逻辑网关，如串行，并行，分支，汇聚，条件并行。支持嵌套编排，设置变更的子流程。提供基于流程即代码的自动化流程编排。

变更流程节点中支持内置一些插件，这样制作一个变更流程将更高效。

- 变更操作的基础插件：定时插件、HTTP 请求插件、发送通知插件、人工确认/暂停插件、审批插件、消息展示等。
- 常用变更系统的插件：CMDB 的插件（删除主机，主机加锁，修改主机服务状态，按照规则修改主机自定义属性，故障机替换，转移故障机至待回收/故障机/空闲机模块，批量更新主机/模块/集群属性，上交主机到资源池等），作业平台的插件（快速执行脚本，快速分发文件，获取任务日志等），ITSM 插件（更新变更日历，关闭变更工单等），监控插件（告警屏蔽插件，解除告警屏蔽插件等）。

通过编排能力，在一个变更流程中，就可以编排多个系统的变更步骤，集中化地实现变更操作的自动化和批量处理。

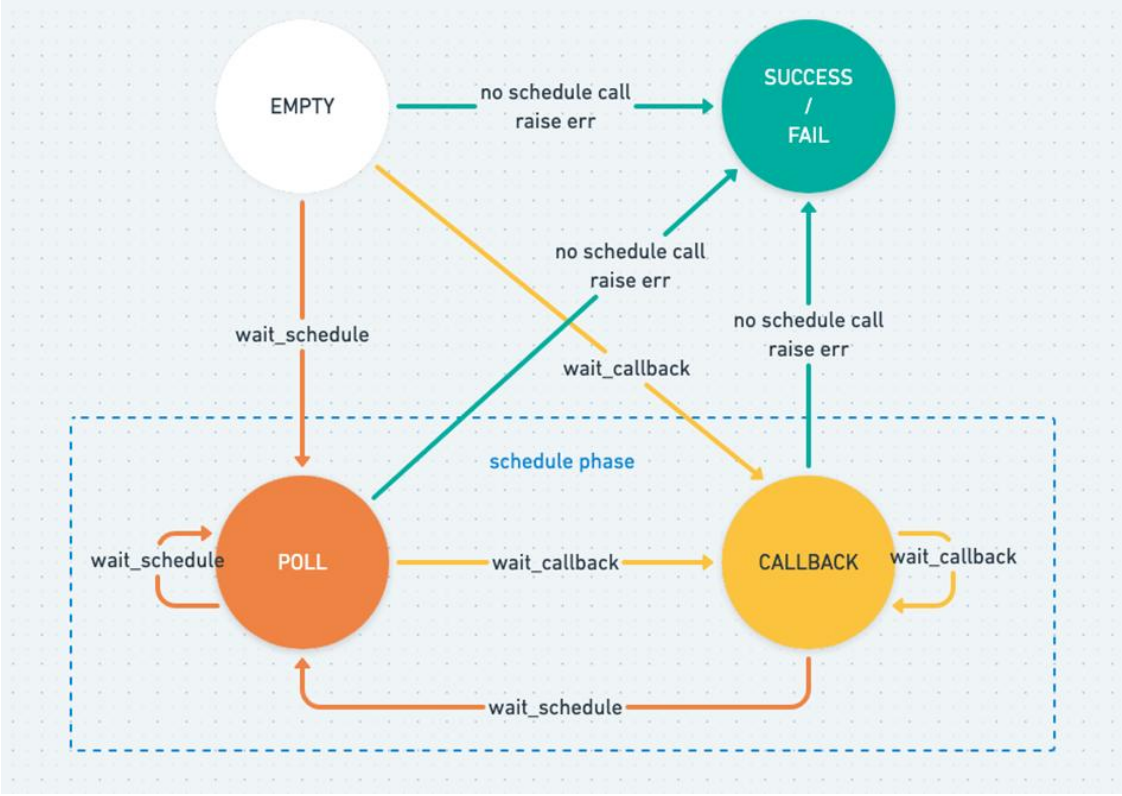
(3)扩展插件市场

当某个系统的变更操作过程较为复杂时，可以通过丰富的表单界面和验证逻辑将企业内部各个系统、各个平台的变更操作过程组装成一个标准插件模板，来实现体验更佳、效率更高的调度编排能力。

插件的开发需要具备完整的开发流程规范，包括高效的可视化低代码的开发方式、插件调试和快速发布能力。

建设插件市场，要使用统一的、轻量化的插件开发框架（plugin-framework），开发者使用该框架进行插件开发，通过调用 plugin-framework 暴露出来的标准接口，完成系统插件功能的实现，并将其部署到变更管理的系统上，就可以使用插件的能力。

一个插件在一次执行生命周期中可能会经过下图所示的状态转换，通过框架提供的方法，开发者可以在插件逻辑中进行任意合法的状态流转，以适配不同的业务变更场景。



通用性的插件可以发布到“插件市场”，共享插件资源。企业可以根据自身需求，从插件市场中选择合适的插件，以实现对不同系统的支持和扩展。

(4)调度编排模板与任务

编排后形成的流程称之为模板，模板应具备全局变量、版本管理、版本审批发布等功能。企业将一系列通用的变更操作和流程节点创建成调度编排模板。企业就可以快速构建和管理变更流程，提高变更管理的效率和准确性，降低变更风险。

支持模板“创建，删除，更新”等管理功能，支持立即执行、定时执行、周期性执行的任务实例，支持“模板克隆”，“导入导出”等共享模板的功能，所有任务实例的执行记录均需要被保存，以便于搜索查看、安全审计和运营分析。

常用的低风险的变更操作，如针对测试环境的变更，可以将变更模板授权给非专业人员执行，即提供生成“轻模板，轻应用”的功能，限制模板的参数，生效范围等，将只能由运维/开发执行的变更操作，授权给测试/运营/策划等职能岗位使用。

变更调度编排模板在跨系统调度编排设计中可以帮助企业快速实现特定场景下的变更操作。企业可以根据自身需求，选择合适的调度编排模板，并根据实际情况进行定制和优化。

4.2.3.6 面向变更管理的可观测设计

变更是导致生产故障的主要原因，因此在变更执行过程中，实时观测并感知因变更触发的各种业务异常如告警、故障等事件，是变更执行者主动并及时解决问题，降低故障影响半径的有效方式之一。为了实现变更的可观测性，需要具备以下几个能力：

- 变更影响分析：实时感知变更对系统或服务的影响，需先建立系统的依赖关系，包括业务依赖关系、应用调用关系和资源关联关系。业务依赖关系指变更对象与周边业务服务的依赖关系；应用调用关系是指微服务之间的调用关系；资源关联关系则涵盖数据库、虚拟机、物理机等资源的关联。有了这些依赖关系，可以在变更过程中对变更对象及其关联对象进行观测，实时感知生产业务运行状态。
- 全栈可观测：建设基于 Trace、Metric、Log 的可观测能力，并将这些能力与变更影响分析结合，实现对变更相关对象在变更过程中的全面可观测。
- 灰度验证：变更通常以灰度方式进行，结合自动化验证确保每批次变更无异常后再进行下一批次变更。根据不同业务与应用特征，可设计不同的批次变更时间间隔。
- 流量闭环：全链路灰度是一种在微服务架构中实现应用版本平滑发布的技术。它不仅需要在网关处做灰度策略，还需要在微服务调用之间实现灰度。通过观

测系统，我们应该能观测到在进行全链路灰度时，相关的流量在相应的泳道内闭环，不产生逃逸。而相关实现上，需要使用到链路跟踪数据。

- 版本质量对比：通过观测手段，分析应用版本在性能、可用性、用户体验等方面的数据对比，以持续改进 CI/CD 链条质量。
- 变更运营分析：从组织、对象、运维场景等多个方面对变更的成功率和服务质量进行统计分析，发现变更脆弱或问题环节并持续改进优化。

变更流程中，通常会采取研发/SRE/运维人员盯盘监控的方式进行业务分批变更及发布上线，这样可以在业务出现变更故障时，及时发现并由 SRE 进行业务止血操作或回滚，防止变更引入的风险进一步扩大，从而影响大量用户。

以下是几个重要的监控指标，需要关注：

- SLO（Service Level Objective）指标：
- 定义和监控 SLO，明确服务的目标水平（如响应时间、可用性等），并在变更过程中实时监控这些指标，确保服务质量不下降。
- RED（Rate, Errors, Duration）指标：
- 请求速率（Rate）：监控每秒处理的请求数量，确保变更不会导致请求速率异常波动。
- 错误率（Errors）：监控请求的错误率，及时发现并处理变更引入的错误。
- 请求持续时间（Duration）：监控请求的响应时间，确保变更不会导致响应时间显著增加。
- 核心业务指标：
- 业务关键性能指标（KPIs）：根据具体业务，定义和监控关键性能指标，如交易成功率、用户登录成功率、购买成功率等。
- 用户行为指标：监控用户行为，如页面访问量、点击率等，评估变更对用户体验的影响。
- 错误日志比例和数量：
- 日志收集和分析：集中收集变更前后的错误日志，分析错误类型和数量，及时发现和解决问题。

特别要提醒的是，除了在变更时需要监控数据外，在回滚后的验证场景也需要进行密切的观测，以确保系统恢复到预期状态。

4.3 变更管理案例

4.3.1 B 站变更防控的设计与实践

SRE Elite 收录点评

该平台利用了 trace 和 CMDB 资源拓扑信息，以关联和聚合应用服务的变更，能够追踪并识别变更对整个服务生态系统的潜在影响，有助于提高故障排查的效率和准确性。

在企业内不同部门存在多套发布变更系统，且短期难以推倒统一重建的情况下，从变更防控的视角，补充建设防控平台，通过统一模型、熔断控制等方式，实现自动化的集中式变更管控。

（一）背景及设计原则

目前，在 SRE 的业界，对变更的重视程度已然达到了顶峰。在实践过程中，基本上是围绕变更方案评审，变更发起前控制（重大活动/节假日等关键时间控制），变更事件记录这些开展变更管理工作。

这种做法应对传统的 IT 架构是足够的，但是在目前围绕云原生+微服务的这套复杂技术体系来看，存在着诸多问题，诸如：

评审围绕流程驱动，在评审环节依靠人的经验来确保方案的正确性，过于依赖经验并难以持续保证效果

由于资源的稀缺性，难以确保每个变更都能够纳入评审，仅能覆盖部门高优重保的大型变更计划

评审行为仅能约束变更的发起，无法保障变更执行过程按照要求切实有效执行，比如强制分区灰度要求，强制观察时间要求等

基于人操作的变更和平台操作的变更，围绕技术平台的基础设施和业务平台的配置变更，散落在各个地方，各自对于变更用了各自的流程定义和描述，难以收归一起管理和分析

单纯的变更记录，难以支撑当下复杂场景下由某个边缘变更、底层变更导致的级联故障定位

由于缺乏完整的变更管控技术框架，难以实现公司内部所有变更的平台托管，即无法知晓到底存在多少变更行为，哪些地方会发起变更，每次变更具体是变了什么东西，变更影响具体影响了什么对象和范围等

面对以上诸多局限，秉承技术问题还需要技术解法的原则，将变更作为一个独立的技术域，抽象和定义了变更的对象概念，规划和设计了变更管控技术框架，围绕这个框架打通从变更元信息定义、变更平台/场景接入、变更防控到变更分析的全流程链路，来实现变更的一体化和多层次管控。

（二）体系设计详情

B 站的变更防控方案基于行业最佳实践，构建了工程化的变更管控体系。通过标准化描述模型和过程模型、分层策略、全方位托管、灵活感知、准入准出检查和熔断机制，实现高效变更管理和价值挖掘。

(1) 变更定义

统一变更描述模型

针对不同角色对变更的理解以及不同业务部门在语义的定义上是一定存在差异的。在 B 站，应用变更、配置变更和基础设施变更分别由不同平台承载。由于这几种变更场景对变更信息的定义存在差异，因此实现不同类型变更统一管控的目标需要抽象出一个通用的变更信息模型。这个通用的变更信息模型将会作为一个中介层，跨越各种平台和部门，以统一的方式描述和定义变更信息，确保所有参与变更的个体和系统都有共通的理解和认知，从而提高变更管理的效率和准确性。

变更描述模型应包含以下基本信息和管控信息作为元数据：

基本信息

- 变更来源：用以标识具体的变更平台和平台唯一标识
- 变更对象：包括变更对象类型(应用/数据库/服务器等)和变更对象唯一标识
- 变更时间：变更具体执行的时间
- 变更人员：变更实际的操作人员
- 变更环境：变更所生效的环境，比如 UAT/ PRE/PROD 等
- 变更内容描述：对变更操作的详细描述，包括 Release Notes 和详情描述，对于传达变更目的和实施细节非常重要。

管控信息

- 变更目标：即期望通过变更来达成的目的，包括日常迭代、BUG 修复、故障处理等;

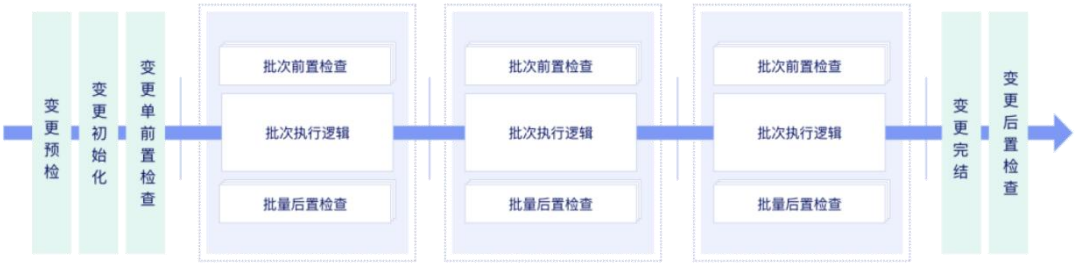
- 变更场景：指具体的变更动作，包括迭代发布、配置发布、服务器上下线等;
- 变更范围：即变更在线上环境的生效范围，包括机器分批生效模式以及流量比例分批生效等;
- 变更影响：指本次变更影响的服务和基础设施

标准化的信息模型使得后续的变更感知、变更预检、变更防御以及变更审计等操作都能够基于统一的信息结构进行，有助于降低管理复杂性，提高协同效率。此外，对于其他技术风险领域的能力建设，这个模型也为故障定位、根因分析等方面提供了通用的数据结构和框架。

统一变更过程模型

统一变更过程模型旨在提供标准化方法，以一致的方式描述和管理从提出到实施的整个变更生命周期。

这个模型通常包括以下几个关键组成部分：



一个变更的生效往往是基于一定范围和流程的，因此我们需要围绕这个过程进行变更流模型的定义。通过流模型定义，我们可以抽象出两类基本的变更流模型，分别适用于逐步生效的变更和无法拆分的变更：

1、无法拆分的变更流模型：

即一次性生效模型，适用于某些无法拆分或拆分成本较高的变更场景，例如 DB 配置类型变更。在这种模型下，变更控制防御能力主要集中在配置变更前后。

2、逐步生效的变更流模型：

按机器分批生效：将变更分为若干批次，在每个批次中逐步将变更应用到不同的机器上。这种方式允许在不同的批次上实施更多的风险管控，以便更精细地监控和评估变更对系统的影响。

按流量比例分批生效：允许按照预定的流量比例逐步引导用户访问到经过变更的系统。这种方式同样可以在不同阶段上实施风险控制，适用于需要更细粒度控制用户流量的情况。

3、变更等级分层

变更等级分层策略根据潜在影响、风险和重要性将变更分级，以便更精确地控制和管理变更过程。这样的分层策略有助于组织更加精确地控制和管理变更过程，确保关键系统和应用的稳定性和可靠性。

等级	含义	数量	生命周期定义	使用场景
G0	仅做事件同步，无管控能力	一个	仅有一个事件同步节点	适用于无任何风险的变更，但数据需要提供给相关人员做检索、审计等场景
G1	仅做单节点的变更前后切面管控	一个	生命周期中仅有一个变更节点	适用于低风险变更，无需复杂风险管控能力，仅做事前准入性和事后完整性检查的场景
G2	有完整的变更工单，且工单下关联了至少 1 个批次的子节点	多个	生命周期分为四个阶段(工单开始>各批次节点开始>各批次节点结束>工单结束)	适用于多数逐步生效的变更，并需要配套进行风险管控的场景
G3	在完整的变更工单感知基础上，增加了变更提单阶段感知	多个	生命周期在 G2 等级的基础上，前置增加了变更提单阶段	
G4	在变更提单感知的基础上，增加变更无人值守的决策能力	多个	生命周期在 G3 等级的基础上，变更提单后增加无人值守决策阶段	适用于需要系统进行无人值守代理执行变更的场景

以上某组织变更参考标准

每个管控等级的变更场景在满足前一级别的基础之上衍生出更多的变更要求和管控切面。基于标准的变更信息模型以及变更等级，我们就可以将变更管控单独抽离出来，在不同的平台业务上构建统一的变更管控平台，使得 SRE 团队能够更专注于变更的风险防控与治理，同时平台业务团队更专注于平台研发本身。

(2) 变更管控平台架构设计

变更管控平台面向不同用户提供丰富的功能，包括面向研发和 SRE 的变更信息感知、检索和订阅，以及面向平台方提供的变更场景接入、托管等能力。以下是对每个功能的具体描述：**变更信息接入：**

通过标准变更信息模型，对整个变更场景进行完整定义，确保变更场景的一致性和可管理性。支持对变更场景的全生命周期进行编排，包括计划、执行、监控和反馈，提高变更管控的可追溯性和效率。

变更信息感知：

覆盖范围广泛，平台涵盖了服务器、网络、数据库、中间件、Pass、系统发布、系统配置、业务数据等多个领域的变更信息。提供变更搜索、变更订阅、影响面分析等功能，使用户能够快速准确地获取关于变更的信息。

变更过程防控：

根据变更管控等级和防御能力，实现对变更风险的发现和阻断。提供灵活的变更防控策略，确保对不同等级的变更采用适当的防控手段。

变更过程分析：

通过深入分析变更实施的各个环节，及时发现潜在隐患，优化变更策略，确保变更的顺利进行和业务稳定。

变更结果度量：

通过对变更防控效果进行度量，包括场景覆盖率、风险检测质量、变更阻断准确率、变更防御执行效率等方面的指标。通过分析结果，持续提高变更防控的各项指标，优化防控策略和执行效率。

通过这些功能，变更管控平台不仅提供了全面的变更信息管理和感知能力，还支持对变更场景的标准化托管和灵活的防控策略。同时，通过变更分析功能，平台能够持续优化变更防控效果，提高整体的变更管理水平。

(3)变更平台/场景接入

变更场景是整个变更管控的核心概念之一，因为所有管控动作都是基于变更场景的，这里也可以理解为变更的作用域。平台层面的接入情况，主要是元信息的录入和描述过程。

在技术层面，我们主要将一个变更场景整体做了如下定义：

```
type ChangeScene struct {
    // 归属变更平台
    Platform
    // 关联变更资源类型
    SourceType
    // 基础信息(变更内容)Schema
    ContentSchema
    // 管控信息(批次内容)Schema
    StepSchema
    // 检查项
    []Navigation
    ...
}
```

```
type ChangeControl interface {
    // 变更初始化
    InitChange()
    // 变更结束
    FinishChange()
    // 变更批次开始
    StartChangeStep()
    // 变更批次结束
    EndChangeStep()
}
```

可以看到变更场景定义了一类变更，包括归属的变更平台，操作的变更资源，管控时所需的基础信息 Schema 和管控信息 Schema 以及检查项等。每一个变更都基于具体的变更

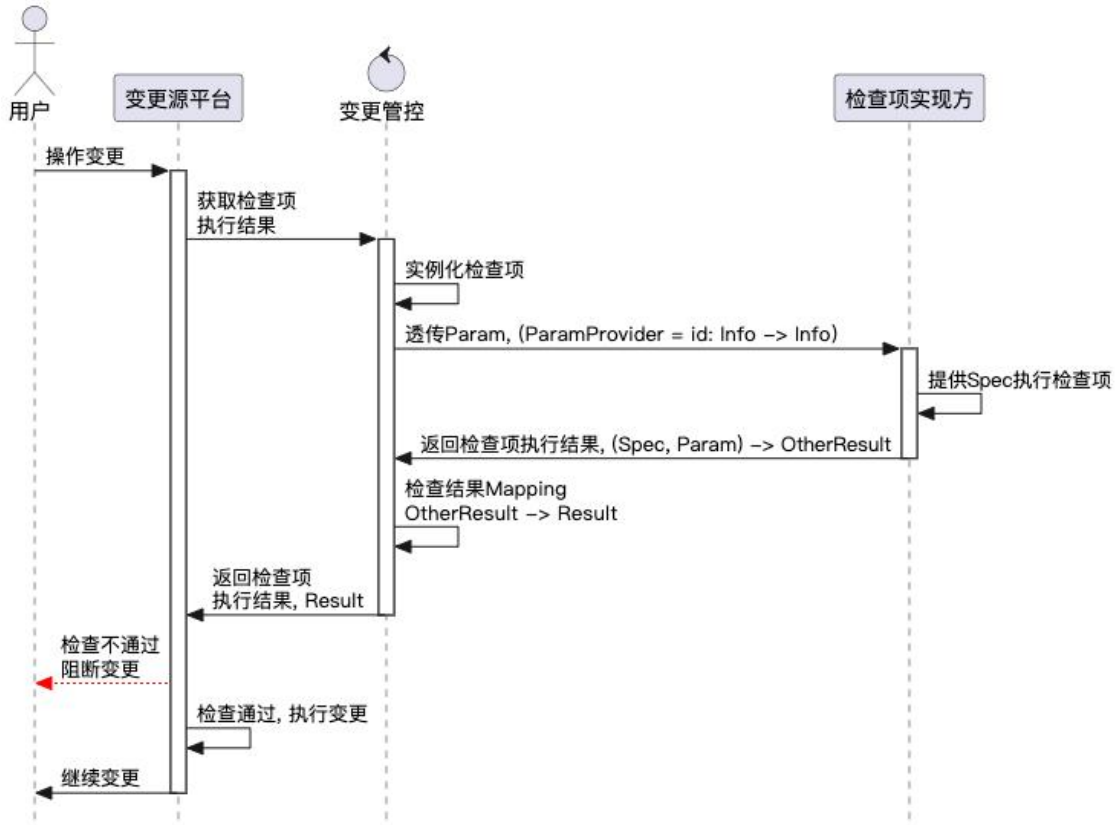
场景进行实例化，执行变更场景定义的管控动作和信息交互。因此,进行变更管控的第一件事就是定义需要管控的变更场景，变更源平台在接入自身平台后需要在 ChangePilot 上托管自身的变更场景后才能进行实际的变更管控。

(4)变更元信息托管

实现变更过程的精细化拆分，拆分前后阶段，拆分批次，拆分批次前后阶段等

基于变更单、批次单的前后阶段增加切面的检查项能力嵌入

通过检查项实现过程的检测和干预



(5) 变更感知

围绕全域变更信息的整合，系统构建了分词与关联机制，通过多元化的检索与触达手段，以实现人与系统间变更信息的灵活流通。

变更检索

在变更检索方面，系统设定了多种场景，包括针对单一对象或平台的时间跨度搜索，以及针对单个变更行为的搜索。

检索场景

- 围绕单一对象的时间跨度搜索

- 围绕单一平台的时间跨度搜索
- 围绕单变更人的变更行为搜索
- 检索的维度涵盖了变更标题和变更状态的修改，变更的源平台、对象和环境，以及操作人员和影响范围。

检索纬度

- 变更标题、变更状态
- 变更源平台、变更对象、环境
- 操作人、影响范围

(6) 变更防控

防控技术方案

变更防控方面，系统依托于变更流模型，实现了链式执行，并通过设置检查项进行准入准出。

执行策略包括观察者模式和执法者模式，以及阻断变更过程、预警提示异常、升级审批阻断变更过程等防控策略。

系统还建立了熔断机制，以应对变更系统请求失败的情况。

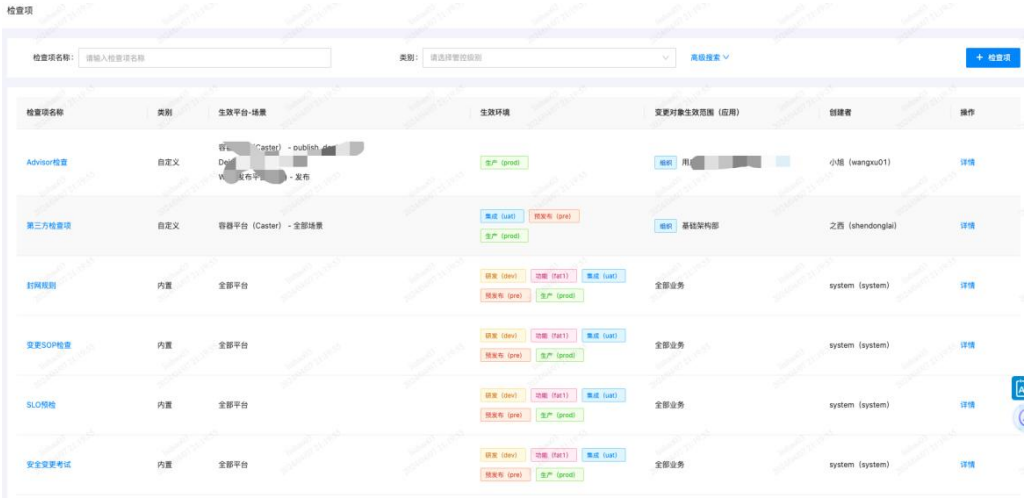
通过以上的设计，达成了变更防控的技术方案。

防控能力概述

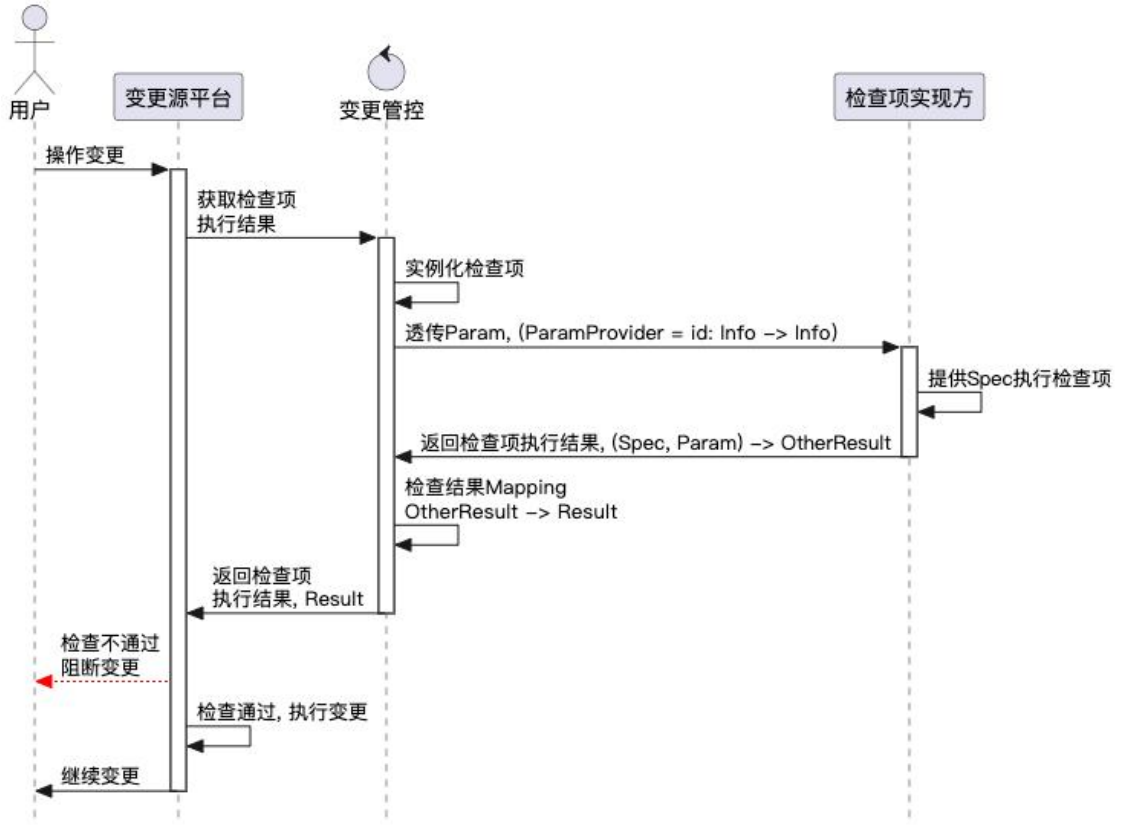
变更系统的防控能力包括通用检查项如封网拦截、SLO 检查、批次检查、环境监察等，以及自定义检查项如安全扫描、业务代码发布 SOP 等。这些检查项旨在防止不符合标准的变更进入生产环境。

举例一，有业务进行灰度变更之后， SLO 迅速下降，这个时候系统就会进行拦截，阻止业务继续进行发布， 避免导致更大的故障。 另外， 可能的场景包括，批次检查避免用户不进行任何的灰度等。

举例二，安全团队，希望对发到线上的制品包有一个安全扫描，则可以使用构建自定义检查项，避免有安全风险的制品流入到生产环境中。



技术交互模式



(7) 变更价值挖掘与可视

通过多维度视角对变更数据进行挖掘，以发现风险点，明确优先级，并可视化项目价值。

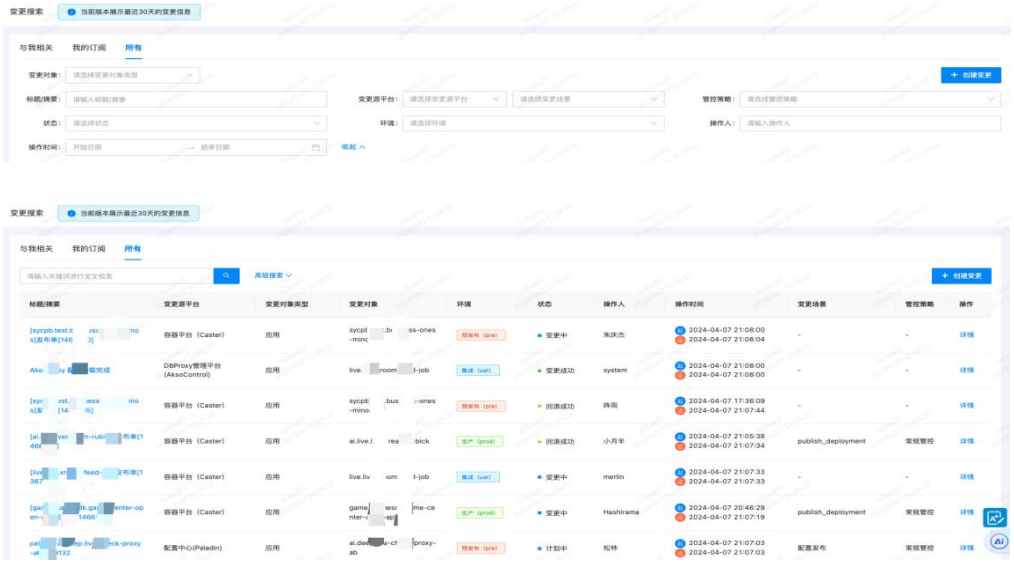
运营面板和检查项面板为团队提供了实时的数据视图。

运营面板

检查项面板

(8) 开放能力

开放能力包括全量或按纬度的变更消息投递，便于让上下游系统对变更相关消息进行记录并消费。例如，可以与监控系统或审计系统进行联动。



- 变更订阅

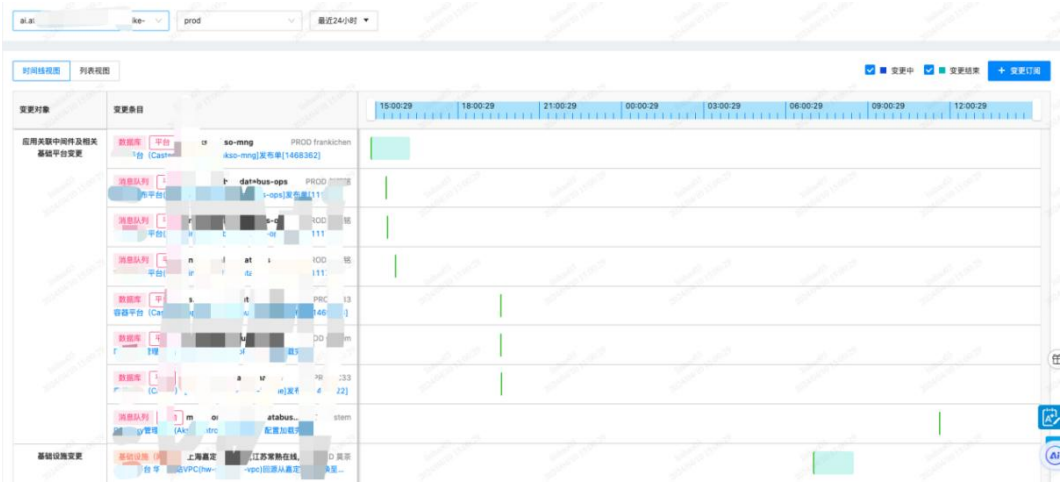
订阅服务允许用户根据服务、业务、组织等多维度进行订阅，包括自身变更、上下游变更，以及变更的平台和场景。触达机制涵盖个人、群组、webhook 等多种方式。

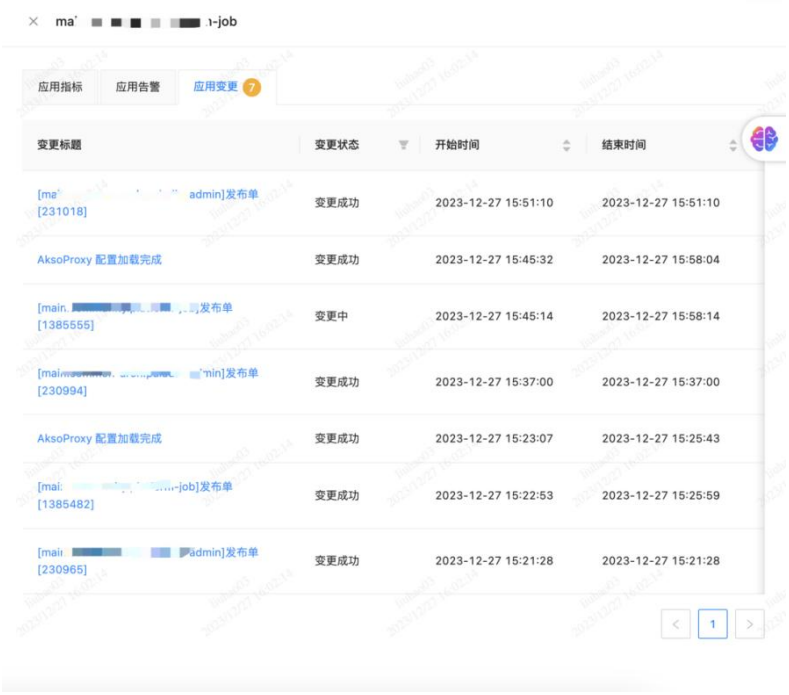
- 变更分析

通过对变更故障的深入分析，我们发现很大一部分异常并非源自本服务的变更，而是由上下游依赖或更基础的设施变更所引起的。为了解决这一挑战，ChangePilot 利用了 trace 和 CMDB 资源拓扑信息，以关联和聚合应用服务的变更。

具体而言，我们利用 trace 分析获取了应用服务的上下游关系，同时通过 CMDB 资源拓扑信息分析得到了应用服务所管理的资源架构，包括存储层（如数据库集群、缓存集群等）、计算层（包括容器、物理机实例）以及网关层等关键资源信息。随后，通过统一的资源标识符进行检索，我们能够有效地聚合变更信息。

这种方法使我们能够更全面地了解应用服务的生态系统，包括其上下游关系和底层资源架构。通过整合这些信息，ChangePilot 能够追踪并识别变更对整个服务生态系统的潜在影响。这不仅有助于准确定位变更引起的异常，还提供了更深层次的可视化和理解，有助于提高故障排查的效率和准确性。




应用指标
应用告警
应用变更 7
变更标题
变更状态
开始时间
结束时间
[231018]
变更成功
2023-12-27 15:51:10
2023-12-27 15:51:10
AksoProxy 配置加载完成
变更成功
2023-12-27 15:45:32
2023-12-27 15:58:04
变更中
2023-12-27 15:45:14
2023-12-27 15:58:14
[1385555]
变更成功
2023-12-27 15:37:00
2023-12-27 15:37:00
[230994]
AksoProxy 配置加载完成
变更成功
2023-12-27 15:23:07
2023-12-27 15:25:43
变更成功
2023-12-27 15:22:53
2023-12-27 15:25:59
[1385482]
变更成功
2023-12-27 15:21:28
2023-12-27 15:21:28
[230965]

（三）总结及展望

经验教训

在变更管理的实践中，我们总结出了五个关键日拱一卒的经验教训：充分设计、找准切入、兼顾效率、保持效用。以下是针对这些教训的优化建议：

充分设计：在实施变更管理前，必须进行彻底的规划和设计。这意味着需要确保变更设计的可持续性和兼容性，避免频繁修改导致合作方的抵触。设计阶段应考虑软件的建模标准，确保内部逻辑的变更不会影响外部接口。

找准切入：在变更管理的初期实施阶段，应选取高频且容易出错的场景作为切入点，以此构建成功案例并推广。中期阶段，将成功模式推广到整个组织，包括低频但高风险的变更场景。通过展示数据驱动的成效，增强变更方案的吸引力。

兼顾效率：在变更过程中，确保不过度干预日常业务和开发工作。通过设置如绿色通道和白名单等机制，优化变更流程，减少对业务运行的影响。其中，绿色通道，主要用于快速止损，仅豁免 1 个小时，即开即生效。而白名单主要用于长时间豁免管控规则，一般用于活动期间的保障场景下一旦出现问题需要快速止损的情况，需要事前提交申请，并审批通过。

保持效用：变更管理应注重实际效用而非理论上的改进。避免过度设计检查项，应聚焦于实际需要，确保变更控制的举措既有效又实用。同时，防止在没有明确需求的情况下引入不必要的变更。

日拱一卒：持续的投入和改进是确保变更管理成功的关键。应建立长期的变更管理机制，将变更管理的标准和实践融入日常操作，逐步培养组织内对变更管理的认可和依赖。

未来展望

愿景：实现 B 站变更防控体系的充分性、高效率和有效性。持续保稳提效，在复杂多变的业务情形下借助数据&智能建立可持续的变更防控机制。

4.3.2 携程云平台基础设施变更管理实践

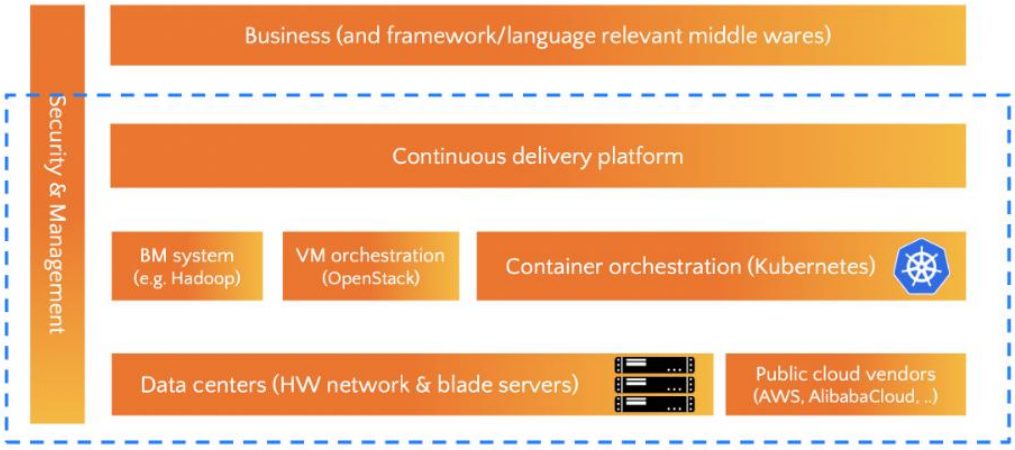
SRE Elite 收录点评

1，这是一个由混合云 SRE 团队提供的案例，作为基础设施质量的把控者，对变更的计划性和标准作业流程要求严格，对于企业内私有云 SRE 管理团队有一定的参考价值。

2，大量使用了 IaC 的方式对基础架构进行管理及变更，其中使用 SaltStack 和 StackStorm 管理配置变更，使用 Kustomize 和 Git Workflow 管理多个集群和环境的基础组件配置文件，对公有云资源使用 Terraform 进行管理，实现了全栈 IaC 落地。

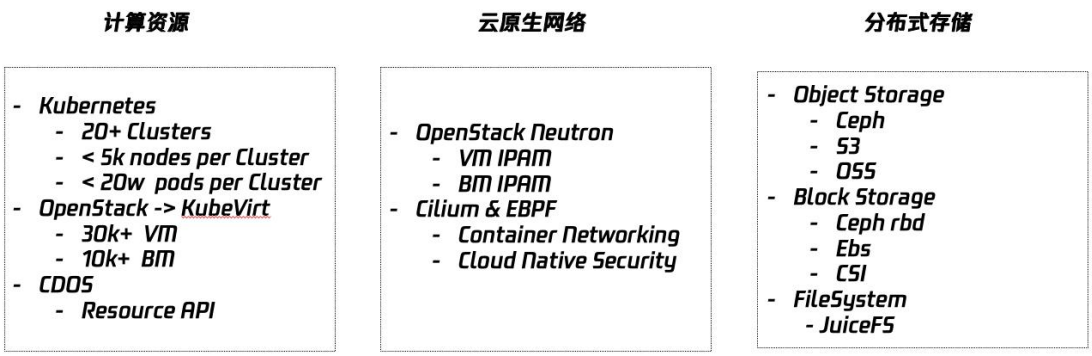
（一）背景及设计原则

携程云平台概览



携程基础设施采用混合云架构，云资源分布在自建数据中心和公有云供应商。携程混合云平台的 IaaS 层提供裸金属管理、虚拟化宿主机（KVM），以及容器宿主机等多种底

层资源类型。其中容器通过 Kubernetes 进行编排和交付。更上一层则是各种业务 PaaS 平台，诸如在线应用、数据库（DB）、ElasticSearch、Kafka 等服务。



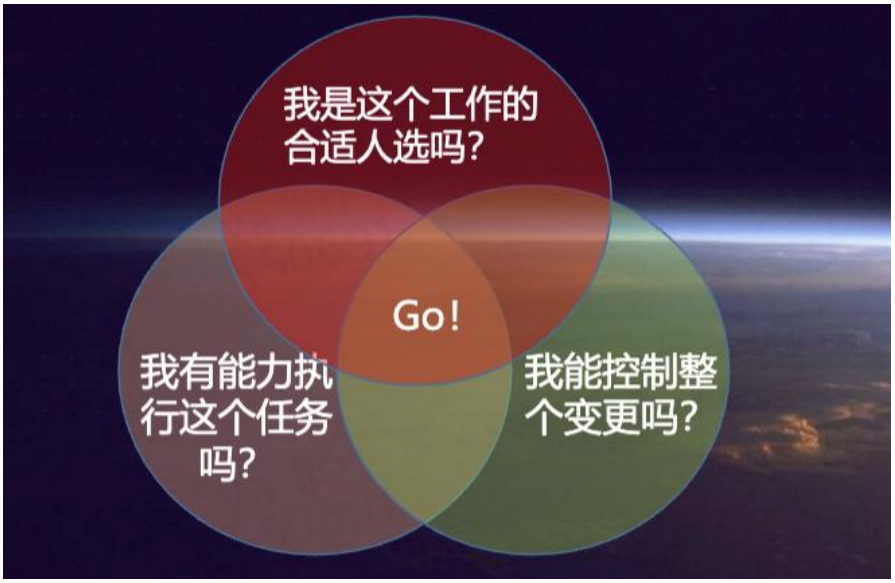
在携程的技术架构中，由云技术团队负责主要计算、网络和存储三大资源类型的管理和交付。在计算资源方面，携程云团队管理超过 20 个 Kubernetes 集群，每个集群的节点数控制在 5000 个以下，以减少单一集群故障对整体的影响。在虚拟化技术方案，携程云团队从使用 OpenStack 过渡到 KubeVirt 进行 VM 资源交付，目前运维管理约 3 万台虚拟机实例。

网络虚拟化技术方面，携程云团队使用 OpenStack Neutron 和 Calico 来管理虚拟机和裸金属服务器的网络。容器网络则主要基于 Cilium 和 eBPF 技术。携程存储解决方案在私有云中主要使用 Ceph，而公有云存储则采用 S3、OSS 等对象存储。文件存储方面，携程采用 JuiceFS 和 FastDFS 来满足不同的业务需求场景。

（二）基础设施变更管理详情

变更失败主要由两大因素引起：人为因素和流程因素。

首先，人为因素并非指操作人员故意造成故障。很多时候，故障是因为变更人员未能遵循既定规范和操作流程，导致变更失败。因此，携程要求变更实施人员在操作之前，自问自答如下三个核心问题，也称为“三步法则”：



- 是否为合适人选：确保变更人员对任务有充分的理解和背景知识，而不仅仅是盲目执行上级的指令。
- 是否具备必要能力：变更人员需评估自己是否有能力设计出完整的变更方案，包括发布计划和应急恢复计划。
- 是否能控制整个任务：在变更过程中可能会遇到意外问题，变更人员需要有能力主导问题的解决和恢复。
- 讨论这三步法则的目的，并不是完全消除人为因素导致的故障，而是为了最大限度减少人为因素的影响。
- 流程因素也极为重要。在设计变更时，应遵循几个基本原则：
- 计划性：技术设施的变更应严格按照计划执行，与普通应用的变更略有不同。
- 灰度原则：灰度执行变更，先小范围后大范围，以控制风险。
- 可回滚性原则：确保每次变更都可以安全回滚至变更前的状态以应对可能的故障。
- 避免循环依赖：变更过程中应避免循环依赖问题。
- 风险评估机制：建立系统的风险评估流程，确保变更的安全性。

计划性原则

与普通应用的变更相比，基础设施变更应该严格按照计划执行，典型例子如Kubernetes 4个月的发布模式，在此参照下，基础架构进行变更前，更应该制定更严格的计划。

灰度原则

- 严格按照 Availability Zone (AZ) 定义，控制变更的灰度
- 不同 AZ 的服务，不依赖同一套基础设施服务
- 控制基础设施变更的爆炸半径
- 生产不同 AZ 的基础设施变更，时间上必须间隔 1 天
- “慢即是快”

携程采用了混合云架构，并建设了国内的同城三中心高可用体系。这意味着我们的基础设施变更必须严格遵循三个可用区（AZ）的原则，确保每个 AZ 的服务都具备高可用性。即使一个 AZ 出现故障，也不会影响到其他两个 AZ 的正常运行。此外，为了控制变更的风险范围，每个 AZ 所依赖的基础设施必须是独立的，不能有共享的部分。

最关键的一点是，不同 AZ 之间的基础设施变更需要有足够的观察期。携程云团队规定，任何基础设施的变更后，必须至少经过一天的时间来观察其效果。变更后需要经历至少一个流量高峰期来验证变更的稳定性，是否能抗住峰值流量压力。尽管看似放慢了变更速度，实际上这种方法能更有效地确保变更的质量和安全，避免了因急于求成而可能带来的风险。这种做法体现了一个重要原则：慢是快，即通过细致周到的计划和执行来实现更快、更稳定的长期发展。

可回滚原则

- 基础设施变更比较复杂，工程师容易忽略回滚策略
- 测试环境需要验证回滚步骤是否可执行
- 部分极端情况，可以将新建集群作为回滚方案
- 依赖快速构建环境的能力+数据备份+配置备份
- 典型的回滚策略
- 代码和配置文件都需要版本管理，杜绝仅回滚镜像
- 引入中间版本
- 谨慎删除旧字段

关于变更管理中的可回滚原则，这是一个在实践中经常被忽视的重要方面。在进行变更时，团队会制定一个回滚计划，但仅在测试环境或小范围灰度发布时未发现问题，往往不会去实际验证回滚计划的可行性。因此，携程云团队强调在测试环境中必须验证回滚策略的有效性。在一些极端情况下，比如之前升级 Ceph 集群时遇到的问题，集群无法启动，不得不重建集群并迁移数据，显示出在紧急情况下能够重建整个集群的能力是多么重要。

携程云团队推崇代码与配置的分离原则，在进行变更时，尽量避免同时修改代码和配置，避免定位困难。

另外，考虑，引入中间版本，在数据库变更中，最佳实践是先添加新字段，迁移数据，然后再删除旧字段，而不是直接修改字段属性。这样做可以减少回滚的复杂性和出错的风险。

总之，确保变更可回滚性不仅是为了应对可能的失败，也是确保整个系统可维护性和稳定性的关键措施。通过这些策略，能够更有效地管理技术变更，减少潜在的业务中断风险。

杜绝循环依赖

- 核心基础设施间的依赖需要定期梳理
- 核心基础设施自愈能力
- 网络&虚拟化控制器，做到无外部依赖自启动
- 容易忽略的依赖项:
- DNS
- 分布式存储

在管理基础设施时，强调减少循环依赖的重要性，因为这些依赖关系往往会导致系统复杂化和潜在的运维问题。为此，携程云团队定期梳理核心组件的依赖关系，并确保这些关键组件具备自愈能力。特别是在网络和虚拟化控制器这两个方面，历史上曾经经历过断电事件、电力恢复后，理论上业务应该立即回复，但实际上却因为这两个核心组件未能及时启动而导致整个系统无法运作。这一经历强调了底层系统的自主性和弹性的重要性，它们需要能够在没有外部依赖的情况下自启动。

此外，在进行技术变更时，经常被忽视的两个关键领域是 DNS 和存储。这些系统虽然在日常操作中看似稳定，但在变更过程中，任何对这些系统的忽视都可能导致严重的业务影响。

杜绝循环依赖

梳理基础设施管理服务，定义 P0 / P1 / P2 服务标准

针对 P0 业务，建立定期风险 review 机制，review 内容包括：

- 故障爆炸半径
- 业务架构
- 业务部署规范
- 业务故障预案

针对 P0 业务，进行故障演练，验证是否满足服务 P0 级定义，并建立常态化演练机制

为了确保核心应用和基础设施的稳定性和可靠性，携程云团队建立了一个详细的风险评估机制，并对服务进行了等级划分，包括 P0、P1 和 P2 级。这些服务级别的定义基于过去几年中对系统故障影响的分析，其中 P0 级指的是对业务影响最大的系统。

对于 P0 级服务实施定期的审查，包括几个关键方面：

故障爆炸半径：评估单个故障点对系统的影响范围，确保高可用性，即单个可用区（AZ）的故障不会影响到其他区域。

业务架构：分析业务依赖的系统和组件，确保这些依赖项的 SLA 也符合高可用要求。

部署规范：检查部署的规范性，确保所有操作不会发生预期外的行为和影响。

标准操作流程手册（SOP）：经过定期演练的标准操作流程手册 SOP，确保应用的所有者或管理者在发生故障时，可以依据 SOP 迅速恢复服务。

此外，携程云团队每年会进行实际的突击演练，例如选择性断开一个机房的网络连接，来测试其他机房是否能够独立承担起 100%业务流量。这种演练帮助携程云团队验证系统的实际表现与预期是否一致。

除了这些集中演练外，携程云团队还鼓励应用的所有者通过混沌工程的方式，定时注入模拟的网络故障（如断网、丢包、网络延迟增大），不断验证并提升自身应用的韧性，确保真正满足 P0 级服务的要求。这种主动的风险管理和应急演练是确保业务连续性和系统稳定性的关键措施。

(1) 基础设施变更计划 – 日常运维变更实施方法

云组件部署机器调整、底层服务参数调优、DR 调整、域名及访问入口调整、备份机制优化、日志轮转调整、内核参数变更等 IAAS 组件日常运维管理相关变更，适用于如下变更流程及变更原则:

1. 生产环境计划内变更，需要在每周四的周会前提交 Release Plan，周会上 review 变更计划。采用一票否决制，有任何不确定因素及反对意见的，变更搁置；

2. 变更计划需要有明确的回退步骤；

3. 对生产环境应用有一定概率产生网络中断等影响的变更，需要安排 outage 窗口、通过评审后实施；

4. 对发布系统有较大概率产生影响时间大于 15 分钟的变更，提前一天做好用户通知报备工作。

基础设施变更模板

- 变更计划由变更委员会 Review
- 关联 Review 通过且 Merged 状态的 Merge Request
- 详细的 Rollout & Rollback Plan
- 需要在发布窗口期内变更
- 实时通知 NOC 变更状态

@sys-ccb （注意：部分项目是 @ sys-dp）

产品

- 填写变更的项目名称（必填）

Changelog

- 填写变更内容（必填）

merge_request

- 填写 merge_requests 链接 （必填）

Migrations

- 根据是否需要做 migration 填写

关联系统

- 影响到的其它系统

发布环境

- 填写发布的环境

Rollout Plan

- 填写发布操作步骤 （必填）

Rollback Plan

- 填写回退步骤 （必填）

发布窗口

- 填写计划发布的时间

实际开始日期

- 填写实际开始发布的日期（必填）

实际结束日期

- 填写实际结束发布的日期（发布周期大于一天时，必填）

预计发布时间

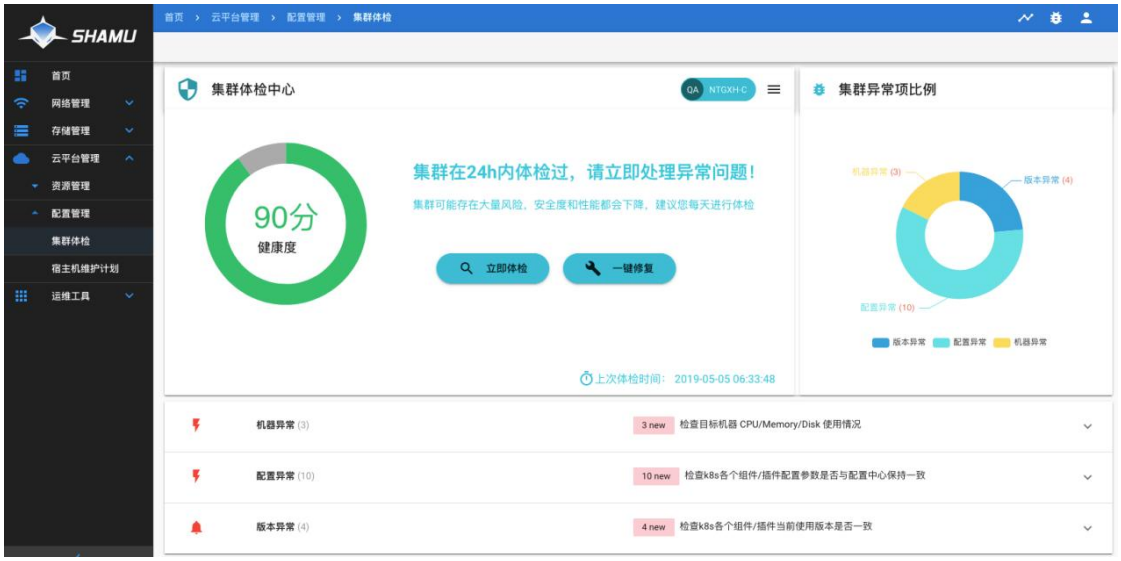
- 20 分钟

发布状态

- [ ] 发布中
- [ ] 成功
- [ ] 回退

携程变更模板示例

一致性巡检



配置不一致是万恶之源，为了解决这一问题，携程云团队投入了大量精力确保所有系统配置的一致性。携程云团队开发了一款基础工具，专门用于监控和检测宿主机上的 Kubernetes 组件配置是否一致。检查项不仅包括各组件的配置，还涵盖了内核参数以及各服务的配置设置。

该工具会定期进行巡检，并将检查结果反馈给相关团队，以便及时发现并解决任何配置不一致的问题。

基础设施变更计划 – 基础设施大版本升级

升级注意事项

https://docs.cilium.io/en/v1.12/operations/upgrade/#id3

1. cilium-operator/cilium-agent 默认 Prometheus metrics 端口号发生了变化
   - cilium-agent 9090->9962
   - cilium-operator 6942->9963
2. 在 Azure IPAM 模式下，--azure-use-primary-address 默认是关闭的，即不使用 ENI primary ip，aws --aws-use-primary-address 也是一样的，需要注意开启
3. 公有云 operator 能够支持通过 instance-tags-filter 配置过滤 EC2 实例
4. cilium-operator pprof 相关的配置进行了重命名
   - pprof -> operator-pprof
   - pprof-port -> operator-pprof-port
5. kube-proxy-replacement 配置里的 probe 选项被标记成废弃，将会在 1.13 中移除
   - https://docs.cilium.io/en/v1.12/gettingstarted/kubeproxy-free/#kube-proxy-hybrid-modes

以上是 Cilium 组件 v1.11 升级 v1.12 的差异分析报告

要进行技术基础的大版本升级，需要非常的谨慎，以下是升级过程中的关键点。

变更内容概述：首先明确升级中哪些内容发生了变化。

参数更新：列出与之前版本不同的参数设置。

功能新增：介绍升级后新增的功能。

功能减少：说明此次升级中减少的功能。

在实施变更时，携程云团队采用持续部署（CD）模式，确保每项变更首先在测试环境中验证，并进行回滚测试。之后，变更会在生产环境中逐个可用区（AZ）实施，每次变更后观察一天以确保稳定性，然后继续在其他 AZ 执行。

另外，如 ECTD 版本升级，K8s APIserver 版本升级，也可以遵循此逻辑。

这样的管理策略旨在确保每次技术变更都是可控和安全的，同时通过严格的测试和分阶段部署来最小化风险。

(2) IaC 实践: StackStorm & SaltStack

为了保证配置和变更的一致性，携程云团队使用了多种配置管理工具，如 SaltStack 和 StackStorm，来管理配置变更。通过 SaltStack 管理基础组件的配置，并通过 Git 追踪配置版本的变化。为了增强系统的安全性和稳定性，携程云团队要求不同的可用区之间不得共享配置文件，这样即使发生配置错误，也只会影响单一的 AZ，而不会波及整个系统。

以下是我们的一些实践

通过 SaltStack 管理云平台基础设施服务

SaltStack Formula & Pillar 机制，标准化配置和敏感信息分离

不同 AZ 不共用 SaltStack Master，缩小变更操作的爆炸半径

Cilium Agent 不依赖 K8s DaemonSet, 用 SaltStack + docker-compose 管理 Cilium 服务

通过 SaltStack 管理云平台基础设施服务

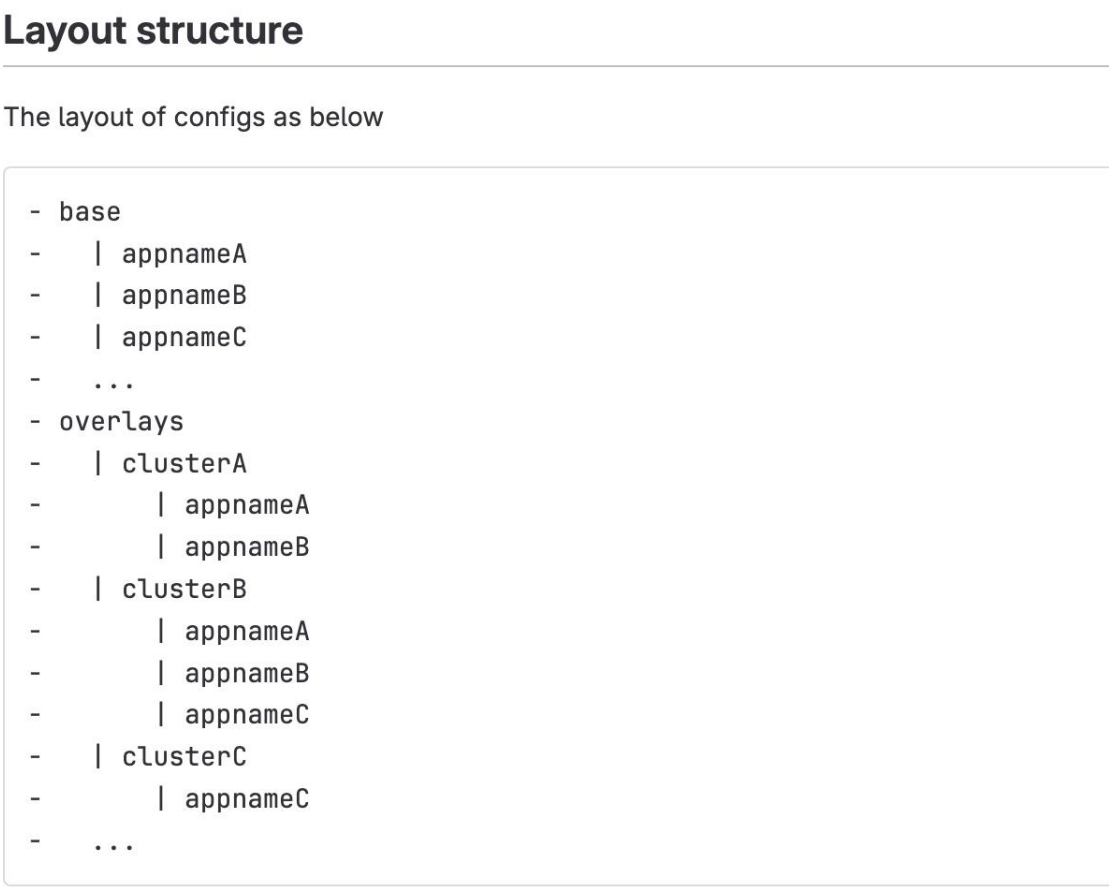
使用 StackStorm 提供的事件驱动和 runbook 技术栈，实践 IaC

Chatbot & Stackstorm 工作流对接

机器配置更新依赖 SaltStack 提供的能力

对 Stackstorm 执行情况进行深入分析，持续优化运维变更的 workflow

(3) IaC 实践: 云基础组件变更管理 GitOps



代码分层结构

使用 Kustomize 来管理多个集群和环境的基础组件 Yaml 配置文件

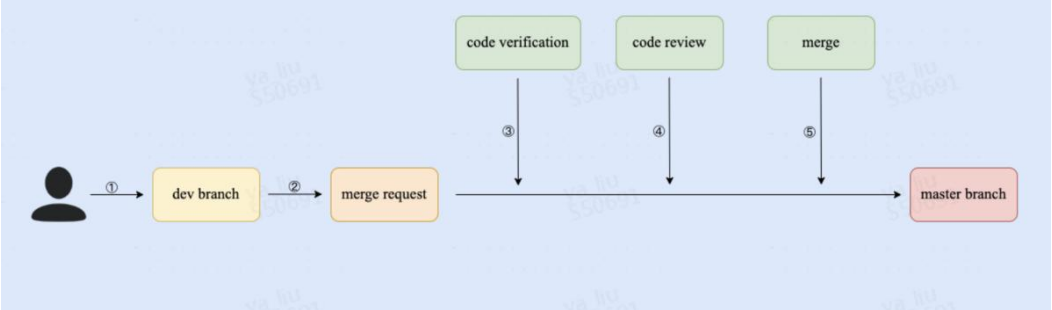
使用 Git Workflow（Tag/ MergeRequest）进行版本控制

基于 GitlabCI，CI/CD Pipeline 将基础组件部署到集群中

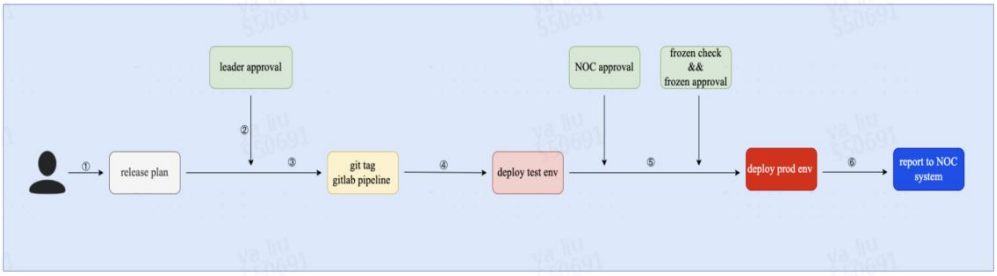
随着云原生技术的推进，携程云团队将更多基础组件转变为 Operator 化部署，几年时间里从最初的十几个增长到现在的五六十个。这些 Operator 的发布和变更管理，采用了 GitOps 技术来落地。

在携程云团队的实践中，所有组件统一存放在一个主仓库中，然后根据集群的维度对每个组件进行 layer 拆分。携程云团队使用 Kustomize 来管理不同集群和环境下的基础组件 Yaml 配置文件，确保配置的灵活性和可维护性。

版本控制方面，携程云团队采用 Git Workflow，包括使用标签（Tag）和合并请求（MergeRequest）来管理代码的版本。这样的配置管理（CM）模型较为简单：开发人员开发完毕后提交代码，进行代码检查和审查，之后提交发布计划供上级审批。



CI 流程



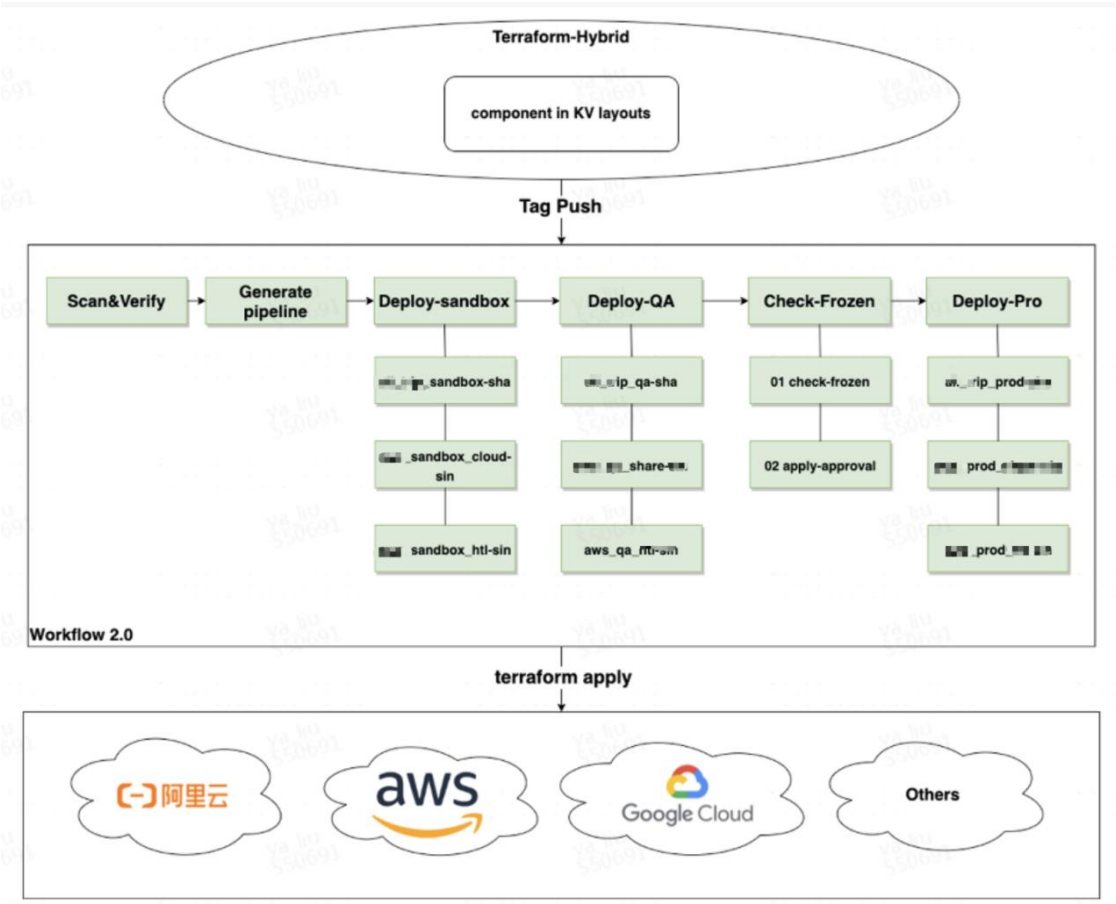
CD 流程

在自动化部署方面，携程云团队基于 GitlabCI 实施 CI/CD Pipeline，将基础组件部署到指定的集群中。在发布过程中，除了需要上级的审批，还必须确保发布操作在预定的维护窗口内进行。最终，产品的部署严格按照单个可用区（AZ）逐一进行，确保发布流程的安全性和可控性。

这样的策略通过分步骤、逐一区域的方法降低了整体风险，确保了系统的稳定性和安全性。

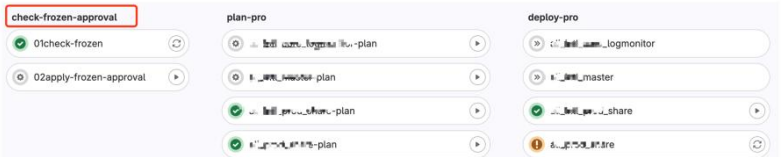
(4) IaC 实践: Terraform

Terraform 目录结构



Terraform 部署流程

携程云团队在公有云资源管理中采用了 Terraform，并结合 GitOps 进行管理。这种方式与之前讨论的 Operator 管理类似，携程云团队按照组件的维度来组织 Terraform 配置文件的目录结构，这样做可以有效避免多人同时开发时产生的冲突。



Terraform Release Pipeline

以上的 Terraform 的工作流程包括几个关键步骤：首先是代码检查，确保配置的正确性和最优化；接着是在 Sandbox 和 QA 环境中进行部署。Sandbox 环境的使用场景主要是在引入新资源时，不应直接在测试环境部署，而应先在 Sandbox 环境中进行验证。验证无误后，再将资源部署到测试环境和生产环境。

在生产环境部署时，携程设有一个“冻结审批”机制。如果部署在维护窗口期内进行，通常可以直接通过；如果在窗口期外，则需要直属领导的审批才能执行部署。这样的安排旨在确保部署的安全性和责任明确，以便在出现问题时能有明确的责任归属。

（三）总结及展望

切记灰度，切忌不做回滚方案，面向灾难设计架构

AI + ChatBot 可以承担更多的日常工作，对于变更可能造成的风险进行智能化提示

对于核心组件定期 Review，主动升级，避免跨多个大版本的升级操作

不畏惧基础设施变更的前提:

具备持续变更能力

对于基础设施有深入的研究和理解

实践出真知

首先，灰度发布是基础设施组件变更不可或缺的环节。在逐步推出新功能的同时，降低风险可能带来的影响范围。其次，必须制定并验证回滚方案。这确保在遇到问题时，变更操作人员可以迅速恢复到之前的稳定状态。

此外，基于 AI 技术，比如聊天机器人，可以帮助工程师进行更多的风险分析和提供智能化的提示。这些工具的应用能够更有效地管理和预测潜在的问题。

第三，对核心组件的定期审查和主动升级非常关键。工程师应避免跨多个版本进行升级，就像之前遇到的从 Kubernetes 1.3 升级到 1.19 过程中遇到的各类挑战。目前，团队面临的挑战是如何将 1.19 升级到 1.28，因为长时间未升级导致积累了许多待解决的 bug。

在进行基础设施变更时，持续的变更能力、深入研究和理解基础设施是必需的。此外，实际操作比理论讨论更为重要，故障演练是一种很有效的风险识别方法。

在人为因素不可避免地影响变更时，工程师应当正视错误，而不是试图掩盖。作为变更执行者，如果出现问题，应该坦诚承认。同时，作为领导者或管理者，应当避免简单责怪下属，而应反思系统设计是否足够健壮，以减少人为错误的发生。

4.3.3 某银行变更管理设计与实践

SRE Elite 收录点评：

这是一个银行变更管理的案例，需要在符合监管的前提下保障可靠性，并对风险控制进行了量化。案例定义了应用和基础设施等变更风险的评分模型和积分制度，并结合评审进行准入控制。

该银行业务变更涉及多方企业人员关联，流程环节复杂，对变更管理所涉及的过程，设立了配套的组织架构闭环治理，例如变更自动化率、耗时、风险评估准确性及流程审批效率等。

（一）背景及设计原则

背景及挑战

通常银行业务信息系统的业务研发团队和 SRE 团队分属于不同的部门，对应的持续集成（CI）和持续交付（CD）也是由两个团队分别建设，中间通过流程管理平台（ITIL）联动，任何变更，通过审批后，才可以部署到生产环境。

相比于互联网行业以快速响应市场变化和用户需求为核心，强调技术敏捷、系统安全和数据保护的变更管理，银行行业受到金融监管机构的严格监管，银行信息系统的变更发布关系到银行业务的连续性和系统稳定性，因此，传统银行的变更管理通常参照 ITIL 标准开展实践，以变更流程的合规性、变更风险管理的谨慎和严格为主。

然而，随着互联网金融的兴起和商业银行数字化转型的加速，银行业务部门对业务系统的变更管理提出了更高的要求。他们期望变更管理不仅要确保变更的稳定性和安全性，同时也要能够快速响应市场和用户需求。这种“既要稳、又要快”的双重要求，无疑给银行信息系统的变更管理带来了更大的责任和挑战。

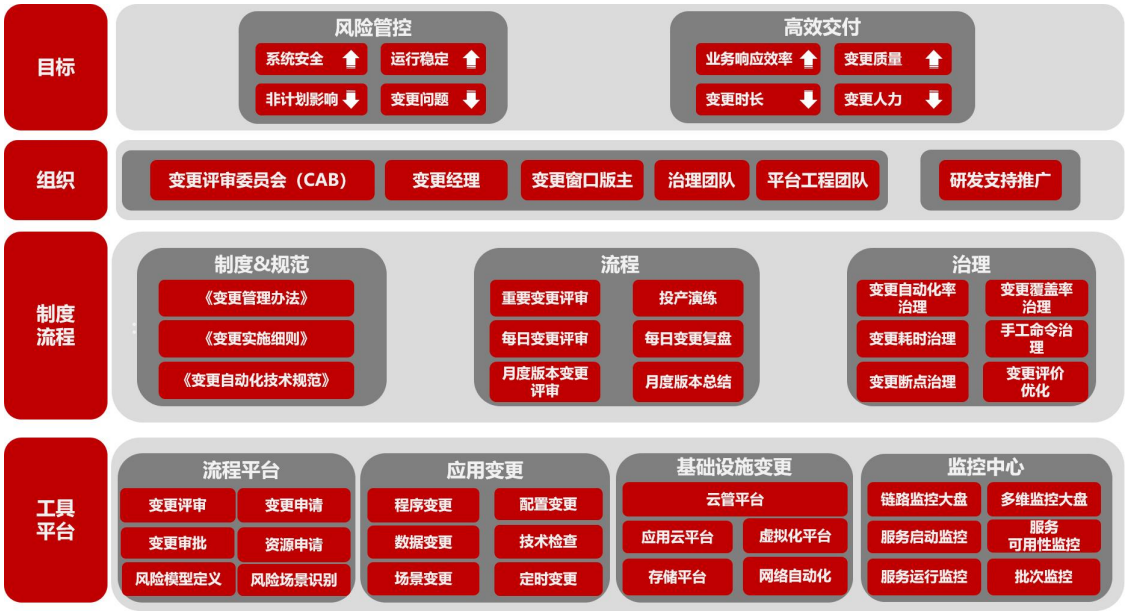
面对这种挑战，银行需要在保持流程合规的基础上，利用先进的技术手段提高变更自动化水平，提升变更的效率和质量，同时，进一步优化变更管理流程，实现变更风险线上化、数字化，强化变更风险的识别和控制，以满足不断变化的市场和用户需求，支持业务快速创新。

设计原则

某银行生产环境，约 700 多个应用，3 万多台 OS，2 万多个容器。从 2013 年开始进行应用变更自动化的建设，最初目的是减少手工操作性工作。

平台能力从最初的应用程序更新自动化，逐步扩大到配置更新、数据更新、技术检查自动化，再到与流程管理、配置库、监控等生态平台的联动，资源交付自动化。

当前，变更管理以变更风险控制、高效交付作为变更建设的主要目标，通过组织、流程、技术平台的持续演进以及常态化的变更治理实现对变更全生命周期的闭环管理。

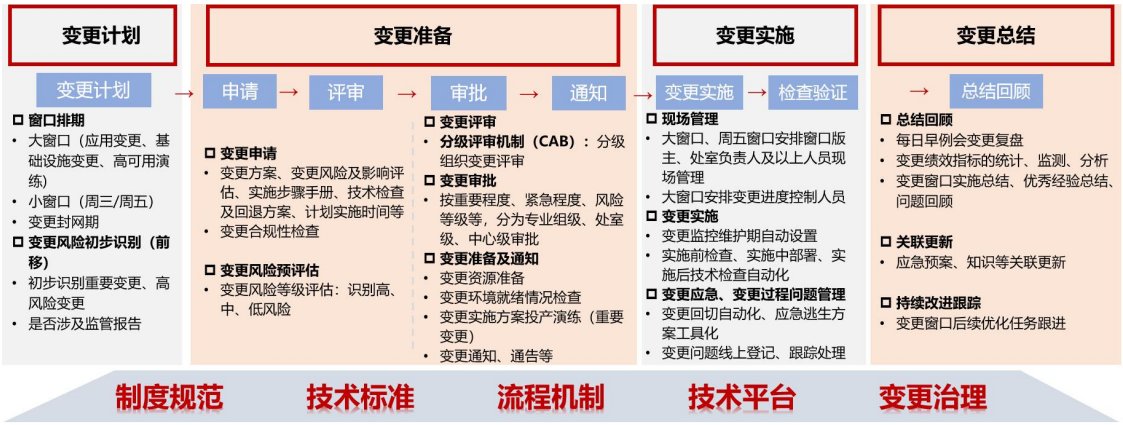


某银行变更管理总体思路和设计如上图所示

- 变更组织围绕变更的核心业务流程进行设计，由数据中心牵头组建，研发中心支持推广团队（注：研发支持推广团队是数据中心与研发中心融合的桥梁）共同参与，主要职能是制定变更相关的制度、规范，组织变更计划制定、评审、总结，进行变更管理相关工具平台建设，开展变更相关的治理活动，驱动变更流程的高效、高质量执行；
- 变更相关制度流程在满足监管要求的前提下，主要目标是指导变更管理、变更治理工作开展，制定变更流水线编排等技术实施规范，是变更技术实现的补充；
- 工具平台则是以风险管控自动化、变更执行自动化、检查自动化为目的，并尽可能将管理手段转变成技术实现。

（二）体系设计详情

某银行将变更流程分为计划、申请、审核、审批、通知、实施、验证、总结 8 个阶段，并围绕这 8 个阶段重点开展变更风险管控以及全流程自动化治理。



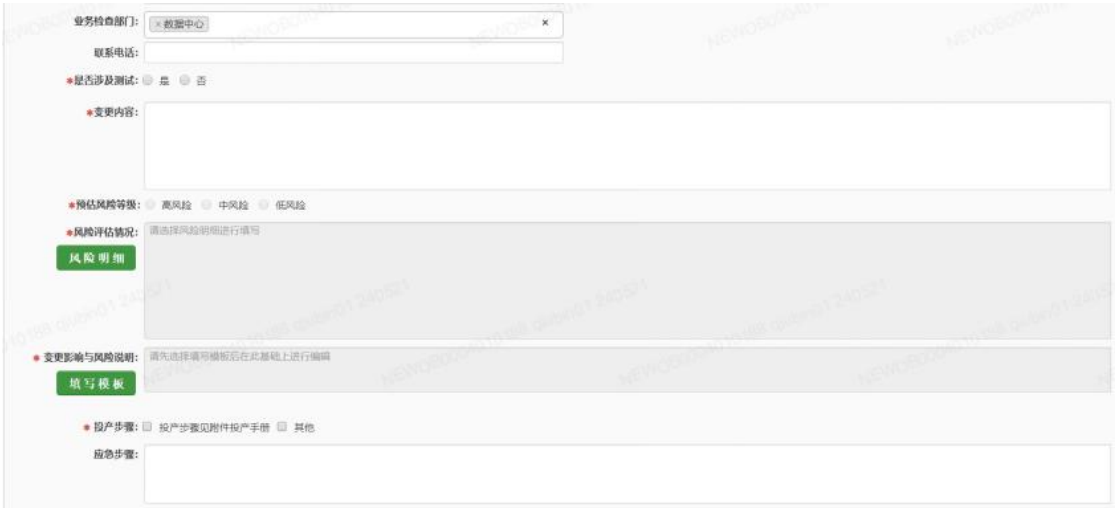
下面主要介绍变更风险管控、全流程自动化的建设及管理思路：

(1) 变更风险的识别和控制

1.变更风险清单化、场景化、线上化管理

通过定义应用变更、基础设施变更风险评分模型，并纳入变更申请的风险预评估环节。对于重要变更，还需要对变更准备情况、变更准入条件、非功能性需求进行评审。

下图为变更申请工单风险评估部分，预估风险等级根据风险评估情况自动生成。



应用类变更内容的往往更加复杂、动态，因此，应用变更的风险预估模型适合采用积分制。某行从**系统重要性**

、变更自动化程度

、部署方式

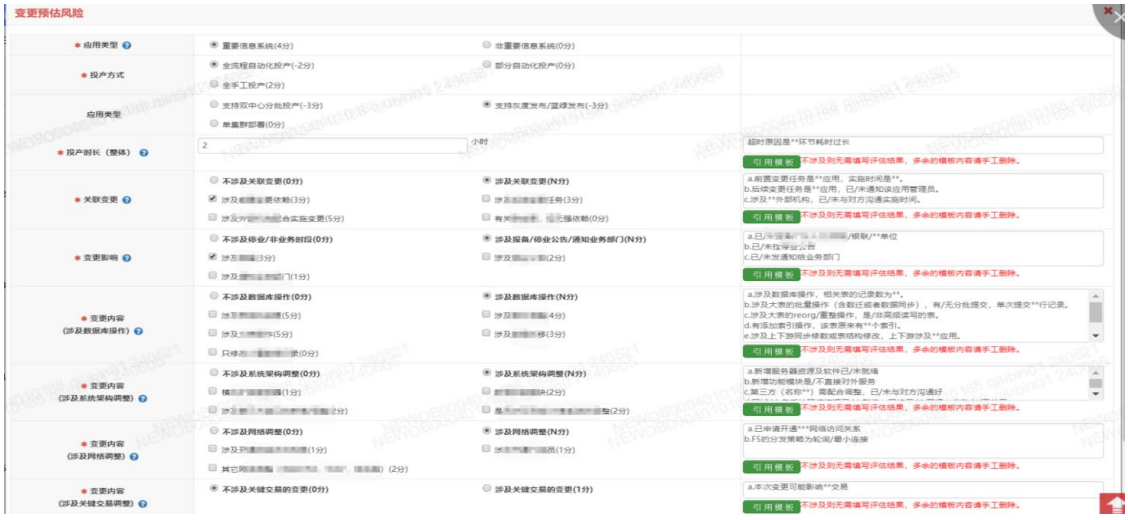
、投产时长

、与其他变更的关联度

、变更影响范围

、高风险变更内容

等 7 个方面进行风险预评估。下图为应用变更风险评估预览：



基础设施类变更内容比较标准、固定，但容易发生全局性影响的生产事件，适用将风险评估场景化，方便评审委员会、审批人员快速判断变更风险等级。下图为基础设施变更风险场景预览：

针对重要变更，某某银行在变更申请阶段增加了针对重要变更的变更准备情况、变更准入条件、非功能性需求实现情况的评审。按照变更分类分为应用类、系统类、机房类、网络类、安全类 5 个大类 80 多个子类，每个子类评审要点分为 A、B、C 三个级别，A 级属于严格执行要求，评审未通过则不允许申请变更。下图为重要应用变更评审要点预览：

变更评审要点

※ 评审等级要求：
※ A级：严格执行评审要求，投产前评审不通过则不予投产。
※ B级：原则上必须执行的评审要求，投产前若不满足要求，则需通过评审会议决策是否投产。
※ C级：需在评审内容中描述需求或实际情况，作为投产准备或投产评估的依据。

搜索 导出

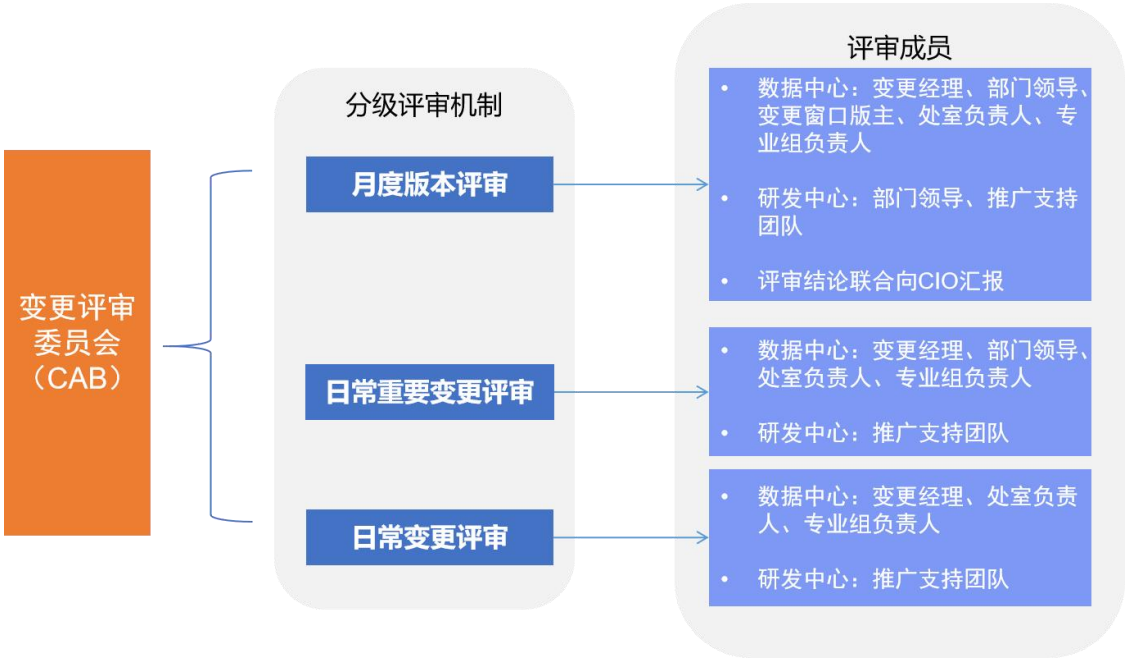
序号	类型	子类	分类	评审细项	评审要求	评审事项	评审等级	责任人	评审内容
1	应用类	应用变更	基础设施	服务器、存储硬件...	需列明服务器类型...		C级		填写模板1:
2	应用类	应用变更	基础设施	系统架构部署要求	系统架构需满足高...		A级		填写模板1:
3	应用类	应用变更	基础设施	系统架构部署要求	除了已规划为公共...		B级		填写模板1:
4	应用类	应用变更	基础设施	系统架构部署要求	采用WEB、应用、...		B级		填写模板1:
5	应用类	应用变更	基础设施	服务器互访要求	1、服务器互访必...		A级		填写模板1:
6	应用类	应用变更	基础设施	互联网应用访问	1、新建互联网应...		A级		填写模板1:
7	应用类	应用变更	基础设施	互联网IPv6协议要求	新建互联网应用必...		A级		填写模板1:
8	应用类	应用变更	基础设施	网络部署规则	新建系统和新增O...		B级		填写模板1:
9	应用类	应用变更	基础设施	IP地址资源相关	是否涉及服务器IP...		B级		
10	应用类	应用变更	基础设施	网络容量评估	1、新建系统须提...		A级		1、非

显示第 1 到第 10 条记录，总共 113 条记录 每页显示 10 条记录

« ‹ 1 2 3 4 5 › »

2.组建变更评审委员会（CAB）

由数据中心 SRE 团队和研发支持推广团队共同组成，按照变更分级评审机制组织开展变更评审。评审内容包括变更材料、变更方案、变更预评估风险及变更影响。



3.应用变更风险管控方案

某某银行应用变更通过变更风险评分、制定新建准入、与研发建立联合变更管理机制，同时，采取变更窗口集中管理、变更时点控制、投产时长控制，避免发生重大业务影响的生产事件，并且通过 SRE 左移落地，确保非功能性运维需求的实现，确保业务信息系统投产后的可靠性、可运维性。

4.基础设施变更风险管控方案

基础实施类变更重点是严防严控基础设施变更引入风险，避免发生全局性业务影响的生产事件。某行对基础设施类变更按场景进行风险等级分类，标准化基础设施变更手册、严格要求配套变更应急措施/逃生措施，并严控基础设施小变更。

(2) 变更自动化与治理

1.应用变更自动化

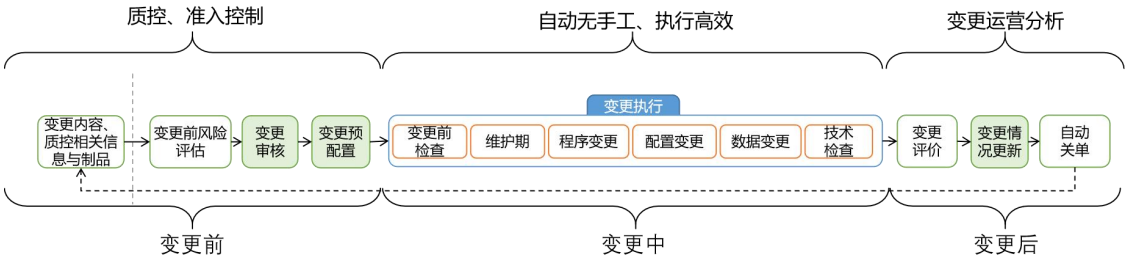
银行业务系统涉及的技术栈多且复杂、三方系统关联多，变更场景复杂，是一个多关联方、多流程、多环节的系统性工作。应用变更除了要面对风险引入控制问题，还要面对投产时间长和工作强度高两方面痛点。表层原因主要是自动化程度不足、投产中断多、手工检查验证多，集中体现在【变更中】环节，根源多产生于【变更前】，包括研发侧引入的制品质量问题、脚本质量问题、部署流水线质量问题；环境因素引入的环境变量、用户/网络权限、文件系统容量问题；自动化平台引入的代理可用性、变更自动化能力、检查自动化能力问题。以下是某银行全流程自动化应用变更实践：

1. **变更流程联动：**

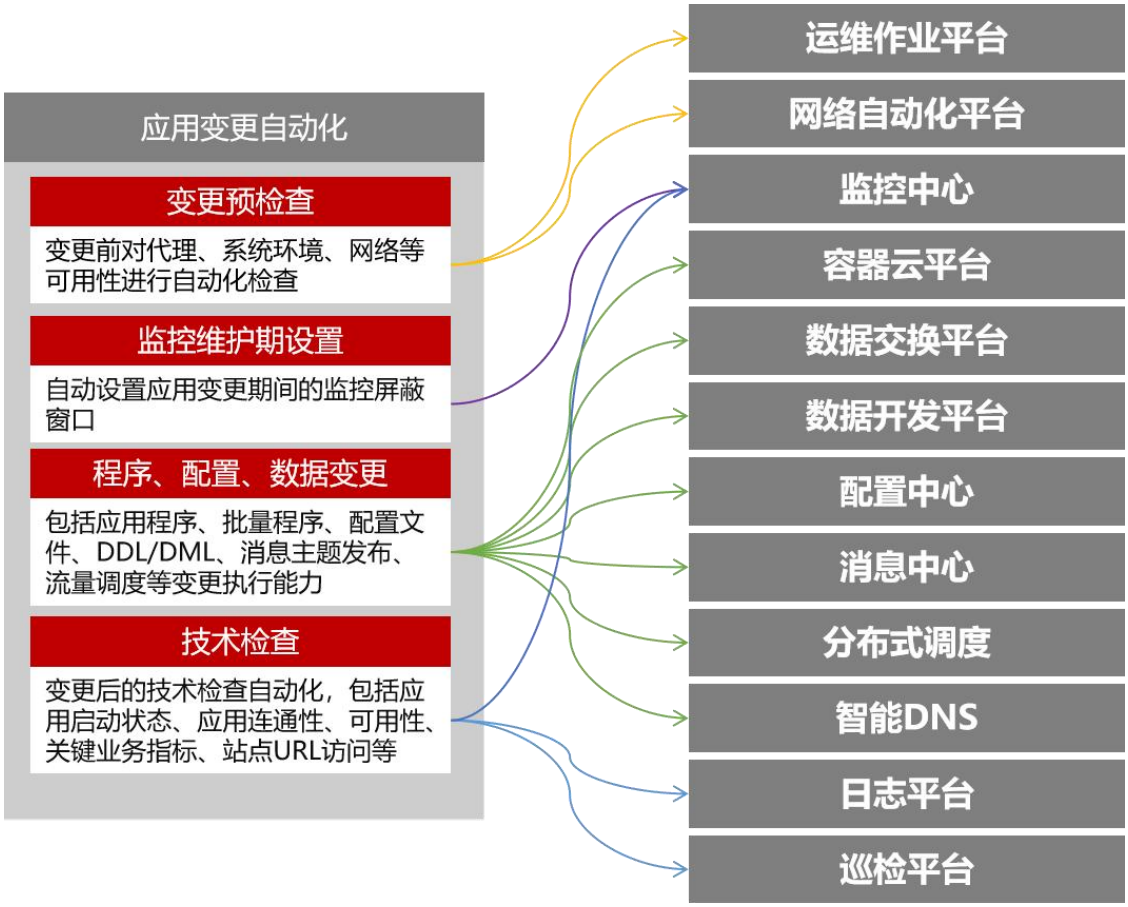
通过建立流程审批、制品库、应用变更自动平台，API 网关之间的联动。运维管理平台接受研发管理平台提交的变更申请，并同步变更制品；应用变更自动化定期同步运维管理平台状态为“实施中”的变更清单，对于定时执行变更单，应用变更自动化平台根据变更执行计划定时发起，并在实施完成后通知 SRE，其他变更单，由应用管理员在应用变更自动化“变更工作台”完成变更单查看、发起、确认更新等操作。

2. 生态对接实现变更执行自动化：

应用变更执行过程由变更前检查、监控维护期设置、程序变更、配置变更、数据变更、技术检查等 6 个环节组成。



6 个环节的自动化执行，主要通过自动化变更平台本身的流程引擎、代理执行能力，以及通过 API 网关与其他运维支撑平台的生态对接，实现平台自动化能力的扩展。



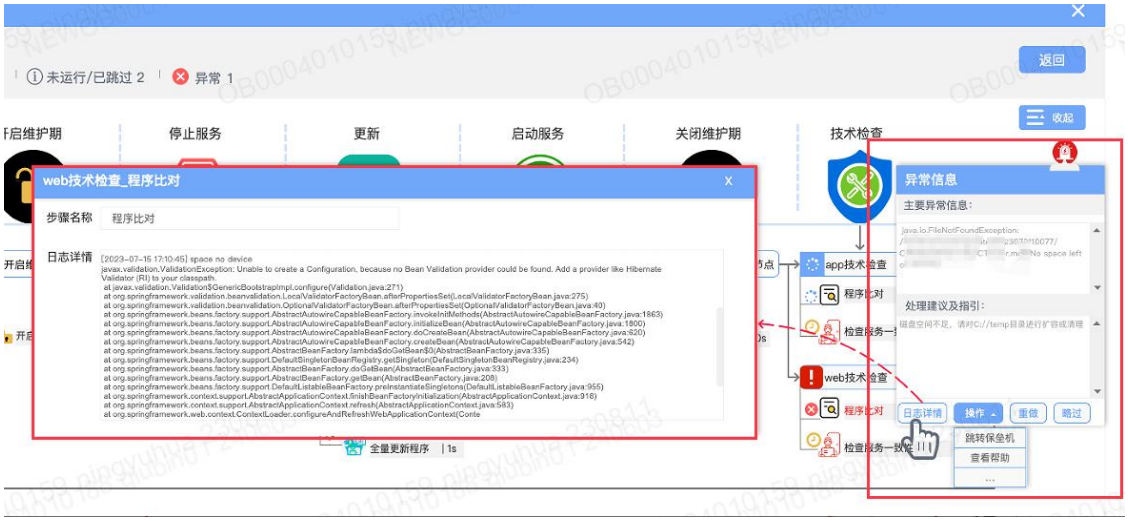
- 变更预检查自动化：变更发起后先进行预检查，在变更前提前暴露问题、解决问题，减少变更过程异常中断，有利缩短变更耗时。

	检查内容
代理	代理是否在线
	代理权限是否正确
系统环境	指定路径是否存在
	数据库密码是否准确
	环境变量是否存在修改
	临时下载目录空间是否足够
网络权限	平台到目标服务器网络连通性
	目标服务器之间连通性
	新增业务网络连通性
	第三方对接平台网络连通性

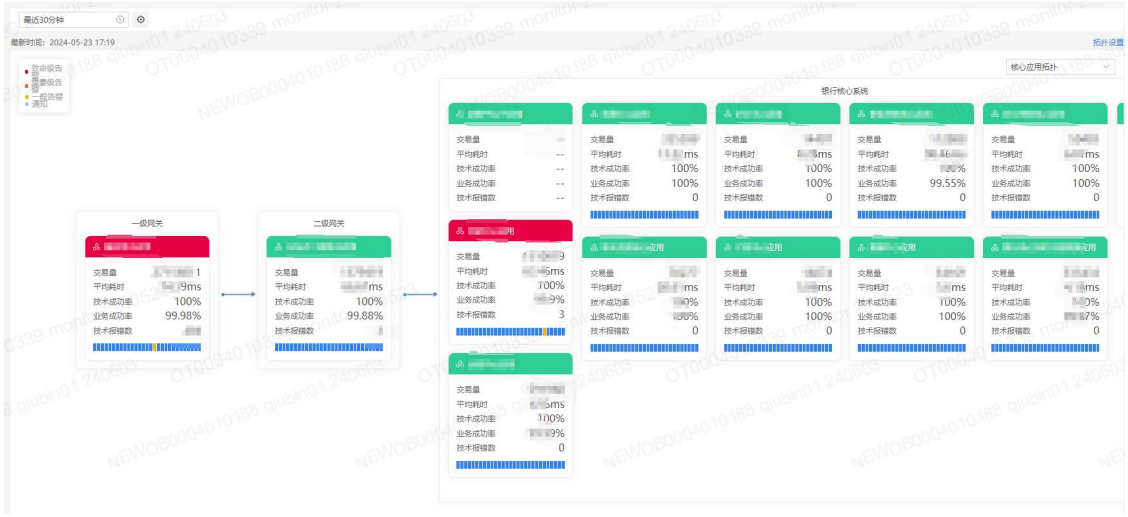
- 监控维护期设置：支持按默认维护期和人工选择个性化维护期，避免变更期间监控误报。

- 变更执行过程可视化：包括执行进度可视化、异常可视化，此外，对于重要业务系统变更，还采用双屏变更方式，一屏用于变更操作、一屏用于业务运行观测。

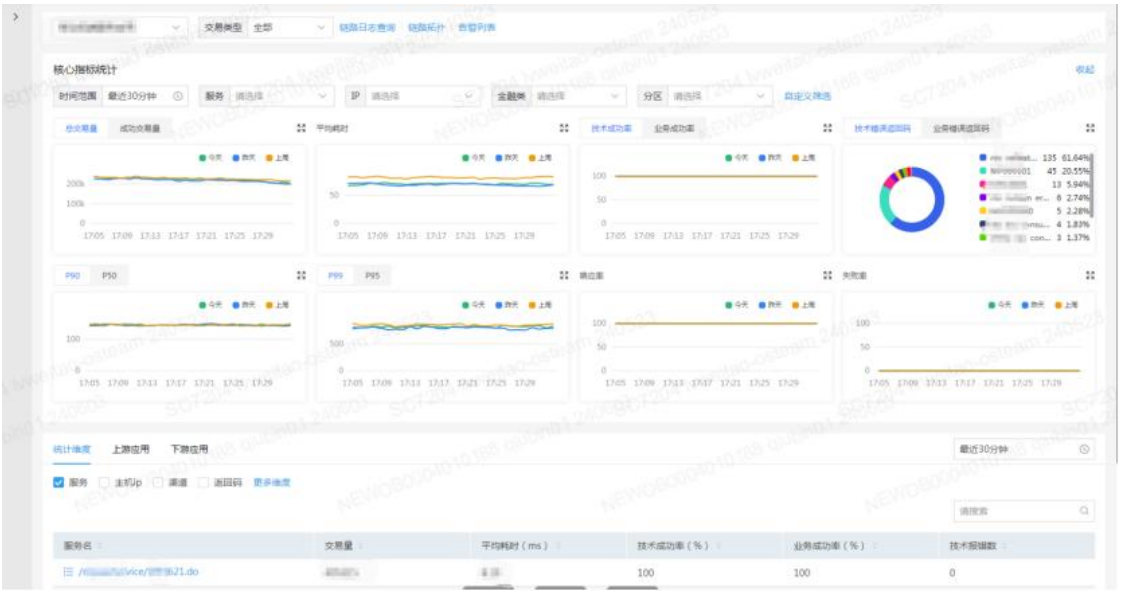
图：云上变更过程可视化



图：变更异常信息回显

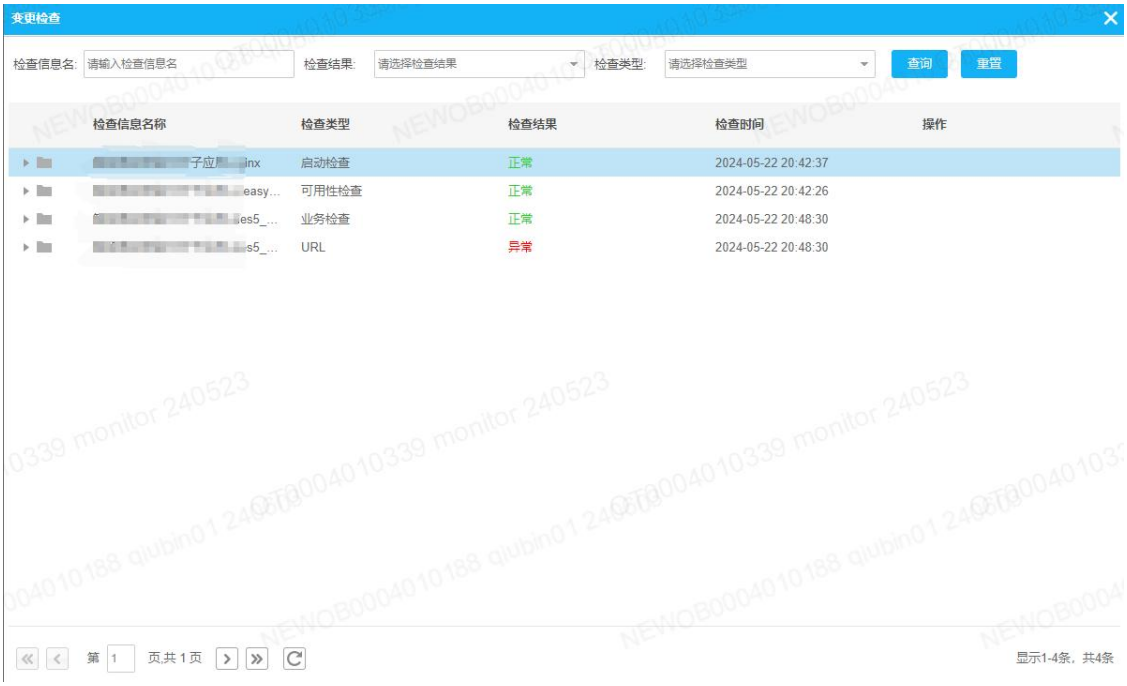


图：应用群监控大盘

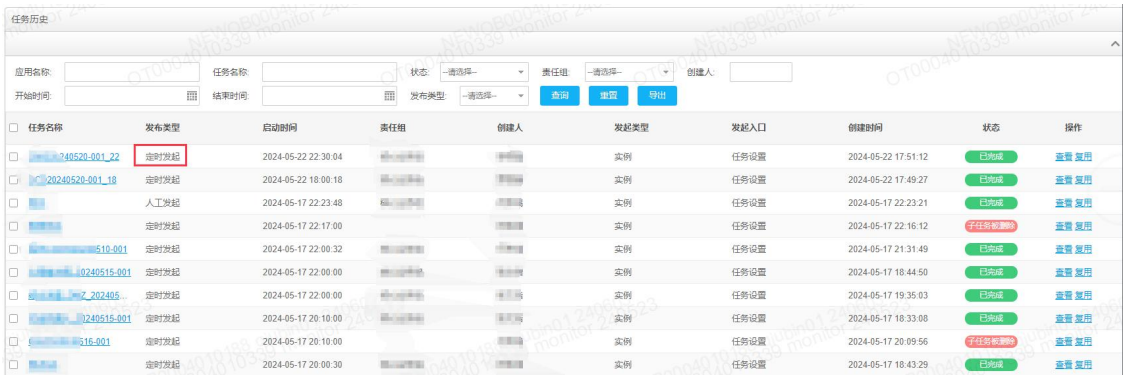


图：应用多维分析大盘

- 技术检查自动化：通过平台自身代理，以及对接监控中心、日志平台、巡检平台实现变更技术检查项的自动执行。

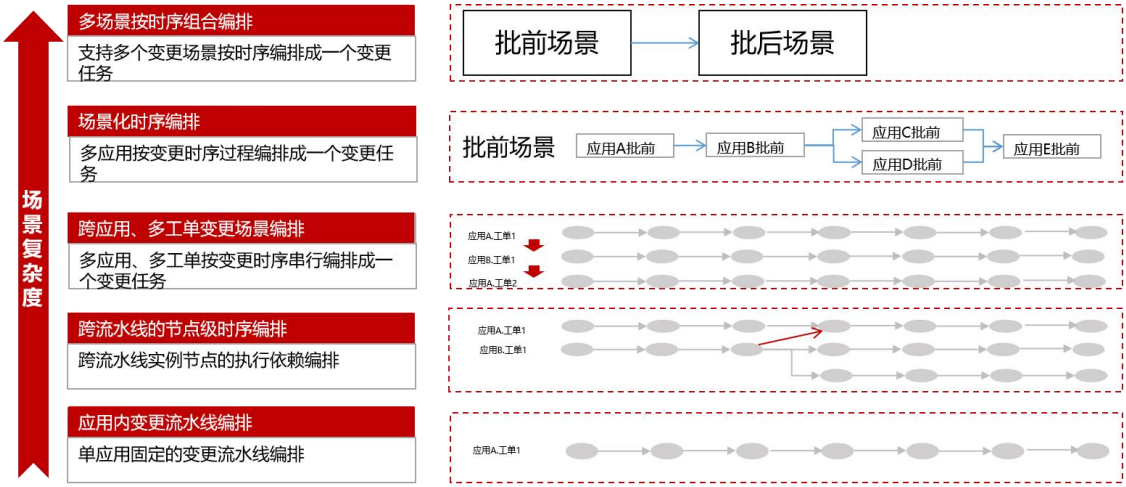


- 变更任务定时执行：对于已实现全流程变更自动化的应用，支持配置变更任务自动发布，变更执行结果通过蓝信通知到 SRE。



3. 灵活的变更策略支持：

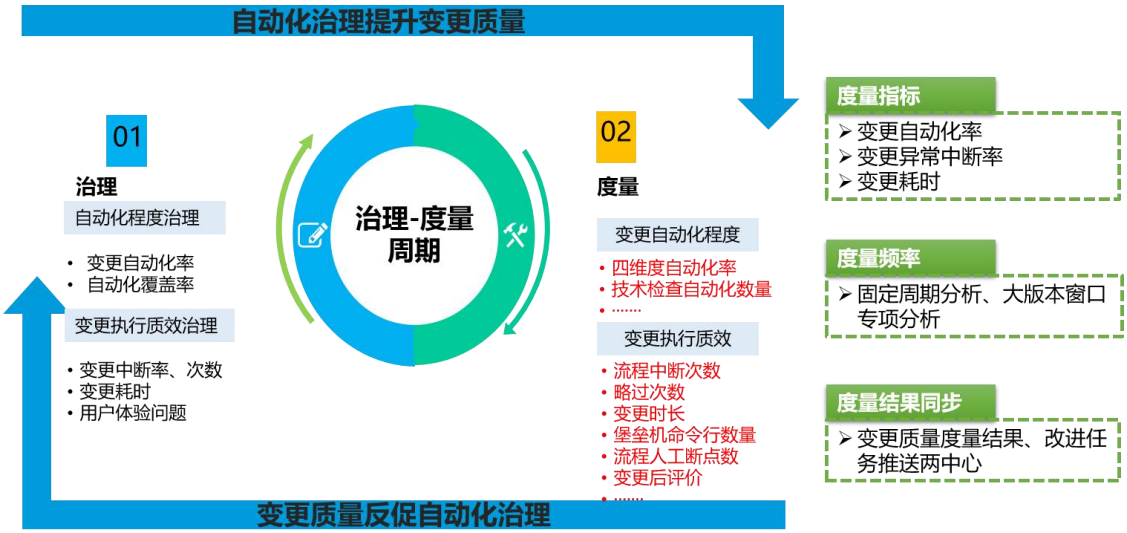
支持跨发布流水线的节点依赖、场景化变更，减少变更次数、加强变更时序控制。



场景化变更实现国产化分布式核心应用群按批前、批后时序，单双机、生产同城环境分批部署，大大提升核心变更效率，缩短核心变更时间：

2.应用变更自动化治理

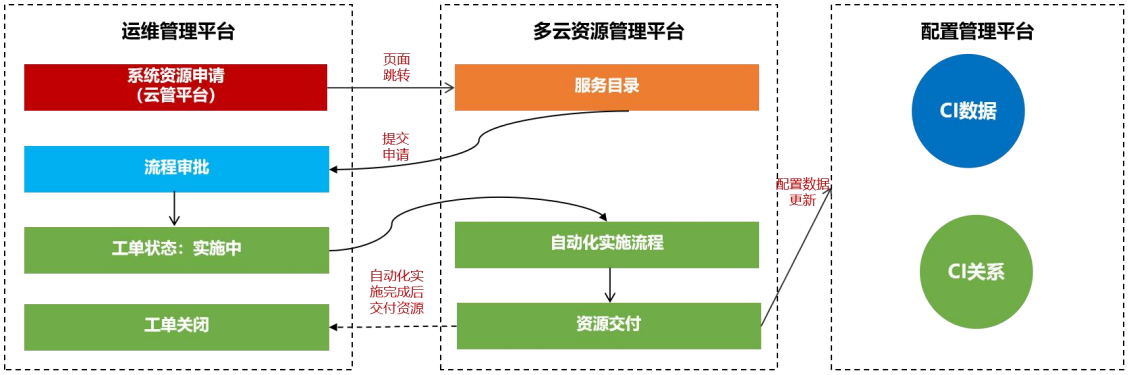
应用变更自动化的治理主要面向6个变更执行环节的执行自动化、异常中断、执行耗时以及变更后评价（变更执行相关用户体验、需求、问题等信息收集）进行治理，治理团队包括变更经理、平台工程团队、研发支持推广团队。



3.基础设施资源交付自动化

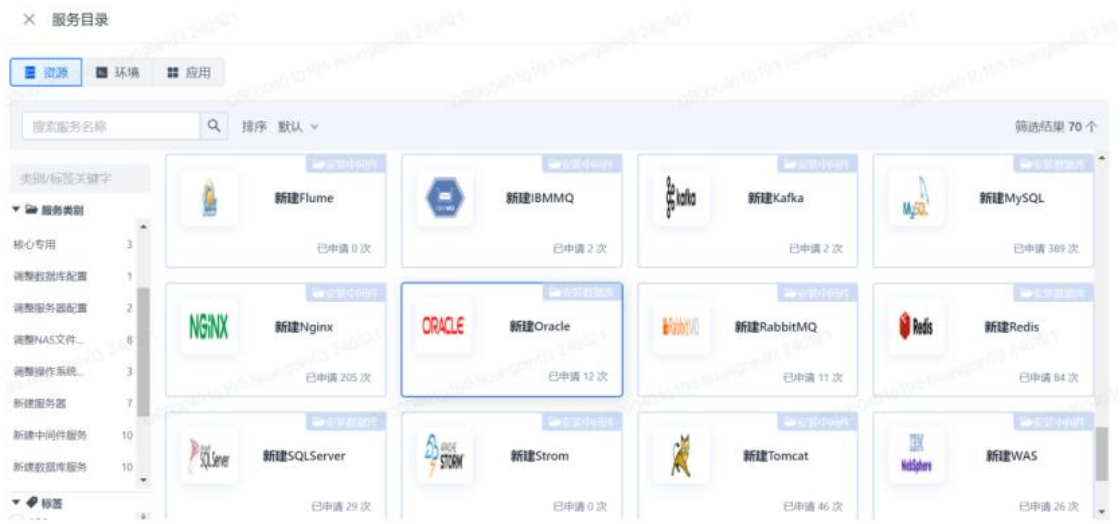
某某银行通过流程平台与多云资源管理平台、网络自动化平台对接，实现基础设施资源申请流程审批通过后自动交付、配置数据自动更新。

1. 系统、软件、存储资源交付自动化



为实现资源的自动化和标准化交付，基于基础实施即代码（IaC）技术，使用代码为每一类资源定义标准的最小化的资源模块，实现资源的代码化抽象：

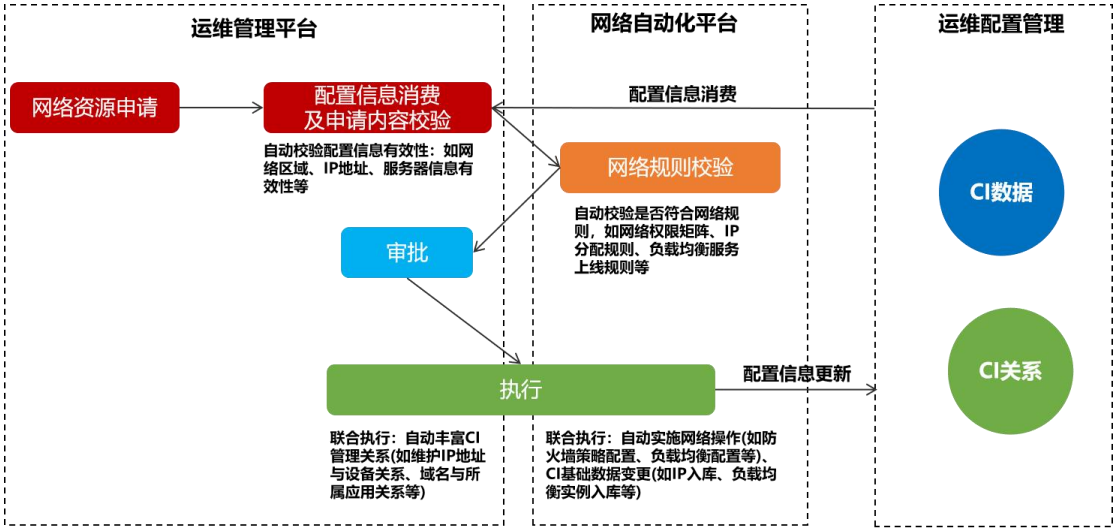
- 按照不同的资源类型建设标准的、可复用的资源模块，包括虚拟机/物理机、操作系统配置、代理、数据库、中间件、存储资源和网络资源等。
- 各资源模块分别管理，不同技术团队按标准配置规范负责对应组件部分的开发和调优，并进行统一管理。
- 资源交付流程以服务目录的形式展现，通过对标准的资源模块进行组装，编排形成通用的部署模板，将部署模板映射发布为服务项，为用户提供资源交付的自助服务：
- 按需组合资源模块，描述资源模块间依赖关系，实现复杂资源定义，满足任意丰富的业务场景需求。
- 提供套餐化的自助服务，简化和标准化申请参数，提升服务体验。



云管平台服务目录示例

容器云资源服务项交付示例

2. 网络资源交付自动化



流程平台对接网络自动化平台、CMDB 实现网络资源申请工单快速填写，域名发布、IP 分配、网络访问权限、负载均衡服务上线等网络资源申请项的自动交付。

网络自动化平台支持任务编排、人工/自动执行、遇错停止、继续执行、结果回显、健康检查和备份前后对比、变更涉及设备的告警一键清理等功能。

网络资源、负载均衡自动交付示例

（三）总结及展望

变更是一个多关联方、多流程、多环节的系统性工程，在安全、数字化、降本增效为主题的当下，变更管理面临最大的挑战，其实是安全、效率、成本的既要又要，在未来1~2年，在变更管理方面，将持续深耕变更风险控制自动化、操作执行自动化、治理分析自动化，提升变更风险管控能力和交付效率，降低变更人力投入，推动业务信息系统的标准化、服务化、容器化改造，以此降低变更管理的难度和成本。

4.4 发布管理案例

4.4.1 中移互联网敏捷发布平台建设实践

SRE Elite 收录点评

1，“中移互联”这个名字就体现了传统行业与互联网的结合，传统IT模式与敏捷互联网业务间的矛盾，促使其从方向上选择卸下历史包袱，推倒零散的工具系统，一步到位建设一体化运维平台。并且对物理机、虚拟机、容器等不同类型的资源及上层应用设计了混合编排模式，为后续的综合算力调度做了储备。

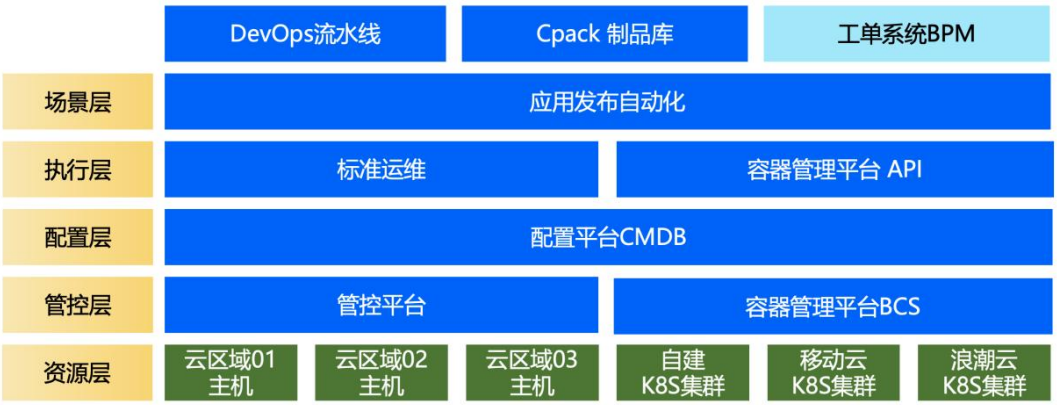
2，在一体化运维建设之后，为了追求更高的质量效率，中移互联SRE团队向研发服务左移，保障了测试环境及生产环境的发布一致性，并且承建了研发工具链，从运维一体化扩展至研运一体化。

（一）背景及设计原则

中移互联网有限公司（以下简称“中移互联”）是中国移动集团公司面向互联网领域设立的专业子公司，目前运营着多款拥有亿级用户规模的产品，包括移动云盘、移动认证、超级 SIM 等。从业务上看，这些业务有持续迭代发版的内在需求，然而，各业务线的发布变更能力存在显著差异，手工与工具并存，工具平台多样化，不仅效率低下，且容易出错，发布能力与业务发展水平不匹配。

为了解决上述问题，中移互联提出了推动构建公司级统一应用敏捷发布平台。该平台旨在实现传统主机和云原生环境的全场景自动化发布变更，构建覆盖测试、灰度和生产环境的一体化应用发布能力，从而推动发布工作从手工到自动化、发布工具从分散化到集约化的转型，提升生产环境的变更成功率和发布效率，为业务团队提供稳定、安全的发布保障，进一步推动公司的数字化转型和业务发展。

（二）体系设计及关键流程



敏捷发布平台基于蓝鲸底座构建，旨在提供 CI CD 集约化一体化的变更管理能力。

平台的架构设计包括以下几个层次：

资源层：涵盖分布在不同机房的物理机、虚拟机及各类的 K8s 集群。目前公司内有 8 个数据中心，包含：广州萝岗、南方基地、北京机房、移动云无锡机房等。

公司产品有全量接入移动云，使用 KCS 容器服务；部分产品接入浪潮 PaaS 平台，使用浪潮提供的 K8s 容器服务；中小产品则自建 K8s 集群进行部署运营。

管控层：使用 GSE Agent 统一管理所有产品的物理机和虚拟机，命令下发、文件分发、脚本执行、数据采集都通过统一的 GSE Agent 去完成，规避了业务线各自使用不同工具和管理方式的弊端。

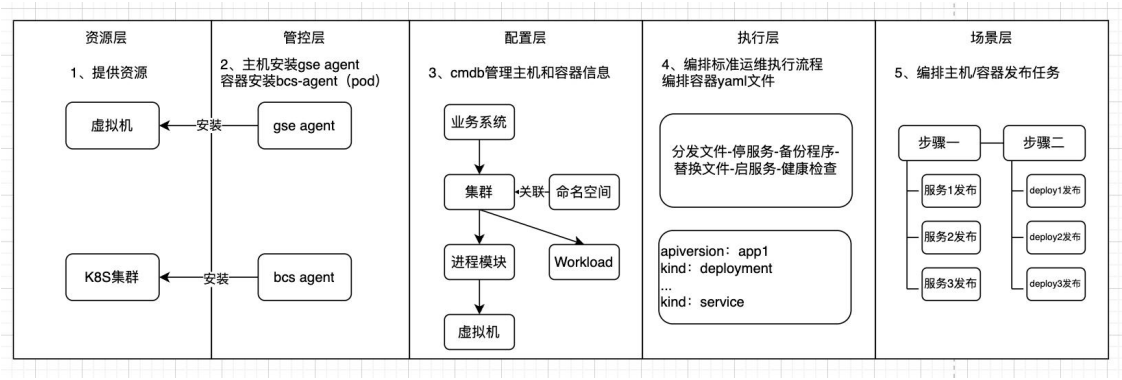
敏捷发布平台通过 BCS 将不同来源的 K8s 集群统一管控，并对外暴露统一的 BCS-API。这一做法一方面屏蔽了不同容器服务（如 KCS 接口或浪潮云 PaaS 接口）的 API 差异，避免了每个产品团队需要单独适配；另一方面也解决了不同版本 K8s 接口的差异问题，因为 BCS-API 本身兼容了不同版本的 K8s 接口。

配置层：统一管理主机和容器资产，并通过应用拓扑将二者关联。在发布变更时，只需选择目标业务系统和服务，系统会自动从 CMDB 拉取目标主机及 K8s 集群的 Namespace，提升配置的准确性和唯一性，降低变更出错的概率。

执行层：通过标准运维编排规范的通用执行流程，各业务线可以复用此流程完成传统主机服务的变更发布，并通过调用 BCS-API 完成容器服务的变更发布。

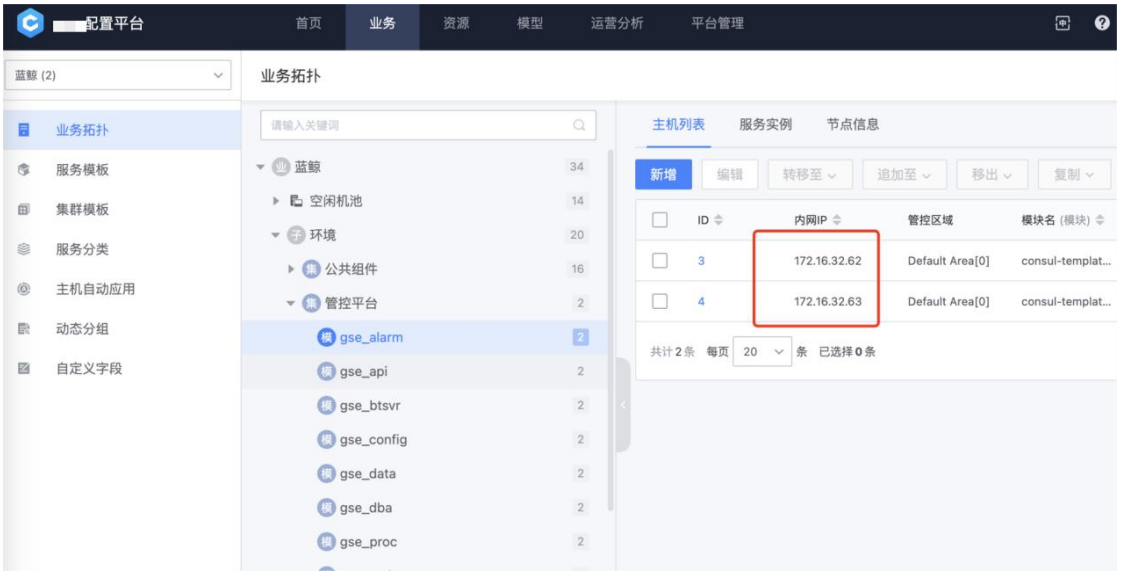
场景层：提供应用发布自动化统一编排任务，包括自动滚动分批发布任务、主机和容器混合发布任务、蓝绿发布任务等。在测试环境中，通过 DevOps 流水线自动完成编译、

构建，并调用应用发布自动化接口完成发布；在生产环境中，集成 BPM 工单审批系统，生产环境的每次变更任务创建时会自动生成 BPM 审批工单，需运维领导审批通过后方可执行。



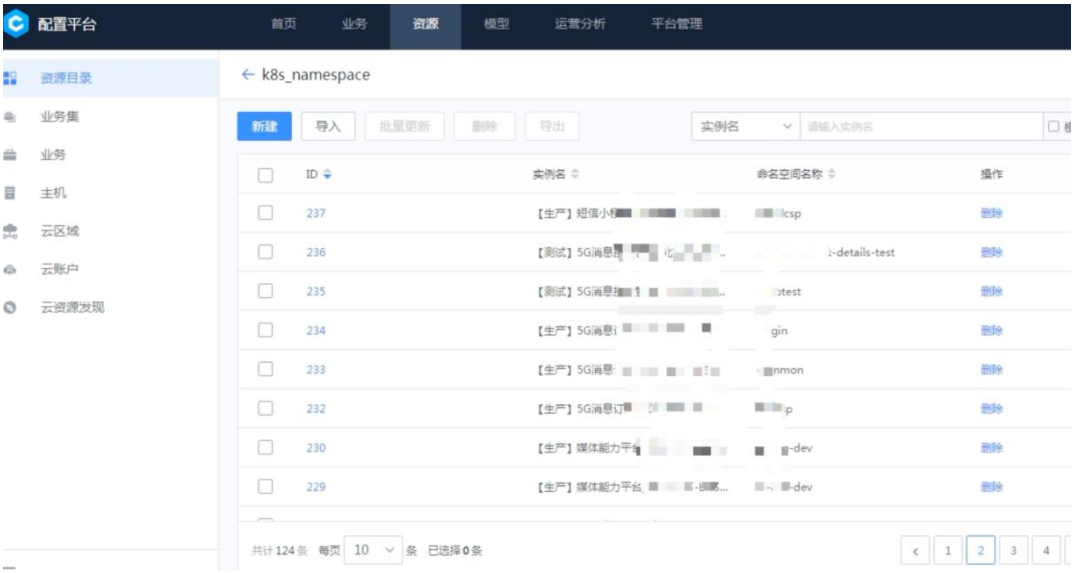
配置层统一管理主机和容器资产，并且通过一个应用拓扑将主机资产和容器资产关联起来。在发布变更时，只需要选择要发布的业务系统和服务，自动从 CMDB 拉取服务要发布的目标主机，以及目标 K8s 集群的 Namespace。统一的变更配置层，提升配置的准确性和唯一性，降低变更出错的概率。

CMDB 维护传统应用拓扑，业务-环境-集群-模块的四层应用拓扑维护数据，管理下业务系统数量，每个业务系统下服务模块数量，每个服务模块部署在哪些主机上。在 CMDB 中维护好这些数据，业务方在发布时，消费这些数据，需求单独维护“应用”和“主机组”的关系。



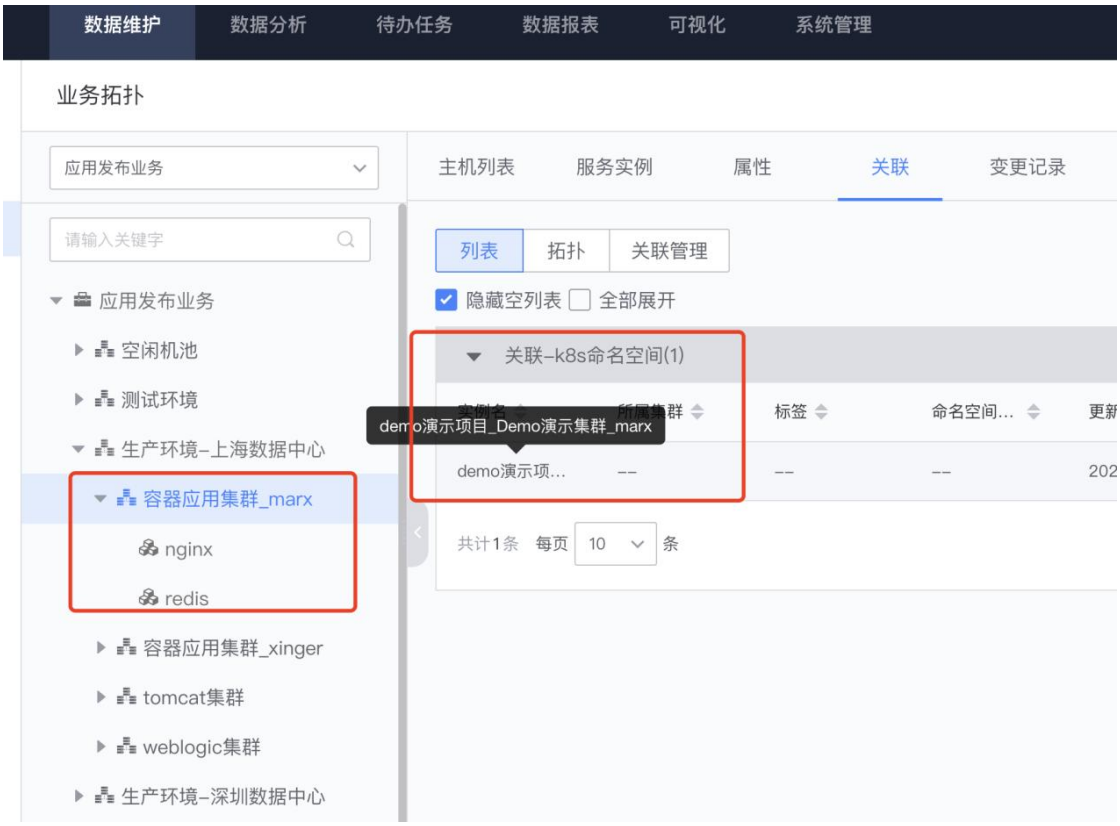
针对容器应用，同样复用这套拓扑关系数据，利用 CMDB 中模型的关联关系，建立集群模型和命名空间模型的关联关系。

在 CMDB 中维护 K8s 的命名空间数据，方式是周期性调用 BCS-API 实时拉取 K8s 集群数据，做比对和更新。



Namespace 资源

将命名空间和应用拓扑中的集群层级关联之后，就可以用相同的应用拓扑管理容器资源，无需额外维护一套拓扑关系。



执行层通过标准运维编排规范的通用执行流程，每个业务参考复用此流程完成传统主机服务的变更发布，通过调用容器管理平台 BCS-API 完成不同容器服务的变更发布。

在场景层，则通过应用发布自动化统一编排发布任务，包含自动滚动分批发布任务、主机和容器混合发布任务、蓝绿发布任务。

虚拟机发布节点配置

发布 10 台主机，选择随机滚动分批

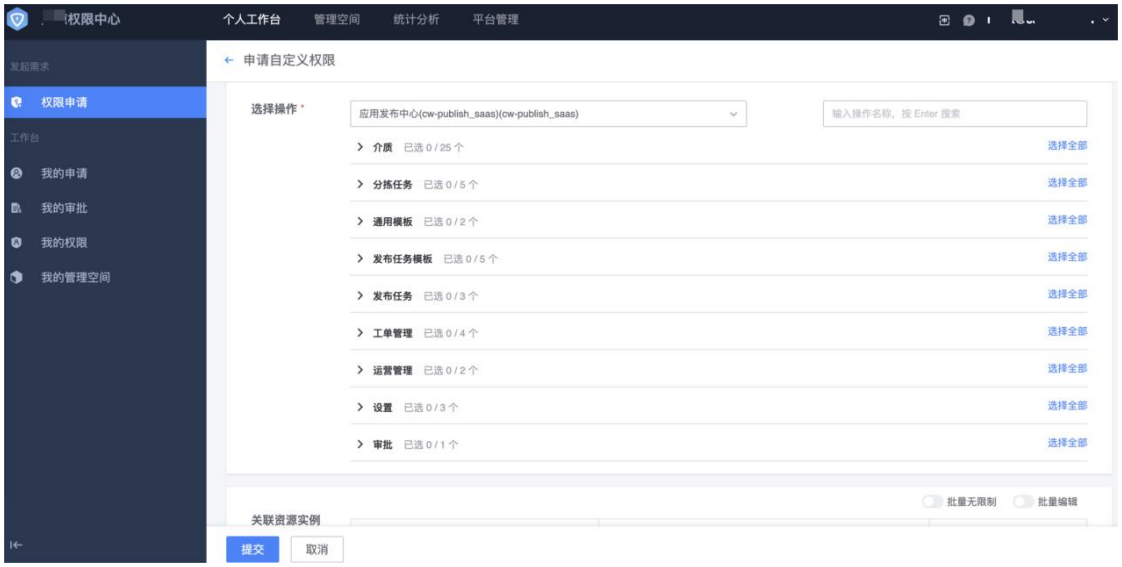
在测试环境，通过 DevOps 流水线自动完成编译、构建，调用应用发布自动化接口完成发布；在生产环境，集成 BPM 工单审批系统，生产环境每次创建变更任务，自动创建 BPM 审批工单，运维领导审批通过之后，才能执行生产变更任务。

研发部和系统部密切协同，构建全网业务从需求、项目、代码、构建、制品、发布等端到端闭环管理，接入公司多个部门 30+业务线，累计纳管近万台主机和 60＋K8s 集群，生产环境周发布近 600 次。

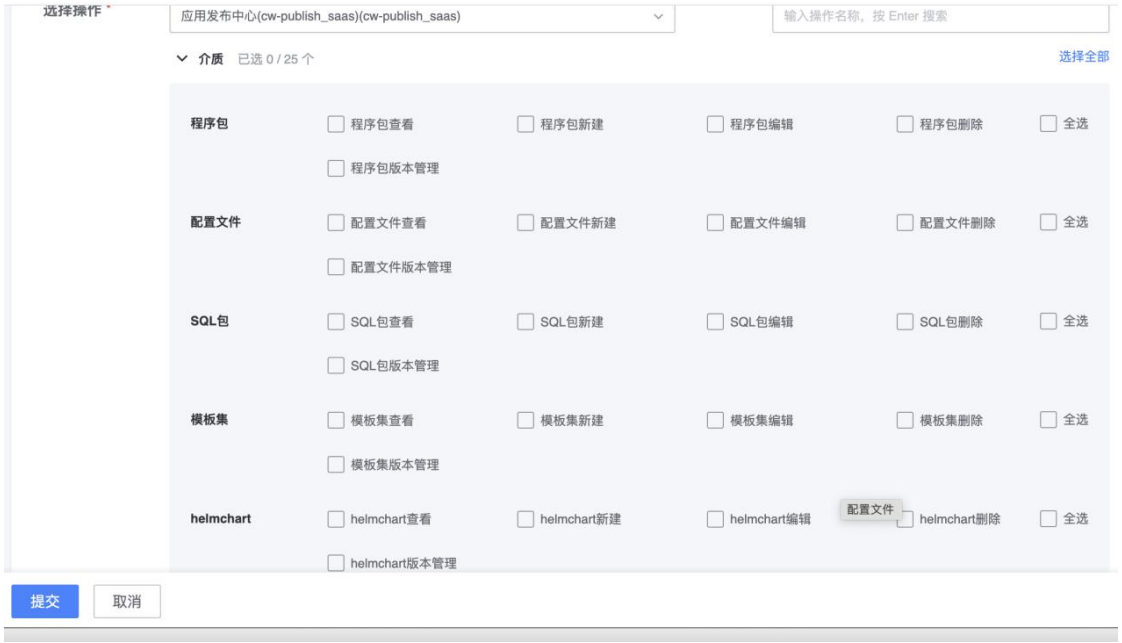
(1) 统一权限管控

无论是自研战略业务，还是中小业务，统一使用敏捷发布平台，这样可以进行统一权限管控，严格控制不同角色的生产操作权限，极大收敛变更权限风险敞口。

将发布权限统一注册到权限中心，核心是介质、发布任务模板、发布任务三大模块。



介质模块包含：程序包、配置文件、SQL 包、模板集、Helm 包的增删改查和版本管理。这些操作的授权范围都是业务系统，即授权用户某业务系统下程序包的增删改查、版本管理权限。



发布任务模板和发布任务，核心是控制执行能力，授权范围也是业务系统，即授权用户某业务系统下发布任务执行权限。

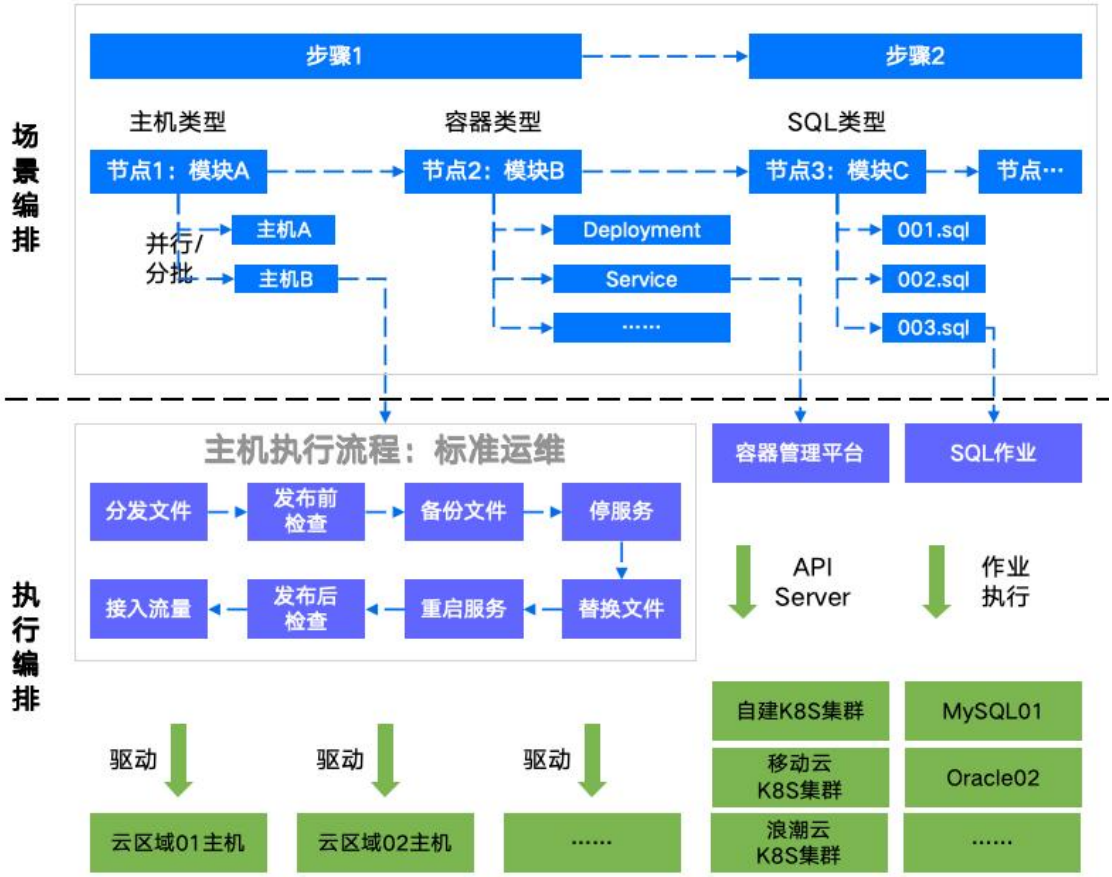
统一变更入口之后，可以对不同业务团队的变更操作进行约束，推动变更操作规范化通用化，降低发布失败率。

此外，统一不可删除的变更记录，既能快速定位问题，缩短变更异常的回滚时间，还能用于度量分析和复盘改进，以及能提供给公司审计。

业务	CI/条	构建次数	总构建次数	CD/条	发布次数	总发布次数	周生产环境发布次数	主机/台	K8S/套
[redacted]	420	504	14227	386	491	13399	118	253	0
[redacted]	560	283	5243	363	294	4679	113	48	5
[redacted]	718	161	13494	429	256	7998	80	623	2
[redacted]	111	300	1101	139	342	1248	74	545	0
[redacted]	187	347	6040	151	312	5473	47	108	0
[redacted]	231	333	4215	150	309	3325	37	188	1
[redacted]	998	1283	19867	670	988	13140	29	3642	29
[redacted]	100	50	3187	108	47	2595	10	24	0
[redacted]	110	49	1415	79	9	145	9	817	2

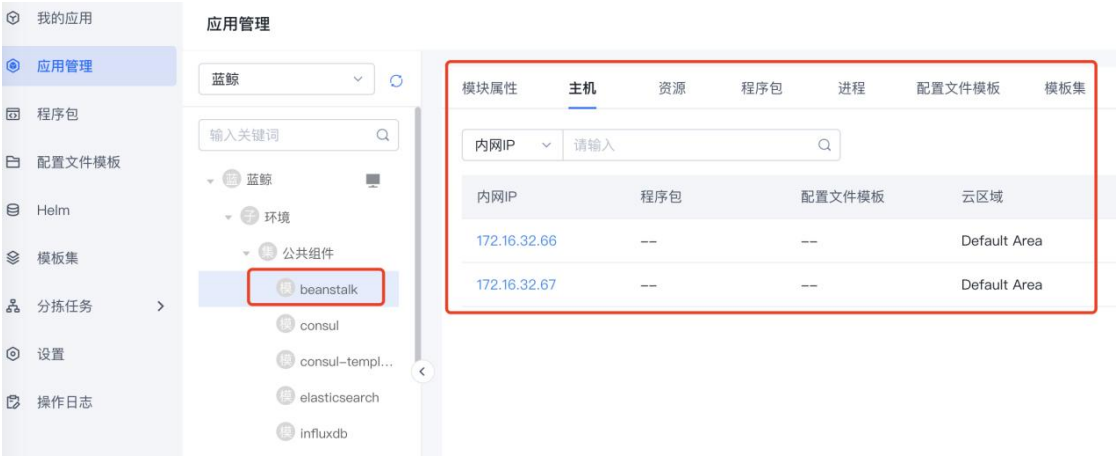
(2) 多场景任务模板

中移互联目前处于云原生转型的过程中，但各团队进度不一：有些业务已全面云原生化，全部部署在 K8s 环境中；有些业务因历史技术原因仍运行在虚拟机上；还有一些业务处于转型中，部分服务在 K8s 环境，部分在虚拟机环境。为实现传统主机服务和云原生服务的变更一体化，敏捷发布平台通过集成标准运维和容器管理模块，提供一套兼容主机和容器任务的变更模板，屏蔽异构环境差异，提升生产环境变更成功率。

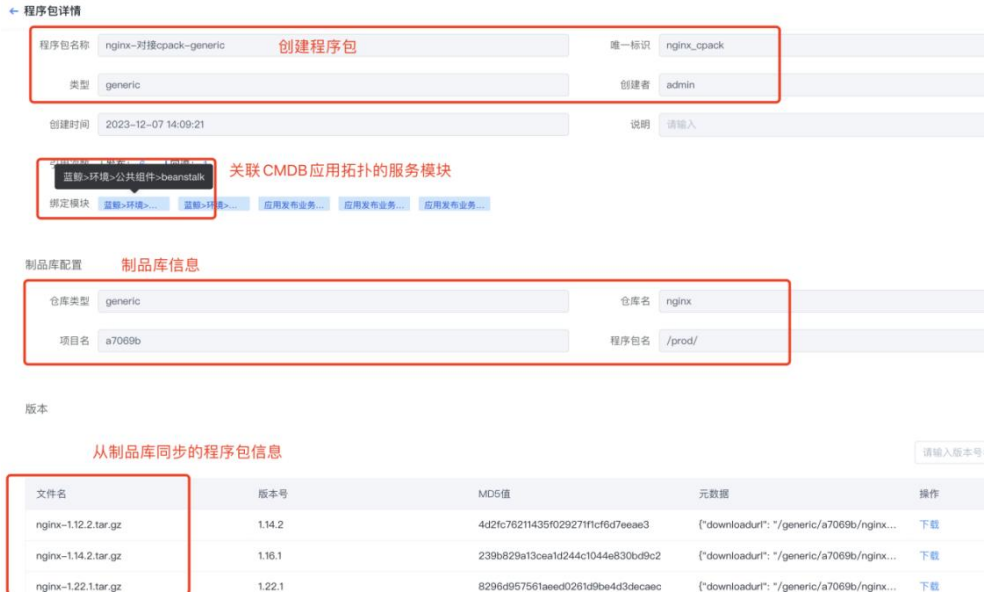


完成一套混合发布任务编排的流程：

步骤一：维护传统应用拓扑关系，服务模块关联主机



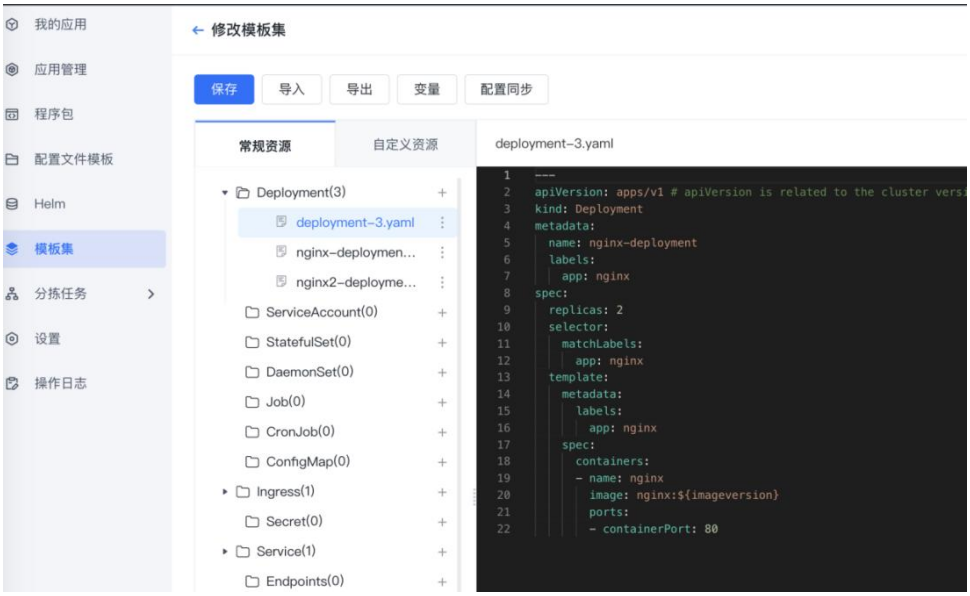
步骤二：创建程序包，从制品库同步程序包版本，并关联应用拓扑中的服务模块



步骤三：在标准运维编排主机执行流

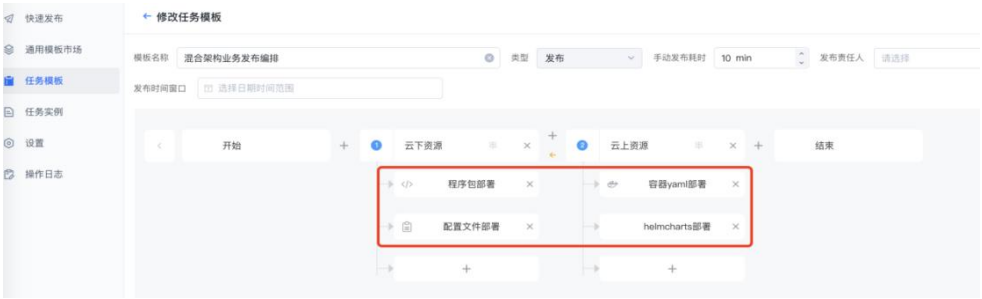
步骤四：创建传统应用发布节点，选择业务、模块、程序包、主机

步骤五：创建容器模板集，导入 YAML 文件，或者从生产 K8s 集群直接导入 YAML 文件。将容器模板集和 CMDB 的服务模块绑定



步骤六：创建容器应用发布节点，选择发布的业务系统和服务模块，选择关联的模板集，选择模板集中版本，以及版本内 Workload、Ingress

步骤七：创建完整的混合编排发布模板，包含主机程序包发布、容器模板集发布，以及配置更新和 Helm 发布（未展开）。

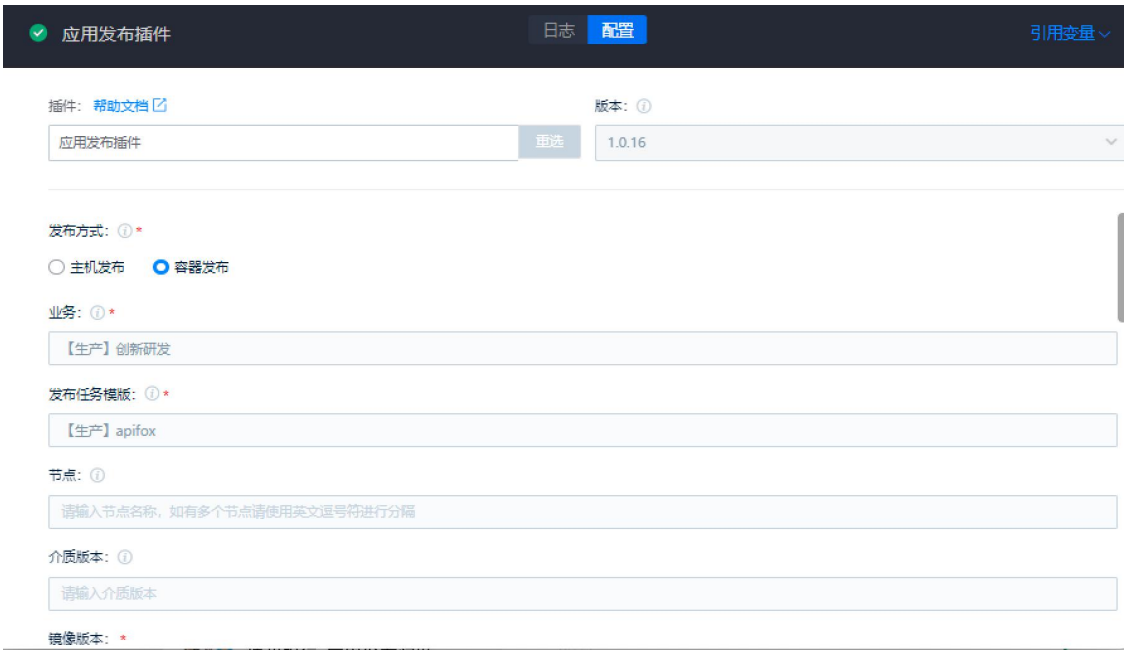


(3) 测试环境/生产环境发布流程打通

研发人员在测试环境发布服务时没有问题，但在生产环境发布时却出现故障。这主要是由于测试环境和生产环境的架构和配置不一致，以及两者使用不同的发布流程。为了提高生产变更的成功率，需要在研发、测试和生产环境中使用相同的部署流程。然而，研发/测试环境强调敏捷性，要求自动构建和自动部署，而生产环境强调安全性，要求变更安全稳定。因此，需要平衡不同环境和团队的需求，才能实现研运一体化。

为解决这一问题，研发产品部和系统工程部密切协同，在一站式敏捷发布平台上实现测试和生产环境的统一流程管控。在测试环境中，使用 DevOps 流水线自动完成编辑、构建、代码扫描、制品入库、持续部署和自动测试，并在持续部署节点调用应用发布自动化接口进行部署。

研发/测试环境流水线



对接应用发布插件

生产环境，BPM 工单系统和应用发布自动化对接,达到生产变更任务的强审批执行。

对接工单系统的审批内容

（三）总结及展望

建设总结

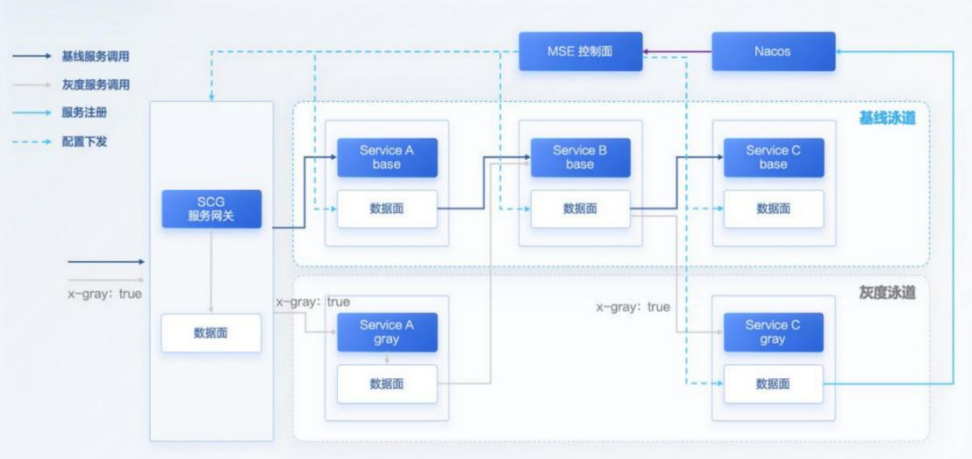
敏捷发布平台接入业务 156 个，累计 1400 多个用户接入使用，平台累计共完成配置 CI 流水线 4402 条、CD 流水线 4044 条，CI 总构建次数 9 万余次，CD 总发布次数 6.5 万次；纳管 1 万余主机节点，60 多个 K8s 集群。

同时，逐步接入各业务线研发/测试以及灰度/生产环境，目前全环境覆盖率为 54%，部分环境覆盖率 44%。

展望方向

由于架构方面原因，目前大部分应用的发布并不能实现灰度，违反了“可灰度，可监控，可回滚”的实践原则，爆炸半径无法最小化。但是由于架构升级改造成本高昂，要进行全量改造并不现实。

因此，研发部和系统部目前正在联合推动基于 Java Agent 的无侵入全链路灰度发布方案的落地，目标是构建灰度泳道管理体系，为业务提供逻辑泳道隔离和泳道流量投递等发布能力，并计划与业务微服务网关对接，实现跨集群、多应用的灰度发布。同时，灰度发布规范体系也在制定及落地中，设置了业务灰度准入规则，旨在扩展发布窗口，帮助业务团队实现不停机的无损发布。



4.4.2 某证券变更一体化平台建设实践

Elite 收录点评：

本证券案例的变更一体化平台建设目标是解决四种场景的变更统一：自研和外购应用的发布/变更一体化、传统和云原生应用的发布/变更一体化；测试与开发环境的发布/变更一体化；敏捷与安全的工程一体化。覆盖场景多，平台整合度高，规划参考性较强。

背景及设计原则

某证券是国内资产规模 Top 10 的证券公司，每年 IT 投入超过 10 亿，有庞大的客户基础和丰富的金融产品，有成规模的 IT 团队和匹配业务发展的变更一体化平台，满足了证券类业务 SRE 在变更场景中关注的特点。

部分传统业务系统是外购，由厂商进行维护和变更；新的业务系统大部分都是自研，通过流水线进行发布。变更平台需要同时兼容外购业务系统和自研业务系统的变更。

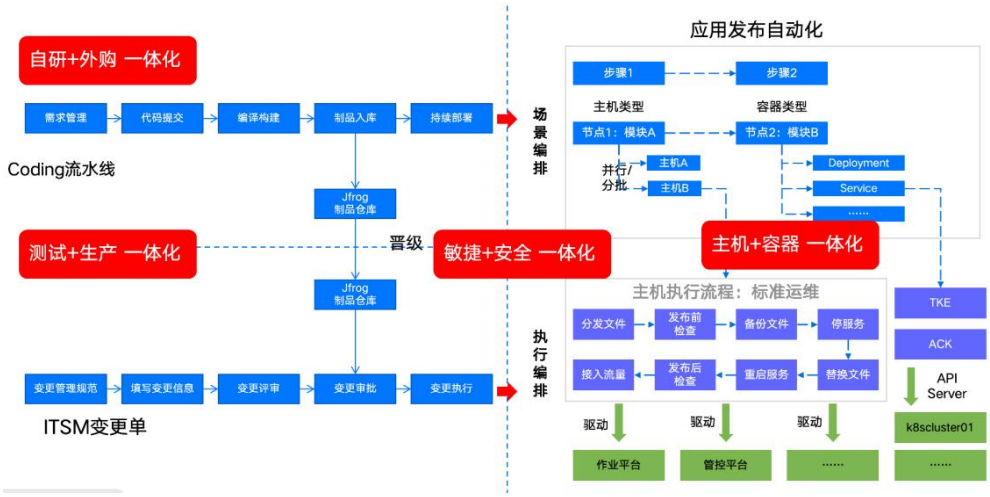
目前大部分业务系统是在虚拟机上通过二进制包方式部署，随着云原生的发展，部分新业务系统部署在 TKE 容器集群上，通过 Yaml 文件或者 helm 包进行部署，未来计划增加 ACK 容器集群。变更平台需要同时兼容二进制部署，和云原生的 yaml/helm 方式。

有存在测试环境使用一套发布流程，在生产环境使用另外一套发布流程，可能出现一个应用版本在测试环境通过测试，但实际在生产环境发布失败。主要原因是测试和生产环境工具和流程的差异性。因此变更平台需要支持同时在测试环境和生产环境使用，支持资源和流程的晋级，保证两个环境的操作一致。

在测试环境中，基于 DevOps 理念，要做到持续部署，达到“敏捷”的要求；在生产环境中，基于 ITIL 理念，要做到变更管理，达到“稳定”的要求，变更平台需要同时兼顾“敏捷”和“安全”。

体系设计详情

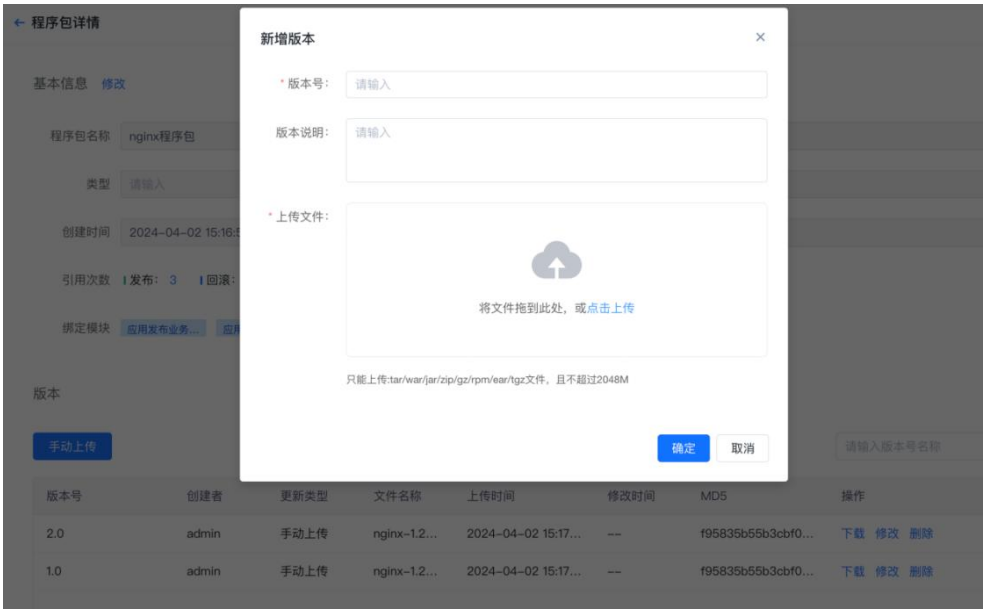
基于上述背景，IT 运维部一直在思考如何建设一个变更一体化平台，抽象出四个“一体化”的思路：自研和外购一体化、传统和云原生一体化、测试和生产一体化、敏捷和安全一体化。



(1) 自研和外购一体化

外购业务系统是由厂商维护的，厂商提供的就是一个制品包，因此需要支持厂商运维人员在平台维护程序包，并完成发布动作。

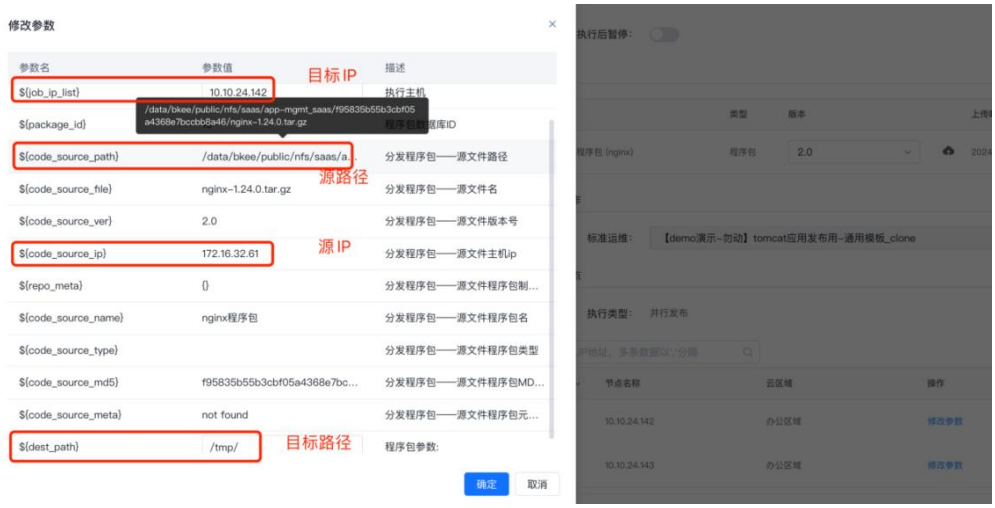
变更一体化平台支持运维人员在界面上手动上传程序包，存储在共享存储服务中。



为了避免存储堆积，可以自定义设置存储清理策略：

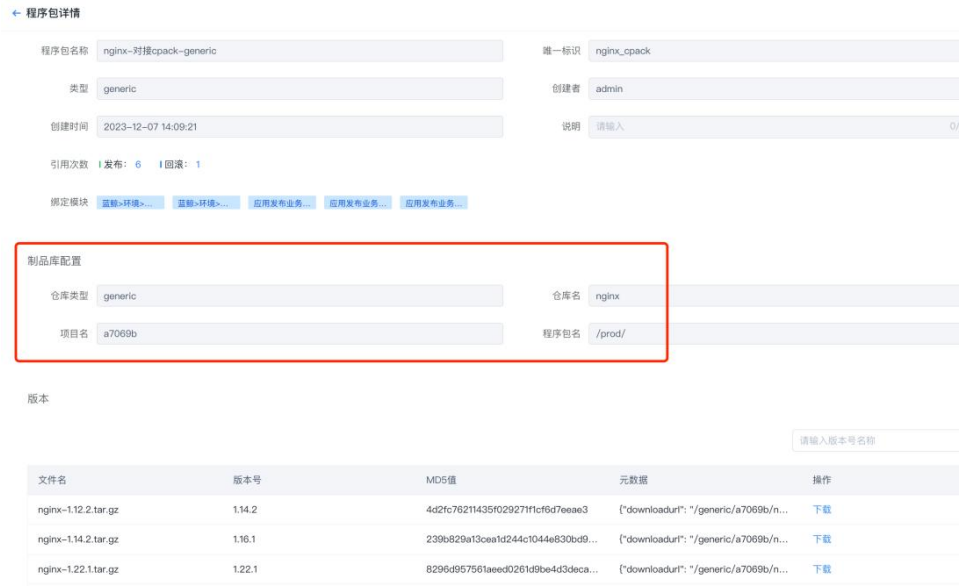
- 永不清理
- 按天数清理
- 按版本清理

在发布任务中，会将程序包存储的源服务器 IP 和源地址，以及分发的目标 IP 和目标路径，作为参数变量传给标准运维流程，这样可以通过作业平台的原子实现，文件的快速分发。



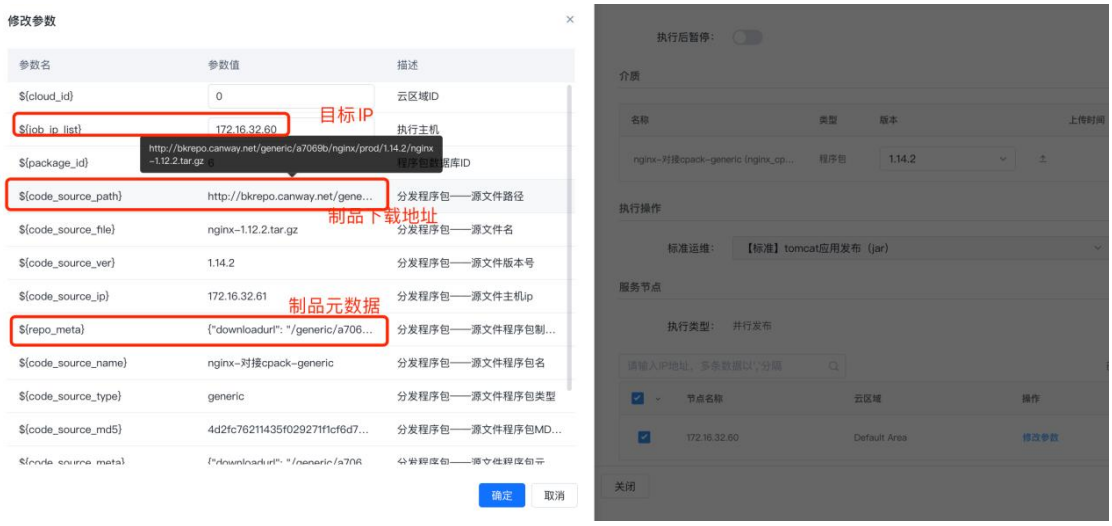
标准运维中引用快速分发文件的原子，利用 GSE 的大文件分发能力，可以快速自动将文件分发到目标机器上。

自研业务系统是开发团队维护，通过流水线打包之后，自动入库到标准的制品仓库中，在测试环境和生产环境中都是存储在制品仓库中。为了维护制品包的一致性，变更一体化平台支持通过制品库插件，对接制品仓库同步程序包信息。



上传制品库插件

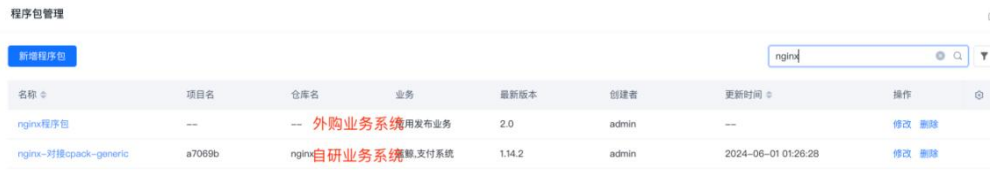
针对这种类型，变更平台不会再存一份程序包，因此是调用制品库接口获取程序包的名称、版本、MD5、下载地址，用于做程序包分发。



考虑到直接开通所有主机到制品仓库的网络策略，存在较大的风险敞口，以及各个云区域到制品仓库的网络速度限制，不会在目标机器上直接执行 curl 命令下载程序包。

SRE 同学编写了自定义的文件分发原子，用一台代理机器，和制品仓库在同一个云区域，先执行 curl 下载命令，将程序包从制品仓库下载到本地，再通过作业平台的文件分发功能，实现大文件的快速分发。

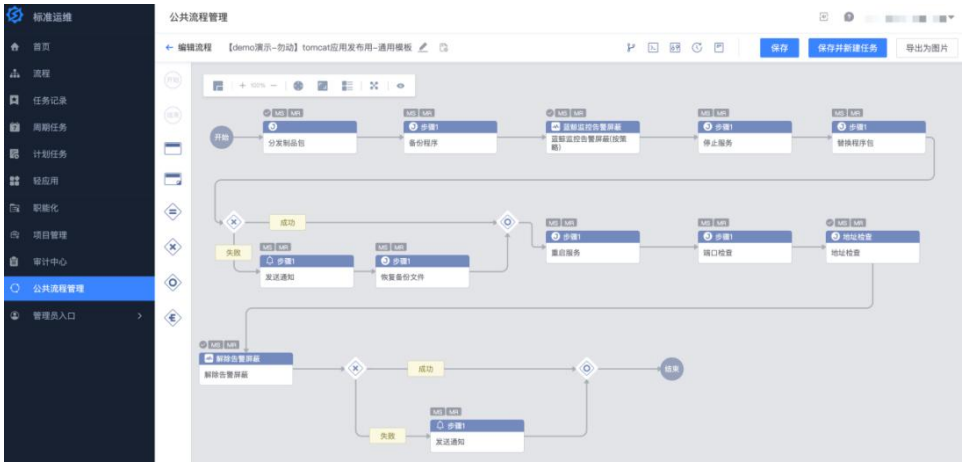
通过兼容设计，将两种类型的制品包统一管理起来，用同一套发布逻辑，完成制品包的分发和部署，实现自研系统和外购系统的一体化。



(2) 传统和云原生的一体化

公司的传统应用部署在主机上，发布时需要登录主机操作，而云原生应用部署在 K8s 集群上，发布时需要执行 kubectl 命令完成。另外部分工具的主机发布任务和容器发布任务存在割裂，无法统一执行。而应用发布中心支持一个任务同时兼容主机发布和容器发布，实现一次审批一次编排。

针对主机场景，使用管控平台纳管所有主机，标准运维编排所有主机发布任务。并且针对同类型的业务系统，规范建议复用相同的标准运维流程，无需每个业务团队自己建设。通用化规范化发布流程建议：



(1)分发制品包：将具体的制品包名称、分发 IP、分发路径设置为变量，由业务系统在发布时传入，这样所有业务系统的制品分发复用这一个规范分发节点

(2)备份程序：完成原有位置的程序备份

(3)屏蔽告警策略：传入目标主机 IP，或者屏蔽策略 ID，停服务前自动完成告警消息屏蔽。

由于变更时存在服务启停，变更期间必然产生大量无效告警，因此需要将变更目标主机的进场告警屏蔽。此操作也是引用一个通用的监控平台原子。

(4)停止服务：执行停服务脚本

(5)替换程序包：替换程序包

(6)重启服务：执行启服务脚本

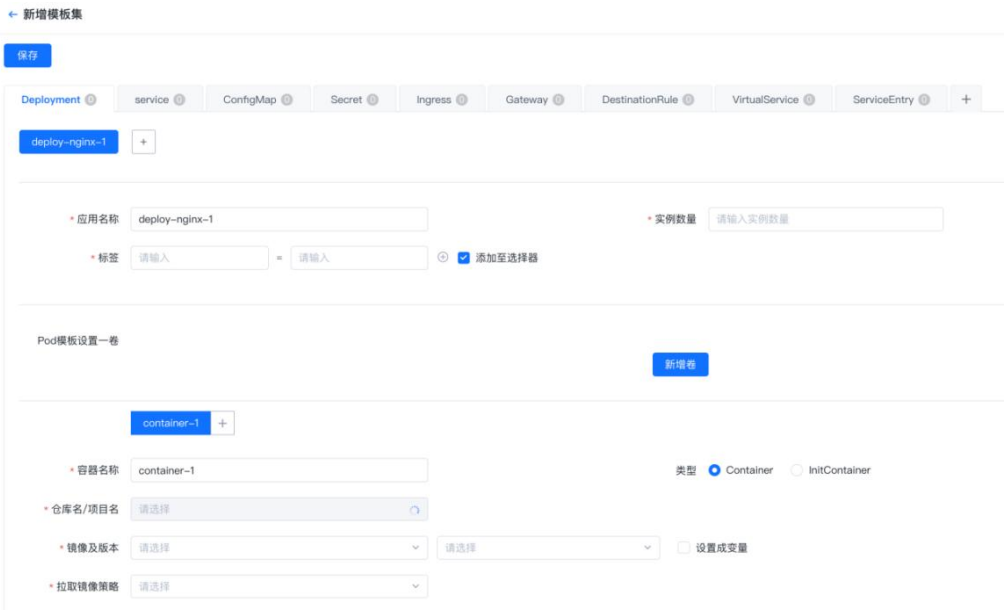
(7)端口检查：检查进程端口是否启动

(8)地址检查：检查业务访问地址访问是否正常，编排一个通用检查脚本，具体 URL 通过参数变量传入，多业务系统复用此脚本

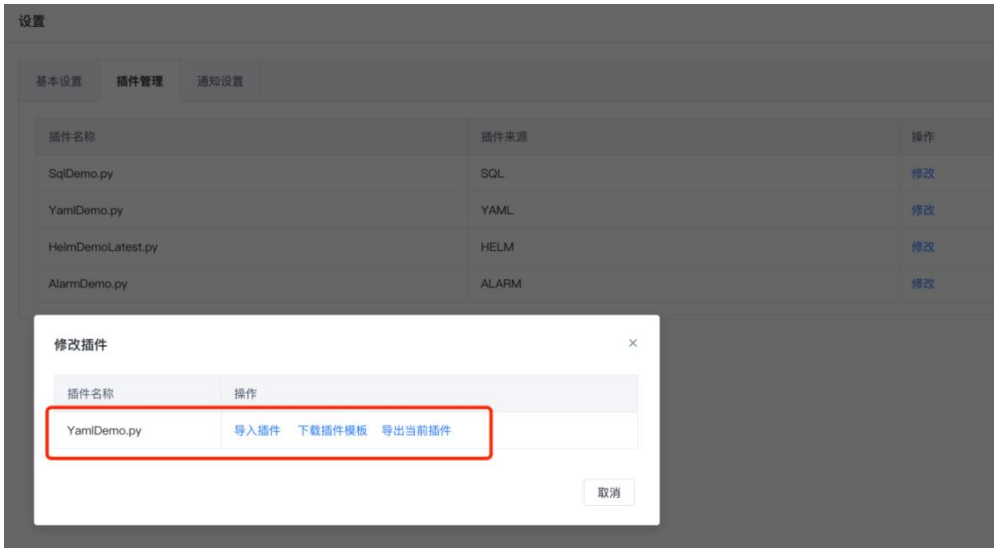
(9)解除告警屏蔽：变更完成之后，自动解除告警屏蔽

(10)发送通知

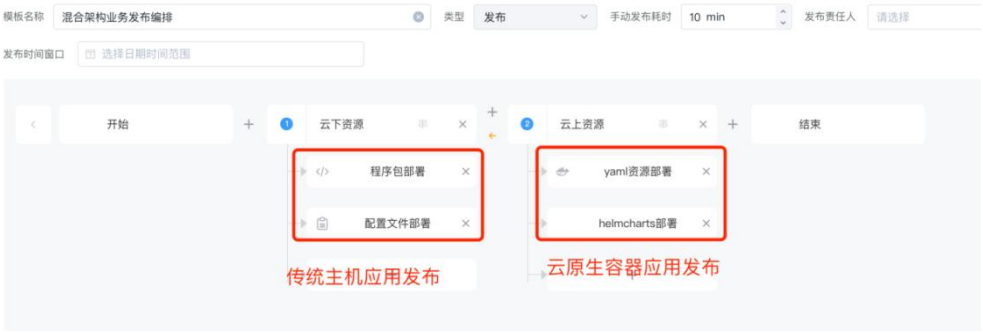
针对云原生的应用，变更平台支持编排容器应用的 yaml 文件，提供两种编排方式，一种是原生管理方案，支持运维人员直接导入 yaml 文件进行管理；一种是将 yaml 文件中的关键参数抽取成表单参数，运维人员只需要下拉选择即可，降低容器应用的编排门槛。



做容器发布时需要调用 K8S 接口，变更平台通过 yaml 插件集成接口实现容器发布。



因此变更一体化平台，支持将传统主机发布节点，和云原生发布节点，编排到同一个发布任务中，实现“传统和云原生的一体化”。



此外考虑到还存在数据库的 SQL 更新、纯配置更新等不同场景，变更平台在调度引擎时，单独设计的插件模块，支持通过可扩展的插件去接入更多发布类型。

(3) 测试环境和生产环境的一体化

为了提高生产环境的发布成功率，要求生产环境的发布操作，尽可能 1 比 1 复刻测试环境的发布操作，达到测试环境和生产环境的一体化。

IT 运维部的 SRE 专家将发布操作拆解成三个环节，每个环节都需要保证测试环境和生产环境的一致性。

环节一：应用拓扑

在测试环境一般只会有一套集群，而生产环境两地三中心，至少存在三套集群，一般会认为测试和生产的环境没办法保持一致，从而忽略测试环境的维护。但实际上可以在应用拓扑层级保持一致，并且按照同一套规范去维护。

公司在测试环境、预发环境、生产环境都使用“业务-子系统-集群-模块”四层拓扑架构，减少架构层面的疏漏。

环节二：制品管理

在测试环境和生产环境都部署独立制品仓库，开通从测试环境仓库到生产环境仓库的单向定点网络，并创建定期制品同步策略。

开发人员在 CI 流水线阶段完成构建打包之后，就会自动入库，将版本包存储到制品仓库中；在测试人员完成测试后，自动触发制品晋级任务，自动将制品从测试环境的制品仓库同步到生产环境的制品仓库，用于生产变更。

这样确保生产环境要发布的制品包，是经过测试，并且和测试环境的版本是一致的，减少运维人员在过程中的手动下载上传干预，全流程自动化流转。

环节三：发布流程管理

发布流程时生产环境变更中的最核心的部分，更要求生产环境的发布流程，是在测试环境中完整执行过的。且为了降低人工重新配置的复杂度以及出错概率，变更平台中的发布任务模板和标准运维执行流程，都是支持从测试环境导出，直接导入到生产环境，保证发布流程的一致性。

变更一体化平台从应用拓扑、制品管理、发布流程管理三个维度，实现测试环境和生产环境的一致性，提高生产环境变更的稳定性。

(4) 敏捷和安全的一体化

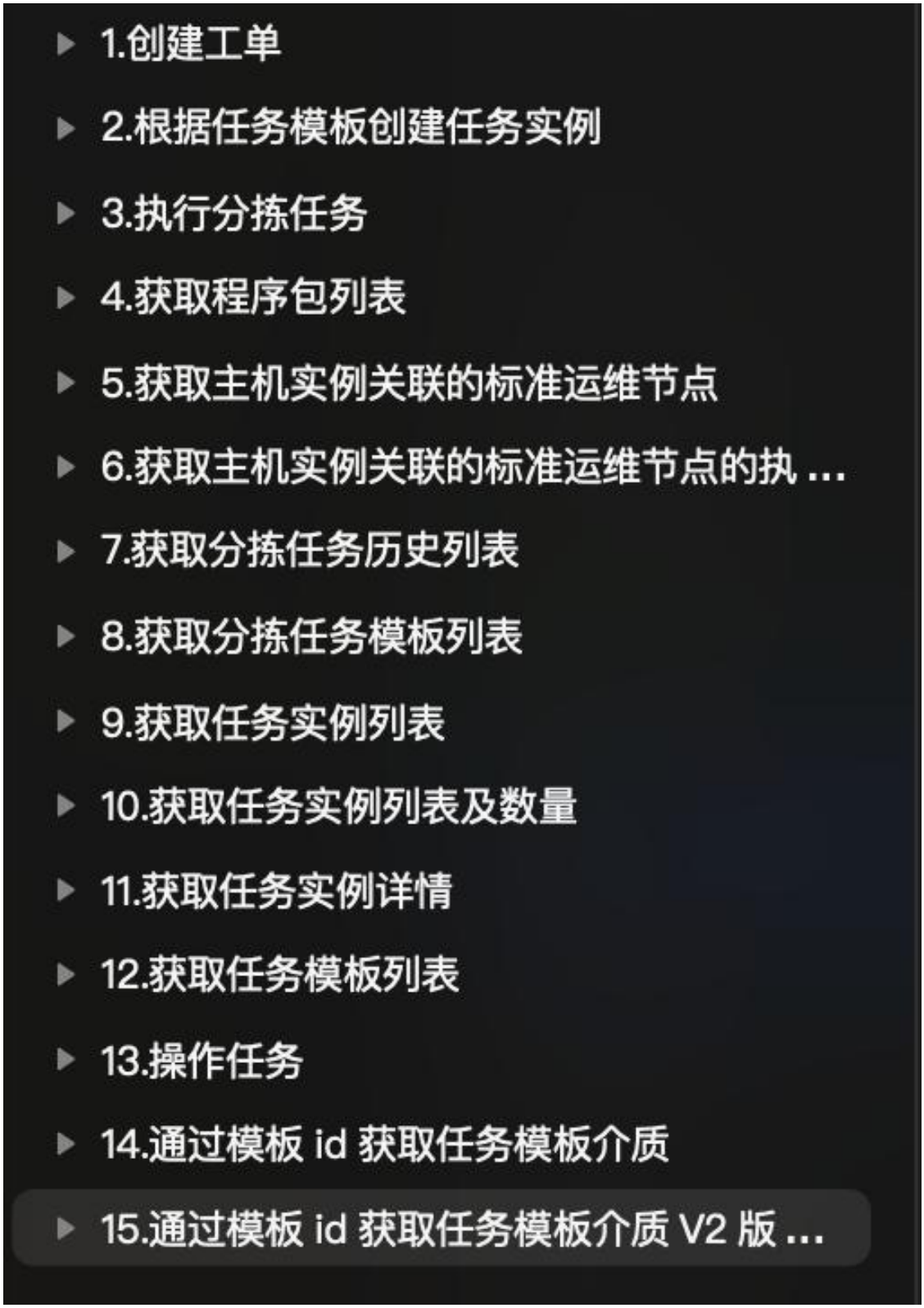
在测试环境，开发人员通过 CI 流水线做持续集成和持续部署，实现快速开发迭代，因此对变更平台的要求是：能够作为 CD 节点完美到 CI 流水线中，实现应用的快速部署。

变更平台开放了丰富的 OpenAPI 去满足 CI 流水线的要求：

search_template 接口支持传入业务系统查询发布任务模板

create_task 接口支持传入任务模板 id 创建发布任务

operate_task 支持操作发布任务，包括执行、暂停、灰度、停止



在生产环境，最重要的要求是稳定安全，因此需要对业务变更操作进行更多管控。一方面是在重要时间段内不允许变更，比如证券交易期间，重大会议期间；另外一方面所有变更操作需要和工单系统对接，运维领导审批通过之后才能执行发布。

创建业务保障策略，在重大会议期间、节假日期间，从技术上限制运维人员对生产环境进行变更操作。目前大部分金融企业都是采用管理规范进行约束，在此期间不允许进行变更，但无法落地到技术层面进行强管控，还是存在业务团队冒险变更的风险。深层原因是，没有一体化的变更平台，每个业务团队使用各自的变更工具进行操作，无从限制。

业务保障策略支持灵活配置满足不同场景需求：

- 支持设置策略状态，默认启用/停用
- 支持限制超管
- 支持选择生效业务，不同业务的安全级别不同，不同时间管控要求不一样，C 端业务强管控，内部系统弱管控；以业务系统级别管控，而不是具体 IP 维度管控，降低运维复杂度
- 支持自定义设置生效时间，日常证券开市时间是周期性周一-周五，节假日是固定时间段

通过工单插件对接现有的变更工单系统，实现生产变更操作的审批。

在测试环境，通过 OpenAPI 集成到 DevOps 流水线中，实现快速发布，达到“敏捷”要求；在生产环境通过业务保障策略和工单审批策略，实现发布管控，达到“安全”要求。

总结及展望

变更一体化平台建设是一套长期动态的系统工程，随着应用系统及周边配套平台联动，还有很多工作在规划中，有很多能力在持续完善，例如度量、复盘、规范化等等。另外一些源自互联网的新方案例如 argoCD 等相对于传统的 K8s 部署方式集成度更高，可以探索其与合规跟质量的结合点。

4.4.3 游戏 GitOps 发布管理实践

SRE Elite 收录点评：

1，这是一个典型的 GitOps 发布案例，对于互联网企业来说很常见，对于传统行业 SRE 具备一定的参考价值。它适用于流程和权限管控相对宽松而执行频率较高的测试环境、预发布环境、体验环境及低社会敏感应用的生产环境等。

2，该案例描述了一个大型 SRE 团队全球化应用部署管理的场景，对不同细分职能的分工和协作模式做了阐述，对不同类型权限的操作者提供了差异化入口及审计方案，同时也兼顾到了不同使用习惯的开源组件集成，体现了鲜明的互联网风格。

(一) 背景及设计原则

为了更好地加速业务创新和解决业务规模化的挑战，云原生应用架构与开发方式应运而生，与传统单体应用架构相比，分布式微服务架构具备更好的、更快的迭代速度、更低的开发复杂性，更好的可扩展性和弹性。然而，正如星战宇宙中，原力既有光明也有黑暗的一面。微服务应用在部署、运维和管理等方面的复杂性却大大增加，DevOps 文化和背后支撑的自动化工具与平台能力成为关键。

在容器技术出现之前，DevOps 理论已经发展多年。但是，如果“研发”与“SRE”不能用相同的语言进行交流，用一致的技术进行协作，那就永远无法打破组织和文化的藩篱。Docker 容器技术的出现，实现了软件交付流程的标准化，一次构建，随处部署。结合 IaC 和 Kubernetes 声明式的 API，可以通过声明式的方式实现自动化的持续集成与持续交付应用和基础设施，大大加速了研发和 SRE 角色的融合。

此外 GitOps 也是一种符合 DevOps 思想的运维方式，GitOps 以 Git 仓库作为唯一的事实来源，储存声明式配置，并通过自动化工具实现环境和应用的自动化管理。Git 实现了版本控制、回滚、多人协作；声明式配置保证了配置的可读性和事务性；自动化部署消除了人为错误，调高了部署效率和准确性，同时也保证了多环境的一致性。所以 GitOps + 声明式配置 能够很好的解决传统运维的痛点，提高部署效率，保证部署的一致性。

（二）体系设计及关键流程

(1) SRE 分层服务模式

在实施 GitOps 的过程中，我们需要对 SRE 团队进行分层的能力建设，相距业务团队由远及近可以划分为“平台 SRE”、“定制 SRE（可选）”、“服务 SRE”、“On-call”四个团队：

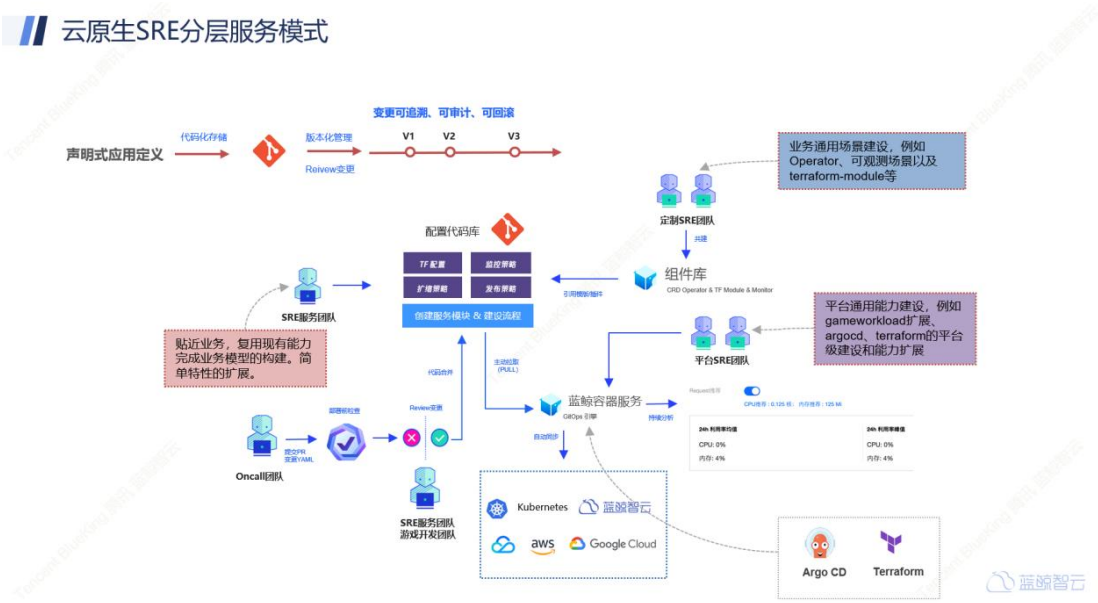
1. “平台 SRE”负责通用能力建设，针对游戏的特性定制和构建容器服务，提供一站式游戏微服务云原生能力为核心，完成游戏场景的 workload 定制、ingress 扩展、GitOps 以及 Terraform 等能力与容器平台的集成。

2. “定制 SRE”是可选组织，人数较少，视企业应用的异构程度及数量，以及“服务 SRE”团队的能力情况酌情设置，其职能可以合并到上下两层，如果设置该组织，可以负责

通用的声明式自助服务场景建设，开发基于业务场景抽象的 Kubernetes CRD Operator、声明式全栈可观测场景，以及可复用的 Terraform module 等。

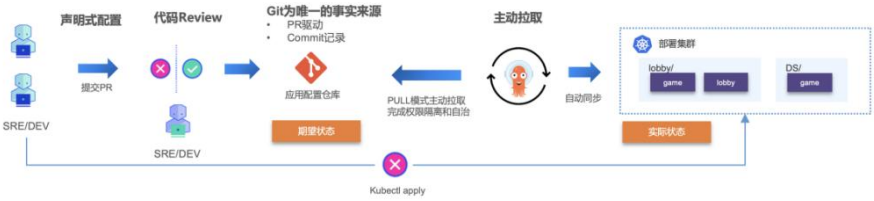
3. “服务 SRE”紧密结合业务需求，利用“定制 SRE”提供的方案、工具和技术对业务的上云改造方案以及预期效果进行评估，并完成业务应用模型构建。实现业务基础设施的编排，业务应用的运行管理，以及业务的扩缩容策略以及流量管理策略的配置。同时将声明式的配置代码存储在代码仓库中，基于 Git 实现配置的版本化管理和权限管理。

4. 全球 On-call 团队则基于统一的 Git 变更流程，对应用以及应用所依赖的基础设施配置进行变更。变更内容在经过代码审核后将被合并到主干，GitOps 主动监测到配置的变更并同步变更到业务环境中，并确保环境配置的一致性。



(2) GitOps 持续交付

GitOps 能力基于 Argo CD 实现。GitOps 作为一种应用交付模式，极大地简化了传统 CI/CD 流程。在持续集成（CI）阶段，SRE 团队的焦点集中在代码编译和容器镜像的构建上，避免了在蓝鲸 CI 平台（蓝盾）中存储 Kubernetes 的认证信息，从而增强了系统安全性。进入到持续交付（CD）阶段，运维人员仅需通过 Pull Request (PR) 或 Merge Request (MR) 向代码仓库提交变更，GitOps 引擎会主动地从 Git 仓库拉取配置，并自动地将这些配置与当前集群的状态进行对比和同步。这种“拉取（Pull）”模式相较于传统的“推送（Push）”模式的最大优势在于，它允许集群主动拉取配置，避免了编写和维护复杂的流水线脚本或手动触发更新流程的需求，简化了操作并提高了自动化水平和系统的安全性。

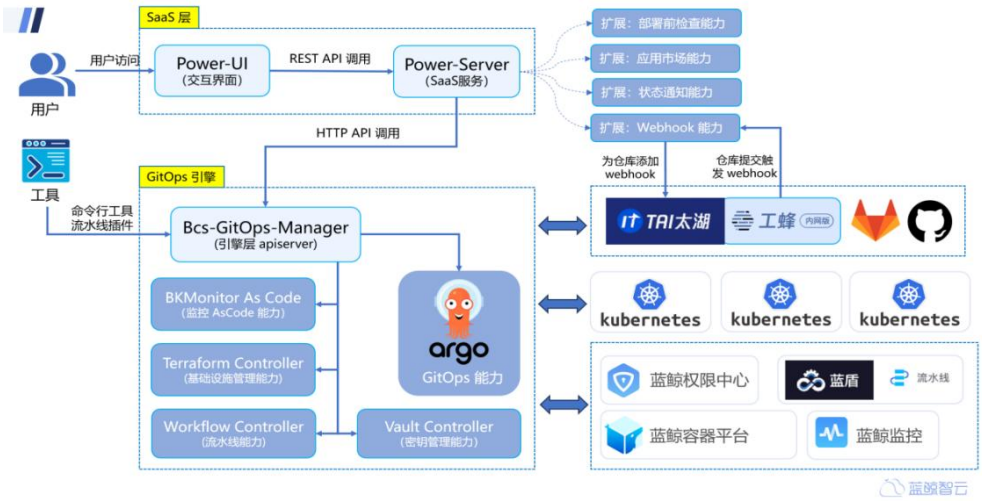


(3) 声明式自助服务能力

针对游戏 SRE 团队而言，我们围绕游戏生态构建了一系列声明式和自助式的服务场景。通过向蓝鲸平台工程靠拢，我们实现了服务的可衡量、可管理和可复制性，同时不断进行优化和创新，以适应不断变化的需求和挑战。

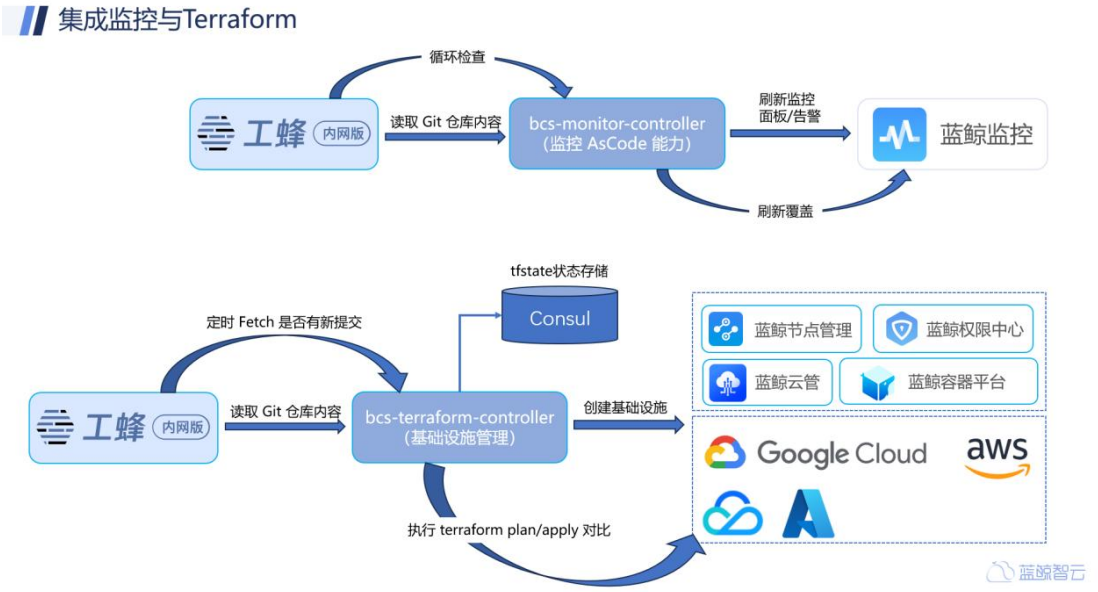
游戏 SRE 团队根据游戏场景提出了云原生架构下应用的三层架构模型，运维能力层声明了应用的交付流程、扩缩容策略，应用运行层声明式了应用的基础元素，包含服务、服务配置和数据库脚本，基础设施层则声明了应用运行时的环境和基础设施，基于 Everything as Code 理论将应用模型存储在代码仓库中，实现统一演进，可审计、可回溯。

在面向用户层，为不同团队设计了多种入口，以蓝鲸 SaaS PowerApp 为例，为业务运维人员提供了更加友好的操作界面和更加完善的权限控制。



除此之外，游戏开发人员可以通过蓝鲸 CI（蓝盾）触发，合作企业工程师可以通过标准运维 App 触发，平台 SRE 工程师可以通过 BCS-SaaS 触发，同时也保留了命令行操作的入口。

最后，我们增加了 ArgoCD 的扩展，例如密钥管理、代码库协议转换、Dry-Run、历史差异对比、动态资源信息等，在基础设施管理层面，集成了监控与 Terraform 等基础能力。



（三）总结及展望

权威机构 Gartner 认为，在云原生环境下，我们需要建立面向服务的 DevOps 理念，通过平台级能力为服务团队提供 DevOps 的支持。这种模式被认为是最适合微服务的 DevOps 服务模式，其核心价值主张就是“X as a Service”。

在这个背景下，游戏 SRE 团队的任务就是为游戏研发团队提供一个稳定的工程平台，帮助他们屏蔽复杂的底层基础设施。这个平台包含开发者所需的各种工具、自助的开发者服务界面、以及可复用的模板，以满足开发团队的日常需求。

在这里，平台的编排构建可以由 IaC 实现统一的编排能力。而基于 GitOps 则解决了声明式配置管理以及持续交付的问题，实现全球地域的一致性游戏应用交付和管理。

而平台工程的出现，其实是 IaC 和 GitOps 技术发展的必然结果。在现代软件开发中，由于技术复杂度和变化速度的加快，以及微服务架构的广泛使用，研发团队不再能够有效地关注所有的技术细节。平台工程就是将这些通用、底层的工作抽象出来，由 SRE 平台团队来提供，从而让研发团队能够专注于他们的主要目标。

5 故障应急

5.1 故障体系

5.1.1 故障定义

故障的定义如下：在日常运营中，无论任何原因导致我们的服务中断、服务品质下降或用户服务体验下降，都被视为故障。但不包括用户环境活自身操作引起的问题。

5.1.2 故障等级

故障等级是衡量故障影响度的标准，数字越小级别越高。通常业界是以 P0 P1 P2 P3 P4 P5 等作为序列， P0 为影响度最高的故障（也有部分公司以 P1 作为最高级别）。

影响度参考指标包括客户投诉量、资金损失金额、受影响用户数，故障时长，业务重要性等。

通常情况下， 故障等级由 GOC、业务开发、质量团队、业务 SRE 和业务稳定性等部门共同制定。而且相关故障等级需要提交业务负责人审批，确保业务方对相关故障级别标准的认可。并定期由 GOC 统一审核故障定义和等级，并公示，确保大家按统一标准执行。

故障等级的作用，我们可以基于这些定义，开展可观测性建设（配置针对性的告警体系以及预案）和平台能力构建（包括而不仅限于可用性架构设计及评估， 压测，混沌演练，容灾演练），目的是提前识别潜在问题，进行预防和保障。

在故障发生后，需要对故障进行评估，并采用就高原则， 如果一个案例符合多条故障定义，通常以最高等级为准。

5.1.3 故障序列

高稳定性团队可能会面临因非技术问题（如业务运营方配置失误，网络运营商的抖动或其他外部因素）引起的故障，因此需要明确技术故障与外部原因之间的区别，因此需要引入故障序列。

故障序列是根据故障的根本原因和实际影响进行分类的方法，区分"故障根因"和"实际影响"。

在定义故障序列时，主要考虑以下几点：**技术责任：**

判断故障是否由技术原因引起，如是，则可能扣减故障预算。我们关注 P1 到 P5 等级的故障，因为它们通常涉及技术责任并影响业务。

故障记录与配额：

评估年度稳定性需考虑故障记录和配额。许多公司限制每年故障次数，超出后可能影响绩效。因此，非技术原因的故障，如业务操作失误，不扣减预算。例如，外部原因的故障标记为"W"，业务操作引起的标记为"S"。公司可根据这些分类制定定义更明确相关的稳定性指标。

5.1.4 关键业务场景

组织有众多业务，究竟该优先保障哪些？我们应聚焦于最重要的业务，而非面面俱到。因此，我们定义了关键服务接口和结果点，即关键业务场景，也称 GOC（Global Operation Center）场景，也有公司叫 NOC (Network Operation Center）)。只要确保这些 GOC 场景的稳定，组织就不会出现重大问题。

关键业务场景通过以下三个领域定义：

- 场景描述：明确需要保障的业务场景及其功能。
- 场景影响评价：设定衡量指标（如成功量、成功率、响应时间）来评估场景的稳定性和性能。可以理解为 SLI /SLO 体系的构建。
- 故障等级：明确场景故障时可能涉及的最高等级。重要业务需重点保障以避免高等级故障。

实例说明

以登录功能为例：

免密登录：使用成功量作为衡量指标，若成功量下降，则可能触发相应故障等级。

账号密码登录：使用成功率进行衡量。

GOC 场景定义的目的：

通过定义 GOC 场景，聚焦关键服务接口和结果点，**驱动和确保公司整体稳定性。**

围绕 GOC 场景，开展容量评估、压测、限流、监控、变更管理等各项工作。

- 业务链路：清晰了解所保障业务的核心目标和完整的业务链路。明确稳定性保障团队，当问题发生时，能够迅速召集相关人员共同解决。
- 容量模型：从接入层到中间件，再到下游数据库，全面了解系统架构；建立性能基线，明确资源成本；描绘流量模型，识别不同时间段的流量特征（如早

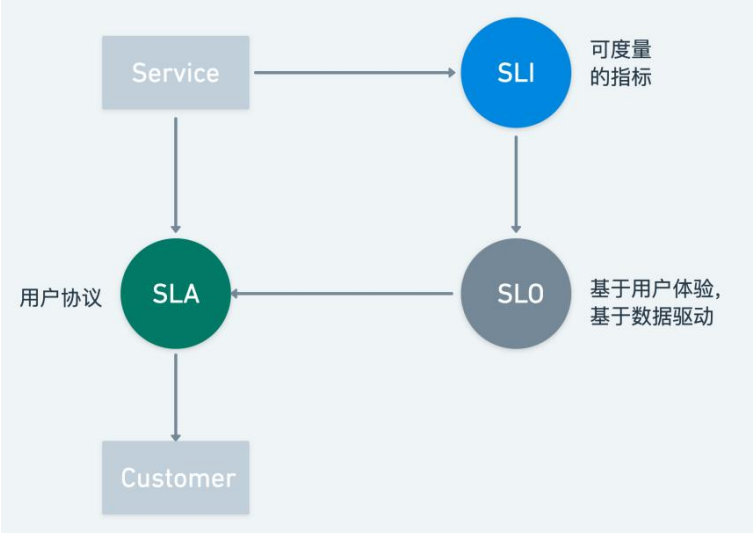
高峰、午高峰、晚高峰）；定期进行全链路压测，评估系统容量，确保在高负载下的稳定性；根据容量模型，设置合理的限流策略。防止突发流量超出系统承受范围，保障服务稳定。

- 可观测性：明确业务的关键监控点和指标（成功量、成功率、响应时间等）。建立实时监控和预警机制，及时发现和处理异常情况。
- 变更防御：建立变更感知体系，构建变更流程机制，对核心业务实施严格变更管理，确保安全可控，保证所有变更可被定位，有迹可循。
- 应急能力：设定应急响应率目标，设定应急组织整形，提升团队快速定位和解决问题的应急止血能力，确保故障迅速恢复。
- 历史故障：对历史故障进行分析复盘，找出稳定性薄弱环境，并进行跟进。

在明确了关键业务场景后，需要进行关键业务的 SLI / SLO 体系的构建。

5.1.5 应用服务 SLI/SLO/SLA

5.1.5.1 什么是 SLI/SLO/SLA



(1) SLI（Service Level Indicator，服务水平指标）是衡量系统或服务性能的具体指标，直接反映服务的质量或稳定性。SLI 通常用来监控特定的服务性能数据，如可用性、延迟、错误率、吞吐量等。它是实现 SLO 的基础，通过 SLI 的实时数据监测，可以判断服务是否达到了预期的 SLO 目标。

一般把这些指标分为 3 层：用户体验、中间件、基础设施。用户体验主要是指和用户强相关的一些功能监控，比如登录、购买等功能，这些往往用户的关键路径，也是我们应

该关注的点；中间件主要指在架构中用到的开源组件等；基础设施就是指网络、物理服务器、物理电源、磁盘阵列卡等，在建设 SLO 时优先梳理用户体验层指标。

- SLO（Service Level Objective，服务水平指标）是对服务质量的量化目标。它定义了系统或服务在一段时间内需要达到的性能或可靠性标准，通常用百分比来表示。例如，如果设定了 99%的 SLO，那么服务在一个月内至少要有 99%的时间可用，或其响应时间需要满足一定的标准。SLO 通常基于用户体验和业务需求来设定，既要满足用户期望，又要在技术和成本上实现可行。
- SLA（Service Level Agreement，服务水平协议）是服务提供商与客户之间达成的正式协议，规定了服务的期望水平以及未达标时的补偿措施。SLA 通常会包括具体的 SLO 目标，例如服务可用性为 99.9%，以及服务中断的赔偿条款。SLA 的目标是明确责任和预期，确保服务提供方和客户之间的透明度和信任。

在特定情况下，可以考虑将 SLA / SLO 与绩效挂钩，这样可以确保团队的目标与组织的服务质量目标一致。团队成员可能更加积极地工作以满足或超越这些目标。同时这样也为团队设定了明确的期望，并能实现数据驱动的评估方式。但是，与绩效挂钩，团队成员可能会过度专注于短期指标，而忽视长期目标和创新。建议每个组织根据客观的实际情况进行使用。

5.1.5.2 如何建设 SLI/SLO

SLO 建设步骤：确认 SLO 预期 -> 梳理关键链路 -> 梳理 SLI（1 原则+1 方法+关键链路）-> 上报 SLI -> 创建监控策略 -> 计算 SLO -> SLO 指标呈现

1. 确认预期

和业务负责人、产品、策划、开发等相关干系人一起确认该功能的预期，即 SLO。需要注意的是，SLO 的设定需要根据服务实际需求来制定。趋近于 100%的 SLO，将会消耗巨大的成本，但是用户可能并感受不到变化。过低的 SLO，将会让用户的感受变差。我们需要确定在不影响用户体验的情况下，确定一个合理的 SLO。

2. 梳理关键路径

梳理用户从功能入口到使用该功能的关键路径，比如：

app 启动 --> app 更新检查 --> app 更新 --> 登录 --> 获取昵称 --> 展示昵称

3. 梳理 SLI

原则：优先梳理用户体验层 SLI，即用户可以感受到的功能、按钮

方法：VALET，从 V（容量）、A（可用率）、L（延迟）、E（错误）、T（人工提单，保底分类）5 个维度来梳理，避免漏掉 SLI

将这 2 个原则套在第 2 步关键路径的每个节点，即可梳理出一份相对完整的 SLI 列表

4. 上报 SLI

对于没有 SLI 需要客户端或服务端从新上报到监控系统，对于已有的 SLI 需要校验其准确性

5. 创建监控策略

针对每个 SLI 需要创建一条监控策略，策略的告警阈值根据第一步定的 SLO 来确定。举个例子，若 SLO 定为 99%，则"app 更新检查成功率"指标的告警阈值为＜99%

6. 计算 SLO

围绕业务稳定性衡量，一共设计了 5 个指标, 多个维度，并基于监控系统计算出相关指标的值。以下的表格是一个指标和维度的事例的组合,供参考，通常 SLO 的指标维度，一定会包含时间维度，服务维度，其他可以根据自己的业务特性自行定义。

维度信息

维度	含义	枚举值
VALET	VALET 方法论，分别代容量、错误、延迟、可用率、工单	Volume Availability Latency Error Ticket
range_time	固定的时间范围，单位：天	1 7 30 180 365
strategy_name	监控策略名	策略名
strategy_id	监控策略 ID	策略 ID
bk_cc_biz_id	业务 ID	
bk_biz_name	业务名	

指标信息

指标	含义	计算公式	包含维度
slo	汇总的 SLO 指标，代表当前场景的稳定性	SLO=（总时间-不稳定时间）/总时间 不稳定时间=多个 SLI 告警时间并集	range_time
slo_error_time_info	各个 SLI 错误消耗详情，根据该指标，可看出哪个指标的不稳	slo_error_time_info=SUM（SLI 告警时间）	velat range_time strategy_name strategy_id bk_cc_biz_id

指标	含义	计算公式	包含维度
	定时间最长 每个 SLI 独立计算		bk_biz_name
slo_error_time	汇总之后的错误消耗	不稳定时间=多个 SLI 告警时间并集	range_time
mttr	告警平均恢复时间，值越小越好	mttr = 故障总时间 / 故障次数	range_time
mtbf	平均告警间隔时间，值越大越好	mtbf = 正常时间 / 故障次数	range_time

7. SLI/SLO 指标呈现

SLO 的展示一定要基于对应 SLI，目的在于让相关干系人能直观的看到当前 SLI 下的 SLO、错误消耗等指标。展示的思路根据不同的用户，大致可以分为三种方式：

(1) 业务场景视角的展示。针对业务某一个 SLI 或者某个场景做深度的 SLO 数据展示。可以分为多个仪表盘，每个仪表盘再按照 VELA 维度展示该功能所有 SLI 指标。达到效果：每个 SLI 的实时数据、可自定义时间范围展示 SLI 指标的曲线及具体值。更聚焦在某个场景或某个 SLI 的深度展示和分析。例如：某业务登录场景的 VIP 延迟、鉴权成功率、用户信息拉取延迟等登录场景核心链路的 SLO 详情展示。大多用于故障定位，排查问题。

(2) 业务整体视角的展示。从业务的角度，展示多个 SLI 的 SLO 数据，将多个功能的 SLO 指标汇总在一起展示。达到效果：通过该仪表盘查看各个功能过去不同时间维度的 SLO，用于判断各个功能的稳定性（slo）；当某个 SLO 低于我们预期时，可以定位到具体影响的 SLI 指标（slo_error_time_info、slo_error_time）；各个功能的告警发生频率（mtbf）；各个功能的告警恢复时间（mttr）。例如：某业务的登录、支付、上传、浏览、语音等核心功能的 SLO 整体展示。大多用于单业务稳定性评估。

(3) 多业务整体视角的展示。从整个公司或集团的更高的视角展示多业务的 SLO 数据，将公司或集团主要业务模块 SLI 和 SLO 定义清晰，并展示在一个大屏上。例如：公司所有业务登录成功率、公司所有业务信息发送成功率、所有业务添加购物车成功率等。用于评估整体技术中台能力，整体服务质量。

整个 SLI/SLO 的展示，从微观业务场景细节，到宏观业务基本状况，根据不同的使用场景，不同的用户需求，可以拆分为不同展示模式，汇总不同量级的数据，灵活展示。

5.1.5.3 如何持续迭代 SLI/SLO

SLO 出来后，可以用实际值和之前的预期做比对，分为以下 2 种情况

1. 实际 SLO < 预期 SLO。比如实际值为 95%，预期为 99%

此时可观察 SLI 的错误消耗，先对错误消耗最大的 SLI 进行调查，如果是指标质量问题导致 SLO 低于预期，那则修复 SLI 后再进行观察

如果指标质量一些正常，也无错误数据，那说明指标的实际情况就是这样，确实达不到预期值。此时又可分为 2 种情况：

业务架构不合理或部署方式不合理导致的 SLI 指标达不到预期：比如架构单点导致用户访问不稳定，集中部署导致用户访问延迟加大，此时应该利用 SLO 数据推动开发进行架构调整，推动运维进行部署优化，从而提升用户体验

SLO 预期高于现实：可能现实条件确实达不到我们的要求，此时应该要放宽 SLO，降低 SLO 预期。否则我们的 SLO 指标会一直处于不达标状态，开发、运维也无能为力，这样的结果会造成团队恐慌，一定不是我们想要的。

2. 实际 SLO > 预期 SLO。比如实际值为 99%，预期为 95%

SLI 覆盖率低：确定关键链路 SLI 是否梳理完整，如果有缺失，则补齐，先提高 SLI 在关键路径的覆盖率

SLO 预期太低：这么导致我们对用户体验的衡量不够准确，可能会出现用户体验很糟糕，但是我们 SLO 数据一切正常，导致与我们目的背道而驰。此时正确的措施应该是收紧 SLO，提高预期

3. SLO 迭代小结

从上面的分析来看，迭代 SLO 主要分为 2 大类：实际 SLO < 预期 SLO 和实际 SLO > 预期 SLO。

实际 SLO < 预期 SLO：一般情况下我们首先要做是看为什么没有达到预期的 SLO，而不是调整 SLO 的数值。只有有充分理由说明之前的 SLO 制定存在不合理，那么我们才能放宽 SLO。

实际 SLO > 预期 SLO：实际的 SLO 比预想的要好，我们可以分析是之前的目标设置的低了，还是我们投入了超过预期的成本。如果这个服务不需要这么高的 SLO，而我们也需要降低一些成本，那么可以适当放宽 SLO。

对于 SLO 的持续迭代，可以从下 5 个步骤执行：

- SLI 质量优化

- SLI 覆盖率提升
- 架构调整
- 部署优化
- 放宽或收紧 SLO

5.2 故障应急体系

5.2.1 故障应急体系设计原则

故障应急体系设计应以快速恢复业务为核心目标。故障应急生命周期通常包括故障发现、故障诊断、故障恢复、故障复盘四个阶段。通过完善的监控与发现机制、有效的故障诊断与定界、及时的恢复措施、全面的复盘与改进，再配套适当的组织流程、定期演练和持续优化，以及相关工程平台的落地，是实现故障应急响应能力提升、保障业务连续性的关键。

5.2.2 故障应急流程设计

5.2.2.1 组织支撑

5.2.2.1.1 组织支撑的重要性

跨部门协作的必要性：故障管理涉及多个团队和部门，单凭某一部门无法独立完成。组织层面的支撑能够打破信息孤岛，促进跨部门的高效协作。

明确职责与权限：通过组织结构的设定，明确各级人员在故障应急中的角色和职责，确保在紧急情况下，所有人都知道自己的任务和权限。

资源的有效调配：组织支撑能够保障在故障发生时，快速调用和调配所需的资源，包括人力、技术和物资。

文化与制度的保障：组织支撑还体现在文化和制度建设上，通过制度的制定和文化的宣导，提升全员对稳定性和故障应急的重视程度。

5.2.2.1.2 构建应急组织架构

1. 设立专门的应急管理委员会

建立一个跨部门的应急管理委员会，负责统筹公司的故障应急工作。该委员会应由各业务单元的代表组成，具备以下职责：

制定应急策略和政策：包括故障定级、责任划分、应急流程等。

协调各部门的应急工作：确保各部门在应急过程中步调一致。

推动应急文化建设：通过培训、宣导等方式，提高全员的应急意识。

2. 明确应急角色和职责

在组织内部，需要设立不同级别和职责的应急角色，因每个公司所在的行业以及业务特性不一样，没有办法一概而论， 因此以下给出的部分参与角色设计仅供参考。

应急总负责人：

负责全公司的应急指挥和协调，具备决策和资源调配的权限。

业务线稳定性负责人：

专注于特定业务线的稳定性保障，负责快速响应和解决问题。

GOC 工程师：

需要 7x24 小时应急盯屏值班工作，针对全站各类风险事件，及时发起应急响应流程，进行有效的应急调度，负责初步的故障处理。根据公司规模， 可能还有分为部门级，公司级等多级的 GOC。

OnCall 工程师：

需要确保自己的电话或其他通讯工具在值班期间始终保持畅通，以便随时接收和处理来自系统或同事的告警信息。在收到告警后，OnCall 工程师需要迅速确认问题的性质和严重程度，并调用相关的方面进行故障应急处理。参与 OnCall 的人员，可能是各个产品线技术负责人，又或者是研发，OnCall 会参与日常问题的响应。与 GOC 工程师类似，OnCall 工程师同样会根据公司及业务实际情况，设立分级的 OnCall 制度并配备相应的人力。

技术应急人：

简单来说，就是研发、质量、SRE，负责本业务的故障跟进及处理；并负责本业务的风险场景识别、监控覆盖、应急快恢能力建设、应急人员培养、风险意识培训。

此外，蚂蚁集团在长期的工作实践中积累了宝贵经验，并首创了**"应急值班长"**

制度。值班长是公司应急响应体系中的关键角色，可以理解为对业务有一定了解，有组织力、决策力，且已经过 3 次以上符合预期的应急处置（包括线上的真实问题和演练问题）的专家，负责全公司的应急指挥和协调，具备决策和资源调配的权限。在应急情况下，值班长需要组织和协调各个团队（如 GOC 工程师、Oncall 工程师、研发、质量、SRE 等），确保快速响应和有效解决问题，实现应急止血。由于相应的要求很高，压力巨大，因此能担任值班长，是相关领域专家的荣誉， 公司也会进行相应的公示及表彰。

3. 建立高效的协同机制

明确的 OnCall 值守体系：设定一线、二线、三线值守人员，明确各自的职责和交接流程。

完善的应急响应流程：制定标准化的应急预案，确保在故障发生时有章可循。

畅通的信息共享渠道：利用即时通讯工具、故障管理平台等，实时共享故障信息和处理进展。

资源保障机制：确保在紧急情况下，能够快速获取所需的技术和人力资源。

5.2.2.1.3 强化文化与制度建设

1. 制度建设

制定应急预案和规范：包括容灾架构设计、演练章程、故障处理流程等。

明确奖惩机制：将稳定性目标纳入各部门的绩效考核，激励全员重视故障管理。

建立知识库和文档：总结故障处理经验，供全员学习和参考。

2. 文化宣导

开展安全生产月等活动：通过专题活动，提高全员的稳定性和应急意识。

培训与分享：定期组织培训、技术分享会，提升团队的整体应急能力。

新人培训：在新员工入职培训中，强调稳定性的重要性和应急流程。

5.2.2.1.4 持续改进与演练

定期开展应急演练：模拟真实的故障场景，检验应急预案的有效性，发现并改进不足之处。

故障复盘和分析：每次故障处理后，进行深入的复盘，分析原因，提出改进措施。

持续优化应急流程：根据实际经验，不断完善应急响应流程和机制。

5.2.2.2 故障发现

5.2.2.2.1 监控发现

监控发现是指 SRE 通过监控主动发现系统运行异常的情况。为此，SRE 应考虑以下内容：

- 监控指标体系设计

监控指标应按层次结构设计，形成一个监控体系金字塔。通常情况下，越靠近塔尖的指标越与业务紧密关联，用于感知用户的真实体验；越靠近底层的指标则与系统运行环境相关，用于故障定位和根因分析。

SLI 定义：

SRE 应优先考虑确定关键服务的 SLI（服务级别指标），这里的关键服务在不同类型业务会有很大的行业特性，如，电商行业会高度关注购物，支付等业务，游戏行业会关注登录，对局等业务。这些业务的指标选择有两大原则，一，优先选择与用户体验强感知的业务对应的指标，如请求成功率、响应时间等，二，能识别一个主体是否稳定，如错误率，容量指标，人工接入工单数等。

应用层指标：

另外，对应用层的监控非常有价值，常见的应用层指标有：延迟、流量、错误率、饱和度，线程池、连接池等指标。

技术组件监控：

对系统中各个组件的监控是确保系统稳定性的关键。组件监控应包括数据库、缓存、消息队列等关键组件的性能指标，如查询时间、命中率、队列长度等。

基础架构监控：

基础架构监控覆盖服务器、虚拟化平台、容器云平台、网络设备、存储设备等硬件设施的健康状况和性能指标。常见的基础架构监控指标包括 CPU 使用率、内存使用率、磁盘 I/O、网络带宽等。

在覆盖率上，SRE 要确保所有关键服务和组件都在监控范围内，包括第三方服务和 API。

- 指标标准化

SLO 的设定：与开发团队合作，基于 SLI 设定明确的 SLO（服务级别目标），如 99.9%的请求成功率或 200ms 以内的响应时间。SLO 是衡量服务质量的具体目标，设定目标有利于团队进行整体的协作。

统一框架：提供一个统一的开发框架，简化指标收集和监控配置的过程，确保关键性能指标的规范化。建议使用 OpenTelemetry 协议及 SDK 。

指标规范：制定明确的指标规范，促进跨应用和 IT 基础设施的协调一致。例如合适的指标名字以及合适的指标标签。

3. 告警触达

SRE 配置的监控策略，应以"故障影响程度"进行分级，不同级别的监控告警应通过电话、短信、IM、邮件等不同的渠道触及 SRE 人员，避免单一渠道告警堆积导致重要告警未被及时关注。当短时间内出现大量告警时，SRE 应对告警实现收敛能力，以便 SRE 可以对告警分类，及时梳理出故障范围。

4. 告警轮值

为了降低 SRE 长期面对告警的心理压力，在团队规模允许的情况下应实施告警轮值。当值 SRE 作为告警的第一责任人，负责组织开展后续故障处理事项。告警触达在第一责任人（A 角色）未在约定时间内响应的时候，可以继续通知第二负责人（备份角色、B 角色）或者第三负责人（C 角色）等。

良好的 SRE 实践应实现所有故障通过监控主动发现，以期达到 SRE 优先介入处理，降低对用户的影响。SRE 可以通过故障主动发现的占比，来衡量监控发现的覆盖程度，该比例越高越好

5.2.2.2.2 巡检发现

在业务系统实际落地过程中，难免存在预期以外的故障发生。所以在监控发现手段以外应安排人员对系统各个环节进行定期巡检，作为监控系统的补充。巡检与监控的区别主要有以下几点：

1. 指标差异。监控更关注状态、性能、容量类指标，巡检可根据 SRE 自身工作需要或者业务需要设置更为个性化的指标（如 Linux 内核版本、Linux 内核参数、系统启动时长、文件系统用户权限等）

2. 指标阈值不同。巡检往往设置比监控阈值更苛刻的指标，以便发现一些隐藏的风险。

3. 周期不同。巡检一般按小时、按天、按周、按月甚至年度执行，监控一般在分钟或者秒级。

4. 深度不同。业务系统、技术组件的深度巡检，通常可覆盖更复杂的隐患检查场景，例如，系统上下游处理容量、关键依赖的健康状态、关键技术组件的性能指标，可作为重要活动、重要节假日前的保障措施，也属于故障预防的一种手段。

SRE 可以考虑通过以下三个方面组织巡检：

1. 巡检周期

针对不同业务模块的重要程度，SRE 可以对各个模块实施不同的巡检周期。对于直接影响用户使用体验的核心模块巡检频率应至少一天巡检一次，非核心模块可按业务特性考虑降低巡检频率。

2. 巡检手段

SRE 应将需要巡检的内容提前整理为仪表盘或其他可视化内容，巡检人员应通过图表可直接了解当前系统指标是否满足 SLO。同时，SRE 应可方便地对历史数据与当前数据进行对比，可直接了解当前指标是否存在异常走势或变化

3. 巡检目标

巡检是为了查找系统中未知的故障模式或者隐患，因此 SRE 除了关注 SLO 是否达标以外，尤其需要关注重要指标的历史趋势是否发生变化、是否存在指标毛刺等，以期通过主动巡检的方式及早发现系统故障或隐患。

良好的巡检可以让 SRE 早于用户发现系统异常，及时止损。因此，可以通过巡检发现的问题与人工上报问题的比例来衡量巡检发现的质量，巡检发现的比例越高质量越好。而从另一个角度，巡检发现是作为监控发现的补充，巡检发现问题与监控发现问题的比例越低表示监控发现越为完善，此时比例越低越好。

总的来说，监控则更侧重于实时的响应和问题解决，而巡检更侧重于预防性的维护。两者都是有效的保证稳定性的方式，通常同时使用。

5.2.2.2.3 人工上报（舆情，客服，运营人员等）

在实际运营中，尽管自动化监控系统可以覆盖大部分的故障发现场景，但总有些情况是自动化监控所无法捕捉的，比如用户体验上的微妙变化、业务逻辑上的复杂问题等。因此，人工上报成为监控系统的重要补充。

人工上报通常涉及以下几个方面：

1.舆情监测：通过社交媒体、论坛、客户反馈等渠道，对用户的讨论和意见进行监听，从中发现可能的系统问题或用户不满。

2.客服反馈：客服团队是与用户直接沟通的窗口，用户遇到的问题通常会第一时间反馈给客服。因此，客服团队需要有一套机制，将用户报告的问题快速传达给 SRE 团队。

3.运营人员反馈：运营团队在日常工作中也可能发现系统的异常情况，他们对业务的了解可以帮助 SRE 团队更快定位问题。

5.2.2.3 故障诊断

5.2.2.3.1 应急协同

系统架构与逻辑的复杂性越来越高，应急协同通常是一个跨团队、跨地域的协作过程，建立高效的应急响应机制是应急协同的重要工作。

1. 协同机制
    - OnCall 值守：一二三线值守，分工明确；
    - 应急响应：建立完善的应急响应流程；
    - 环境资源保障：各环节资源投入及升级流程清晰，能快速提供需要的资源。
2. 岗位设置
3. 应急总负责人（二/三级部门主管）
    - 统筹协调应急响应工作
    - 发起研究重大应急决策和部署
    - 决定实施和终止应急预案
2. 应急指挥小组（三/四级部门主管）
    - 接受应急总负责人的领导，传达和落实应急总负责人的各项指令，汇总和上报应急信息
    - 负责应急执行小组成员的协调沟通，协调应急事件处置工作中的重大问题
    - 向业务及其他相关方同步必要的应急事件处理相关信息
3. 应急执行小组（四级部门主管、工程师）
    - 落实应急总负责人及应急指挥小组布置的各项任务
    - 组织制定应急预案，并监督执行情况
    - 掌握应急事件处理情况，及时向应急总负责人和应急指挥小组报告应急过程中的重大问题

向应急指挥小组确认事件恢复效果，关闭事件

4. 值班人员（OnCall/值班人员）
    - 响应所有事件，包括通过电话、自动监控等渠道上报的事件
    - 完整记录所有接收的事件信息，包括：用户信息、事件描述、发生时间和地点等
    - 为事件进行适当的分类、为事件分配优先级等属性
    - 使用知识库等手段对事件进行初步诊断和分析，并进行尝试解决
    - 必要时联系相关方参与事件处理
    - 如果不能解决事件，应当将事件分配给合适的二线支持并及时升级
    - 检查事件记录的处理进度，保持与用户的联系，适时通知事件处理状况
3. 协同过程

- 启动应急响应机制
- 组建应急指挥部
- 信息共享与同步
- 资源调配
- 决策与指挥
- 持续监测、评估、反馈

5.2.2.3.2 故障定界

在微服务架构下服务通常会有多个模块组成，当其中一个模块异常时往往会导致整个服务多个模块产生异常告警。而故障定界时指 SRE 确认故障是由哪个模块异常导致整体故障的过程。SRE 在进行故障定界的过程中，应注意以下事项：

1. 故障定界并非故障定位

故障定界时，SRE 应将精力放在确认故障模块上，为后续故障恢复提供操作依据，过程中主要追求更高的时效性。故障定位则主要偏向确认故障根因，为后续代码级或架构级别的调整提供依据，过程中追求的是准确性。因此，SRE 在故障定界的过程中需要时刻谨记首要任务是恢复中断的业务，而非对故障原因刨根问底，绝大多数的故障定位是可以事后做的。

- 辅助定界手段

1. 告警信息丰富

告警是 SRE 触达故障的关键渠道，也是直接有效又容易忽略的提升故障定界效率的第一步。告警信息除了异常本身的信息描述，（例如，CPU 使用率 100%）还可通过丰富告警对象信息、上下文信息，让 SRE 更快分析、定位产生告警的源头，例如，SRE 是否已经做到所有告警都能展示对应的主机信息、IP/容器信息，是否有告警对象的集群、主备信息，交易类监控指标是否能够同时在一条告警信息里，查看当时的技术成功率、延迟、流量、错误率、饱和度等信息。

2. CMDB 辅助定界

CMDB 是生产机房镜像库，SRE 需构建以配置为中心的运维一体化建设满足各个运维场景对象可以追溯到 CMDB CI。每条告警信息中携带了 CI，当有告警产生时，可以初步定界监控直接的问题对象，注意准确的定界还依赖于快速验证。

3. 链路跟踪手段

链路跟踪通过对符合 OpenTelemetry 标准的原始链路日志进行实时采集与流式计算，还原精准链路拓扑，应用于全链路实时观测、基于链路的业务指标计算、故障定界。

链路跟踪准确定界故障源，比如应用通常会因在组件异常问题、交易相互依赖、因数据库造成上层应用队列堵、多只交易缓慢的问题等通过链路跟踪定界故障。

4. 可观测故障决策树

在链路拓扑的基础上整合 CMDB 数据关联业务系统具体组件的日志、指标信息，故障定界时可以进行多层下钻挖掘故障源头，并通过关联故障排查脚本作业的方式支持在链路上做进一步的故障信息收集。结合 CMDB 拓扑、链路拓扑的组件关系、服务关系，将常规的故障排查步骤编排成故障决策树，将专家经验沉淀为系统判断规则基于可观测数据和排查任务反馈的信息进行故障定界判断

5. 快速验证设想

当 SRE 有怀疑的故障模块时，应该通过一定的手段（如：调整流量、配置参数、增加实例数量等）对设想进行验证。SRE 需要注意验证过程中应只影响模块的部分实例、流量或请求。操作实施后，SRE 应通过故障观测手段来确认设想是否符合预期，如不符合预期应考虑是否操作参数需要调整或模块猜测存在偏差。

3. 明确定界故障边界

筛选已知故障定界场景，提前确定定界组件，SRE 明确在明确微服务架构下各个组件后，常见的如云 PaaS、数据库、中间件、主机、网络、存储等，整理筛选出可能的组件故障点，依赖于事先人工分析和以往故障案例分析形成故障边界。故障边界的确认是尽可能减少不确定性，把定界点考虑在前。

4. 复杂情况定界故障

对于复杂的故障场景，可能是多种组件故障混合的。一般可以通过 CMDB 拓扑链路关系确认最下层组件，首先分析最下层组件并进行快速恢复验证。

5. 详细记录操作步骤

SRE 应记录定界验证的操作内容，作为后续故障恢复或切换的依据。因此，SRE 应通过运维工具进行操作，避免由于上机操作导致操作记录散落无法进行归档。同时，详细记录的操作步骤可以避免当验证产生非预期效果时，或多个操作叠加产生负效果时，可作为应急回滚的依据。

故障定界是SRE故障处理中的核心过程，只有准确的故障定界才能保障后续有效的故障恢复方案。而且由于过程中会涉及操作行为，SRE需要对一切的操作行为保持谨慎避免由于操作导致额外的故障发生。可以通过准确性、耗时和操作次数来衡量定界质量

5.2.2.3.3 影响评估（影响人数，范围，上报级别）

故障影响评估是多方面的，主要看对服务功能的影响、影响时长、故障发生的时段、对用户影响的范围。

(1) 如果对应用服务无任何影响，可以快速恢复的生产问题的，这些问题MTTR为0且对用户无感，事后报告直接领导。

(2) 对于应用服务有影响的，且影响到客户但范围可控的，定性为生产事件，需要及时上报到系统运维中心生产运行情况沟通群并报告处室领导（互联网行业一般到部门总监），并在事件问题群跟进事件解决进度。如果影响时间过长，需要进一步报告中心领导。

(3) 对于应用服务有影响，且影响到客户且范围过大的，定性为生产事件，需要及时上报到系统运维中心生产运行情况沟通群并报告中心领导。在互联网行业，通常上报至部门总经理。

国家网信办就《网络安全事件报告管理办法（征求意见稿）》公开征求意见，按照《网络安全事件分级指南》，属于较大、重大或特别重大网络安全事件的，应当于1小时内进行报告。 网络和系统归属中央和国家机关各部门及其管理的企事业单位的，运营者应当向本部门网信工作机构报告。属于重大、特别重大网络安全事件的，各部门网信工作机构在收到报告后应当于1小时内向国家网信部门报告。

5.2.2.4 故障恢复

在SRE中，故障恢复是一个重要的概念。当SRE收到监控告警信息并确认故障发生（排除误告警）后，应第一时间确认评估故障影响范围，同步故障到相应干系人，并根据相应的优先级对故障进行恢复，并在恢复后需对故障原因进行根因分析，做好排除隐患，尽量避免二次故障。最大程度减少系统停机时间，降低MTTR，保障业务可用性，保证业务连续性。

故障恢复通常包括架构自愈、应急预案、应急维护和恢复验证等方面内容。

5.2.2.4.1 架构自愈

故障是不可避免的，架构自愈的理念是在系统设计之初以及迭代过程中考虑系统的健壮性，使其具备完善的逃生能力，通过一系列技术手段来实现系统的高可用性，架构高可用是满足架构自愈必要条件，包括如下两个方面。

1. 部署架构高可用

在业务部署上通过跨交换机/机房/可用区/地域等粒度的容灾，如通过云的置放群组策略控制业务部署的亲和性，实现 IaaS 层部署架构的高可用；当 IaaS 层发生故障时，无需人工干预，可快速自动切换调度。常见的部署架构有两地三中心，即系统划分为三个数据中心，两个位于同城，一个位于异地，同城的两个数据中心分别承担主备角色，异地数据中心则作为备份，保证高可用性和容错能力。

2. 业务技术架构高可用

业务应通过不断迭代，优化业务架构，排除单点模块，实现业务本身高可用，如典型的负载均衡、多活、热备自动切换等实现故障转移。或者业务架构可以充分利用云的弹性伸缩 AS 以及云原生能力（Workload、HPA/GPA/VPA），使用扩容、限流、降级等自动解决系统过载、流量增加等场景的故障。

5.2.2.4.2 应急预案（已知故障场景）

基于已知的常见故障场景，SRE 复盘历史上出现的生产环境故障，提取故障特征，梳理业务相关依赖（关联哪些设备、哪些组件、相关依赖）沉淀成专家经验的故障处理模板，落地到连续性管理平台（运维手册）中。SRE 通过对手动故障处理操作流程进行总结提炼，开发故障处理工具脚本，将其编排落地到自动化故障处理系统中。通过不断的迭代，丰富故障识别场景，实现当故障发生时，通过监控、链路跟踪等可观测指标匹配识别故障，执行自动化处理流程，无需人工干预进行故障恢复。

由于系统会经常地发生变化，所以相应的应急预案会可能出现老化失效的问题， 因此，需要进行持续的版本迭代并进行演练。

5.2.2.4.3 应急维护（未知故障场景，人工干预）

针对无法自动化处理的故障，需要 SRE 快速人工介入处理，及时卷入周边关联团队协助处理，并按照故障响应处理的流程规范对故障进行升级，同时定期对故障处理进展信息进行同步。在故障恢复后对问题根因进行分析复盘，并评估是否落地到自动化故障处理中。

5.2.2.4.4 缓解及恢复的措施

缓解及恢复措施是在生产环境中，当系统发生故障或重大问题时，为降低用户影响、恢复业务功能而采取的紧急行动与恢复策略。其目标是通过快速"止血"减少问题影响，并在此基础上迅速实现服务的全面恢复。

核心原则：

- 快速止血：优先采取缓解措施，减少用户感知的影响，而不必立即解决根本问题。
- 服务恢复：在缓解措施生效后，尽快通过稳定的方法恢复系统正常运行。

典型缓解与恢复方法：

- 应用回滚：将服务恢复到已知的稳定版本。
- 数据回滚：还原数据库或配置到健康状态。
- 重启：重启相应的业务服务器及服务
- 熔断：通过启用熔断机制，在调用失败率过高时自动停止调用故障服务，避免连锁反应导致更大范围的系统问题。
- 限流：动态限制用户请求速率，防止高并发或恶意请求导致系统过载，确保关键功能的正常运行。
- 降级：关闭非必要功能，集中资源保障核心业务运行。
- 扩容：增加系统资源以应对高负载或资源瓶颈。
- 切流：将流量引导至备用系统或健康实例。
- 屏蔽：暂时屏蔽问题模块或功能，降低故障扩散风险。
- 隔离：将故障模块隔离，保护其他服务的稳定性。

以上仅仅是部分缓解与恢复的手段，实际应用中需要根据具体场景灵活选择或组合使用，以最大化降低影响并恢复服务。

宗旨：

缓解及恢复措施的核心是通过临时性手段迅速减少故障影响，并争取更多时间来分析问题根因与设计长期修复方案。它强调行动的高效性与精准性，以最大化减少对用户和业务的干扰。

5.2.2.4.5 恢复验证

故障解决后，SRE 团队应对业务进行健康检查，验证故障恢复情况；持续观察一段时间，查看 SLI 各项指标是否正常，包括观察性能指标（延迟、吞吐量、处理速率、时效性），可用性指标，质量指标，运营指标等，确认服务已恢复正常。

5.2.2.5 故障复盘

复盘源于围棋术语，也称「复局」，指对局完毕后，复演该盘棋的记录，以检查对局中招法的优劣与得失关键。这样可以有效地加深对这盘对弈的印象，也可以找出双方攻守的漏洞，是提高自己水平的好方法。故障复盘是指生产故障发生后，对故障处理过程、影响和原因分析、制定有效改进措施的过程。故障复盘是一次宝贵的从历史故障中汲取经验教训的机会，通过及时复盘，举一反三，确保下次类似的问题不会出现甚至扩大化，故障复盘对业务稳定性建设非常重要，技术团队需要不断加强和培育事后总结的文化，理清所有的根源性问题，最关键的是落地有效措施避免未来重复发生或降低故障重现的概率及爆炸半径。

故障解决后，需要及时针对引发本次故障的根因及处理过程做一次详尽的复盘，识别到其中的风险，并制定及跟踪后续对这些纰漏的修复及警示。故障复盘作为稳定性建设一个非常重要的环节，需要管理层自上而下的认同和支持，不能把事后总结当成例行公事，需要让工程师们真正认同事后总结是把系统服务变得更可靠的一次机会。任何故障发生后，都会产生一篇故障复盘的文档记录到故障管理系统中。针对复杂或者严重的案例，需要通过正式的会议形式组织故障复盘，提升工程师们重视度和面对面讨论引发更多深入思考。正式复盘会议之前通过标准化的故障总结模板收集复盘需要的信息，由故障相关的团队一起协作参与完成复盘材料的准备，会上主要还原事实，聚焦改进，针对故障关键信息达成共识，复盘过程中要避免相互指责，建立"对事不对人"的事后总结文化。

下面主要针对正式故障复盘会议的组织、流程和平台工具进行详细介绍说明。

5.2.2.5.1 复盘组织

故障复盘已成为一项标准的稳定性服务，需要关注服务的时效性，一般建议重大事故的复盘在根因明确后的第 1 个工作日内完成复盘，非重大事故建议 7 天内完成复盘总结。正式故障复盘会议由质量管理团队组织和主持，会前发起正式的会议邀约，建议邀请故障相关方的技术研发、测试、SRE 等角色参加，同时建议按需邀请：网络、DBA、UED、业务运营、客服、值班长等角色。如关键角色因时间冲突不能到现场，建议更换时间，避免

因关键角色的缺席导致信息的不对称，有重开的风险。重大故障的复盘务必邀请故障相关方有决策权的管理角色参加。

1. 复盘流程

下面我们将从盘前准备、盘中主持、经验传承三个方面来介绍故障复盘的流程，尤其是盘前准备对整体复盘质量至关重要。

2. 盘前准备

复盘组织人在故障复盘会正式召开前需按照标准的复盘模板准备好故障的复盘材料，增加正式故障复盘会议条理性，复盘模板包含但不局限于以下内容，复盘材料可由故障涉及相关团队一起按照统一复盘框架协同完成。

过程回溯：回溯问题是一个抽丝拨茧的过程，一般包含几个关键时间节点的信息回溯：故障注入、故障发生、故障发现、故障通告、故障响应、初因定位、故障止血、影响消除、根因定位等。

原因分析：回溯完成过程之后，可以为分别从故障根因、触发原因、造成影响放大原因三个层面进行剖析。通过 5-Why 方法抽丝剥茧层层扒开迷雾，找到问题的根源，从根源上制定有效改进措施，避免问题再次发生。

影响分析：故障影响分析帮助我们从更全面视角了解故障带来的损失，需要更关注每一起故障对客户和业务造成的影响，如是否产生了社会舆情及客诉，是否产生了业务的资损等；其次是技术视角，系统的可用性是否受损，关键技术指标下跌程度。

经验总结：为了避免核心关键优化措施遗漏，可以按照故障发生的整个生命周期进行讨论分析，并落地执行：一方面从系统质量层面关注需求设计和评估是否充分，代码是否经过 Code Review 并充分测试，变更执行过程是否有效灰度和观测，是否有回滚预案；另一方面关注故障处置方案是否合理，如故障是否监控发现，是否及时应急，是否有更快的恢复方案等。为避免类似的故障重复发生，需要给出短期治标方案，又需要长期治本方案，以及沉淀经验和教训。

3. 盘中主持

故障复盘主持人需要对故障信息有全局性的理解，在会议中能有效识别并记录故障关键信息和改进点。针对大家的讨论进行全程的控场，能收能放，当大家针对某一个问题的讨论过于发散，要及时控场，引导发言的同学聚焦故障相关核心问题讨论；对于重大责任争议故障，建议复盘主持人邀请高级管理层参加，建立"聚焦改进、对事不对人"的复盘氛围，避免相互指责，鼓励各自关注自己团队的问题和改进。

4. 经验传承

针对一些经典的故障分析，可以通过内部的运营平台或者技术论坛，进行更大范围的公布和分享，目的是希望大家都能够"以史为鉴"，对共性问题进行反思，形成故障深入复盘的文化氛围，更多的部门和团队可以从经典故障案例中汲取教训，从而获益，共同推动线上业务的稳定性建设。

5. 平台工具

我们首先需建立故障总结的标准模板，通过流程管理工具将故障模板中的关键节点信息，如故障的描述、过程节点、影响及原因分析等结构化沉淀到系统中，我们一般该系统为故障管理系统，基于故障管理系统沉淀历年的故障复盘记录，基于故障管理系统沉淀的历史故障，数据结构化和线上化，可以进行更多维度的故障数据分析，将分析结果分享和赋能给工程师们，进一步提升业务的稳定性。同时基于故障管理系统我们可以做更多场景的拓展，如通过自动化数据收集，一方面提升故障记录效率，也可以将故障复盘记录和变更信息、定位和恢复信息，以及业务应用等信息进行关联，提升变更质量，提前做好变更观测，也能推进定位及恢复系统建设，提升恢复时效。故障管理系统虽然只是一个流程管理工具，却为我们很多系统稳定性工具建设提供了丰富实战数据来源。

5.2.2.5.2 根因分析

1. 根因分析的目的

根因分析指为了确定适合的解决方案而探查问题根本原因的方法，用以逐步找出问题的根本原因并加以解决， 而不是仅仅关注问题的表征。与解决临时问题和被动应对相比，以系统化方式确定和分析问题原因，找出问题解决办法，并制定问题预防措施等，使得对解决根本性问题更为行之有效。

2. 根因分析的步骤

因为引起问题的原因通常有很多，物理条件、人为因素、系统行为、或者流程因素等等，通过科学分析，有可能发现不止一个根源性原因。首先我们需要找出：定义问题（发生了什么）、找出原因（为什么发生）、改进措施（什么办法能够阻止问题再次发生）。

1. 如何定义问题

在根本原因分析之前，必须确定并定义发生的问题，可以通过提出以下几个问题来探寻根因：

- 谁首先发现了这个问题？
- 到底发生了什么，尽可能详细地描述？

- 在这个过程中的什么时候发现了这个问题?
- 问题是什么时候发现的?
- 到目前为止发生了多少次?
- 有没有发生的模式?
- 问题是如何被检测到的?

通过收集这些信息，找出具体的细节，以便对问题有更全面的了解。此时可能需要采取短期遏制措施或纠正措施。将需要审查所有收集的信息，以便根据事实和数据确定可能存在的问题，一旦定义清楚了发生的问题，就开始了根因分析过程。

2. 如何找出根因

进行根本原因分析时，一种较为常见的技巧是"5 问法"，得到每个问题（"为什么"）的答案后，继续提出更深入的问题（"好，那为什么"）。通常情况下，经过大约五次提问即可找到最根本的原因，但具体次数取决于实际情况：有时 2-3 次便足够，有时可能需要多达 50 次。

举个例子：淘宝商品详情页面无法展示库存数量。

首先立刻可以反应出的一个问题：为什么库存数无法展示？这是第一个"为什么"。

第一个答案：因为库存调用链路异常。

第二个"为什么"：为什么库存调用链路异常?

第二个答案：因为 A 服务调用 B 服务未得到响应。

第三个"为什么"：为什么 A 服务调 B 服务会出现无法响应?

第三个答案：因为 A 服务的机器异常导致服务不可用。

第四个"为什么"：为什么 A 服务的机器会异常?

第四个答案：因为 80%左右机器出现了 FullGC。

第五个"为什么"：为什么机器会出现 FullGC?

第五个答案：因为今天的代码发布引入了一条全表扫描，慢 SQL 导致请求堆积，导致机器出现 FullGC。

3. 制定改进措施

我们找出问题根因的核心目的是针对根因制定改进措施，避免同样的问题再发生。

5.2.2.5.3 制定改进

1. 举一反三

故障改进是故障复盘中的关键一环，故障改进不仅仅是处理当下问题，一定要去看本质的问题，目的是降低故障再次发生的概率，但降低概率不等于不会发生，当故障真的来临的时候，也需要有足够的能力和定力来应对。

从问题入手，是治标；从根源入手，不仅可以解决眼前的问题，还可以解决更多隐性的未知问题，是治本。故障改进工作就是一次对系统和业务问题的治本思考。

2. 如何做好故障改进

首先重点要关注"谁？什么时间？完成什么任务？交付什么结果？"

改进项必须有负责人，一个任务可能多人配合，但必须指定一个主要负责人，负责人对任务的过程、结果负责。

改进项必须明确完成时间，需要明确给出改进项在某个具体时间节点的完成时间。

明确改进的输出，任务的目标、交付结果必须明确，作为任务完成的验证标准，且可衡量。如设计方案的改进措施，要求输出方案文档，可以进一步明确方案文档必备的内容等。

同时一个好的改进措施需要符合 SMART 原则：

Specific：即改进项。我们需要改进的事项或指标。

Measurable：即验收标准。改进完成后通过评审、演练、主管确认等方式验收。

Attainable：即改进项是否可落地。需要避免出现一些喊口号、无法落地的改进。

Relevant：即和故障本身有相关性。避免已在计划内的或非故障识别的改进项纳入充数。

Time-bound：即预期解决时间。建议最长不要超过三个月，避免改进期间故障再次发生。

3. 改进措施的记录

为确保改进项得到有效跟踪和验收，需要结构化地记录在系统中，一个完整改进措施的内容包括以下内容：

标题

计划完成时间

负责人（及其团队或协助处理人）

验收方式及验收人

跟踪人

改进措施的类别

具体改进内容描述及验收标准

4. 改进措施的分类

措施的分类主要有基础架构类、产品设计类、发布类、变更执行类、编码类（前端代码、客户端代码、服务端代码）、测试类、监控类、容量/灾备类、安全类、流程规范类、数据库类、中间件类等。在每次故障复盘的过程中，监控类是每一起故障必考虑的改进项，技术同学会从精益求精的角度提升系统监控发现，及时有效预警，为业务的恢复争分夺秒。

5.2.2.5.4 问题跟踪

故障复盘梳理出来的故障报告需要细化事件问题跟踪项，包括事故情况、根因、责任人和后续改进事项。首先距离落地还需要持续跟踪明确处理人，预计完成时间等，避免事故跟进事项成为空谈。其次，SRE 建立定期提醒机制，确保事故跟进事项的落地。一些事后问题修复的跟进可以通过文档管理平台落地开发任务。

5.2.3 故障应急工程体系设计

5.2.3.1 面向故障应急的监控设计

在 SRE（Site Reliability Engineering）中，面向故障应急的监控设计是确保系统高可用性和业务连续性的重要组成部分。故障应急监控设计的核心目标是能够及时发现潜在问题，迅速定位故障根因，并且通过有效的告警机制和监控覆盖，保障故障响应的及时性和准确性。

以下是对面向故障应急的监控设计的详细扩展说明。

(1) 监控体系结构设计

监控体系应按层次结构设计，形成一个完整的监控金字塔模型，涵盖从底层基础设施到用户体验的各个层级。具体设计包括：

用户体验层监控：在监控体系的顶层，监控应聚焦用户体验相关的指标，如请求成功率、响应时间、系统可用性等。此层监控旨在直接感知用户的真实体验，通过 SLI（服务级别指标）和 SLO（服务级别目标）来衡量服务的整体质量。

应用层监控：中层监控覆盖应用服务的性能指标，如延迟、错误率、吞吐量和饱和度等黄金指标。这些指标能够反映应用的健康状态，并帮助识别潜在的性能瓶颈和资源限制。

组件和中间件监控：进一步下层，监控应涵盖关键组件和中间件，如数据库、缓存、消息队列等。常见的监控指标包括数据库查询时间、缓存命中率、消息队列长度等，这些指标对于保障系统的稳定性至关重要。

基础设施层监控：最底层的监控体系应覆盖服务器、网络设备、存储设备等硬件设施的健康状况。此类监控包括 CPU 使用率、内存使用率、磁盘 I/O 和网络带宽等指标，确保基础设施的稳定运行。

(2) 指标标准化与统一框架

为了确保监控指标的可操作性和可比性，必须在组织内部推动指标的标准化：

SLI 和 SLO 的设定：SRE 团队应与开发团队合作，基于 SLI 设定明确的 SLO。这些 SLO 不仅要符合业务需求，还要考虑用户体验的敏感性，如请求成功率的目标设定为 99.9%或更高，响应时间限制在 200ms 以内。

统一监控框架：通过使用统一的监控框架（如 OpenTelemetry），简化不同系统和服务的指标收集与监控配置过程。统一框架有助于促进跨团队的协同，避免数据协议不一样而构成的成本。

指标规范化：制定组织内部的监控指标命名和标签规范，以确保不同团队和系统间的监控指标能够互通，并且便于后期的分析与调优。

(1) 多层级告警策略

为了在故障发生时快速响应，告警机制应具备分级处理和多渠道通知的能力：

告警分级策略：根据故障影响的严重程度，将监控告警分为多个级别，并定义不同级别的响应策略。例如，最高级别的告警可能需要通过电话和短信同时通知值班 SRE，而次级告警可以通过 IM 或邮件通知。

多渠道告警通知：为了防止单一通知渠道的拥堵或失效，告警应通过多个渠道同步触达相关人员。SRE 团队可以利用电话、短信、IM、邮件等多种手段，确保告警信息在第一时间得到关注和处理。

告警收敛与分类：在高峰期，系统可能会产生大量告警。此时需要通过告警收敛技术，将同类告警合并，以减少告警噪声。SRE 团队还应对告警进行分类，快速识别出影响范围最大的故障，从而优先处理。

(2) 轮值与应急响应机制

为减轻 SRE 团队成员的负担，建议实施告警轮值机制：

告警轮值：设立轮值制度，确保团队成员轮流作为当值 SRE，负责第一时间响应告警。轮值制度不仅可以减轻个体的心理压力，还能保障每个故障都有专人负责。

应急升级：当值 SRE 在规定时间内未能响应告警时，系统应自动升级通知至备份 SRE 或更高级别的管理人员，确保告警不被遗漏。

(3) 综合监控覆盖与冗余设计

为了实现故障的全面监控，SRE 团队应确保监控的全面覆盖：**关键服务与组件的全覆盖：**

所有与业务密切相关的服务、组件和第三方 API 都应在监控范围内，确保无论何种故障发生，都能在监控体系内被及时发现。

冗余监控设计：

针对特别重要的业务系统，建议设计冗余监控路径，例如使用多个独立的监控系统交叉监控同一业务，以防止单一监控系统失效导致的漏报。

通过以上设计，SRE 团队能够构建一套面向故障应急的高效监控体系，从而显著提升系统的故障发现与响应能力，确保业务的连续性和高可用性。

5.2.3.2 面向故障应急的作业平台设计

面向故障应急的作业平台设计是 SRE 体系中的核心环节，旨在通过自动化、标准化和流程化的手段，提高故障处理的效率和准确性。一个高效的故障应急作业平台应能够整合监控、告警、诊断、恢复和复盘功能，为 SRE 团队提供一站式的故障处理支持。以下是对面向故障应急的作业平台设计的详细扩展说明。

(1) 作业平台的功能模块设计

作业平台应包含多个功能模块，分别负责不同的故障应急任务：**监控与告警集成模块：**

此模块负责与监控系统（如 Prometheus、Grafana）和告警系统（如 PagerDuty、Opsgenie）进行集成，确保所有告警信息能够实时同步到作业平台中。平台应支持告警的自动分类和优先级排序，便于 SRE 团队快速识别和处理最紧急的故障。

自动化作业模块：

该模块提供自动化操作的支持，包括自动化脚本执行、批量操作和预定义的应急流程调用。自动化作业模块可以大幅减少手工操作的错误概率，提高故障处理的效率。常见的自动化操作包括服务重启、资源扩容、流量切换等。

恢复与切换模块：

当故障定位完成后，平台应支持一键恢复或切换操作，帮助 SRE 迅速执行应急预案。恢复与切换模块应与自动化作业模块高度集成，确保关键操作的安全性和可控性，同时支持回滚机制，以应对未预期的操作失败。

(2) 自动化与流程编排

自动化是提升故障处理效率的关键，作业平台设计应支持高度自动化的流程编排：

预定义应急流程：平台应支持 SRE 团队预定义各种常见故障场景的应急流程，包括诊断步骤、自动化操作和通知路径等。预定义流程应可以灵活调整，适应不同故障的特定需求。

动态流程调整：在故障处理过程中，作业平台应允许 SRE 根据实际情况动态调整应急流程，添加或跳过某些步骤，以应对复杂或未预见的故障场景。流程调整应具有日志记录功能，确保所有操作均可追溯。

多级流程审批与控制：对于高风险操作，平台应支持多级审批机制，确保在关键步骤执行前获得必要的授权。此功能有助于避免因误操作导致的故障扩大或其他次生问题。

(3) 可视化与交互设计

一个高效的作业平台应具备良好的可视化与交互设计，使 SRE 团队能够直观地了解系统状态并高效执行操作：

仪表盘与实时监控：作业平台应提供综合的仪表盘，展示关键性能指标、告警信息和故障处理进展。实时监控数据应与仪表盘集成，使 SRE 团队能够实时观察故障处理对系统状态的影响。

交互式操作界面：平台应设计为用户友好的交互界面，支持拖拽式的流程编排和一键执行操作。复杂的操作步骤应通过可视化流程图展现，帮助 SRE 团队理解操作顺序和影响。

移动端支持与跨设备操作：为了提升应急响应的灵活性，作业平台应支持移动端操作，使 SRE 团队可以在任何地点、任何设备上执行紧急操作。跨设备的无缝切换确保在紧急情况下，团队成员能够迅速接管并继续故障处理。

(4) 安全与权限管理

作业平台必须在设计中充分考虑安全性，确保故障处理过程中的操作安全、权限控制合理：

分级权限管理：平台应支持精细化的权限管理，确保只有授权人员才能执行关键操作或访问敏感数据。权限管理应根据角色进行分配，确保不同级别的 SRE 和管理人员能够在权限范围内执行相应操作。

操作日志与审计：所有在平台上执行的操作应被详细记录在操作日志中，支持事后审计和回溯。日志数据应被妥善保存，并具备查询和导出功能，以便在安全审计和复盘中使用。

安全加固与防护机制：平台应具备多重安全防护机制，包括但不限于身份验证、双因素认证、IP 白名单等，确保作业平台的安全性不受外部威胁的影响。

(5) 平台集成与扩展性

为适应不断变化的技术环境和业务需求，作业平台应具备良好的集成与扩展能力：

第三方工具集成：平台应支持与多种第三方工具和服务集成，如 CI/CD 工具、云服务平台、配置管理工具等。通过 API、Webhook 等方式实现数据和操作的无缝衔接，提升平台的功能多样性。

模块化扩展：作业平台应采用模块化设计，允许 SRE 团队根据需要添加或移除功能模块，以适应不同规模和复杂度的系统需求。这种设计使得平台能够随着业务的增长和技术的演进进行快速扩展。

微服务架构与容器化部署：平台应支持基于微服务架构和容器化的部署方式，确保平台自身的高可用性和可扩展性。在云原生环境中，作业平台应能够自动伸缩，适应不同规模的负载需求。

通过以上设计，SRE 团队能够打造一套高效、可靠的作业平台，有效支撑故障应急响应的各个环节，从而大幅提升系统的稳定性和业务的连续性。

5.2.3.3 面向故障应急的 ITSM 设计

在 SRE 体系中，面向故障应急的 ITSM（信息技术服务管理）设计是确保系统稳定性和业务连续性的关键环节。故障应急的 ITSM 设计应围绕"快速响应、有效处置、持续改进"的目标，提供故障处理流程的全生命周期管理。以下是对面向故障应急的 ITSM 设计的详细说明：

(1) 故障处理流程管理

ITSM 系统应具备完善的故障处理流程管理能力，确保故障从发现到解决的整个过程都有清晰的操作路径和责任分配。关键要素包括：**故障报告与记录：**

支持故障自动检测和人工上报，记录详细的故障信息，包括故障类型、影响范围、发生时间、相关组件等。

故障分类与优先级设定：

依据故障的影响程度和紧急性进行分类，如关键故障、严重故障、普通故障，并设定处理优先级，确保关键故障优先响应。

故障处理流程：

预定义故障处理流程，涵盖故障的初步诊断、详细分析、修复操作、验证与关闭等步骤。流程应具备灵活性，允许根据实际情况动态调整。

(2) 故障应急响应机制

为了在故障发生时迅速响应，ITSM 系统应具备高效的应急响应机制：**多层级响应机制：**

根据故障的严重程度，启动不同层级的应急响应。一般分为一线响应、二线技术支持、三线专家支持，确保复杂问题能够快速升级并得到有效处理。

故障升级与通知：

在故障未能在规定时间内解决时，系统应自动升级通知至更高级别的支持团队或管理层，并通过多渠道（电话、短信、IM 等）确保关键人员及时响应。

协同处理与资源调度：

支持跨团队的协同处理，提供实时的沟通工具和资源调度功能，确保在故障处理过程中各相关方能够有效合作。

(3) 故障知识库与决策支持

知识库是 ITSM 系统中的核心模块，记录了历史故障的处理经验和最佳实践，支持在故障处理中快速参考和应用：**知识库构建：**

整理和归类历史故障处理案例，涵盖故障原因、影响范围、解决方案、恢复时间等信息。知识库应按故障类型、影响范围、组件类型等维度进行分类。

智能推荐与自动化：

利用 AI 技术，从知识库中智能推荐最优的故障处理方案，并支持自动化执行常规操作（如服务重启、流量切换），提高故障处理效率。

持续更新与优化：

故障处理结束后，系统应自动将处理过程和结果更新至知识库，确保知识库内容的实时性和实用性。

(4) 故障复盘与改进管理

故障复盘是提升系统稳定性的重要步骤，ITSM 系统应支持对故障的全面复盘与改进管理：**复盘流程：**

在故障解决后，系统自动启动复盘流程，收集故障处理的所有相关数据，包括操作日志、故障影响评估、处理步骤等。

根因分析与改进措施：

采用 5 Whys 等分析方法，深入探究故障根因，制定并追踪改进措施，确保类似故障不再发生。

改进执行与验证：

针对复盘确定的改进措施，系统应生成具体的任务清单，分配责任人，并设定完成时限。完成后，通过验证机制确认改进效果。

(5) 故障应急演练与预案管理

为了确保应急响应的有效性，ITSM 系统应支持故障应急演练和预案管理：**应急预案管理：**

建立涵盖不同故障场景的应急预案库，定期更新并与实际故障处理经验相结合，提升预案的实效性。

演练计划与执行：

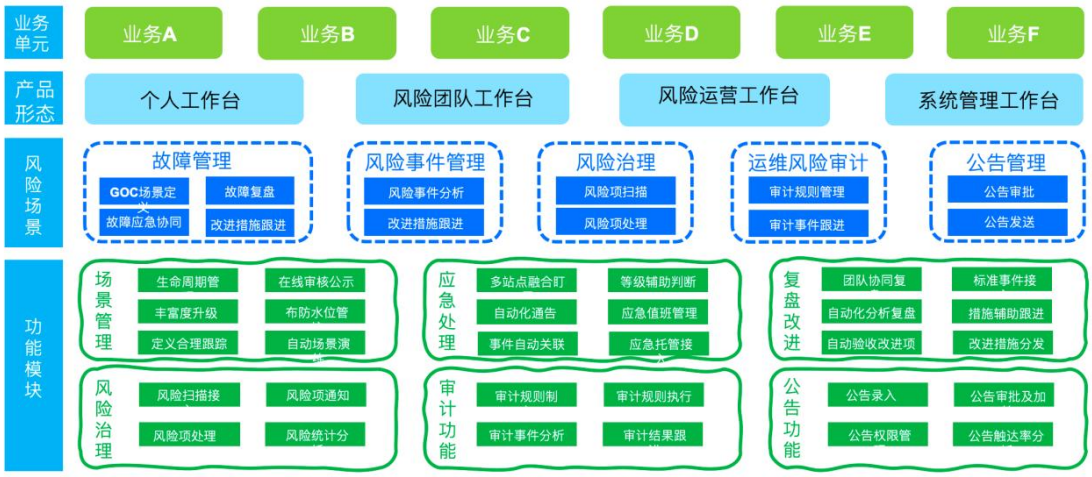
定期组织故障应急演练，通过模拟真实场景测试预案的有效性，演练结果应记录在案，并用于优化应急预案和处理流程。

演练结果分析：

对演练结果进行分析，识别潜在的问题和改进机会，确保在实际故障中能够快速而有效地应用演练中的经验。

通过上述设计，SRE 团队能够构建一套面向故障应急的高效 ITSM 系统，有效提升系统的故障响应能力和业务连续性。

5.2.3.4 面向故障应急的风险管理平台



技术风险管理平台应面向各个业务部门，涵盖了众多业务单元（BU）。他们均能在该平台上进行操作。产品形态主要服务于普通的 SRE（Site Reliability Engineer）人员以及业务接口人员。稍后我会介绍我们的整体应急响应架构。针对不同的用户角色，我们设计了多种工作台形式。例如，一些负责 GOC 的同学或类似职能的同事可能会在系统中输入特定故障定义的场景，为此我们建立了风险运营机制。

故障管理：

故障场景、故障定义等信息用平台沉淀起来，同时通过故障复盘、Action 跟进之类的的管理能力整体形成功能闭环**风险事件管理：**

风险事件是指未被定义为故障、影响较小的业务事件，可以归类为风险事件。

风险治理：

集中处理一些风险巡检系统的风险产出物，业务稳定性同学可以基于风险项信息进行风险的跟踪与治理**运维风险审计：**

对违规操作，进行审计。

公告管理：

平台提供公告功能，用于发布重要的稳定性信息。

5.3 故障应急案例

5.3.1 小米故障应急响应经验分享

SRE Elite 精选原因

小米拥有很强的硬件基因文化，因为如果硬件出现质量问题，相关的修复成本将会非常巨大。所以其质量有独特的要求，小米拥有独立 QA 团队，对运维质量进行考核及管控，构成了其独特的故障管理体系以及复盘的体系,可供有类似业务特性的组织进行参考。

（一）背景及设计原则

小米的最新战略目标是构建一个"人-车-家"全生态系统（Human x Car x Home），旨在满足全行业的需求。我们希望在这个生态系统中，能够像星辰大海一样广泛覆盖，做好每一个细节。为了体现我们小米 SRE 工作的复杂度，以下是一些关键数据：

- 我们拥有 6.99 亿台智能设备。
- 涵盖 200 多个智能硬件品类（仅小米品牌，不包括小米生态链）。
- 小米生态链大约覆盖了用户生活场景的 95%。

小米 SRE 面临的挑战

小米作为全球第三的手机制造厂商&全球领先的消费级 AIoT 平台，SRE 挑战巨大

米家760万MAU

7亿联网设备

小爱1.14亿MAU

- 目前有 7 亿联网设备（国内和国际）。
- 米家平台的月活跃用户数（MAU）为 760 万，作为互联互通的中枢。
- 小爱的月活跃用户数为 1.14 亿。小爱不仅仅是一个音箱，其后端也在适应 ChatGPT 的大潮流。
- 我们有 20 多个业务线，可以认为是集团级的组织单元。
- 全球范围内有 12 个核心机房，服务于各个地区。
- 使用了七家公有云，阿里云，金山云，腾讯云，火山云，AWS，Azure，GCP，积累了丰富的多云管理经验，但是在过程中也因为各云标准不统一在对接过程中非常痛苦

故障发现

故障诊断

故障恢复

一分钟内发现问题

五分钟内定位问题

十分钟内控制损失

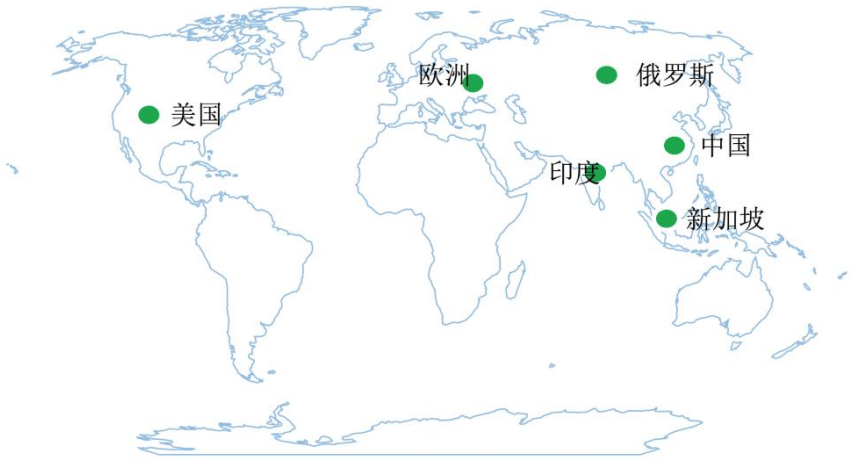
故障复盘

在故障应急方面，我们的目标是做到快速发现、快速定位和快速控制损失。具体而言，阿里的专家提出的经典目标是：1 分钟内发现故障，5 分钟内定位故障，10 分钟内控制损失。这也是我们 SRE 团队的追求目标。

关键能力点拆解

为了实现上述目标，我们需要拆解出几个关键的能力点：

- 故障发现：及时发现故障是应急处理的第一步。
- 故障诊断：在发现故障后，迅速进行故障诊断，确认故障场景。
- 故障恢复：在确认故障场景后，快速进行故障恢复。
- 闭环管理：在故障处理完成后，进行故障复盘，做好问题管理和缺陷管理。



（二）解决方案详情及案例

构建哪些能力应对故障响应

(1) 故障发现

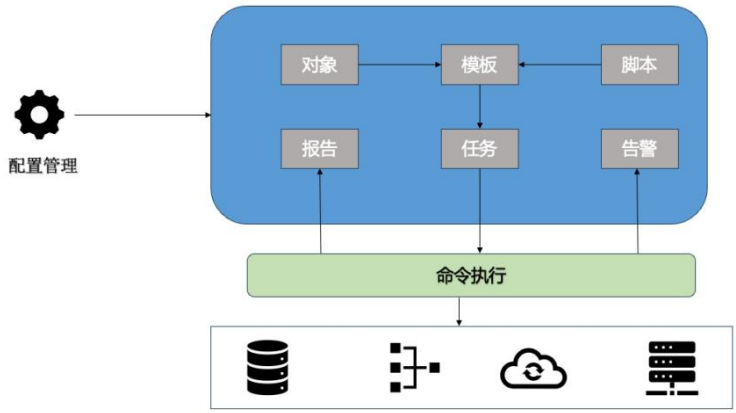
故障发现是应急响应的第一步，包含监控、巡检和故障上报三个方面：**监控：**

使用各种监控工具如 Prometheus、Grafana、DataDog 等进行基础设施、程序、日志、用户体验和网络性能的监控。

- 基础设施监控：混合云环境下的 IT 基础架构信息实时监控（性能/可用性）
- 应用程序性能监控：应用程序的功能/运行状况监控，APM，跨微服务，主机，容器和 Serverless
- 日志管理：为应用，系统和云服务提供日志管理，可创建索引，查询可视化
- 报警：支持机器学习，实现预测和阈值自定义等能力
- 用户体验监控：模拟客户监控产品的体验和可用性的真实用户监控
- 网络性能监控：云或者混合环境的网络流量进行分析和可视化处理

业界中监控的相关工具很多，但关键在于将监控系统与企业的 CMDB 资源数和业务流程系统协同，把产品用好，不停提高产品的易用性。

巡检：



以上是一个经典的巡检模型

抽象巡检配置：首先，我们需要定义和抽象出巡检的配置。

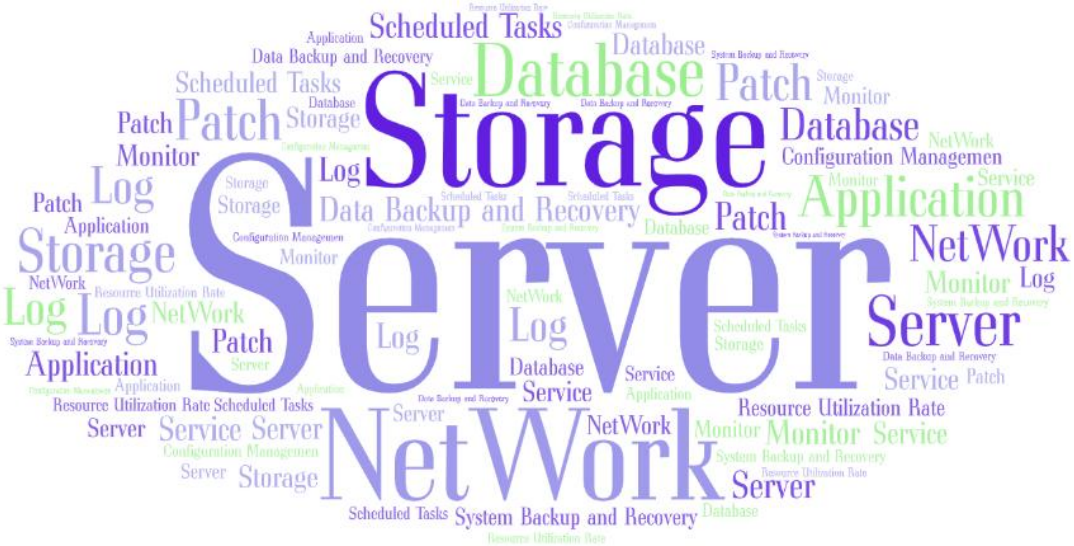
抽象巡检对象：接着，明确巡检对象的范围和具体内容。

抽象巡检模板：然后，制定巡检的模板，以便标准化巡检流程。

在实际操作中，我们可以借助命令执行系统（例如蓝鲸的作业系统）来进行巡检。这些巡检对象包括各种资源、数据库、网络、云服务以及基础设施主机等。

通过这些指标，我们可以全面监控和维护系统的健康状态，确保服务的稳定性和可靠性。

以下是我们根据我们巡检指标所构建的词图，包括数据备份，数据库， 网络， 日志，补丁等等。



故障上报：

自动化监控无法解决所有问题，用户体验上报是重要的补充手段。关注舆情监测、用户投诉和客服统计数据，与运营角色保持信息互通。

- 舆情监听
    - 媒体信息收集
    - 论坛讨论热点
- 客服反馈
    - 用户投诉上报
    - 客服统计数据
    - VIP 重点服务关注
- 运营反馈
    - 新功能上线检查
    - 日常操作反馈

(2) 故障诊断

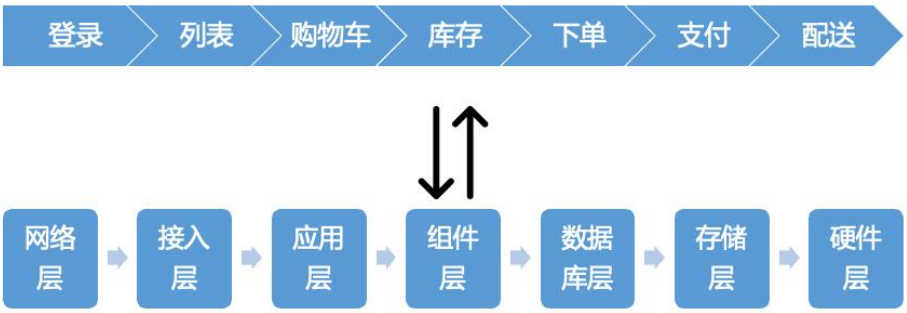
每一位资深的 SRE 同事都必须牢记这句话：如果在发生故障的第一时间无法立即恢复，那么在故障排查过程中，一旦想到能够恢复故障的方法，应优先进行故障恢复，而不是急于找到故障的根本原因。在故障诊断阶段，我们应从以下四个方向进行拆解：

- 故障范围：确定影响模块及其逻辑功能依赖和上下游服务依赖。
- 根因确定：通过关键告警和日志分析，找到故障的根本原因。
- 影响评估：评估故障的影响范围，及时向上级报告关键信息。
- 组织能力：在故障场景下需要强大的组织协调力，确保各部门协同工作。

故障范围界定

作为 SRE 团队的基本理念，每位同事脑海中都必须清晰地记住两个关键概念：横向和纵向的依赖关系。

关键词：SRE向上要理解业务功能模块，向下要精通云基础技术组件



横向依赖指的是业务层面的逻辑关系。例如，在一个典型的商城系统中，业务逻辑的依赖关系包括登录、商品列表、购物车、库存管理、下单、支付和配送。这些模块之间存在着前后端和上下游的依赖关系，理解这些关系有助于我们快速识别和定位故障。

纵向依赖涉及系统的技术层级，从底层到顶层的依赖关系，包括网络层、接入层、应用层、组件层、数据库层、存储层和硬件层。每位 SRE 同事应熟悉这些层级的依赖关系，了解每一层之间的交互和依赖。

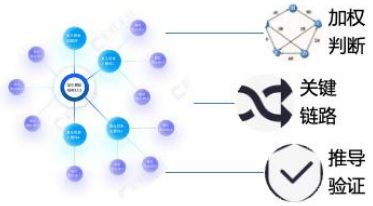
基于上述横向和纵向的依赖关系，我们能够更准确地界定故障范围。在发生故障时，清晰的横向业务逻辑和纵向技术层级的结构图能够帮助我们迅速确定故障点，评估影响范围，并采取有效的恢复措施。

根因确认

关键词：AIOPS最卷的就在这里

归纳分析〔点：聚类〕

演绎推理〔线和面：关联〕



知识图谱

在故障诊断和根因确认方面，无论是依靠人工还是借助 AI OPS，核心的方法其实可以归结为两大类：归纳分析和演绎推理。

归纳分析的核心在于将分散的信息聚合起来。例如，当我们收到大量的告警日志时，首先需要通过归纳分析将这些零散的告警信息进行汇总和筛选。这个过程的目标是过滤掉无关紧要的噪声，提取出有价值的告警信息。通过这种方式，我们可以更高效地识别出潜在的问题区域。

演绎推理则是基于归纳分析得出的点，通过逻辑推理将这些点串联起来，形成一个完整的故障路径。知识图谱是演绎推理的一个典型应用。通过知识图谱，我们能够将归纳分析得到的关键点有机地结合在一起，构建出一个系统的故障模型。这种模型不仅能够帮助我们找到故障的关键路径，还能最终确认故障的起因。

总的来说，归纳分析和演绎推理是故障诊断过程中的两个重要步骤。归纳分析帮助我们从大量的告警信息中提取出有价值的数据，而演绎推理则帮助我们将这些数据进行逻辑串联，找到故障的根本原因。通过这两种方法的有机结合，我们可以更加准确和高效地进行故障诊断和根因确认。

影响评估

关键词：及时升级！

用户请求损失	用户无法使用功能造成的损失
影响用户数量	实际影响的用户数量
影响用户时长	精确的故障时长
客服来访数量	用户通过客服的报障数
用户反馈数量	用户通过反馈渠道的投诉反馈
直接经济损失	因为故障造成的经济损失
政府监管	抽检，召回，立案
市场舆情	维权事件，媒体报道，社会话题
售后维修	维修工单/销量
集团经营影响	产能，GMV金额，核心业务时长

在故障处理过程中，影响评估是一个至关重要的环节。对于那些没有多年经验的 SRE 来说，可能对这部分的认知还不够深刻。然而，影响评估实际上决定了故障处理的优先级和策略，影响到整体的决策过程。

首先，影响评估帮助我们确定故障的严重程度。这包括评估故障对业务运营的影响、对用户体验的影响以及对公司声誉的影响。例如，一个影响支付系统的故障可能比一个影响非关键功能的故障更为严重。通过评估故障的影响范围，我们可以决定是否需要将故障上升到更高的优先级，并调用更多的资源来处理。

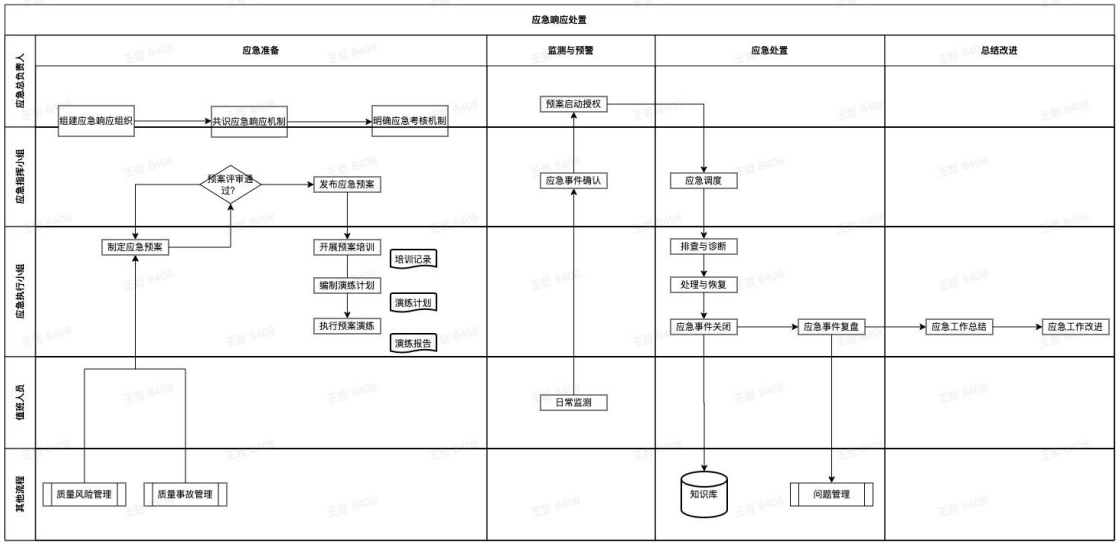
其次，影响评估也涉及决策的层级。根据故障的严重程度，我们需要决定是否需要通知高级管理层，甚至是公司高层。这一过程需要综合考虑多方面的数据，包括政府监管的要求、市场舆情的反馈、故障对经营的影响、受影响的用户数量以及影响的持续时长等。

例如，如果故障可能导致数据泄露或违反监管政策，就需要立即通知相关的法律和合规团队。如果故障影响到了大量用户，可能需要启动市场公关团队来管理用户的反馈和舆论压力。

此外，影响评估还帮助我们合理分配资源。在资源有限的情况下，了解故障的影响范围可以帮助我们优先处理最关键的问题，确保在最短的时间内恢复关键业务功能。越大的故障，处理时间可能越长，因此需要更精确的资源调配和时间管理。

应急响应组织能力

不同的层次都由不同的团队来负责运维管理，同层次不同的硬件/系统/应用都由不同的小组来负责运维管理。



在小米的应急响应流程中，监测预警和应急处置是关键环节。值班人员负责初步处理，确保问题在萌芽阶段得到控制。应急执行小组处理复杂故障，需高效沟通和定期演练。应

急指挥负责整体协调和资源调配，总负责人负责对外发布信息，确保一致性。建议优化信息流，使用统一平台，引入 AIOPS 技术，提升预警和处理效率。定期演练和评估，建立详细文档和知识库，确保信息时效性和准确性。

(3) 故障恢复

故障恢复的目标是迅速恢复系统功能，减少故障影响：**三板斧：**

重启、回滚和扩容。重启适用于单个或多个机器上的服务恢复，回滚用于撤销最近的变更，扩容用于增加系统承载能力。

关键词：每一个动作前都应该知道预期的结果，每一步真正的操作后都需要充分检查

升级版三板斧：

限流、降级和熔断。

关键词：每一个动作前都应该知道预期的结果，每一步真正的操作后都需要充分检查

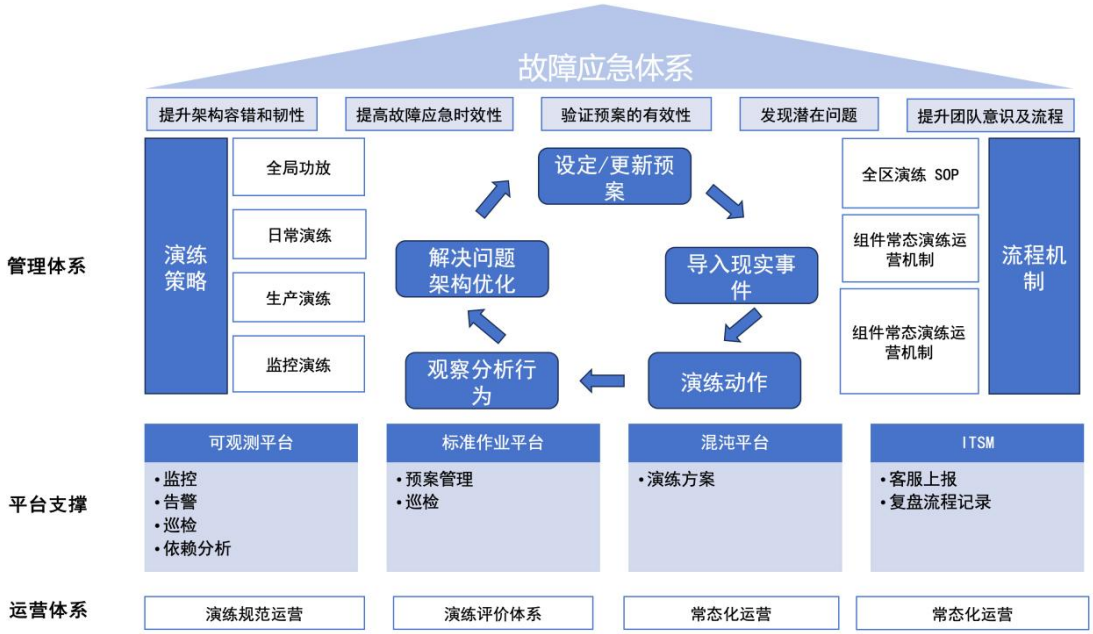
限流的目的是控制入口流量，减少超出后端承载能力的请求，并丢弃新请求。限流可以在不同层次上实现，例如在四层和七层网络协议上，或者在引擎层和数据库层上。通过限流，我们可以有效地防止系统过载，确保系统稳定运行。

降级是指有意识地降低系统部分功能和服务质量，以确保核心功能和关键服务的持续运行。例如，在大型促销活动（如 618 和双十一）期间，京东和淘宝通常会对非核心链路进行降级，以确保核心链路有足够的容量承载流量。实现降级通常通过配置管理来完成，需要业务 SRE 和研发团队共同配合。

熔断是一种较为危险的操作，只有在明确了解关键业务核心链路的情况下才应实施。熔断的目的是在依赖的第三方服务出现异常时，主动认为自身服务也异常，或者返回固定值，从而避免因响应时间过长导致的级联效应。熔断技术可以有效防止系统因单点故障而导致的全面崩溃。

这三项技术是我们在传统"三板斧"之外，与研发团队协作提升系统可靠性的关键措施。通过限流、降级和熔断，我们可以更好地保障系统在高负载和异常情况下的稳定运行。

故障恢复要以练养战



在此次会议中，我们讨论了故障管理和恢复的核心，即制定和执行预案以确保系统稳定性。预案有保鲜期，需定期更新和验证其有效性，通过故障演练实现。演练大盘涵盖演练策略、流程机制、管理体系、工具体系和运营体系。具体包括日常演练、生产演练、监控演练、全局演练 SOP、组件常态演练、重点业务演练、监控报警、故障恢复和复盘改进。通过这些措施，提升系统在故障情况下的应对和恢复能力，确保业务的连续性和稳定运行。

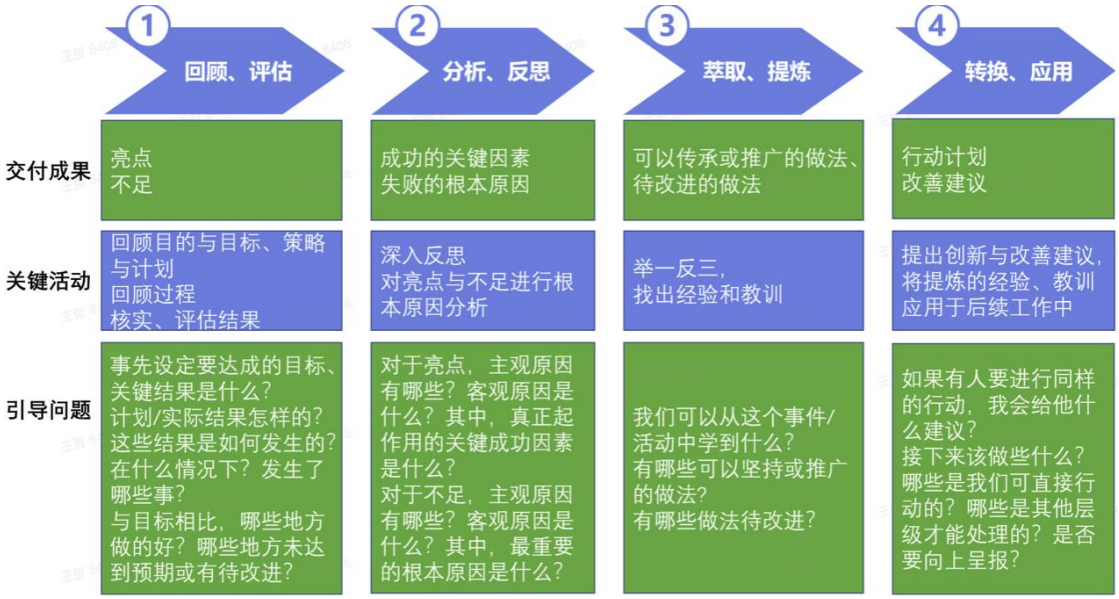
声明：

上图为作者从网上引用材料，如有版权问题，请联系 SRE 精英联盟。

(4) 故障复盘

故障复盘是防止同类问题再次发生的关键：

- 还原事实：记录故障处理过程中所有核心动作和依据的流程机制标准。
- 问题反思：反思故障中存在的问题和解决措施。
- 原因分析：分析主观和客观原因，强调管理动因。
- 重来一遍：思考如果重来一次，哪些环节可以加速或避免。



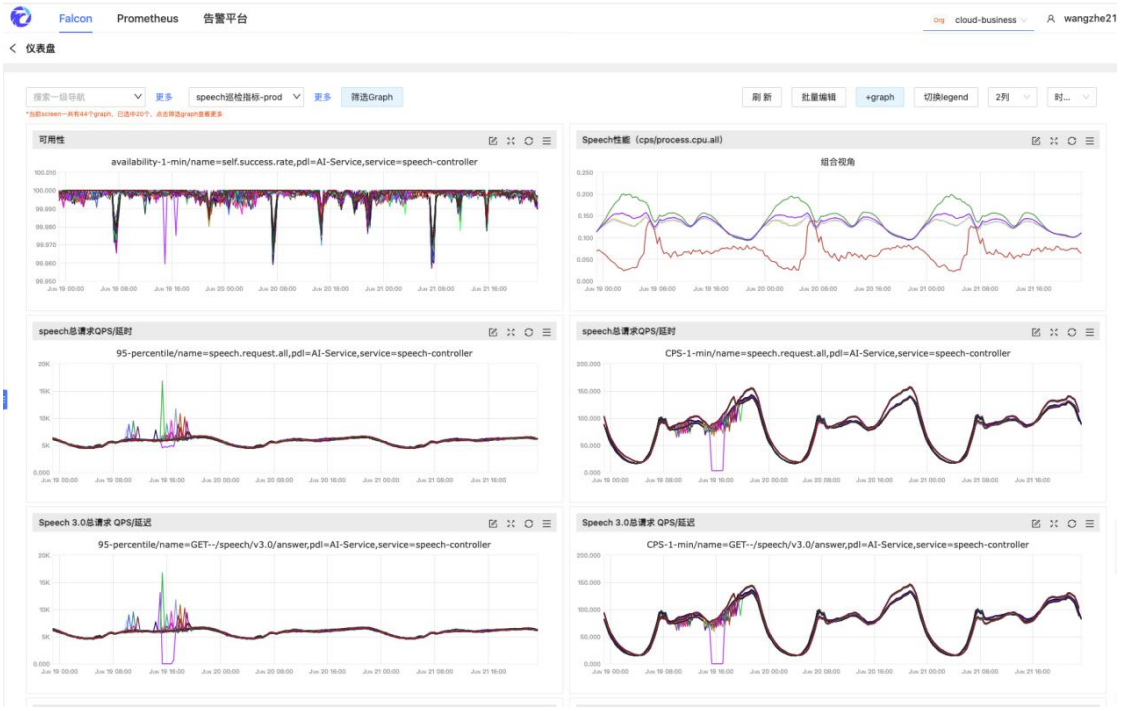
故障复盘是确保系统稳定性的重要环节。许多公司在故障复盘方面做得不足，但在小米，故障复盘被视为核心工作，尤其是在智能硬件制造企业中，质量问题的召回成本和舆情影响巨大。小米将故障复盘作为质量体系的重要组成部分，并在互联网及基建组织中推广。

在复盘过程中，强调每个步骤的执行和记录，确保每一个动作都有明确的结果和影响。通过格式化复盘问题反思、原因分析以及假设重来一遍，确保问题不再重现。使用关键引导问题（如 5 个为什么）深入骨髓地进行复盘，确保每个故障都能得到全面分析和改进。

通过以上措施和体系建设，我们能有效提升系统在故障情况下的应对和恢复能力，确保业务的连续性和稳定运行。

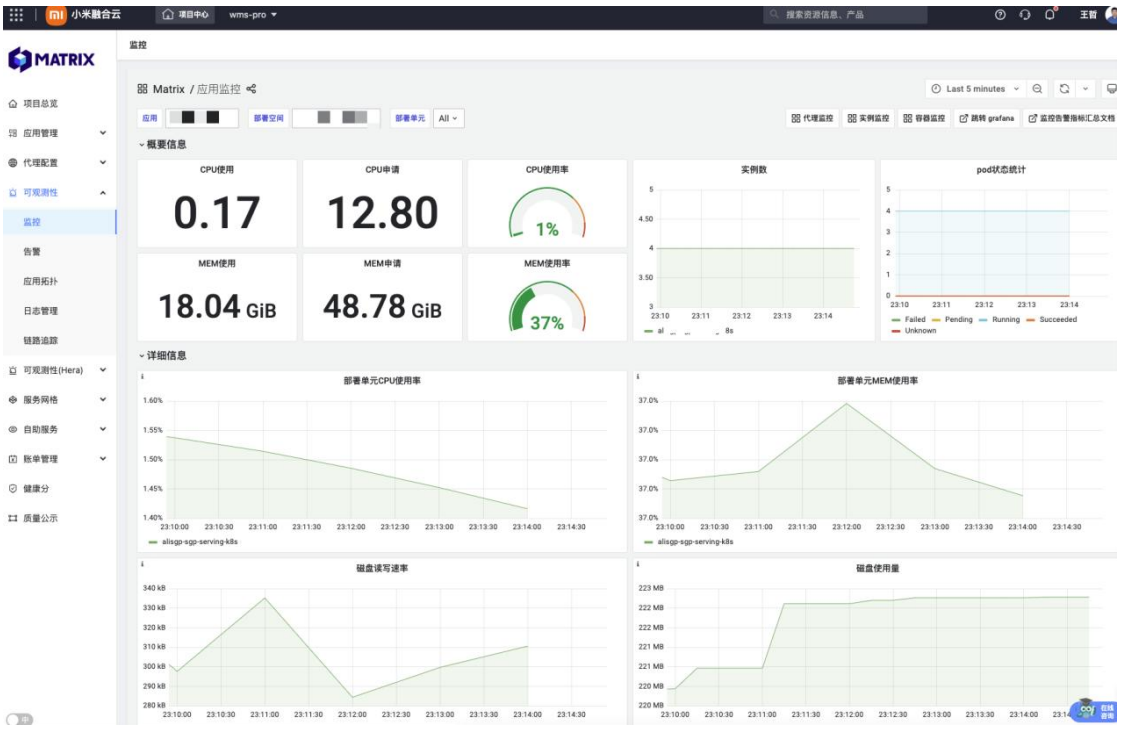
(5) 工具和平台

监控



众所周知，著名的开源软件就是在小米进行孵化的，这就是 Open-Falcon 进化到 2024 年的样子，目前在小米监控体系中主要定位用于基础监控。

另外我们的系统中还整合了 Prometheus，利用了 Prometheus 的生态支持众多监控指标，进行容器监控，业务监控等。

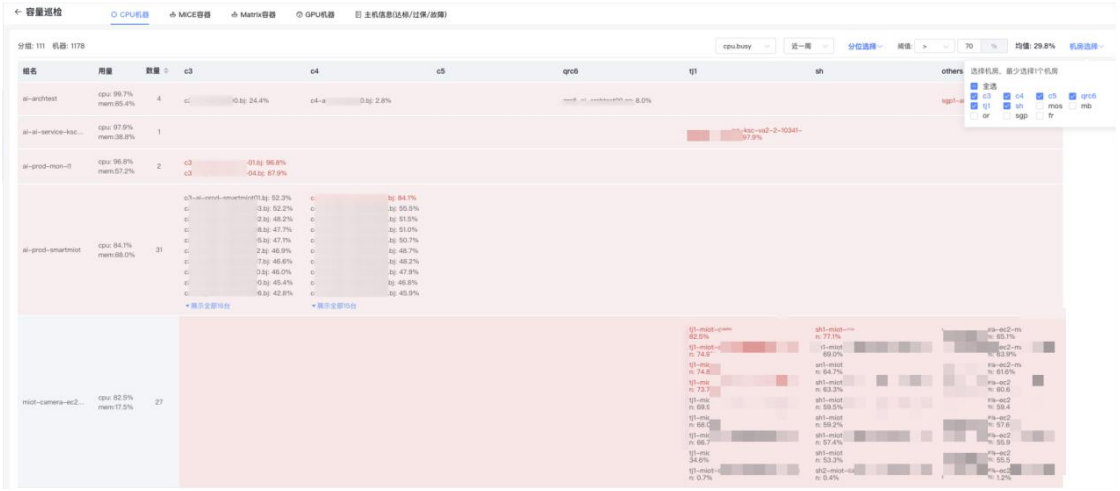


我们在小米内部实现了一个一站式的平台工程，包括容器管理，应用管理，服务治理等能力，并将部分可观测能力进嵌入。

告警

在告警方面，我们使用飞书作为基础平台，并基于其能力开发了自有的告警功能。我们的告警系统支持图表查看、关联分析和告警路径编辑。当告警触发后，工程师可以认领告警，其他人无需重复处理。此外，告警还可以被屏蔽，以避免重复通知。

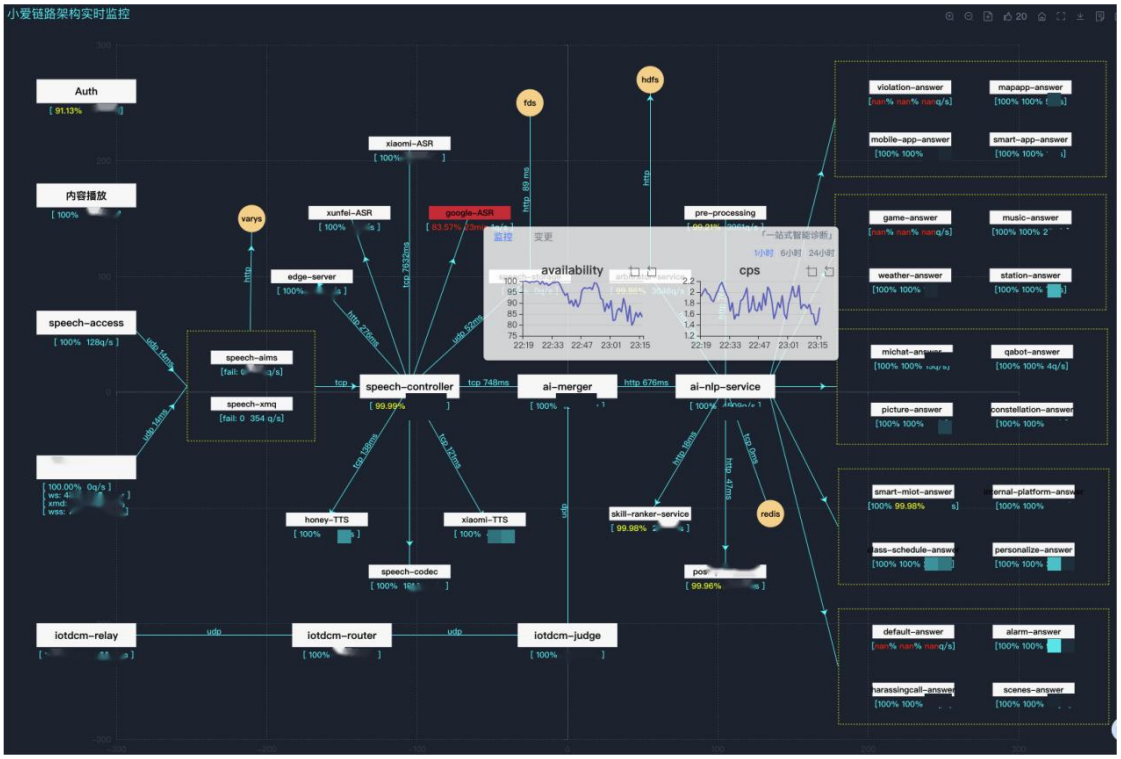
巡检



巡检是我们日常工作的重要组成部分。目前，我们内部有多个巡检平台。昨晚我截屏时发现其中一个平台无法使用。这种情况在 SRE 工作中时有发生。虽然我不打算详细展开，但可以肯定的是，做好巡检是避免故障的最有效手段。

建议大家充分利用监控工具，并合理编排各种巡检任务。只要管理好巡检和变更，你们公司的系统就能保持稳定，减少故障的发生。

链路架构



目前的链路监控系统在设计和实现上存在一些理想化的成分。在 SRE 精英联盟的讨论中，部分同事建议使用 Skywalking 进行抓包，并自动生成业务逻辑图。然而，除非公司拥

有非常强大的语言框架并统一应用，否则只通过配置管理实现抓包并生成业务架构图会较为困难的。

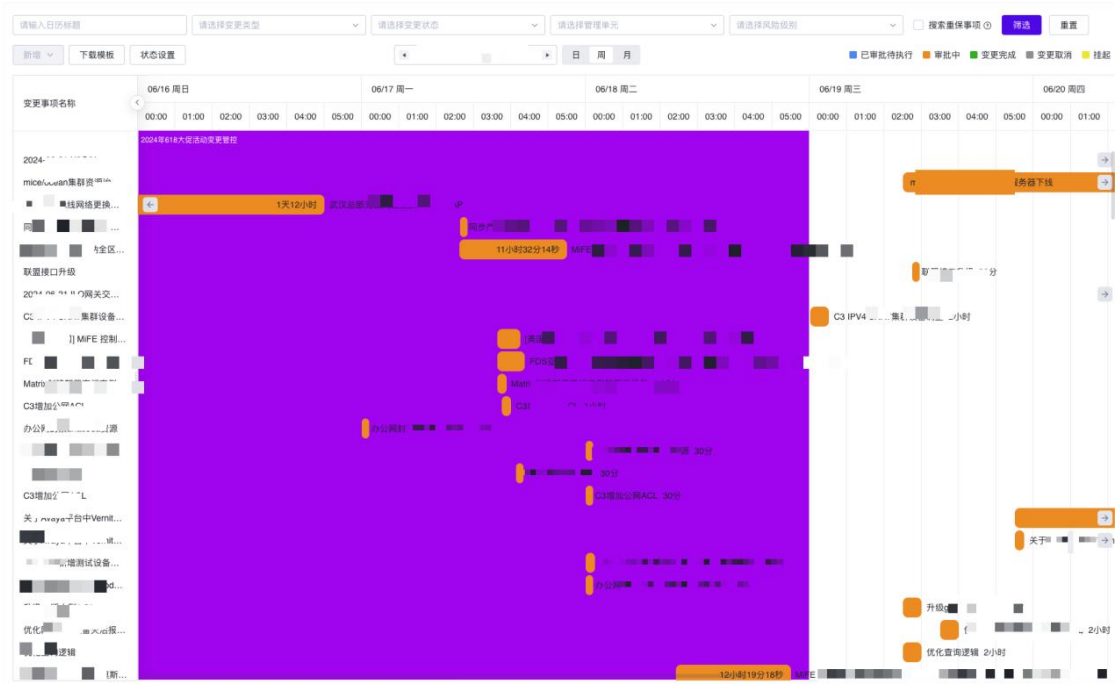
对于那些允许使用多种框架和语言的开放型公司而言，包括小米，从研发侧生成架构链路并不现实。更为可行的做法是由 SRE 团队基于业务逻辑架构绘制核心链路图，并将其与监控系统进行对接，提供支持和全局监控能力。这是目前大多数公司在链路监控方面较为理想的解决方案。

接下来，我们需要解决的问题是如何在功能发布后，确保链路监控与现有系统保持一致，形成一个闭环。实现这一点后，链路监控系统将能够更好地满足公司日常监控和异常处理的需求。

磁盘自愈

以上截图展示的是是小米一个磁盘自愈的小工具。

变更日历



我们的变更管理体系非常严格，质量组织对变更的要求极其严苛。

例如，周一和周五不允许进行某些级别的变更，特定时间点也禁止变更。此外，我们还对双十一、618、4 月米粉节等重要节点进行严格管控。

从图中可以看到，我们在特定时间段实施了变更管控，所有变更需要总监级以上审批。我们还根据时间维度记录了所有变更，便于在异常发生时快速查找根因。SRE 团队通常会查看这些时间节点，确认公司在特定时段内进行了哪些变更，以决定是否需要回滚并进一步查找问题根源。

故障上报

我们建立了一个故障上报系统，要求在故障发生并初步判定影响后，立即通过该系统上报故障。系统具备一键生成故障复盘文档的功能，按照标准化的复盘格式记录。

在故障处理完成后，例如云平台要求在故障处理后一天内完成内部复盘。接着，我们会与业务方进行第二轮复盘。根据故障的严重程度，可能还需要质量委员会进行进一步复盘。最后，完整的故障报告将提交给集团相关的质量组织。

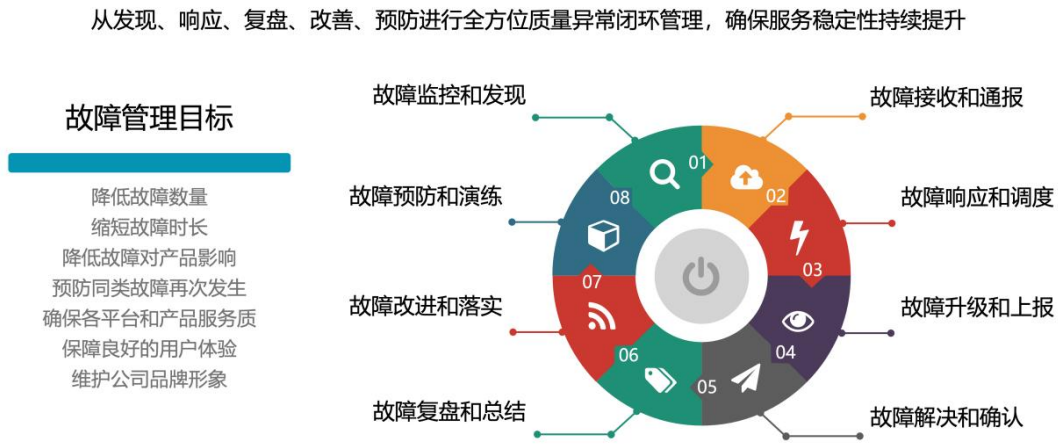
故障处理

为提高故障处理效率，我们统一了故障上报入口，并建立了大部及公司级故障应急响应中心。小米拥有数万名工程师，我们设立了一个成员超过 5000 人的故障服务通报群。当遇到无法判断的故障情况时，可以在群里发布信息，确保所有相关人员能够及时看到。

质量管理 在小米，质量管理是独立于基础设施的，有独立的组织作为监管方，监督技术、运营和产品。

质量管理团队有自己的目标和执行计划，尽管他们的工作可能对日常工作效率和人员舒适度带来挑战，但他们遵循行业标准，如信通院的标准，确保安全、隐私、合规和质量要求。

我们首先满足质量管理团队的诉求，再在这个框架下寻找妥协和执行的空间。这种独立的质量管理视角，尽管带来挑战，却是确保系统安全、合规和高质量运行的关键。



（三）总结及展望

故障应急是对 SRE 体系综合能力的考验，涵盖信息采集、快速反应和决策、稳定准确的处置、资源整合能力及日常演练形成的流程体系。

关键能力

信息采集：及时、准确地获取系统状态和故障信息。

快速反应和决策：迅速分析并做出决策，确保问题及时解决。

稳定处置措施：执行稳定、准确的应急措施，确保系统尽快恢复。

资源整合：整合内外部资源，迅速调动所需资源处理故障。

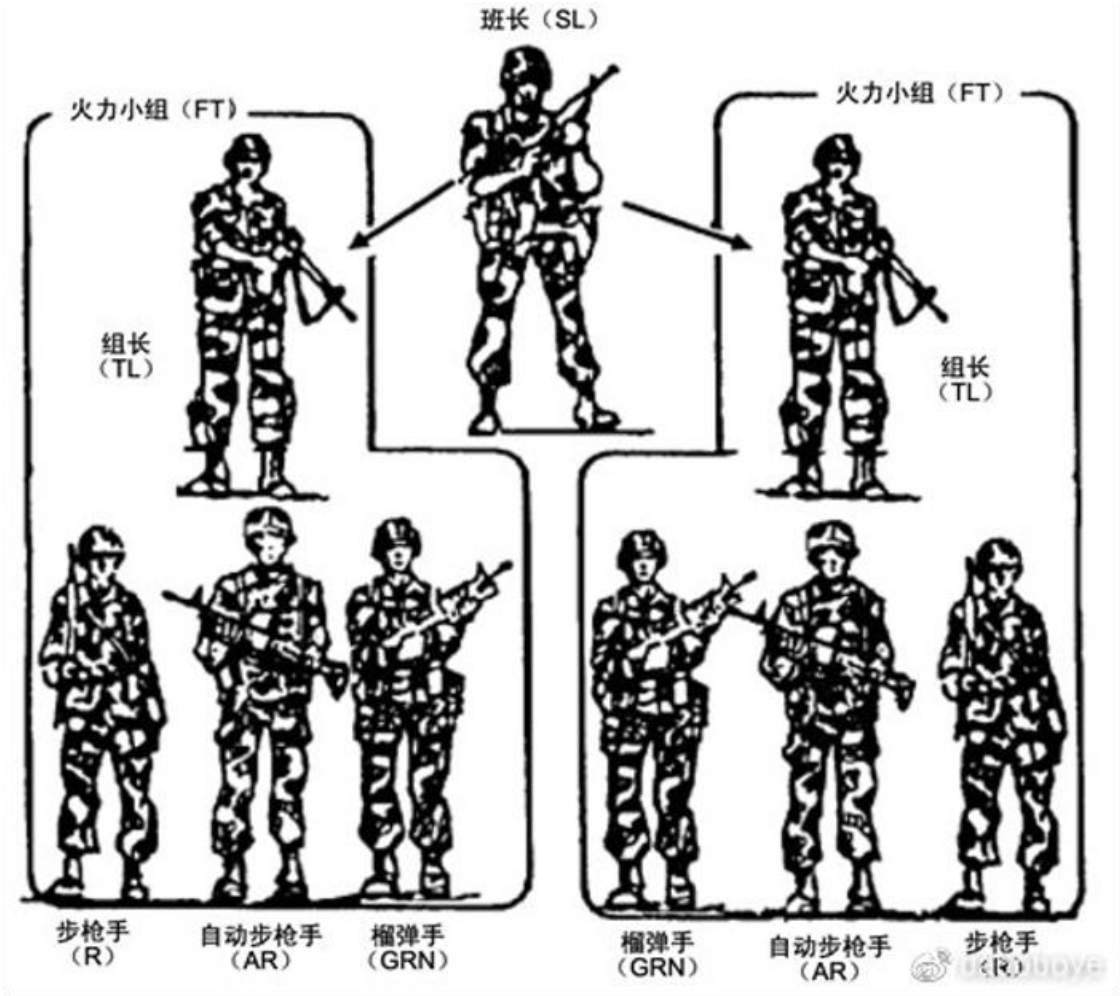
日常演练：通过日常演练，形成规范的流程体系，提高团队应急效率。

团队的重要性

经验丰富、能力强大的专家团队是故障应急的关键。每位成员在故障应急中扮演重要角色，确保系统稳定运行。

2004年在伊拉克的一个美国海军陆战队步兵班，里面有3挺M249。照片中有12人，第13人是拿相机的

类比 SRE 团队之于海军陆战队的步兵班：



分工明确：团队中有专门负责信息采集和执行操作的成员，形成三角形决策团队。

资源整合：如步兵班呼叫精准打击，SRE 团队迅速调用资源解决故障。

日常训练：通过日常演练提升应急能力，每位成员都是关键。

通过这些措施和体系建设，提升系统在故障情况下的应对和恢复能力，确保业务连续性和稳定运行。

5.3.2 中国联通数字化监控平台稳定性保障实践

SRE Elite 精选原因

中国联通作为国家重点央企，长期以来以其庞大的业务体系和稳健的运营著称，面对数字化转型的浪潮，中国联通积极推动核心业务系统向云原生架构大规模演进，面临着技术革新的复杂挑战，还需确保转型过程中的系统稳定性。此案例探索并构建了一套符合稳态企业的稳定性保障方案、策略及平台，并深度融合可落地的智能化 AIOPS 能力，提升运维效率与故障处置速度，为其他稳态企业提供了可供参考的路径。

（一）背景及设计原则

中国联通是国内知名的四大运营商之一（原三大运营商现加入广电成为四大运营商），中国联合网络通信有限公司软件研究院（简称"中国联通软件研究院"）是中国联通的子公司，成立于2015年，是中国联通集团总部提升自主研发能力的重要战略规划，主要承担中国联通集团的内部业务支撑系统、管理支撑系统、大数据以及前瞻性技术研究与应用，负责B域、M域、D域系统的建设和运营，

这里稍微解释一下，各个域名是什么意思。

B域（业务域）= business support system，

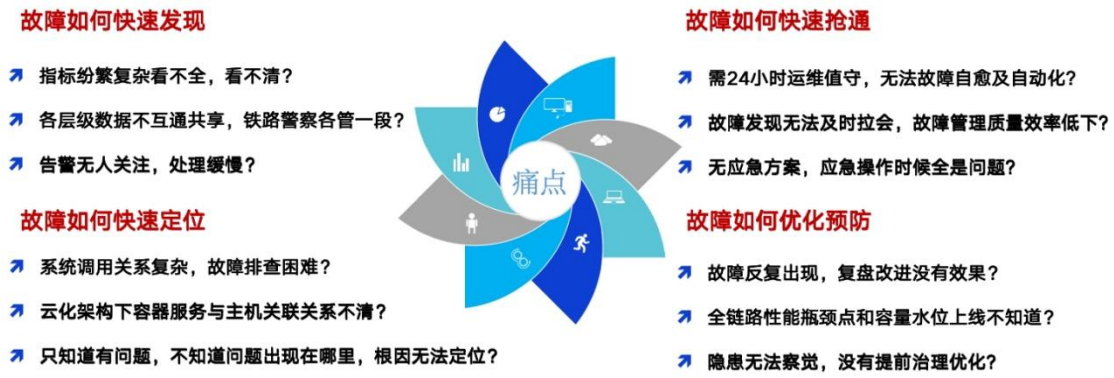
M域（管理域）= management support system。

D域（数据域）= Data support system

其中，B域系统主要包括营业厅办理业务的系统。例如，用户去营业厅办理开卡业务，使用的系统就是我们的业务系统。此外，用户每月交话费和出账单的系统也是由我们公司开发和运营的。

本案例提供者主要负责数字化生产运营保障体系的建设，以及天眼数字化监控平台的架构设计和演进。中国联通作为四大运营商之一，拥有4.3亿用户，每天有千万级的调用量，而且联通的系统是一个集约化的系统，31个省份都在同一个系统上运行，作为国计民生的基础设施，对系统的稳定性和故障处理提出了极高的要求。

运维人员的问题痛点

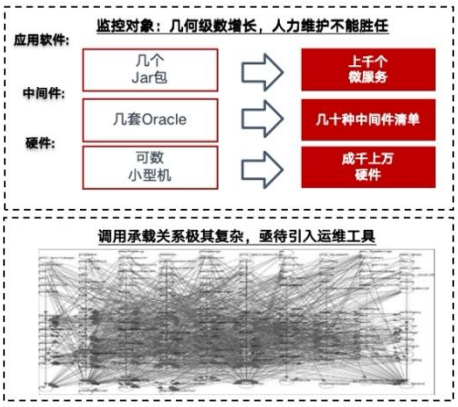


例如在故障过程中，如何快速发现问题。我们面临的一个挑战是指标众多，但难以全面掌握。不同层级——基础设施层、云平台层和应用层——之间的数据不共享，如何实现它们的统一共享？是否存在各自为政的情况？即使告警已经发出，但无人处理的情况又该如何应对？在微服务和容器化的环境中，服务之间的调用非常复杂，我们是否有快速排查

故障的手段？服务和主机之间的关系是否明确？当问题出现时，我们能否迅速定位？我们是否有 24 小时值班的运维人员？是否具备自动化快速自愈和止损的能力？在故障过程中，能否迅速调度故障专家进行处理？应急预案是否有效？故障复盘后，除了整改措施，是否有进一步的优化？我们需要挖掘和治理哪些风险隐患？以上均是我们的运维人员面临的挑战。

云原生下运维的问题挑战

随着云原生技术的不断成熟，企业数字化转型也在不断加速，企业 IT 架构进入云原生时代，多云多集群部署已经成为常态和趋势，几何增长的云资源、微服务以及复杂化的调用关系与业务场景，传统人肉运维难以为继，如何保障系统的全面稳定，保证业务流程的高效运转，为系统运营提出了不小的挑战。



接下来，我将举一些典型案例，说明白不同故障在不同层面的办理一些案例截图

故障根因在 SaaS 服务下的实例，具体某个主机上的服务失败调用增加。

故障根因在 PaaS 组件，有时候，服务本身没有问题，但是依赖的下游组件出现了故障。例如，依赖的 Redis 突然出现了雪崩，或者调用的 ES 突然查询变慢。这些问题又该如何处理？

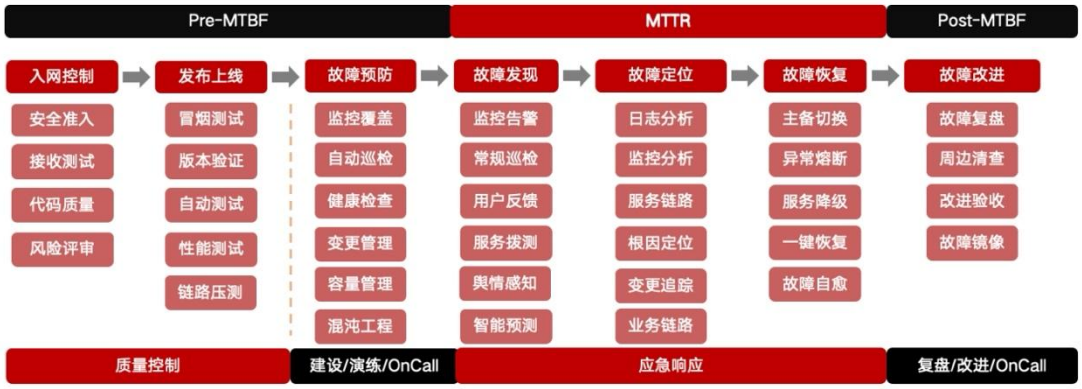
故障根因在外部接口案例：外部接口故障：除了自身系统内部的问题，还可能会遇到外部因素的干扰。例如，依赖的第三方接口出现了问题。这样的问题又该如何快速发现，快速感知，快速排查呢？

（二）解决方案详情及案例

数字化监控平台的核心能力三个解决方案

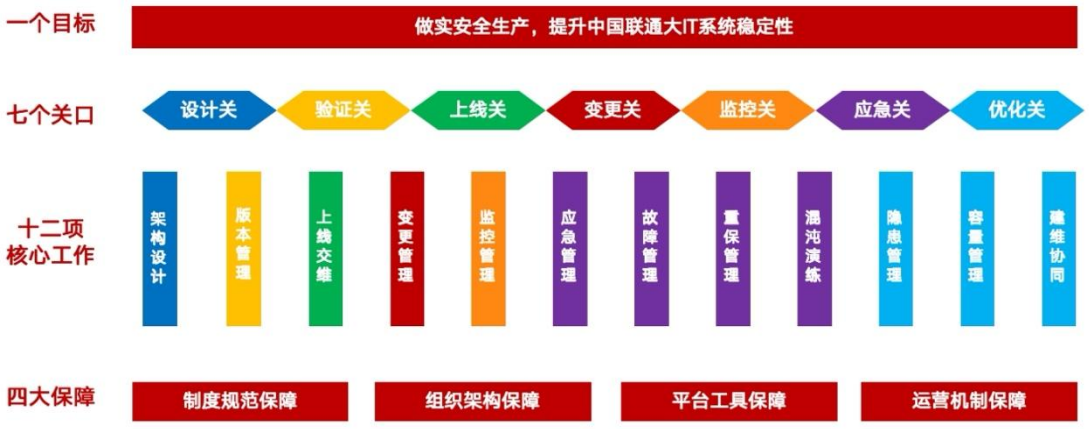
稳定性保障解决方案

解决方案，将安全生产稳定性保障左移，在入网控制时介入，对入网控制、发布上线、故障预防、故障发现、故障定位，故障恢复、故障改进提供端到端工具支撑。



生产运营保障体系结构框架

生产安全保障体系：一个目标，依托四大保障，聚焦研运流程中十二项核心工作，严格把控七个关口。



设计关关键点：架构设计阶段，确保包含冗余设计、灾备设计、熔断限流等稳定性设计。

验证关关键点：上线前进行严格的版本管理和验证，确保系统符合稳定性和安全性要求。

上线关键点：确保研发与运维之间的无缝衔接，规范上线流程，确保系统平稳上线。

变更关键点：变更前进行流程审批，变更中严格执行规范，变更后进行全面检查，确保变更过程安全可控。

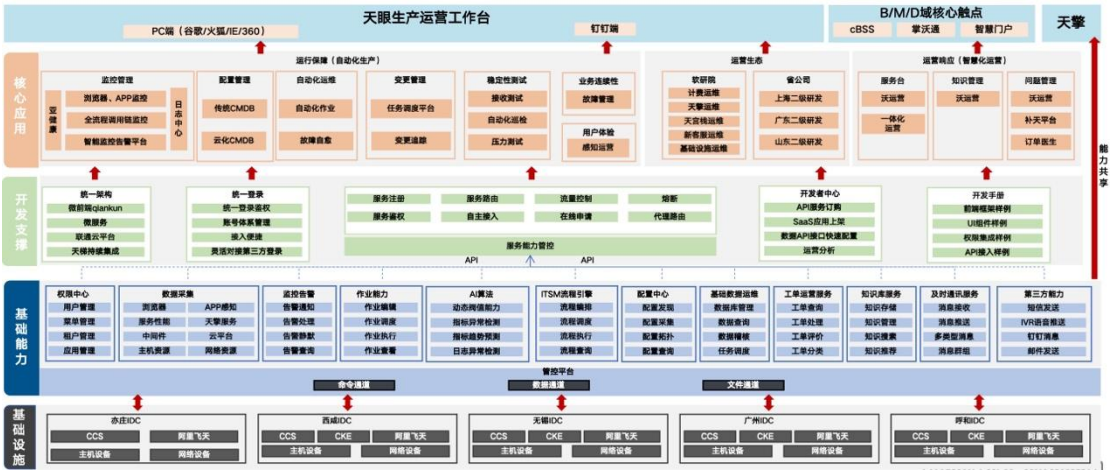
监控关键点：建立核心的可观测性能力，实时监控系统运行状态，及时发现和处理异常。

应急关键点：制定并演练应急预案，涵盖故障上报、调度和复盘，确保在故障发生时能够快速响应和恢复。

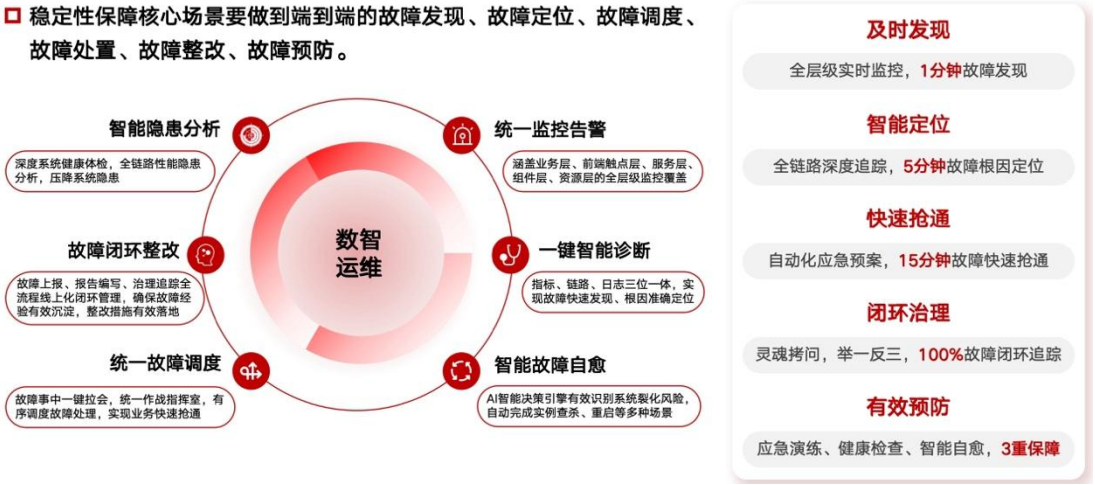
优化关键点：持续优化系统，定期进行隐患排查和治理，评估和管理系统容量，确保系统稳定性和高效性。

数字化监控平台架构

基于云原生下的生产运营支撑平台，以全局运营视角解读 IT 运维，提供端到端、全层级的运维工具支撑，依托大数据与人工智能技术，助力企业数字化业务高效、稳定运行，从传统运维向自动化生产、智慧化运营转变。



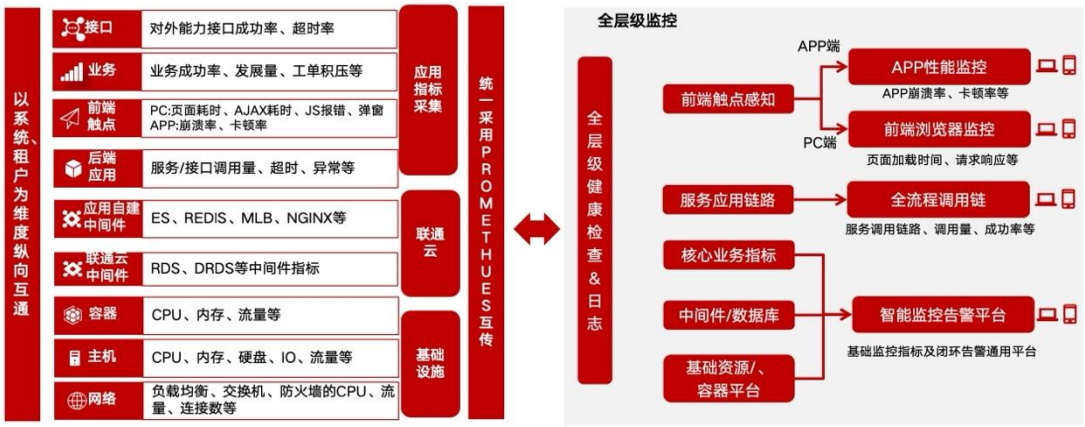
稳定性保障的场景应用实践



1-5-10 目标是由阿里提出的，联通则设定了"1-5-15"目标，即 1 分钟内发现故障，5 分钟内判断故障根因，15 分钟内快速恢复。虽然尚未完全实现 15 分钟内恢复，但通过多年的努力，目前故障 15 分钟内恢复的比例已达到 65%。

为了实现这些目标，我们构建了六个核心场景，来指导我们的工作。

统一监控告警

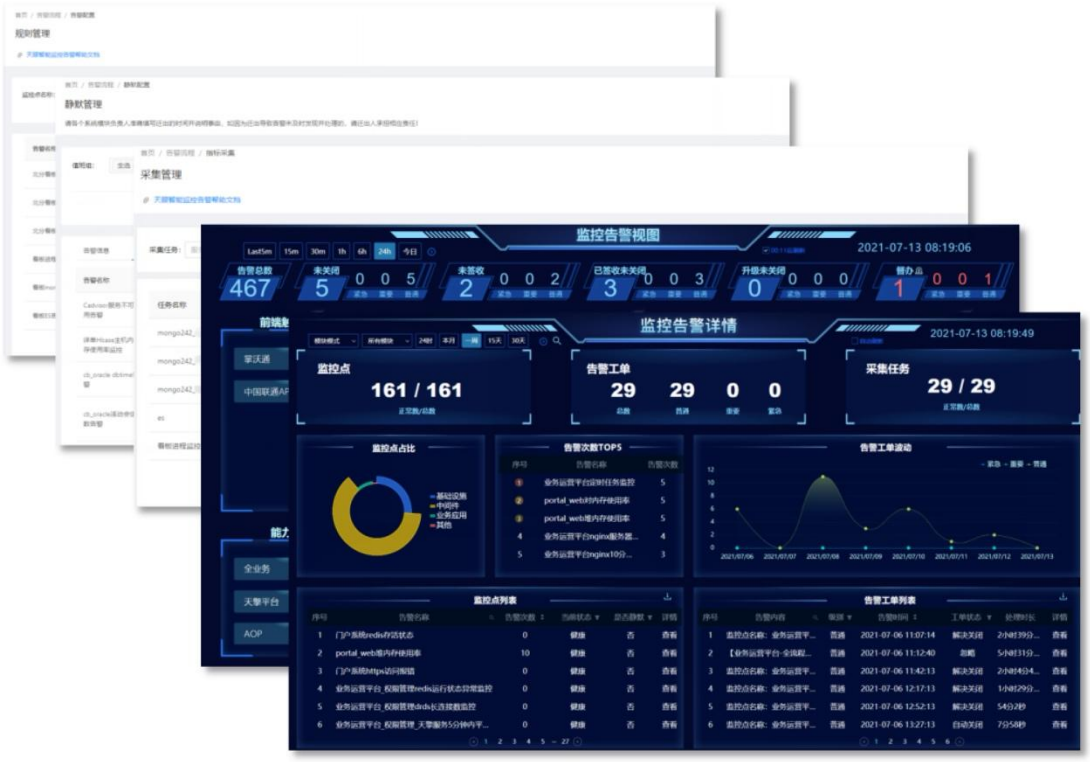


要进行统一监控,我们统一采用 Prometheus 协议进行监控数据的采集，并按照系统和租户的维度进行监控，以实现不同层级指标的统一管理，所有这些监控数据最终通过 CMDB 进行关联和汇聚，确保监控的全面性和准确性。

图示中展示了全层级监控标准，打破分散割裂格局，实现全层级、全链路、端到端的性能监控和链路追踪。

智能监控告警平台

平台提供 IaaS、PaaS、SaaS 各层级监控能力，实现多层级运维数据互通，支持全流程可视化配置，多渠道告警通知，工单闭环管理，用户快速实现监控接入，为系统日常生产运行提供保障。



数据采集：采集组件管理、私有数据仓库接入、租户自定义采集

- 监控配置：告警规则、收敛条件、告警内容
- 静默管理：多维静默管理（全量、监控点、监控实例）
- 告警通知：告警工单推送、电话催办
- 告警处理：双终端工单处理、工单闭环管理
- 告警大屏：系统监控告警全景图、告警工单处理进度

我们的告警平台具备全面的统一纳管能力，涵盖基础设施层、平台层、组件层、服务层、触点层和业务层。通过统一的数据拉通和监控告警，我们实现了全面的系统监控。

联通具有天然的优势，例如短信和外呼能力。这些能力已集成到我们的短信中心、通知中心和外呼中心中，能够快速通知相关值班人员。与互联网公司不同，我们更强调责任问题，确保故障通知到具体值班人员，而非简单地发到群里无人认领。

具体流程如下：

- 故障通知：故障发生时，系统会自动通知相关值班人员。

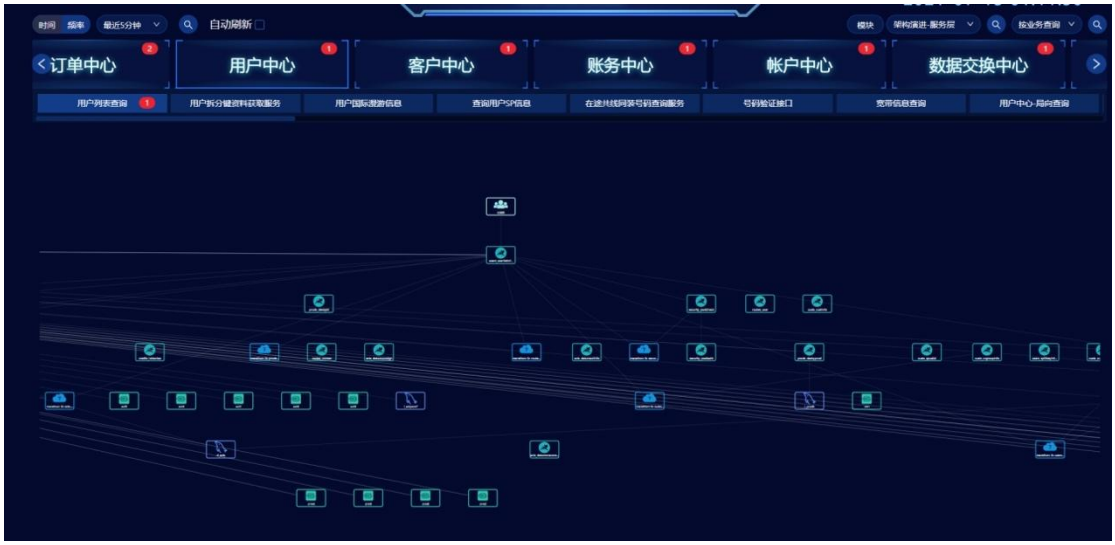
- 故障处理：如果值班人员在 10 分钟内未签收处理，系统会自动升级通知给其负责人。
- 逐级升级：若负责人仍未处理，系统会继续升级通知至部门经理，直至故障被处理。
- 7x24 小时值班：我们有专门的调度值班组，确保外呼能及时联系到值班人员。如普通外呼未能联系成功，将换值班人员进行外呼，确保告警及时签收处理。

此外，我们有一套运营机制，通过这套机制压降签收处理时长，确保告警得到及时处理，保障故障 1-5-15 目标的实现。

针对这些全层级的监控指标，需要了解每个指标的定义和标准。为此，中国联通软院制定了 414 项的全层级指标规范，包括每个指标的推荐配置、必须配置的指标、每个指标的配置阈值等。

1.全流程调用链监控

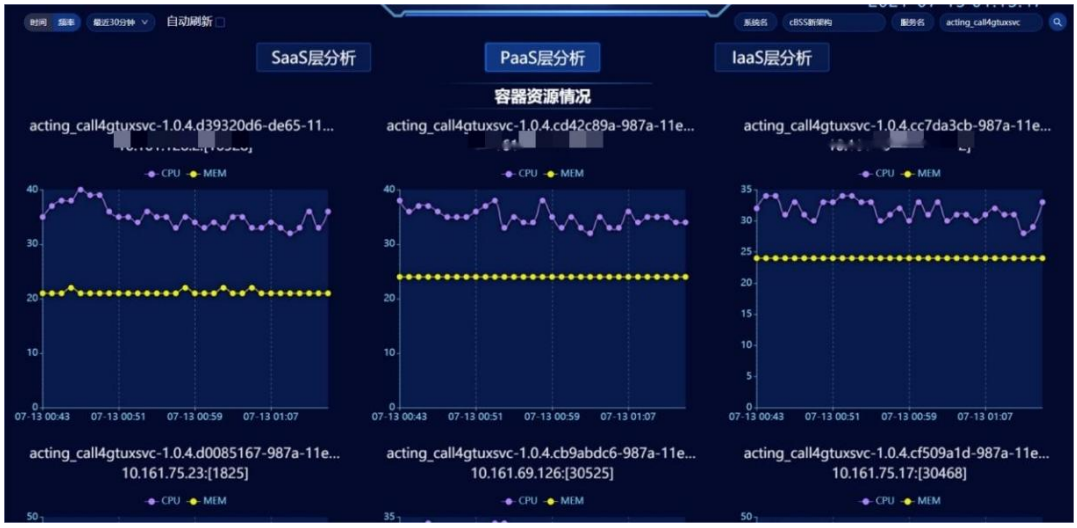
我中国联通的集中系统包含 5000 多个服务，链路拓扑复杂，需依赖技术手段进行梳理，我们通过全流程调用量监控(即 APM)实现对系统的全面监控。最初采用 PinPoint，去年开始升级为基于 OpenTelemetry 的系统，兼容两种形式，采集黄金指标并绘制链路拓扑。通过相关数据，在出现问题时，我们能迅速定位问题指标，如调用量下降、失败率上升或接口超时，并及时发出告警。通过 CMDB，我们可以定位到具体实例，快速重启实例以恢复故障，并查看应用所在的容器和主机的详细指标。



调用拓扑 全流程调用链拓扑自动生成，分租户管理

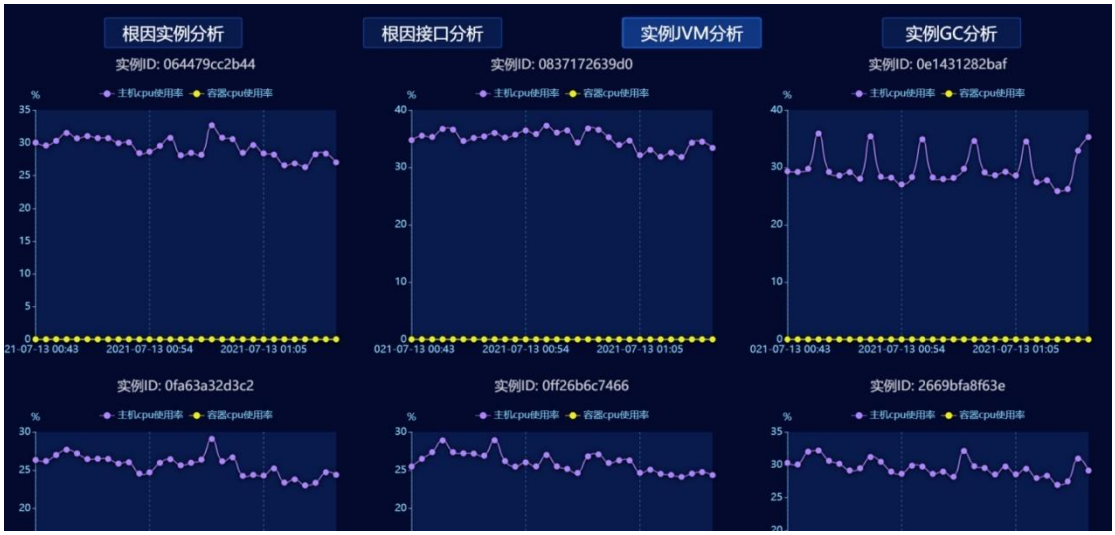
服务趋势/报错异常：服务调用关系、趋势图、报错分类（系统/业务）

实例/接口分析：调用链与云化 CMDB 做关联，关联到容器与主机



SaaS/PaaS/IaaS PaaS 层组件、平台容器资源情况，IaaS 层主机资源

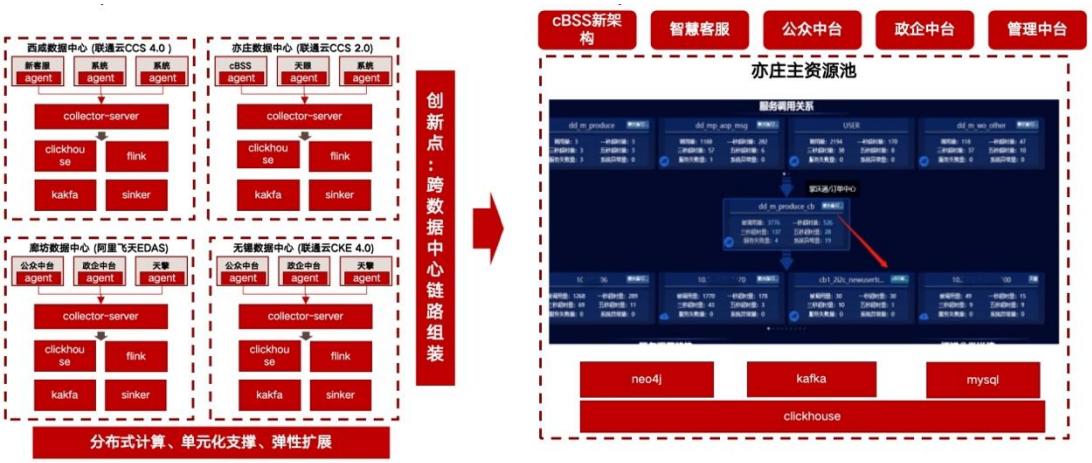
JVM/GC 分析 服务实例 JVM 与 GC 情况分析



告警配置：调用量、超时、异常黄金指标多指标自由组合

跨系统分布式追踪

通常的链路跟踪系统串联范围通常只局限在某一云平台之上，但我们通过分数据中心汇总串联，实现了分数据中心汇总串联，支持跨系统、跨云平台（CKE/CCS/EDAS）、跨数据中心（亦庄、西咸、廊坊、无锡）链路拓扑，完成跨系统调用实时追踪和方法清单级根因定位，日均处理近千亿数据。



前端触点监控

采用 JS 埋点的方式，采集用户访问过程的性能指标，获取浏览器端的真实用户行为与体验数据。包括页面加载、点击、弹窗、JS 报错、ajax 等用户全轨迹跟踪，通过大数据分析，应用于院内故障定位、安全分析、终端分析、感知分析、异常分析等场景。

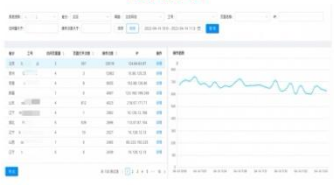
系统总览

页面性能分析

弹窗分析

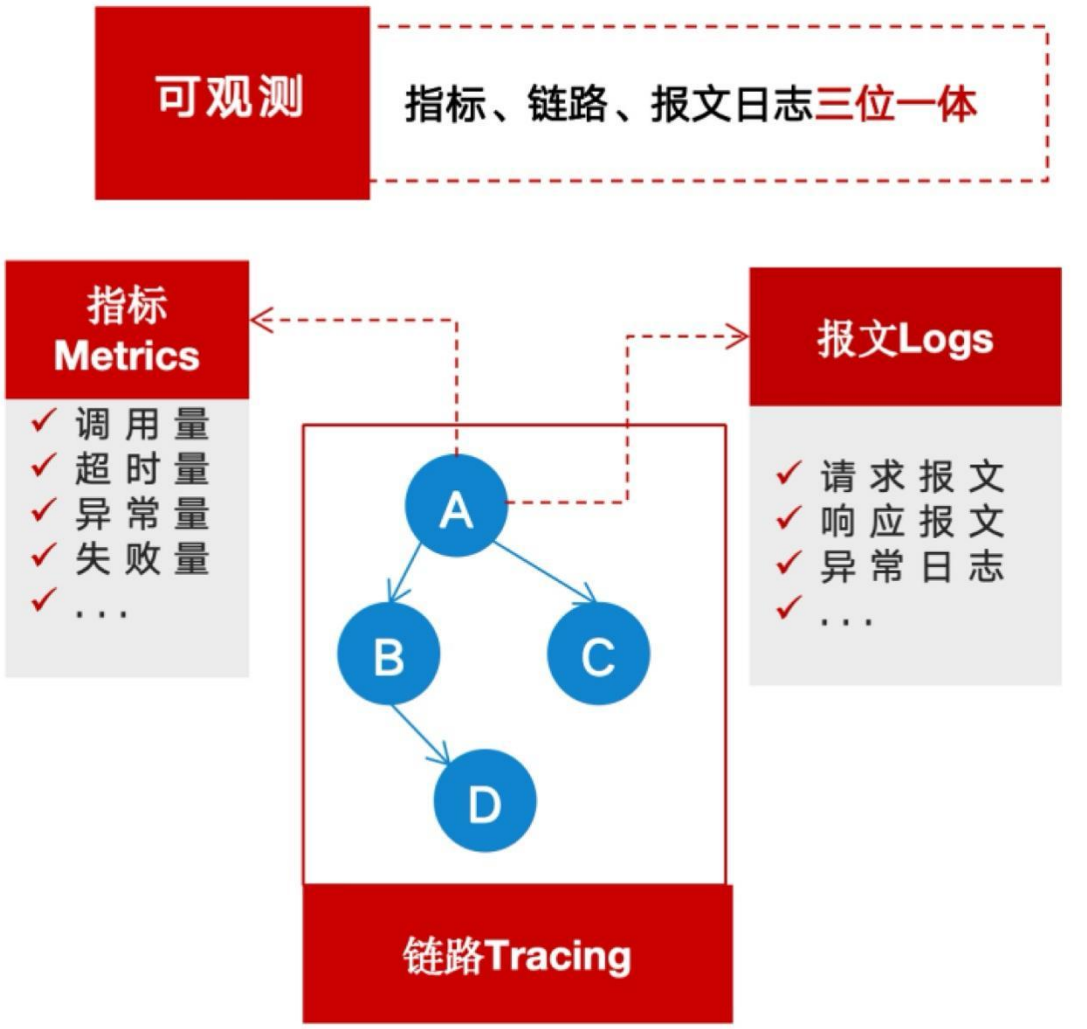
AJAX分析

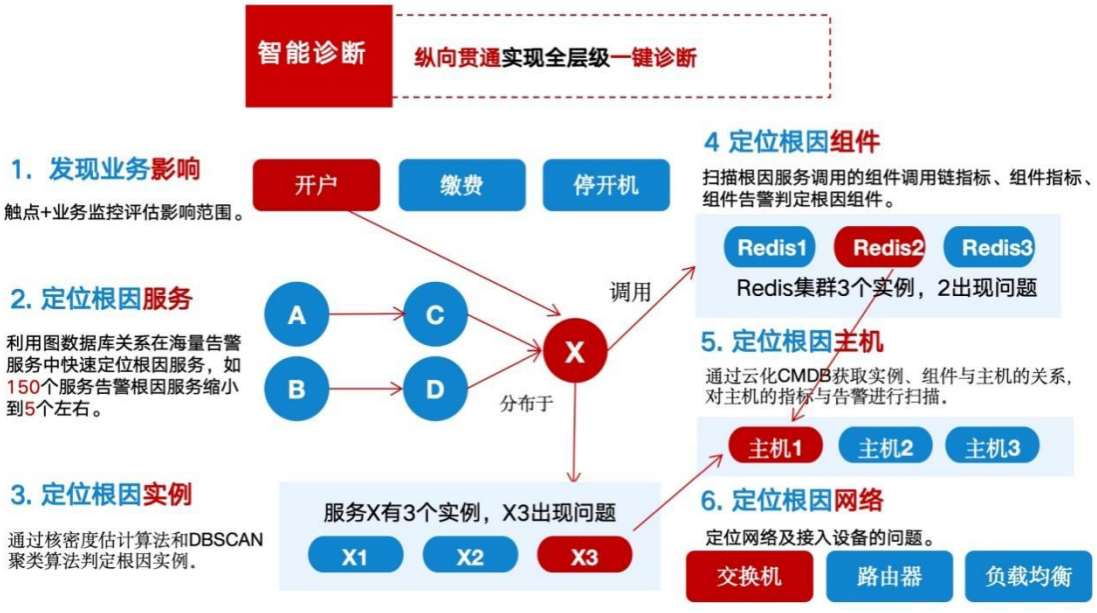
用户轨迹分析


工号稽核

一键智能诊断

通过采集指标、链路、报文日志，实现三位一体的可观测性，在系统纵向全层级方面实现触点层、服务层、组件层、平台层、主机层、网络层纵向贯通，结合云化 CMDB 关联定位，实现全层级一键诊断，端到端快速定位问题根因。


可观测
指标、链路、报文日志三位一体
指标
Metrics
✓ 调用量
✓ 超时量
✓ 异常量
✓ 失败量
✓ ...
报文Logs
✓ 请求报文
✓ 响应报文
✓ 异常日志
✓ ...
A
B
C
D
链路Tracing


智能诊断
纵向贯通实现全层级一键诊断
1. 发现业务影响
触点+业务监控评估影响范围。
开户
缴费
停开机
2. 定位根因服务
利用图数据库关系在海量告警服务中快速定位根因服务，如150个服务告警根因服务缩小到5个左右。
A
B
C
D
X
调用
分布于
3. 定位根因实例
通过核密度估计算法和DBSCAN聚类算法判定根因实例。
服务X有3个实例，X3出现问题
X1
X2
X3
4 定位根因组件
扫描根因服务调用的组件调用链指标、组件指标、组件告警判定根因组件。
Redis1
Redis2
Redis3
Redis集群3个实例，2出现问题
5. 定位根因主机
通过云化CMDB获取实例、组件与主机的关系，对主机的指标与告警进行扫描。
主机1
主机2
主机3
6. 定位根因网络
定位网络及接入设备的问题。
交换机
路由器
负载均衡

故障诊断案例

依托全层级监控指标数据、全层级链路调用、云原生 CMDB，建立故障传递模型，以服务层为故障起点进行纵向串联，配以规则+AI 的能力实现全层级一键智能故障诊断。目前，中国联通一键智能诊断准确率大概在 70%左右。

服务实例异常：根因服务实例耗时突增实例 GC 引发故障

Oracle 异常：Oracle 会话数突增导致服务连接超时增多

ES 异常：ES 进程负载率突增导致上游服务连接超时

RDS 异常：RDS 慢 SQL 突增导致节点状态异常

Redis 异常：Redis 耗时波动引起上游服务连接超时

快立方异常:根因服务下游调用快立方告警异常

主机宕机异常：LB 所在主机宕机导致 lb 实例销毁重启服务波动

网络异常：网络带宽使用率指标打满引起访问受限

智能故障自愈

将"监"、"管"、"控"工具能力融合，告警信息结合 AI 判定算法，触发自动化作业能力，实现故障自愈流程，有效缩短故障处理、恢复时间。

以下是案例：

16:20:45 应用告警

16:20:45 自动触发诊断

16:20:47 推送实例查杀、重启工单

16:21:29 运维人员确认操作

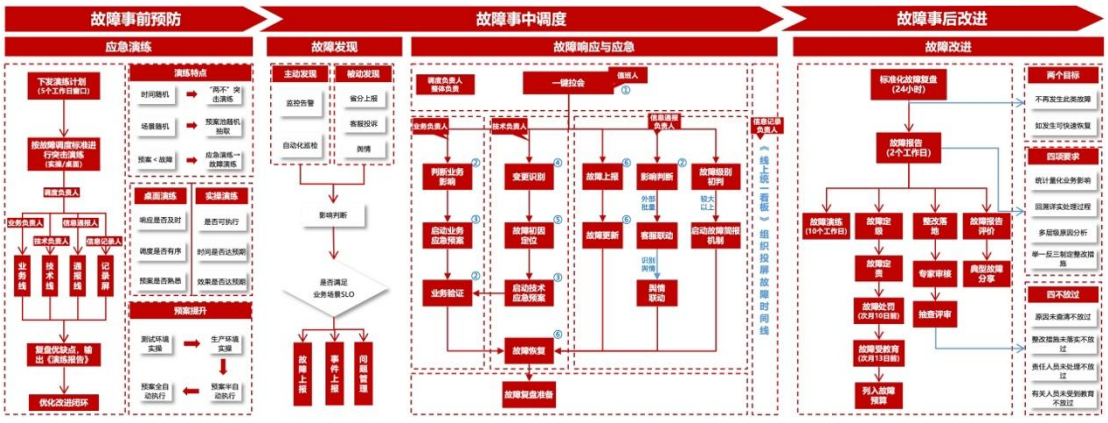
16:21:34 应用恢复

可以看到，从收到告警到恢复仅用 47s，与手工操作相比减少了 4 分钟故障恢复时间。

故障闭环管理

故障事前、事中、事后全流程线上闭环管理，提升故障管理质量和效率，降低故障时长及次数，提升业务连续可用率。

在事前阶段，我们会编制全面的应急预案，并定期进行应急演练。这些演练不仅有助于我们预防故障，也能确保应急预案非一纸空文，而是真正可执行的。我们会进行桌面演练和实操演练，前者用于检验流程的有效性和响应速度，后者则用于验证应急预案的执行能力和预期效果。通过这些演练，可以提升整个应急预案的可用性。



在故障发生阶段，主要通过监控告警、巡检、省分报告、客户投诉和舆情等方式发现故障。一旦发现业务 SLO 被触发，将进行故障上报，并通过一键拉会功能召集相关值班人员进行会议。这些人员包括故障调度负责人、业务负责人、技术负责人、信息通报负责人和信息记录负责人，他们各司其职，确保故障得到及时、有效地处理。

在故障结束后，我们会进行故障改进。在 24 小时内进行标准的故障复盘，在 2 个工作日内完成故障报告编写，包括故障信息、根因分析、操作处置流程及问题整改措施等内容。在 10 个工作日内，我们将对故障进行演练，并根据出现的问题进行整改闭环追踪。

作为央国企，我们严格遵循"四不放过"原则，即：

1.原因未查清不放过：必须彻底查明故障原因，确保问题根源明确。

2.整改措施未落实不放过：确保所有整改措施得到有效落实，避免问题再次发生。

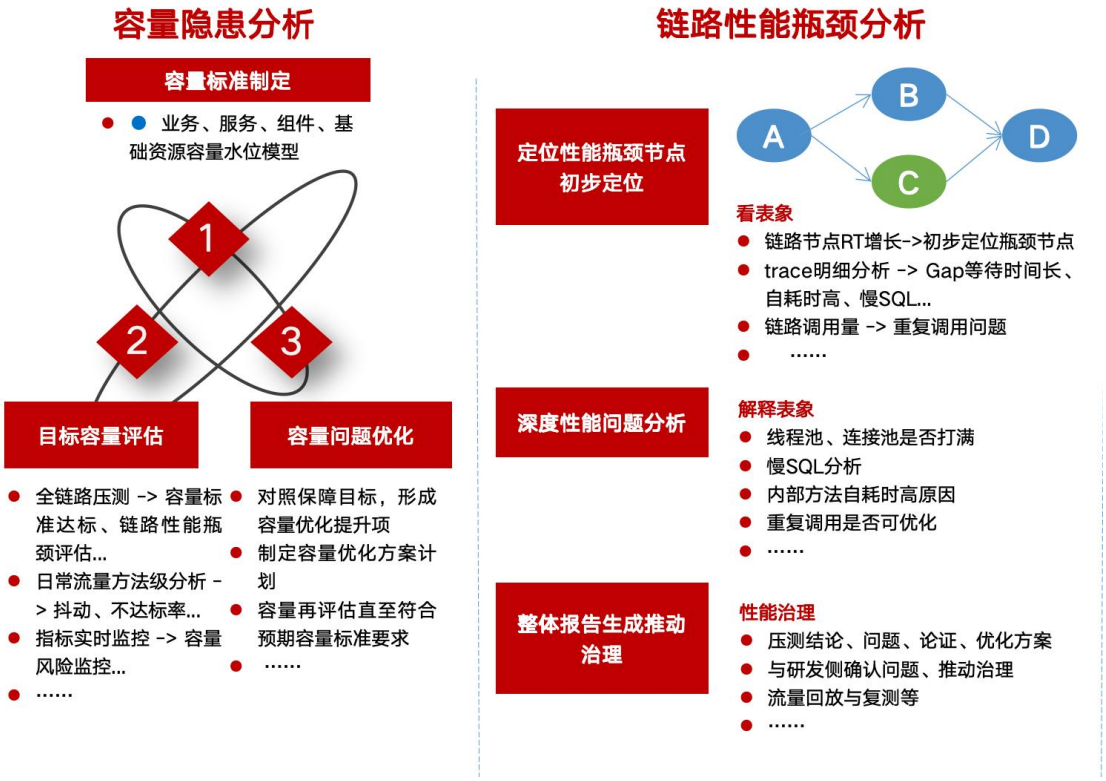
3.责任人员未处理不放过：明确每个故障的责任人，确保责任人员得到应有的处理。

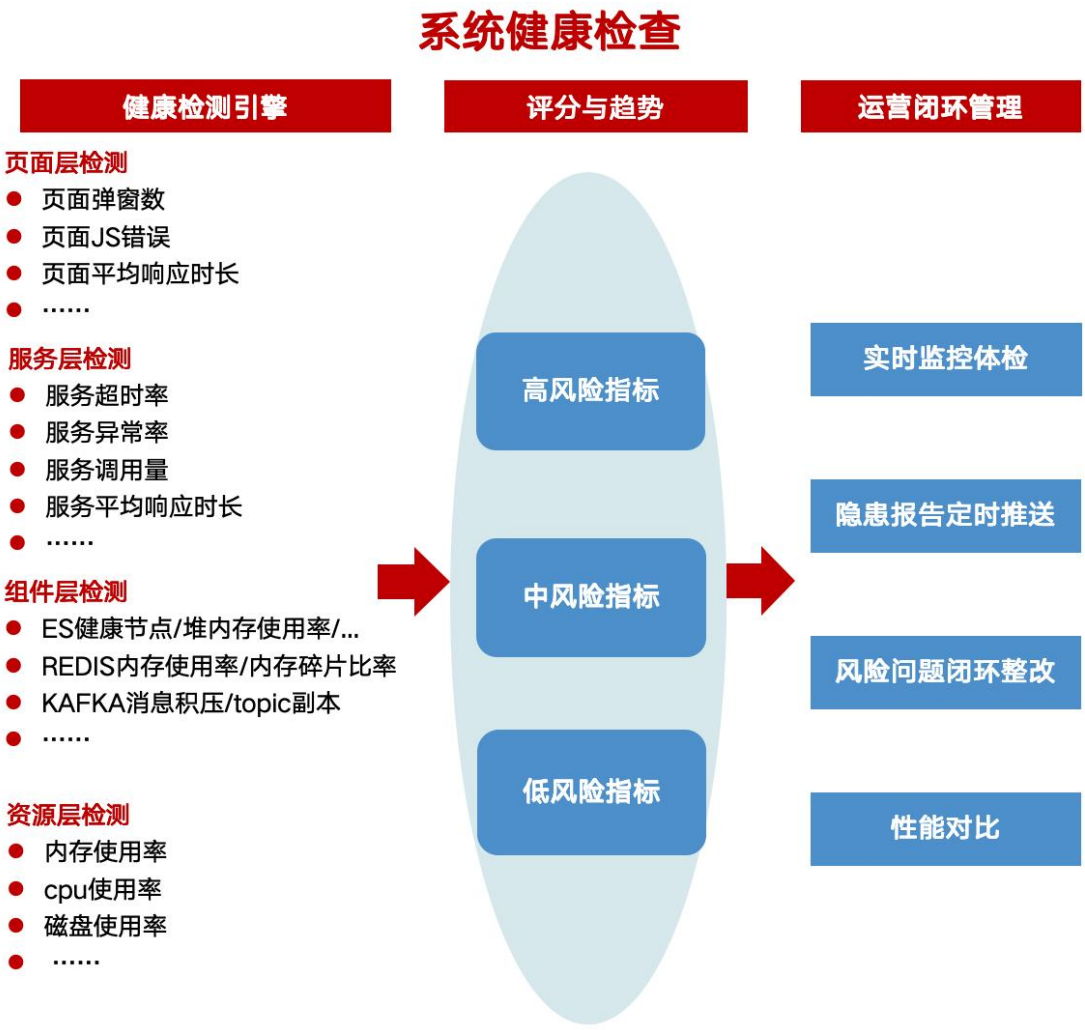
4.相关人员未受到教育不放过：对相关人员进行必要的教育和培训，提高整体防范和解决问题的能力。

以上就是我们在故障的事前、事中、事后全生命周期的线上闭环管理方式。

智能隐患分析

结合监控指标与容量指标，定期开展容量隐患评估，通过核心业务链路的全链路压测，分析链路性能瓶颈，建立健康度算法模型，识别与治理系统潜在风险隐患，保障系统健康稳定。





亚健康检查案例

上面是一个系统亚健康的实践案例分享。通过全层级指标，包括网络层、平台层、组件层、服务层及触点层等不同层级的黄金指标，运用 AI 算法分析来建立健康度算法模型。

例如，如果 CPU 使用率超过 90%，将发出告警。但如果 CPU 使用率一直维持在 80%左右，即使没有达到告警阈值，也能通过风险等级判断其为高风险隐患，从而识别出是否为亚健康状态。

通过全面的隐患分析，可以实现系统健康状态的档案化管理。可以通过日、周、月等不同的时间维度来统计系统问题和风险情况，观测系统在各个阶段的运行情况。也进行实时的系统体检和计算，通过不同的阈值判断指标异常以及风险程度。

还可以进行版本前后的性能对比。例如，如果版本更新前有 10 个风险隐患，更新后增加到 20 个，那么此次更新就是失败的，因为它增加了风险隐患。我们需要提前规避这些问题，提前治理。如果风险隐患减少到 5 个，那么这次更新就是有效的，可以看到它解决了哪些风险隐患。通过不同时间段的对比，可以看到整个系统性能变化的趋势。

还可以定期自动生成体检报告，通过工单形式发送给相关系统的负责人，让他们明确了解系统的风险隐患以及如何进行治理。体检报告会列出相关风险指标，并提供解决方案，助力提升整个系统的稳定性。

（三）总结及展望

数字化监控平台依托生产运营保障体系，构建稳定性保障解决方案，实现了从统一监控告警到智能隐患分析的六大核心场景闭环，显著提升了系统稳定性与运维效率。不仅实现了故障的快速发现与精准定位，还通过一键智能诊断与自愈功能，大幅缩短了故障恢复时间。统一故障调度与闭环整改机制，确保了问题得到彻底解决，有效预防了类似故障的再次发生。

展望未来，随着 AI 技术的深入融合，聚焦生产运维核心应用场景，充分发挥大模型优势，基于大模型+AI Agent 能力，通过提示词工程、检索增强和智能体构建等技术手段，打造生产运维智能体系，通过 AI 工具+大模型能力的赋能生产运维新模式，基于 AI 技术的产品重构和系统之间的拉通，平台将更加智能化，实现故障的提前预警与主动干预，为业务连续性保驾护航，提升生产保障效率，从而提升系统稳定性，引领运维管理迈向新高度。

5.3.3 腾讯全球化游戏故障管理实践

SRE Elite 精选原因

腾讯游戏在全球运营的多个游戏业务中，统一使用了 SLO /SLI 方法论，对业务进行业务导向的监控可视化，并使用了 eBPF 等技术，对业务进行无死角的观测，实现了业务

服务的标准化度量，故障的快速感知及定位。并能通过蓝鲸平台，实现了部分固定场景的自愈，实现了监控与批量作业的联动，降低了 MTTR，相关实践具备较强的落地性及可参考性。

(一) 背景及设计原则

在腾讯全球化的游戏业务中，故障管理面临着许多独特的挑战。这些挑战主要集中在网络复杂性、文化差异、云资源使用以及法律法规的合规性上。

范围	网络	监控指标	文化差异	云资源	法律政策	时区差异
全球化业务	网络环境复杂	指标和维度更多	文化多样	混合云部署	法规多元	多时区
国内业务	网络环境相对稳定	指标和维度相对单一	文化差异不大	单云部署	法规统一	单时区

首先，网络问题是全球化游戏的主要挑战之一。由于各国的网络质量和运营商服务差异显著，导致监控指标需求增加。例如，在国内游戏环境中，专线质量通常由专门部门保障，而在海外，SRE 团队必须自行监控并分析玩家的网络质量，包括国家、运营商和城市等多个维度的网络性能差异。

文化差异也对故障管理带来挑战。例如，在印尼，由于大部分人口信奉伊斯兰教，玩家在特定的祈祷时间段内会集中掉线。这种现象在特定时间段内频繁发生，最终通过了解当地文化得以解释。同样，在泰国，每天早上 9 点的升旗仪式期间，学校的学生和老师全体站立，无法使用手机，导致游戏的在线人数显著下降。这些案例表明，了解并适应当地文化习俗对于解决全球化游戏业务中的故障至关重要。

此外，使用多种云资源也是一个重要的挑战。为了满足全球合规要求，游戏公司需要在不同地区使用多种云服务，如 GCP、AWS 和微软云。这不仅增加了资源管理的复杂性，也对 SRE 团队提出了更高的要求，需要他们具备强大的数据分析和处理能力，以应对多维度的监控指标。

法律法规的差异也是一大挑战。各国的用户隐私保护法规，如欧盟的 GDPR 和美国的 CCPA，要求游戏公司在数据管理和上报时间的一致性上严格遵守相关规定。这需要 SRE 团队具备深刻的法律合规意识，并采用相应的工具来确保合规性。

因此我们建立了事前监控，事中应急，事后总结复盘的机制，来应对相关挑战。

（二）解决方案详情及案例

事前：如何以 监控、自愈、处理的流程的角度，构建故障管理体系

大规模全球监控体系构建实践

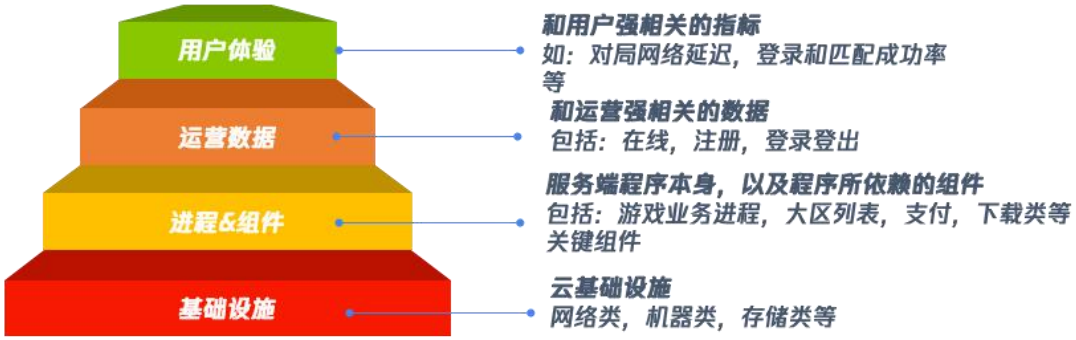
全球化监控的挑战实现快速的故障发现的挑战：

- 复杂的系统多样性：指标多，游戏登录配对对局支付成功率耗时，网络延迟、丢包率，带宽使用率抖动，技术设施可用性、性能
- 数据渠道碎片化：对象多，公共服务，游戏模块，组件模块，网络质量。
- 多维度指标：地理位置，运营商类型，策略类型，问题类型，用户类型，玩法模式类型，地图类型。
- 严格的安全合规要求：操作合规， 数据合规，代码合规……
- 渠道多，云监控，自定义监控，组件监控，日志监控，

首先，复杂的系统多样性和大量数据需要强大的数据分析和处理能力。此外，多维度的指标拆分需要 SRE 对游戏业务有深刻理解，否则上报数据无意义。严格的安全合规要求需要有强大的安全意识和工具来避免风险。

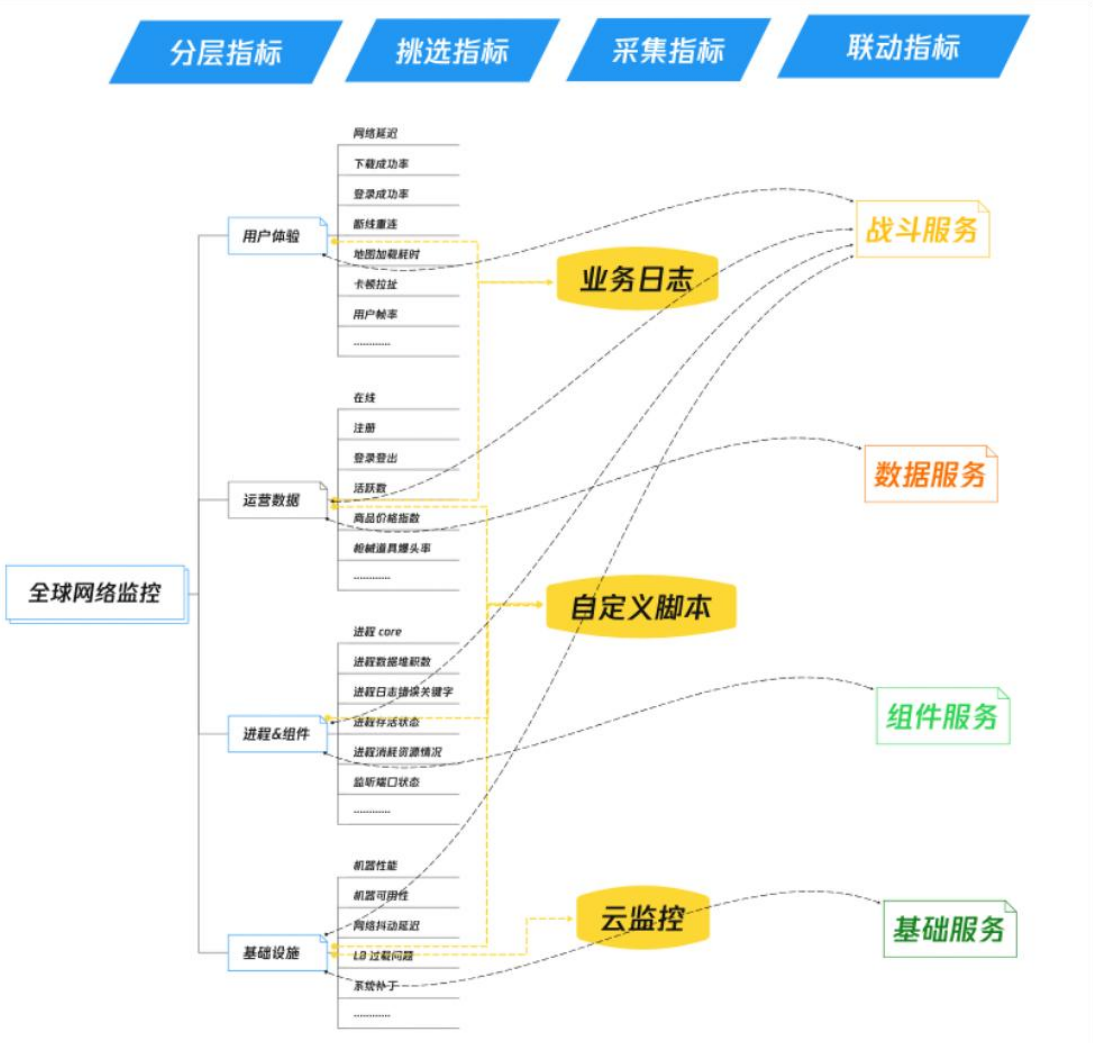
全球指标体系的建立思路

Step1 ：基于蓝鲸监控体系下，业务指标如何管理-指标分层



我们把业务监控指标做分层，从最下面的基础设施层到最上面用户体验层，总共四层，基础设施主要涵盖网络，机器类，存储类，进程和组件是游戏，或者组件程序本身的，运营数据包括在线，注册，最上面用户体验包括用户对局，匹配网络延迟

我们把业务监控指标做分层，通过这样的分层，我们能清晰的了解每一层的指标数据，能更立体，更全面地掌握业务质量和健康度 **Step2：建立监控指标体系**

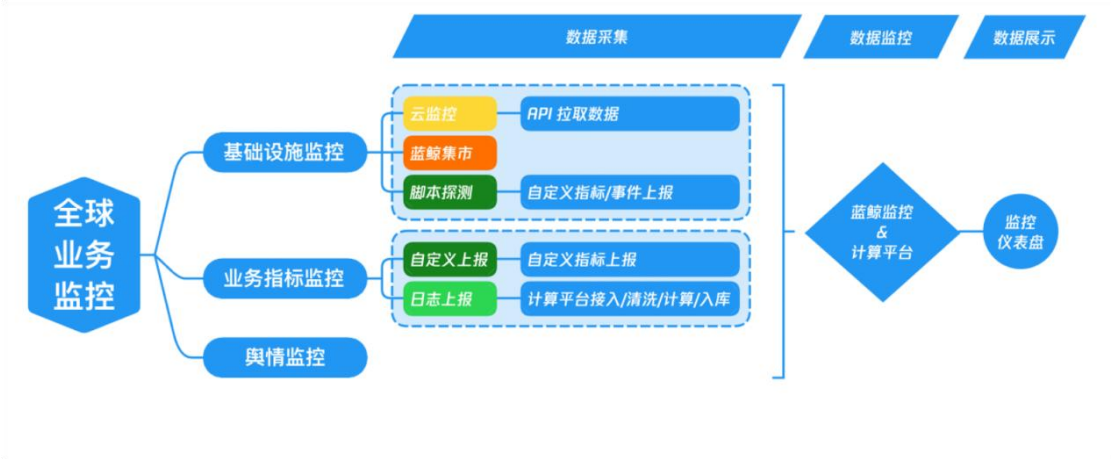


有了上面的分层指标后，接下来就要把二级指标最重要的挑选出来了，哪些指标能直接反应该层的性能和健康状况，而且可衡量，关联性强的，我们就选择。

下一个就是把指标采集下来，这些指标很多来自于业务日志，包括程序日志，运营数据日志，自定义脚本上报数据，还有云监控拨测数据等

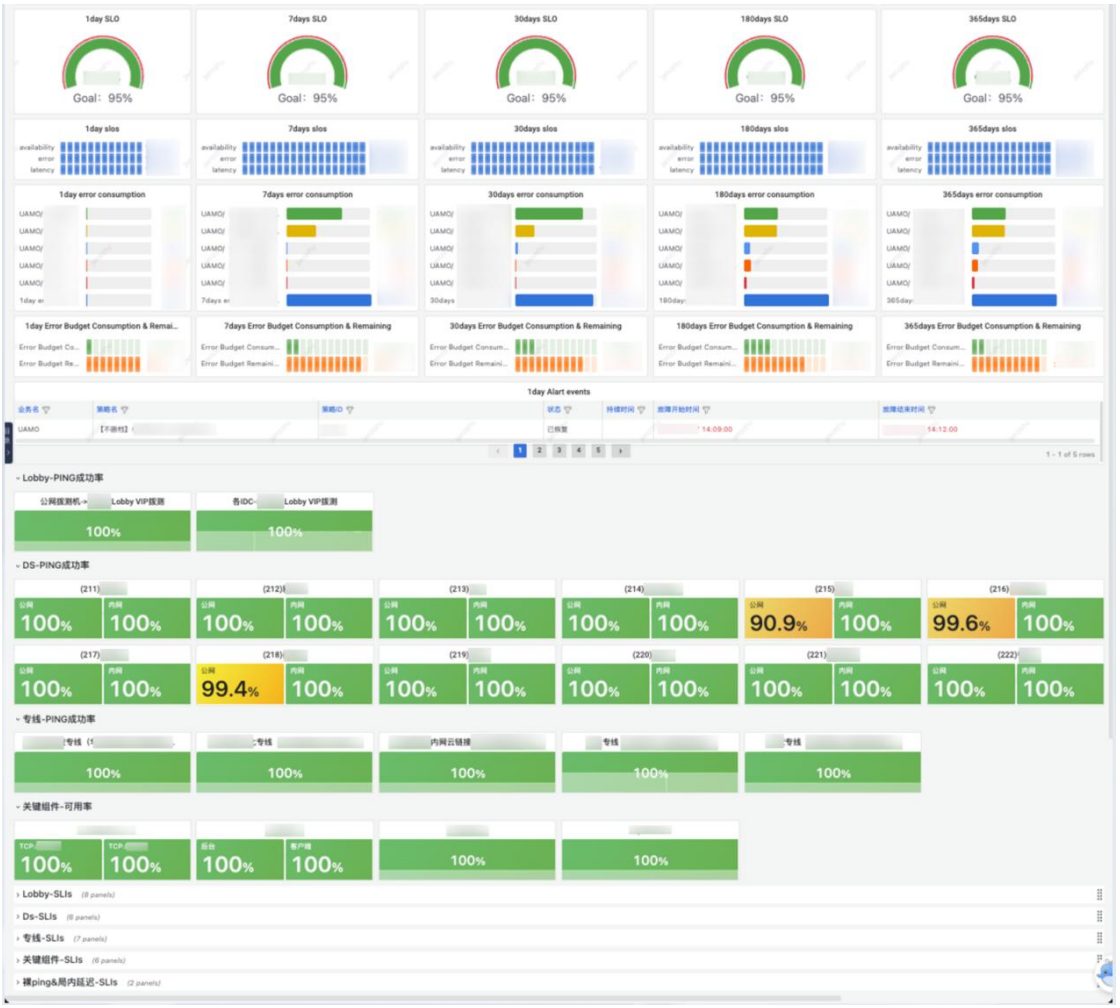
接下来需要将指标服务关联起来。这意味着你需要理解每个指标如何影响你的服务，以及服务之间的依赖关系。这可以帮助你更好地理解你的服务的性能和健康状况。

Step3：数据采集，统一管理



我们把全球的不同渠道的数据通过上报到计算平台和监控后，利用计算平台强大的数据计算，分析和存储能力，提高数据质量的安全性，提高效率。

Step4：建立网络质量的可观测大屏，快速发现和定位问题



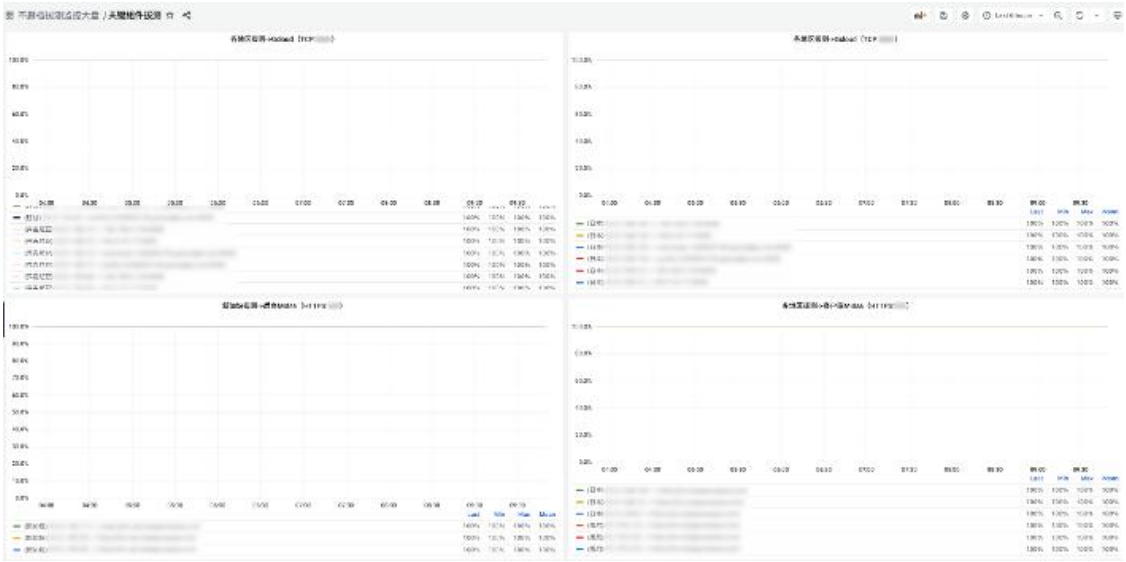
蓝鲸监控仪表盘显示的 SLO

最后，我们通过把指标和服务联动起来，通过可观测大屏呈现给用户

这是一张全球网络质量的可观测大屏，能快通过这张大屏很容易了解到关键组件，关键模块，专线的网络质量

最上面的 SLO 是代表整体，按照一天，天，一个月的时间来展示。出现问题后，可以通过下面的视图，快速定位到哪里到哪里的外网或者内网的 PING、是否有问题

Step4-1：基础设施可观测大屏



这里的基础设置，值得是游戏的基础组件，通过蓝鲸拨测解关键组件网络质量，相关组件可用性指标可能包括

- 下载服务可用性
- 登录，大区列表服务可用性
- 支付服务可用性
- 排行榜服务可用性

Step4-2：业务对局体验可观测

通过对游戏关键组件服务的端口拨测，来监控服务的可用性

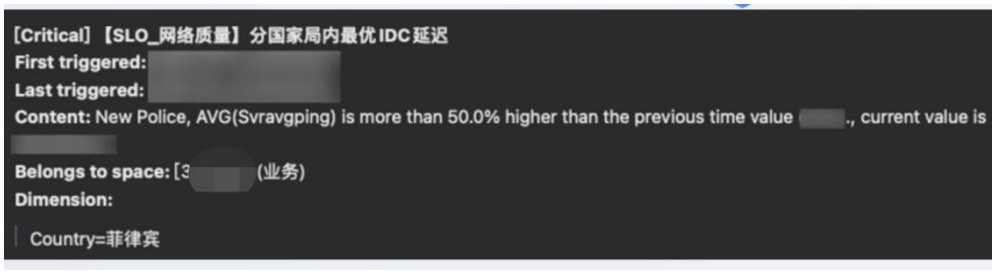
热力图是一种非常有效的数据可视化工具，可以帮助我们理解和解释复杂的数据集。

识别最佳和最差的 IDC 网络：热力图可以清晰地显示出哪些 IDC 网络的性能最好，哪些最差。颜色的深浅可以代表性能的好坏，比如深色可能代表性能好，浅色代表性能差。这可以帮助你快速地识别出最佳和最差的 IDC 网络。

- 提供决策支持：通过识别最佳和最差的 IDC 网络，可以做出更好的决策。例如，可能会选择将更多的资源分配给性能好的 IDC 网络，或者优化性能差的 IDC 网络。
- 跟踪性能变化：热力图不仅可以显示当前的性能，还可以显示性能随时间的变化。这可以帮助你跟踪和理解 IDC 网络的性能趋势。

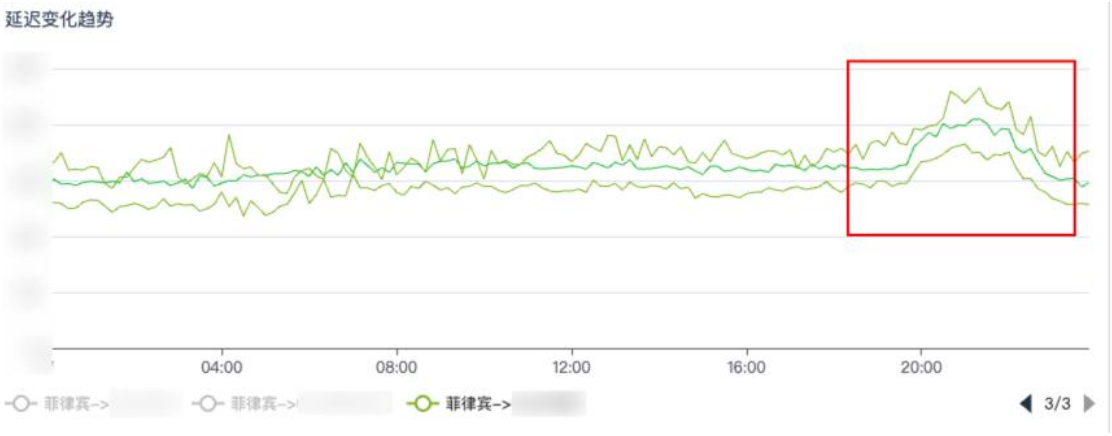
案例 1：维度下钻和数据关联，定位菲律宾网络问题

e.g. 晚上黄金时分，菲律宾玩家网络延迟为何突然升高？



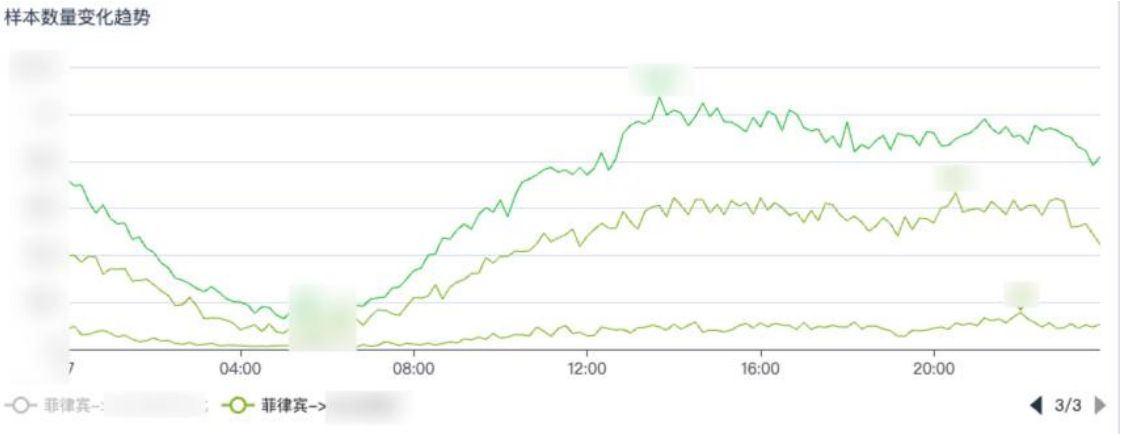
1. 1. 告警触发：

菲律宾最优 IDC 对局延迟升高 ->**触发监控告警**



2. 维度下钻：**检查分 IDC 对局延迟时序图**

-> 对局分配重点 IDC 中，延迟均升高且趋势一致**-> 基本排除 IDC 问题**

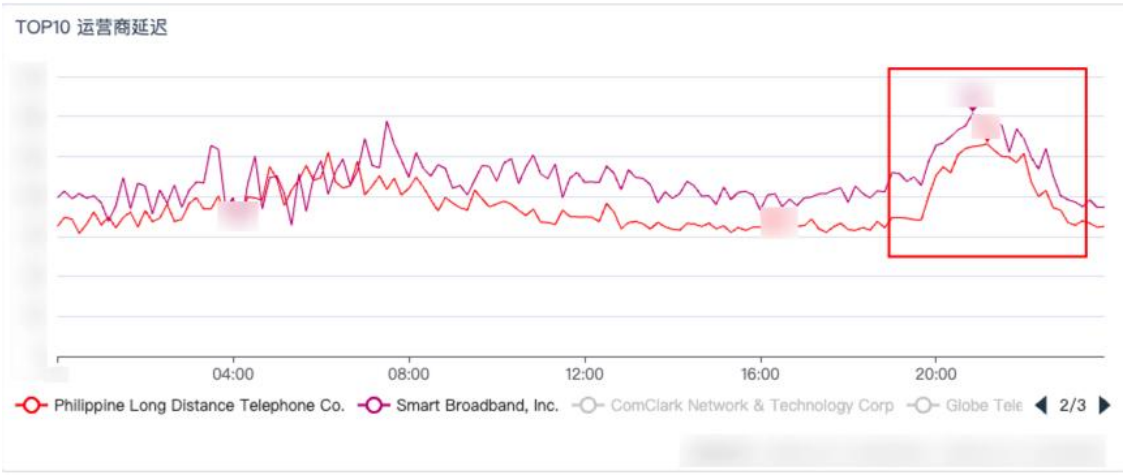


3. 关联检查：检查对局样本数量 -> 样本数量无异常

检查网络质量拨测数据 -> 无异常

检查菲律宾在线数量 -> 在线无影响

检查裸 ping 延迟 -> 有同样上升趋势**-> 基本排除业务网络链路问题，怀疑是公共网络故障**

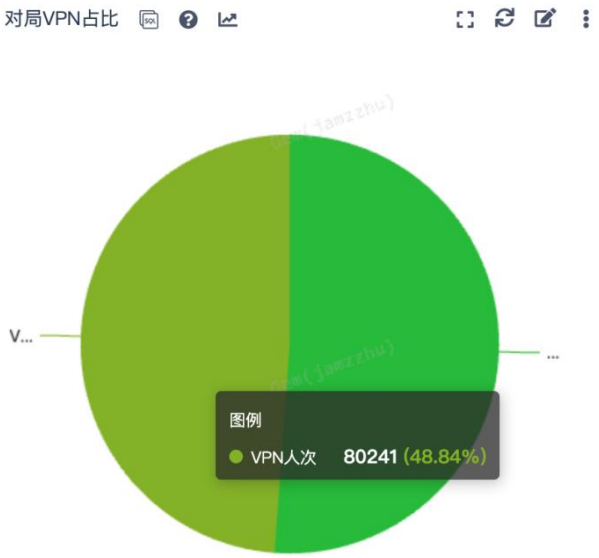


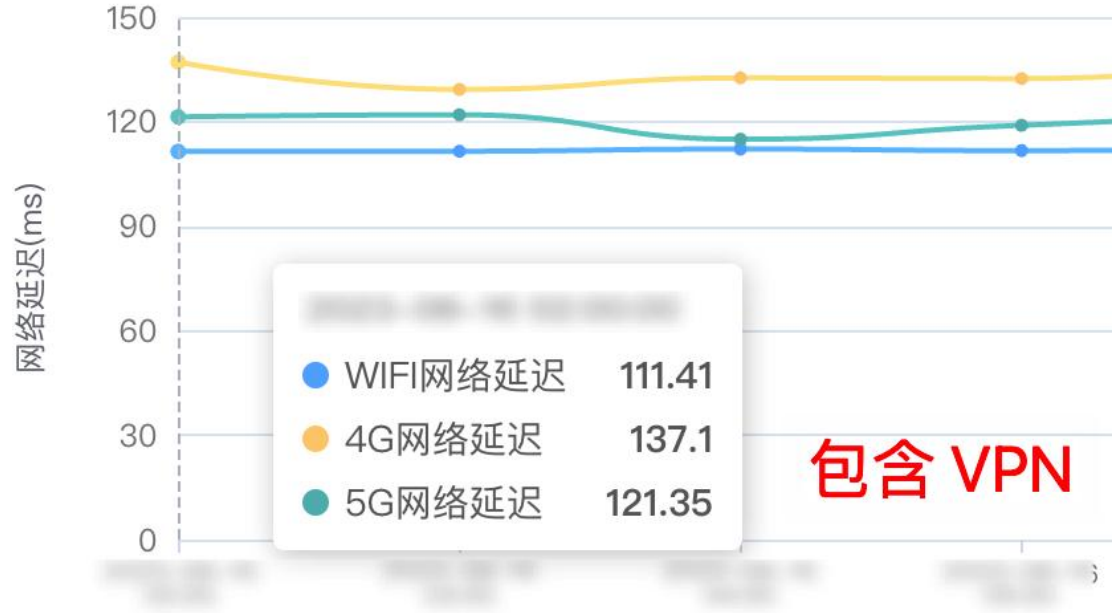
4. 换一个维度下钻：检查分运营商对局延迟时序图 -> 有两个 TOP 运营商延迟上升趋势和总体上升趋势一致**-> 初步判断为运营商问题**

5. 公共网络问题求证：找云厂商核实 -> 确认为运营商光缆故障导致

案例 2：网络可观测，数据层层剥离，破解网络疑难杂症，提供运营决策

1.新加坡玩家明明网速流畅，为何大盘延迟居高不下？

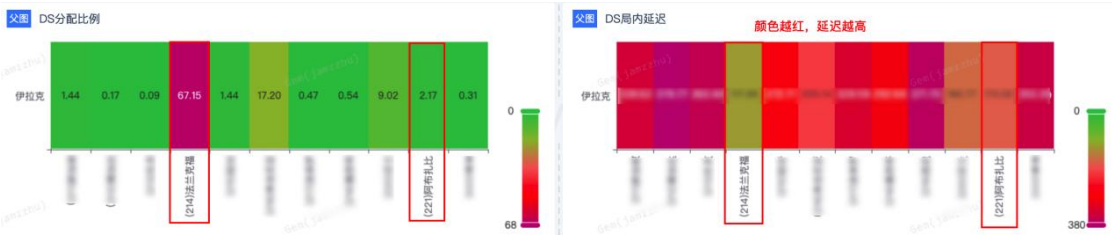




第一个例子是关于如何通过网络可观测性来解决网络疑难问题。我们以新加坡的玩家为例，新加坡的网络环境很好，但延迟却居高不下。通过分析，我们发现很多玩家使用 VPN 来访问游戏，这些 VPN 用户的 IP 地址显示为新加坡，但实际上来自其他国家。排除这些 VPN 用户后，发现实际的网络延迟显著降低，Wi-Fi 延迟只有 72 毫秒，4G 网络为 97 毫秒，5G 网络为 34.67 毫秒。

这表明问题在于使用了 VPN 的用户，并为我们的运营决策提供了依据，如针对 VPN 用户数量多的国家进行不同的运营策略。

2.伊拉克玩家为何不就近匹配，反而绕路去了欧洲节点？



伊拉克是中东国家，但它的玩家为何不连接到最近的服务器，而是连接到法兰克福？通过 trace 路由和节点分析，我们发现伊拉克的运营能力较弱，且与周边中东国家没有直接的海底电缆或点对点连接系统。结果，网络流量被绕道到法兰克福。尽管如此，伊拉克到法兰克福的延迟约为 100 毫秒，这对某些游戏来说是可接受的，因此许多伊拉克玩家选择连接到法兰克福的服务器。

案例 3：业务关键路径质量可观测，快速发现登录问题

用户旅程与关键路径——首先，我们按照用户旅程来分析业务关键路径。这个概念源自谷歌的用户旅程分析。以游戏应用为例，用户旅程通常包括以下步骤：

下载游戏->登录->匹配->对局->支付

SLO 大盘-下半部分

对应指标的实时曲线，明确不稳定的点具体信息**问题发现与告警机制**

在左上角的图表中，我们可以看到用户登录的成功率（SO）。当登录成功率出现红色警示时，我们的告警系统会立即触发告警。

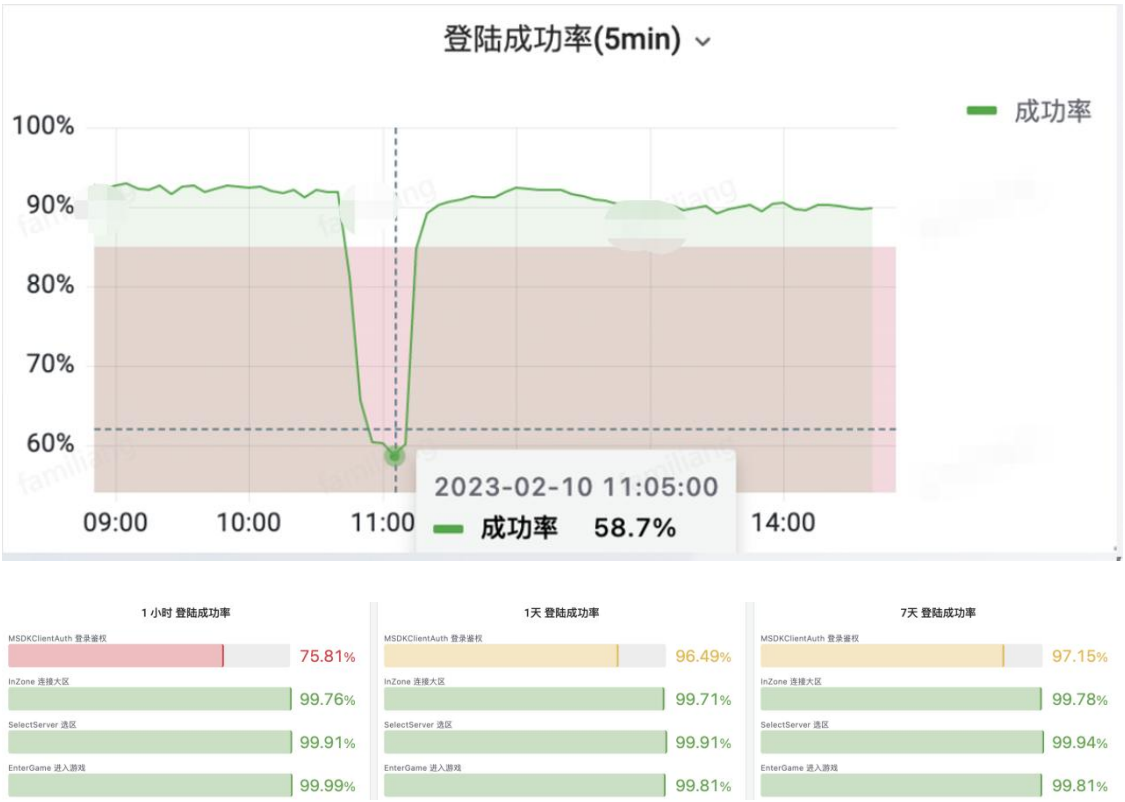
指标分析

收到告警后，我们首先检查左下角的各种指标，包括在线用户数和系统容量等。确认这些指标没有问题后，我们进一步分析右侧的细节。

图表显示了登录组件（MSDK）的成功率。我们注意到，成功率从 96%骤降至 93%，并继续下降至 80%左右。这表明 MSDK 组件出现了问题。

关键节点

Time	1、msdk鉴权成功率	3、Lobby进程存活性	4、Lobby主机CPU使用率	4、Lobby主机MEM使用率
2023-02-10 11:30:00	98.68	1.00	7.32%	15.28%
2023-02-10 11:25:00	98.38	1.00	7.26%	15.27%
2023-02-10 11:20:00	98.01	1.00	7.38%	15.25%
2023-02-10 11:15:00	96.54	1.00	7.33%	15.22%
2023-02-10 11:10:00	88.37	1.00	7.01%	15.20%
2023-02-10 11:05:00	86.03	1.00	6.83%	15.20%
2023-02-10 11:00:00	86.57	1.00	7.34%	15.21%
2023-02-10 10:55:00	86.72	1.00	6.83%	15.22%
2023-02-10 10:50:00	87.73	1.00	6.95%	15.22%
2023-02-10 10:45:00	93.19	1.00	7.02%	15.22%
2023-02-10 10:40:00	96.32	1.00	7.07%	15.20%
2023-02-10 10:35:00	98.89	1.00	6.96%	15.18%



时间轴与监控图

在下方的五分钟监控图中，我们可以看到问题发生的具体时间段，大约在 10 点到 11 点之间，尤其是 10:15 左右，成功率显著下降。

问题定位与根因分析

通过进一步探查，我们发现最底层的监控图显示成功率通常在 99%以上，但在问题发生时，成功率降至 75%。这再次确认是登录组件的问题。

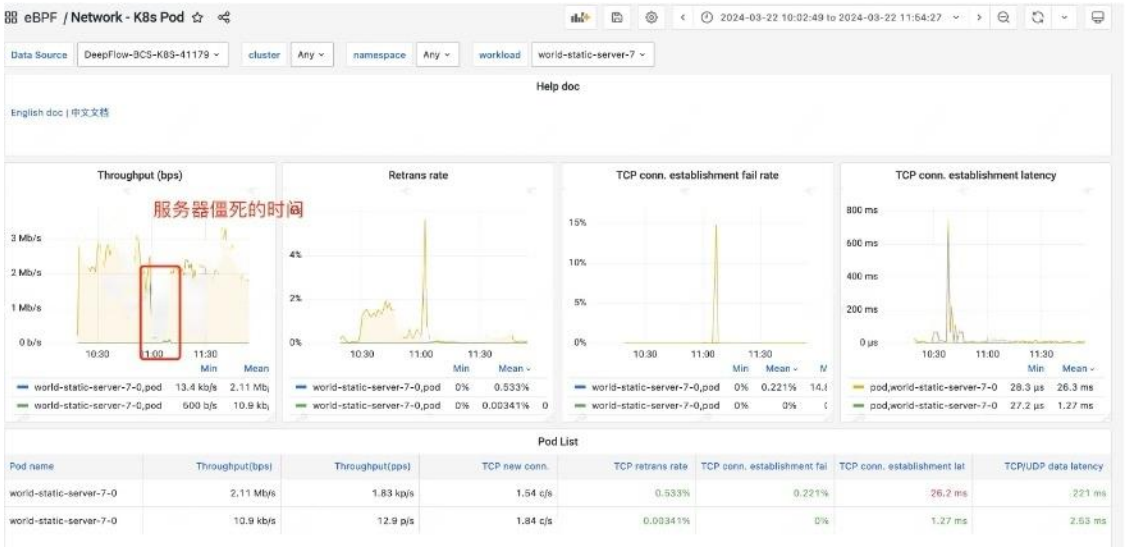
最终，我们确定问题的根因是由于 QQ 平台的周年庆活动导致系统过载，影响了我们的游戏登录鉴权服务。通过这种用户旅程的分析方法，我们能够快速定位并解决问题。

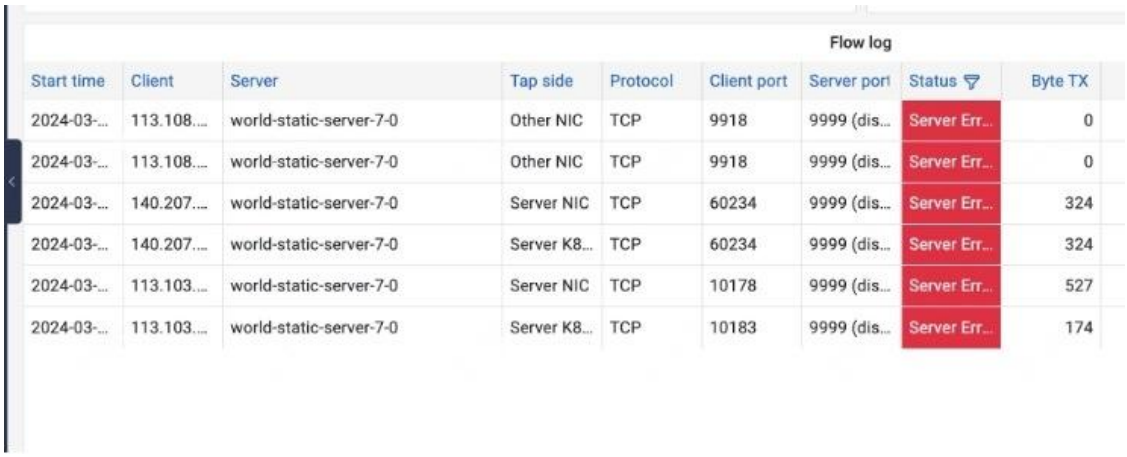
案例 4：使用 eBPF 可观测，快速发现服务器端程序问题

1.在某个副本中进行战斗的玩家突然集体卡住

确定服务器是否在运行中：正常运行

确定服务器出入流量是否正常：流量掉 0





- 从玩家到服务器的业务流量全部是 server Error 状态
- CLB 到服务器的健康检查流量全部是 success 状态
- 是不是可以说明，只有涉及到业务协议的请求包才会出问题？

Flow log

Start time	Client	Server	Tap side	Protocol	Client port	Server port	Status	Byte TX	By
2024-03-2...	9.166.19.65	world-static-server-7-0	Server NIC	TCP	37778	9999 (disti...	Success	228	
2024-03-2...	1.13.155.2...	world-static-server-7-0	Server NIC	TCP	36208	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server NIC	TCP	55580	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server K8s...	TCP	30772	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server K8s...	TCP	34109	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server NIC	TCP	24666	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server NIC	TCP	46574	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server K8s...	TCP	55580	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server NIC	TCP	30772	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server NIC	TCP	34109	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server K8s...	TCP	46574	9999 (disti...	Success	120	
2024-03-2...	114.222.1...	world-static-server-7-0	Server K8s...	TCP	24666	9999 (disti...	Success	120	
2024-03-2...	1.13.155.2...	world-static-server-7-0	Server K8s...	TCP	37884	9999 (disti...	Success	120	

进一步分析 Server 的错误日志，发现了一个关键的错误类型：Server half close flow。

错误类型分析

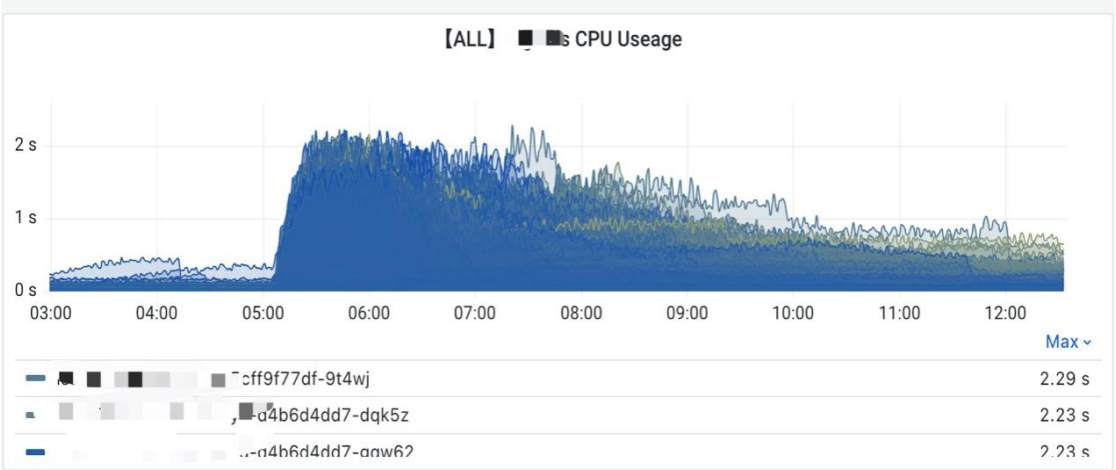
通过分析错误日志中的关键字，我们发现这是一个典型的 TCP 协议中的半连接异常。具体来说，这种情况发生在 TCP 连接的关闭过程中。

我们将错误码和分析结果反馈给开发团队，指出这是一个 TCP 协议处理中的问题。经过进一步排查，发现问题源自发行商在处理协议时的错误实现。

2.整个链路每个组件的 CPU 使用率过高

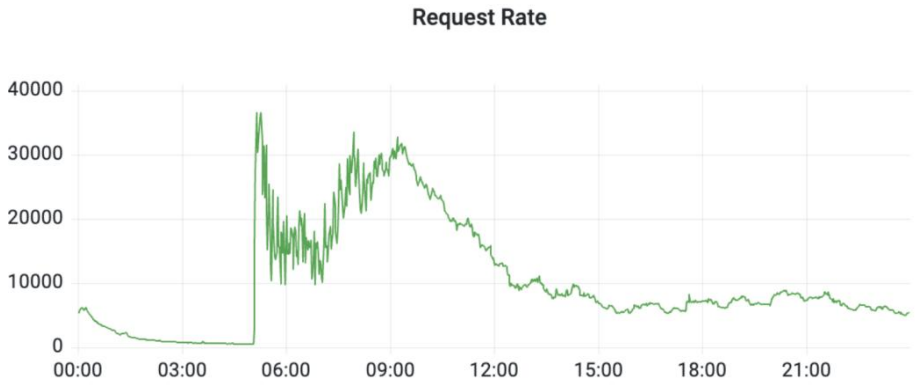
某一天，我们发现不仅仅是游戏登录模块，整个游戏链路，包括匹配模块和战斗模块等，都出现了 CPU 使用率突然增高的问题。以下是详细的分析和解决过

- 怀疑玩家数增长：相较同期无明显增长
- 怀疑服务器产生过多 gc：不符合整个链路均增长的表现



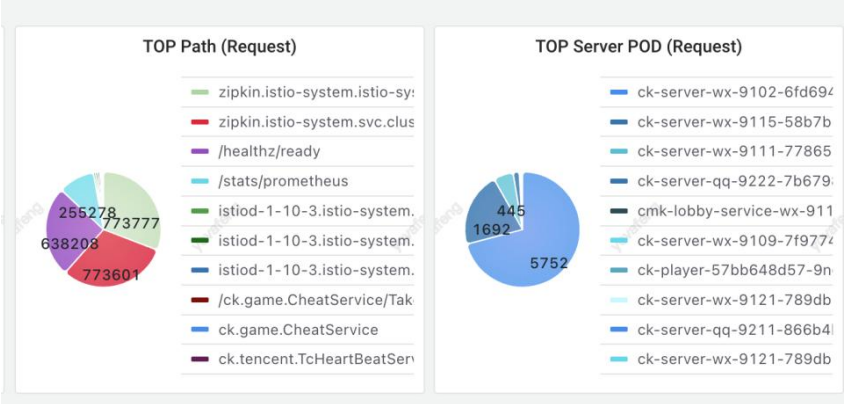
- 在玩家总数没有增长的情况下，是否有可能是单个玩家发起的请求数增多？

整个链路均增长，说明导致问题的原因与网络有关



- 通过 request rate 确认，确实存在过高的 qps

我们使用 eBPF 的 Top Server Pod 面板，找出请求数最高的前 10 个 Pod。



- 通过 top path 定位到 某 pod 的接口异常非常多

我们将问题反馈给研发团队，并核实日志，定位到问题的根因如下：

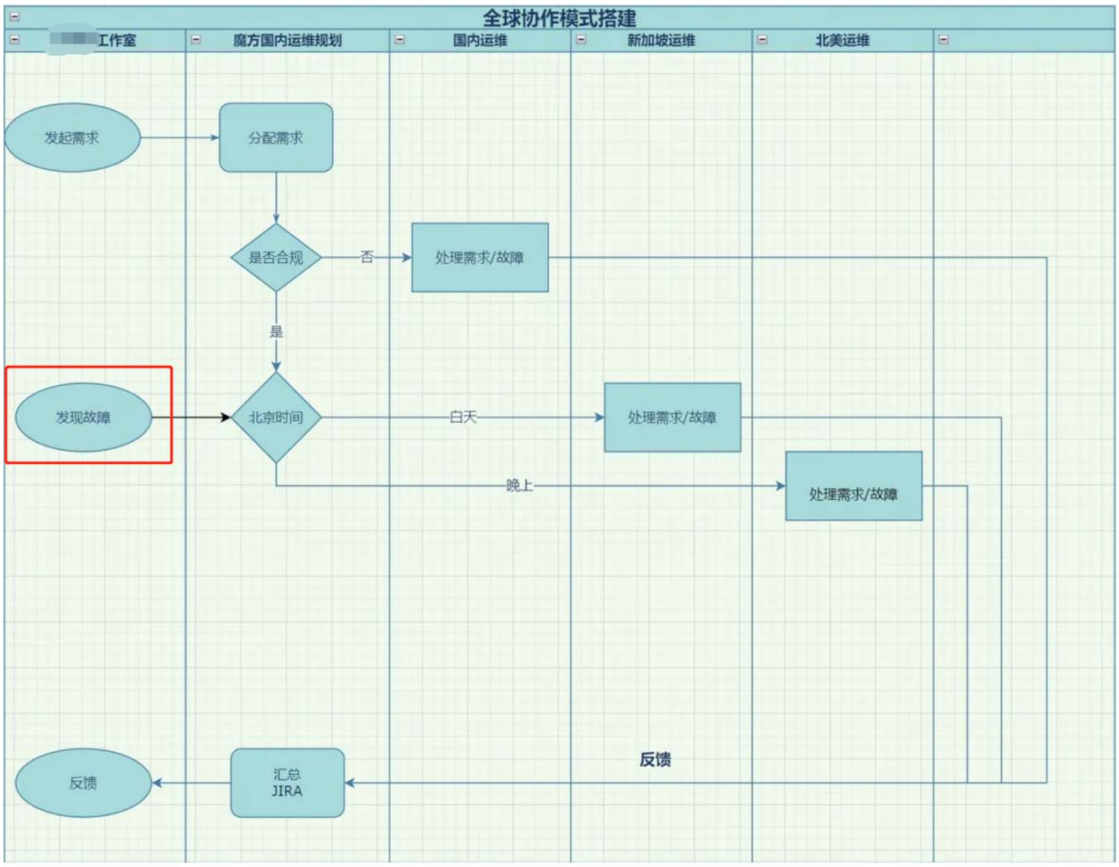
- 某个 SDK 的发布新版本时，未能正确兼容游戏版本。
- 由于不兼容，SDK 不断重发请求，导致整个链路的流量激增，最终引发 CPU 使用率飙升。

eBPF 能够在内核态进行高效的网络数据包过滤和监控，帮助我们分析协议层的问题。它类似于高级的抓包工具，能够替代 Wireshark 等传统工具，极大地提高了问题定位的效率。如果没有 eBPF，我们可能需要逐个检查每个服务和模块的流量，耗时且难以迅速定位问题。

事中：故障快速响应并处理

建立 ONCALL 模式，保证故障得到快速响应

全球化协作流程，保证业务能得到 7*24 小时全天候服务



在我们的 oncall 模式中，告警信息的管理和收敛是关键。以下是我们如何优化告警系统，减少信息洪流，并提高告警的有效性。

建立有效的分级告警机制，减少告警信息洪流

问题 每天收到 N 多告警，等于没告警 收到告警，该谁接收和处理？

如何解决告警风暴问题：

-区分告警等级，使用不同告警策略

-告警责任人自动识别，关注点分离 大多业务噪声降低 90%+

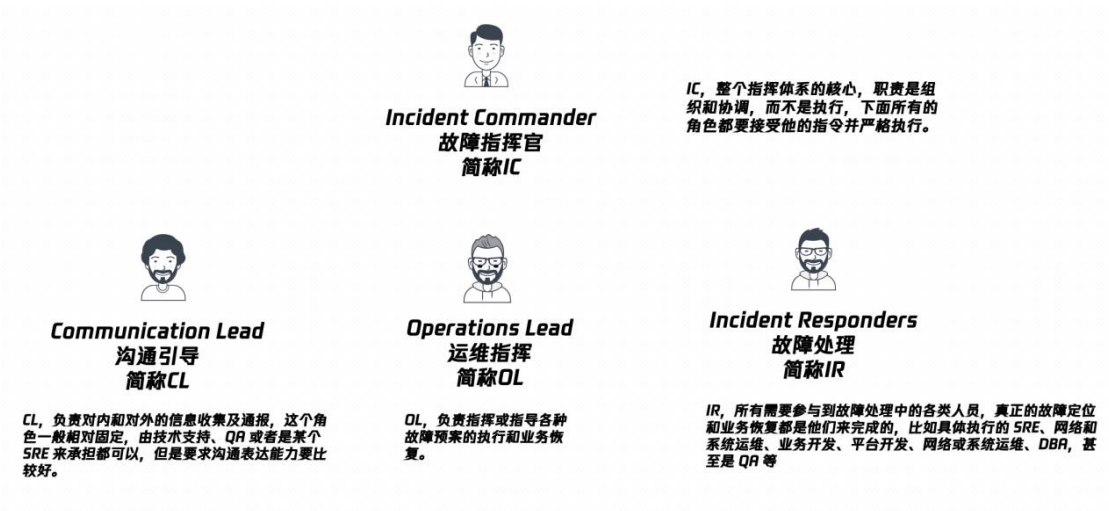
告警等级	判断依据	ONCALL	告警渠道	告警主要责任人
致命	生产环境为主：严重影响游戏用户体验，收入类等问题。比如大面积网络和组件类故障，关键模块机器重启，程序严重 BUG 类	SG（8：00-21：00）	企微致命告警群+电话告警+短信+邮件	基础设施：SRE
致命	生产环境为主：严重影响游戏用户体验，收入类等问题。比如大面积网络和组件类故障，关键模块机器重启，程序严重 BUG 类	NA（21:00-8:00）	企微致命告警群+电话告警+短信+邮件	程序类：对应模块开发
致命	生产环境为主：严重影响游戏用户体验，收入类等问题。比如大面积网络和组件类故障，关键模块机器重启，程序严重 BUG 类		企微致命告警群+电话告警+短信+邮件	公共数据类：SRE 和开发
预警	问题堆积后，可能会产生致命问题，比如生产环境容量，性能类告警	SG（8：00-21：00）	企微预警群+短信	基础设施：SRE
预警	问题堆积后，可能会产生致命问题，比如生产环境容量，性能类告警	NA（21:00-8:00）	企微预警群+短信	程序类：对应模块开发
预警	问题堆积后，可能会产生致命问题，比如生产环境容量，性能类告警		企微预警群+短信	公共数据类：SRE 和开发
提醒	信息事件为主，比如日志错误统计类，非生产环境告警	SG（8：00-21：00）	企微提醒群	基础设施：SRE
提醒	信息事件为主，比如日志错误统计类，非生产环境告警	NA（21:00-8:00）	企微提醒群	程序类：对应模块开发
提醒	信息事件为主，比如日志错误统计类，非生产环境告警		企微提醒群	公共数据类：SRE 和开发
过滤	无效信息，不告警			

- 告警等级分类：

- 致命告警：涉及生产环境的重大问题，如大面积网络故障、服务器重启、关键模块故障、严重程序 bug 等。
    - 预警类告警：短期内不会产生致命影响，但潜在风险较高的问题，如磁盘使用率超过 50% 或 60% 开始预警，Kubernetes 中 Pod 调度策略的预警等。
    - 提醒类告警：主要是一些错误日志，开发人员在排查问题时需要查看，但不影响游戏正常运行。
- **告警渠道优化：**
- 致命告警：通过全渠道通知，直到告警接收人确认。例如，主责任人未响应时，通知备份责任人，备份责任人未响应时，通知团队领导，逐级上报直至总监或 GM。
    - 预警类告警：在企业微信群中同步即可。
    - 提醒类告警：仅在需要时查看，不做紧急处理。
- **责任人区分：**
- 基础设施类告警：由 SRE 团队负责处理，如服务器重启、网络故障等。
    - 程序类告警：由对应的后台开发人员负责处理，提前预设告警名单，有针对性地通知相关人员。
    - 公共数据类告警：根据具体情况分配给相关团队或人员。
- **故障治愈小程序：**
- 在故障治愈小程序中，可以直接 @ 相关的 SRE 或开发人员，提高响应效率。

建立故障应急处理分工协助流程

Google 的故障指挥体系



谷歌的故障指挥体系的话，它是参考 1968 年这个美国森林大火的这样的一个体系，**属于腾讯游戏的故障指挥体系**

当一个问题被定性为故障，这时我们就要成立 War Room，如果是在办公时间，大家可以聚集到同一个会议室，或者同一块办公区域集中处理；

如果是非办公时间，可以是视频、电话或微信会议的方式，形成虚拟的 War Room。

但无论是真实还是虚拟的 War Room，根本目的是快速同步信息，高效协作。在特别时期，比如业务上线这样的关键节点，咱们做法一般是在一个会议室，运维和项目组坐在一起。在已经形成双十一文化的阿里，他们把这样的办公地点称为"光明顶"，各大高手齐聚于此，共同保障业务和系统稳定。

我们一般要求以团队为单位，每隔 10～15 分钟做一次反馈，反馈当前处理进展以及下一步行动。

如果中途有需要马上执行什么操作，也要事先通报，并且要求通报的内容包括对业务和系统的影响是什么。最后由 IC 决策后再执行，避免忙中出错。

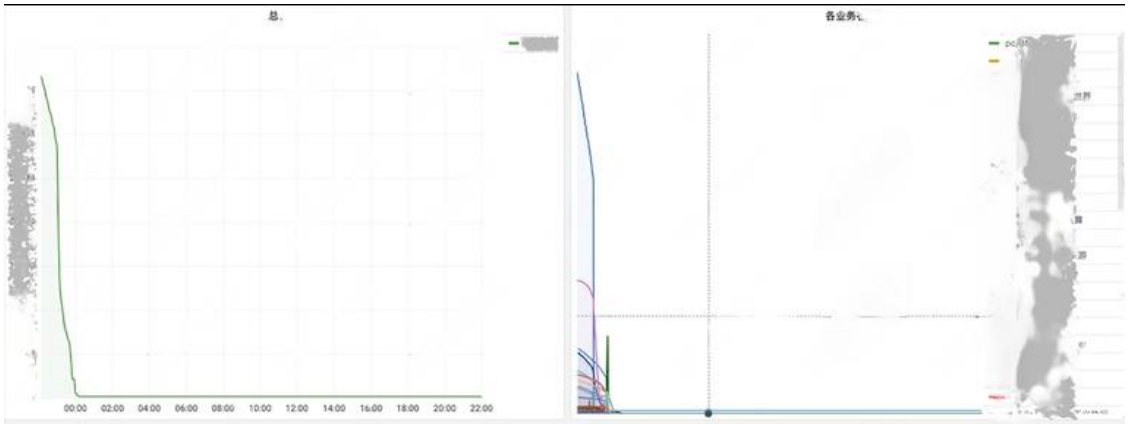
强调一点，没有进展也是进展，也要及时反馈。

很多团队和成员往往会抱怨，我专心定位问题还没结果呢，有什么好反馈的呢？

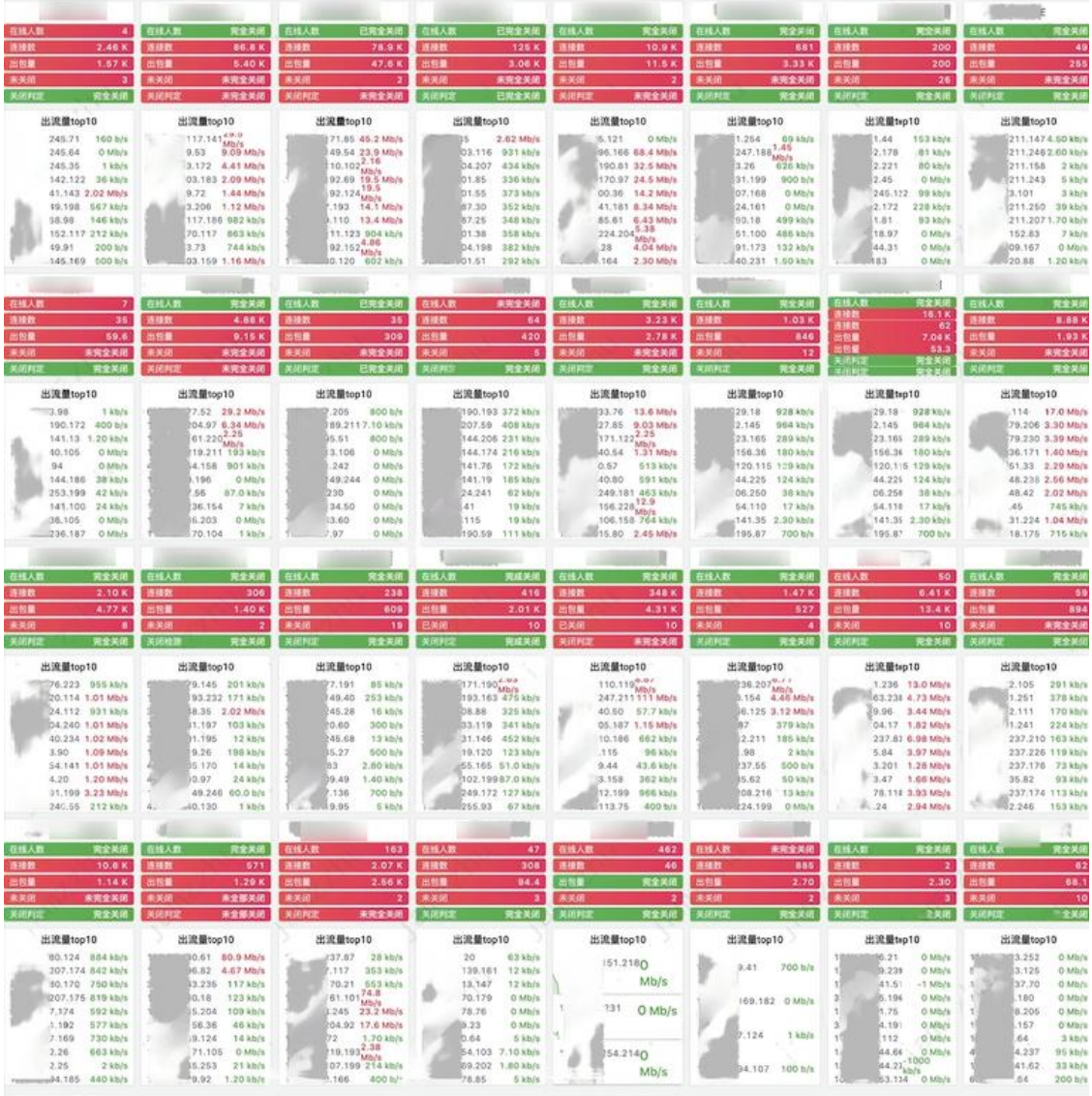
但是没有进展也是进展，IC 会跟 OL 以及团队主管决策是否要采取更有效果的措施，比如 10 分钟之内还没定位结果，可能就会选择做有损的降级服务，不能让故障持续蔓延，这个时候，反馈就显得尤为重要。

War Room 成员组成	War Room 成员组成	War Room 成员组成	War Room 成员组成	War Room 成员组成
角色	IC	CL	OL	IR

War Room 成员组成	War Room 成员组成	War Room 成员组成	War Room 成员组成	War Room 成员组成
负责岗位	总监	Leader/运营规划	运营规划	各系统开发/运维/SRE



重点业务VIP开启情况



所有业务全局指标

从实践经验来看，如果是大范围的故障，一般就是总监来承担 IC 职责，接下来他就可以从更高的层面组织和协调资源投入并有效协作。

这时运维回归到 OL 的职责上，负责组织和协调具体的执行恢复操作的动作。

利用蓝鲸故障自愈快速恢复业务故障

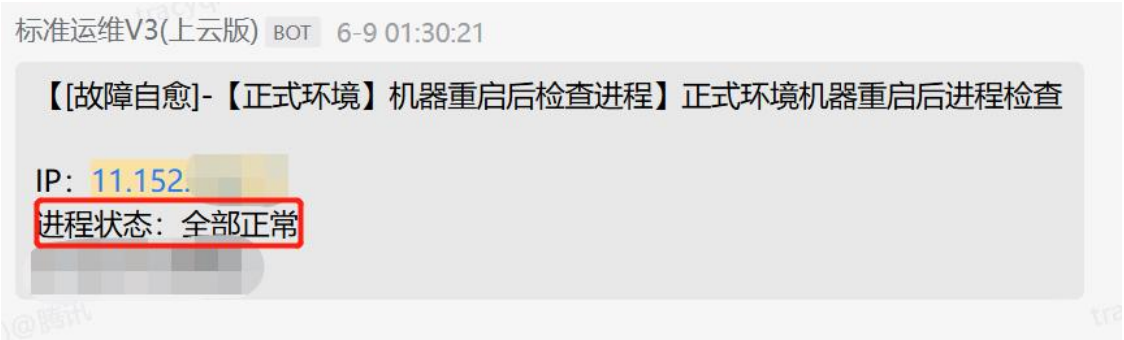
故障自愈是一个非常重要的特性，它可以使系统在出现故障时自动恢复，而无需人工干预。这是一个非常强大的功能，可以显著提高系统的可靠性和可用性，同时也可以减少运维人员的工作负担。

为了应对这些挑战，需要建立一个完整的故障管理流程，从故障发现到改进，特别强调故障治愈，优化 MTTR 时间，通过系统自动化减少人工干预。这些措施能够有效提升全球游戏运营的稳定性和安全性。

某业务使用故障自愈恢复业务进程

1 分钟自动处理，减少 MTTR 时间

ID	套餐名称	套餐类型	关联策略	触发次数(近 7 天)	最近更新人
#107001	【标准运维】【现网】屏蔽集群扩容告警策略	标准运维	1	10	
#99592	【标准运维】【现网】zonesvr通知在线人数	标准运维	1	–	
#90275	123	一键拉群	–	–	
#67588	正式环境致命告警ITSM建单	流程服务	198	–	
#69188	【标准运维】【现网】故障自愈拉起单主机进程	标准运维	2	2	
#69163	【标准运维】【预发布】故障自愈拉起单主机进程	标准运维	1	–	
#65697	【通知套餐】【运维】xwork捕获CVM运行异常通知	标准运维	1	–	
#63429	【快捷套餐】转移主机至空闲机模块	标准运维	–	–	
#63187	【快捷套餐】重启网管Agent	标准运维	–	–	
#63186	【TCM】重启idipsvr和midas_ntf_svr（指定IP地址...	标准运维	–	–	



利用蓝鲸故障自愈快速扩缩容

在复杂的游戏运营场景中，尤其是全球游戏的运营中，快速响应和资源管理至关重要。我们通过故障治愈系统，实现了自动化的扩缩容，确保游戏在不同区域的资源使用最优。

场景背景

游戏上线后，不同区域在不同时间点的资源需求会有所不同。例如，一个区域在某个时间点可能资源富余，而另一个区域可能需要扩容以应对玩家增长。为了高效管理这些资源，我们引入了 AI OPS 能力，通过智能监控和预测模型，实现自动化的扩缩容。

处理套餐说明： 通过告警策略可以触发处理套餐，处理套餐可以与周边系统打通完成复杂的功能，甚至是达到自愈的目的。

ID	套餐名称	套餐类型	关联策略	触发次数(近 7 天)
#16410	故障自愈-全球战斗服自动扩缩容	标准运维	1	--
#9305	【不删档】【非DS Error】gdb coredu…	标准运维	1	--
#6286	【不删档】【非DS Error】corefile日志…	标准运维	2	--
#6285	【不删档】【tbuspp】corefile日志内容…	标准运维	2	--
#3992	【不删档】【DS Error】corefile日志内…	标准运维	2	1
#5806	【不删档】【自愈套餐】发送 CPU 使…	标准运维	3	--
#4204	【测试环境】corefile通知	标准运维	3	--
#3923	【CBT2】corefile日志内容通知	标准运维	--	--

每天定时跑检查结果

记录扩缩容，节省成本直接展示

自动扩缩容实际执行结果

时间	执行状态	大区	实际扩缩容数量	执行时常	缩减成本/月/天
2024-05-10 15:46:00	成功	弗吉尼亚	-2	3 hour	¥-2
2024-05-10 15:47:00	成功	弗吉尼亚	-2	3 hour	¥-2
2024-05-10 15:48:00	成功	弗吉尼亚	-2	3 hour	¥-2
2024-05-10 15:49:00	成功	弗吉尼亚	-10	3 hour	¥-1

事后：复盘总结，从故障中学习并改进

复盘是我们在运维和故障处理中不断改进和提升的重要环节。

简单重复 10000 次，不如有效复盘 1 次，把经验转化为能力，将失败转化为"有意义的失败"

故障复盘底层逻辑

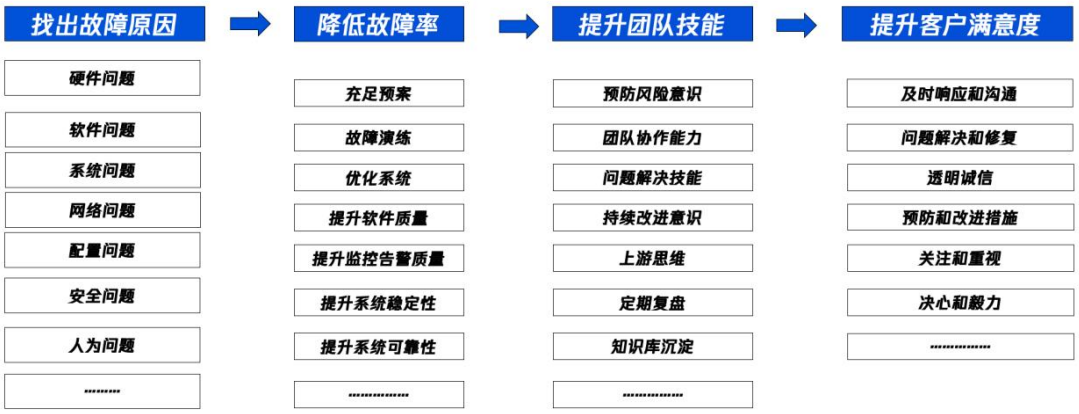
- 故障是无法完全避免的

- 杜绝重复犯错，包容失败
- 根因分析要全面**在故障复盘中，发生故障后分析原因，总结经验教训，并制定改进措施，以防止类似故障再次发生。**

一个经典的复盘实践案例来自航空业。几十年前，美国航空事故频发，死亡率高达25%。通过引入"黑匣子"记录飞行数据和语音通信，专业团队对事故进行详细复盘，找出问题根源并改进。如今，航空业的安全性大幅提升，每年因空难死亡的人数降至3-400人，飞行已成为地球上最安全的交通工具之一。举个例子，如果有人需要每天坐一次飞机，那么大概3200多年才会遇上一次空难。

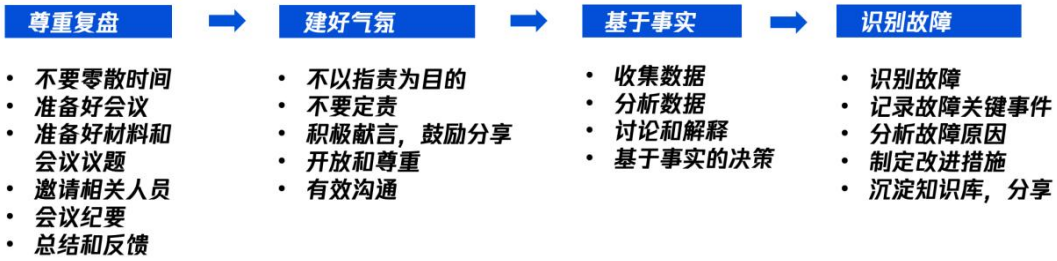
航空业能想到这么细致的原因，不是因为他们想象力丰富，而是能够及时把过去曾经发生过的错误，最大程度转化为未来可以小心和避免的方法，这类行为其实就是我们今天要重点讨论的故障复盘。

故障复盘的目标



复盘的主要目标是通过系统性的分析和总结，找出故障原因，降低故障率，提高系统的稳定性和可靠性。

故障复盘关键要素



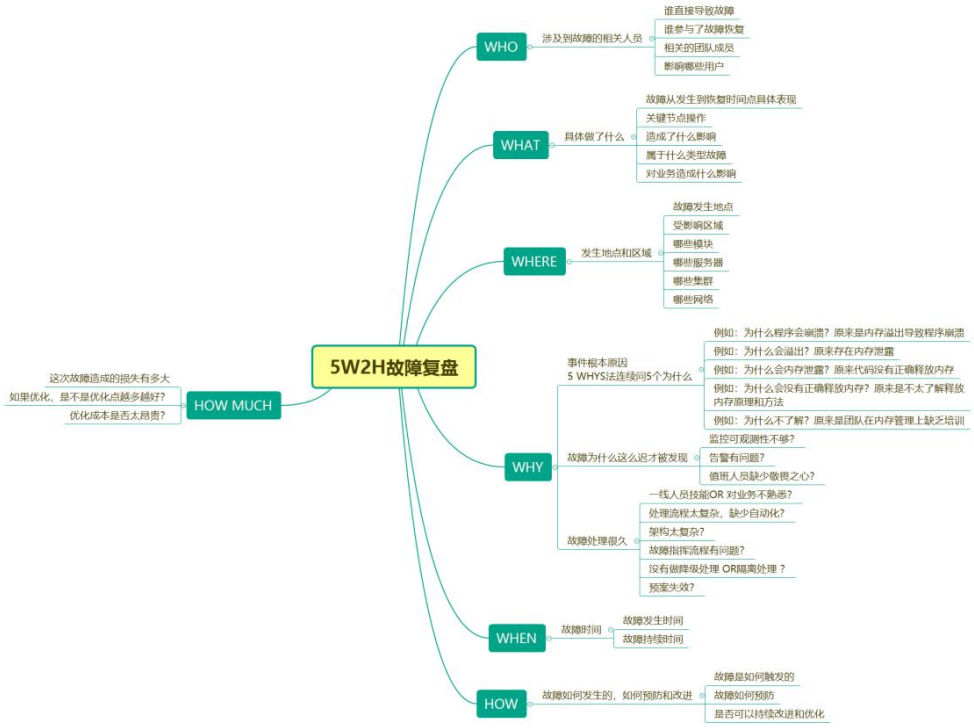
复盘方法

我们采用"5W1H"方法进行复盘：

- Who（谁）：涉及哪些人？
- What（什么）：发生了什么事情？
- When（何时）：事情发生在什么时候？
- Where（何地）：事情发生在什么地方？
- Why（为什么）：为什么会发生？

在实际操作中，我们会：

- 记录故障数据：详细记录故障发生时的所有数据和日志。
- 召开复盘会议：邀请相关人员参加，按照"5W1H"方法进行讨论。
- 制定改进措施：根据复盘结果，制定和实施改进措施。
- 跟踪改进效果：持续跟踪改进措施的效果，确保问题得到解决。



复盘模板参考

某故障复盘模板

故障概述

1. 主题：〔简述什么时间点发生了什么动作，做了什么操作，造成什么影响〕
2. 业务：xx业务
3. 时间：xx时间-xx时间
4. 责任归属：xx
5. 事故影响：xx

故障原因

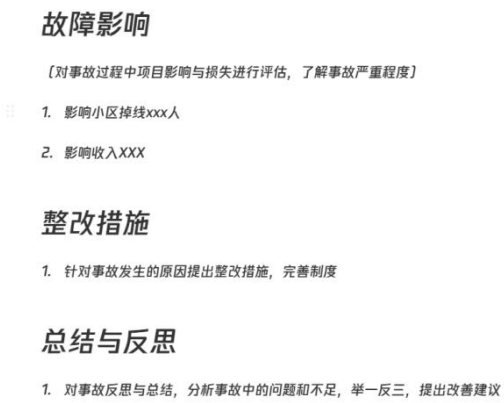
〔分析故障的直接原因和根本原因，包括人为因素、权限因素等，有针对性得预防类似事故再次发生〕

1. 平台权限控制xxx问题
2. 安全操作流程执行不严格
3. xxx

事件回顾-时间线描述

〔详细描述事件从需求产生、发生事故、处理过程、故障恢复的整个流程〕

1. xxx时间点，收到XX告警
2. XXX时间点，收到XXX告警
3. XXX时间点，运维开始处理故障
4. XXX时间点，确认是XX模块故障
5. xxx时间点，运维重启该模块进程，故障恢复



（三）总结及展望

总结

最后，我们回顾下今天主要分享的内容。全球化业务所面临的挑战主要包括网络环境的复杂性，指标和维度的增多，文化的多样性，多云部署，以及法规的多元化。这些问题也给全球业务的故障管理带来了诸多挑战，如系统的复杂性，数据渠道的碎片化，多维度指标，以及更严格的安全合规要求。

为了解决这些问题和挑战，我们提出了四步法：分层指标，挑选指标，采集指标和联动指标。通过可观测性，我们能够展示关键路径和关键网络质量大屏，以便于发现和定位问题。我们还分享了一些典型案例，通过关联检查，维度下钻，以及 eBPF 等技术手段来发现和定位问题。

在故障快速响应和处理方面，我们采用了 Oncall 机制，在全球三地协作中，根据告警时段和告警严重等级进行分类处理。我们建立了故障知会体系，一旦出现故障，我们会建立 War Room 作战室，多人协同快速同步和处理故障，以尽量减少影响范围。

最后，我们需要养成故障复盘的好习惯，明确故障复盘的目标和目的，使用 5W1H 复盘法，以提高复盘的效果。

展望

随着人工智能技术的不断发展，如何通过 AI 模拟人类思维，通过机器学习，深度学习，大模型等能力来理解系统的复杂性，从而预测故障，以及快速发现和定位故障，处理故障将是一个非常有挑战性，也非常有价值的方向。

目前腾讯游戏结合蓝鲸平台 AIOps 能力，已经初步落地了资源智能扩缩容，业务全球化智能选择节点，异常检测，故障维度推荐等场景。另外，我们也在积极尝试利用 eBPF 全链路关联数据的特点进行根因定位。相信技术的发展，人工智能在故障治理场景上落地场景将会出现一个百花齐放的局面，期待未来能与大家有更多的分享。

5.3.4 XX 银行应急管理一体化平台建设实践

SRE Elite 精选原因

XX 银行是中国乃至全球规模排名前列的商业银行，业务众多，客户群体遍布全球，且适逢整体 IT 架构升级，数字化转型深入，技术挑战巨大。在这种背景下，XX 银行构建了符合金融行业强监管特性的三个一体化的应急管理平台：通过"一体化技术平台"实现了底层能力平台 PaaS 化，满足各种底层操作原子化包装的需求；通过"管理操作一体"，实现应急管理思想和自动化操作的同步；通过"数据融合一体化"，实现应急决策所需配置数据、执行数据、性能数据、变更数据的统一管理和展示 。通过以上三个一体化，降低了业务稳定性的风险，值得广大金融行业参考。

(一) 背景及设计原则

xx 银行是中国最大的商业银行之一，在全球范围内运营，提供包括存款、贷款、信用卡、财富管理、外汇交易等多样金融服务，客户群体遍布各个经济领域。在信息化投入方面，XX 银行一直走在国内银行业的最前列，最早进行核心银行系统升级，最早建立多数据中心，最早推广云计算技术，提升业务的弹性和响应速度等。通过这些信息化的投入，XX 银行提高了自身的运营效率，也在客户服务、风险管理、市场反应速度等方面取得了显著的进展，保持了行业领先地位。

随着 XX 银行信息化建设和数字化转型深入推进，IT 部门数字化系统规模不断扩大，各类 IT 设备数量呈指数级增长，信息网络和系统架构复杂度不断提高，运行风险持续增加。为提高业务系统可用性和持续性，公司建设了应急管理一体化平台，支撑业务运维及应急响应体系高效运转。

在此之前，XX 银行存在着大量业务系统应急处置以线下维护为主的特点，并且各系统应急处置组织无系统固化，事件应急预案管理不完善，对应急处置实操性不强；缺少常态化的培训和演练机制，业务支撑人员应急技能掌握不足，导致在应急处置时过程协调复杂、难度大，应急处置时间长，故障易上升到高程度的事件级别。故障事件处置过程没有

形成知识经验，无法及时查找应急操作指引，事后复盘没有彻底执行及改善。基于此，建立线上、流程化的应急管理响应一体化平台，实现应急事件的全生命周期管理，为内部整体信息系统稳定运行提供基础保障。

（二）解决方案详情及案例

基于上述背景，数据中心相关领导一直在思考如何建设一个应急管理一体化平台，通过提升 MTBF（平均无故障时间）时间，降低 MTTR（平均恢复时间）时间，就可以有效地管理应急全流程。这里我们总结了以下"1-2-3"设计原则：

图.应急整体业务流

一个目标：快速止血，第一时间恢复生产。

两大场景：覆盖"事件应急"和"灾备应急"两类场景。

三个一体："技术平台一体"，实现底层能力平台 PaaS 化；"管理操作一体"，实现应急管理思想和自动化操作的同步；"数据融合一体化"，实现应急决策所需配置数据、执行数据、性能数据、变更数据的统一管理和展示。

一个目标，快速止血，保障业务稳定性

应急管理的总体原则：

（一）第一时间恢复生产：生产事件发生后，事件发生单位应第一时间确认影响，根据应急预案组织开展故障恢复。

（二）统一领导、分级负责：各应急小组接受应急领导小组统一指挥，按照事件不同影响程度，分别启动应急组织架构。

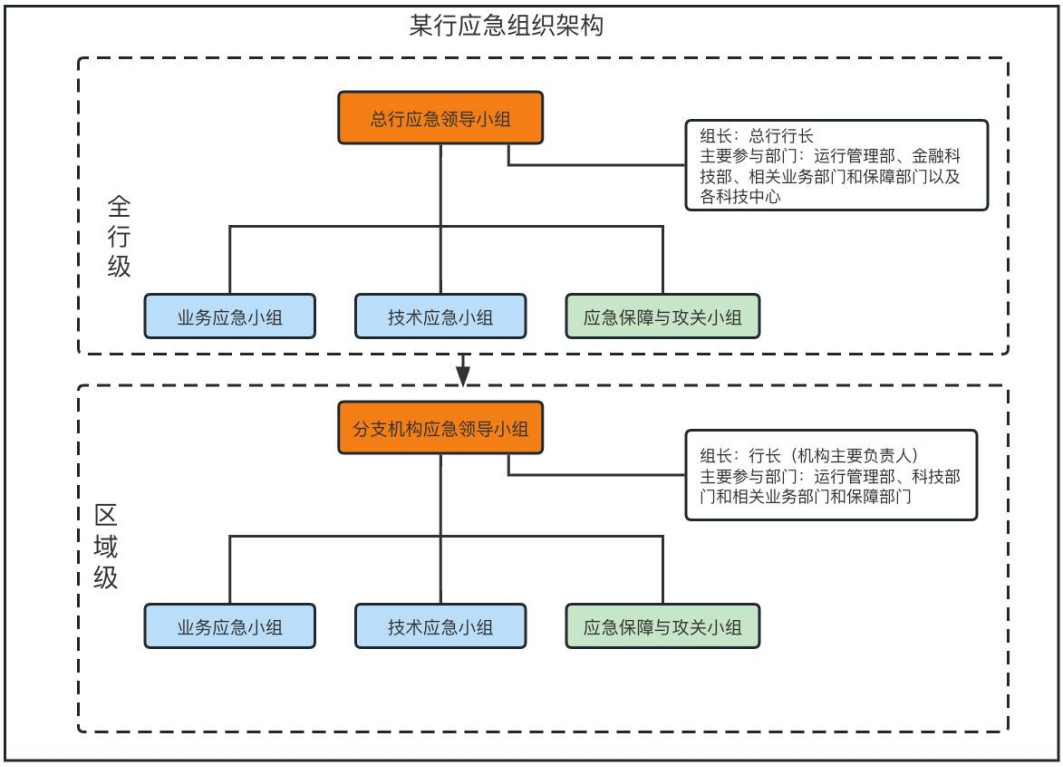
（三）预防为主、演练结合。

在应急事件产生时，可以按照不同维度设置不同群组管理应急人员，通过多种方式通知到应急人员，一、二线协同应急，该应急组织相关角色会迅速返回数据中心，通过移动端签到等方式了解人员到位情况，针对业务故障，采取暂不问责，暂不问询根因的原则，进行业务恢复快速处置

利用故障根因和故障处置的应急专家库匹配，根据不同故障类型，推导决策提供用户故障处置建议，常见的故障处置类型如下：

- 点状故障:某容器问题、某模板问题、某宿主机问题、某服务问题，则可通过容器重启、宿主机隔离、服务管理平台禁用服务
- 面状故障:园区故障、K8s 集群故障、Ceph 故障，则通过 F5/DNS 切换处理;
- 变更故障：版本升级问题，则直接提供 PaaS 平台回退接口;
- 容量不足：CPU/内存不足/ DSF（分布式服务框架）日志中存在连接池满，则提供 PaaS 扩容接口;

根据《商业银行业务连续性监管指引》外部要求，XX 银行制定了行内应急组织架构。应急处置组织架构分为全行级级应急组织架构和区域级应急组织架构。



应急组织各角色职责：

- 应急领导小组：负责制定应急备份与恢复的目标，负责应急决策，负责应急过程的总体指挥调度和组织协调，负责督导应急处置实施；
- 技术应急小组：包括应用、系统、网络、硬件、安全等专业，负责技术应急处置，负责应急过程中提供决策依据和技术支持，负责做好相关专业之间应急处理工作的沟通协调，负责应急过程处理步骤的记录，负责汇报应急处理进展；
- 业务应急小组：负责业务应急处置实施工作，负责做好相关专业之间应急工作的沟通协调，负责组织账务修复、验证和确认等；
- 应急保障与公关小组：负责与其他小组的沟通和协调，负责提供应急所需的人力、物力、财力等资源保障工作，负责信息对外报告、披露和解释、媒体公关、安全保障、秩序维护、法律咨询等。

两大场景，覆盖“事件应急”和“灾备应急”两类场景

应急管理一体化平台覆盖事件应急和灾备各自有其独特的作用和流程，二者紧密相连，共同构筑起企业应对突发事件和灾难的完整防线。事件应急通常是灾备应急中提取的部分处置步骤，灾备应急也同样是事件应急的最后一道防线。事件应急旨在快速响应和处理突发的信息系统事件，如网络攻击、数据泄露、系统故障等，以尽量减少对业务的影响，场

景多样，因此需要不断完善运营，丰富其预案下的不同场景，以便在故障事件发生时，能够快速检索推荐处置策略，实现业务快恢。

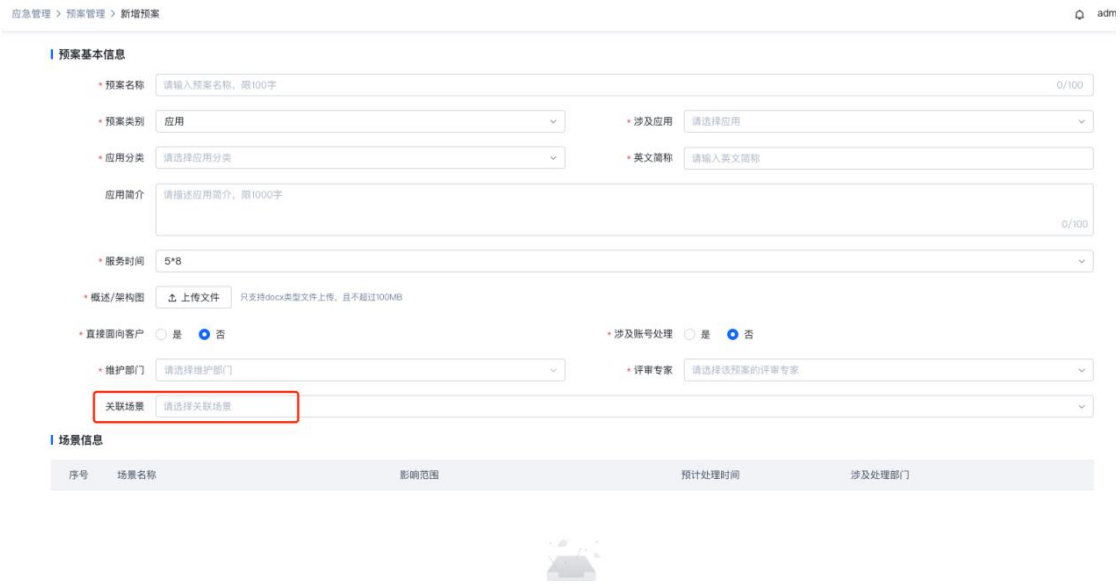


图.应急预案

应急场景是对事件的模拟，旨在通过具体的情景测试应急预案的有效性和可操作性。应急场景可以帮助识别预案中的漏洞和不足，提供实践经验。应急场景的演练结果可以反馈到应急预案中，使预案不断优化和更新，确保其在实际事件中更加有效。

图.应急场景

经过多年的应急演练和生产应急实践，沉淀了面向应用运维和基础运维的各类应急场景 3000+，如信用卡应用系统相关的应急场景：

序号	预案名称	应用分类	场景名称
1	信用卡产品应急预案	A 级	信用卡产品调整：服务限流与服务禁用
2	信用卡产品应急预案	A 级	信用卡缓存异常
3	信用卡产品应急预案	A 级	信用卡产品联机交易异常调整日志级别
4	信用卡产品应急预案	A 级	信用卡产品日切前批量处理未完成作业
5	信用卡产品应急预案	A 级	信用卡产品数据库服务器异常
6	信用卡产品应急预案	A 级	信用卡产品数据库开关切换
7	信用卡产品应急预案	A 级	信用卡产品调用服务开放切换
8	信用卡产品应急预案	A 级	信用卡产品联机信用卡某服务大范围报错
9	信用卡产品应急预案	A 级	信用卡产品异常切换连接集群
10	信用卡产品应急预案	A 级	Agent 进程异常
11	信用卡产品应急预案	A 级	重新信用卡产品应用的容器
12	信用卡产品应急预案	A 级	发卡应用修改信用卡产品数据库配置
13	信用卡产品应急预案	A 级	缓存异常
14	信用卡产品应急预案	A 级	批量控制台手工提交作业
15	信用卡产品应急预案	A 级	……

面向各类基础资源对象的应急场景，以 MySQL 举例，如下：

序号	MYSQL 数据库应急场景名称
1	自动清理作业异常
2	MySQL5.5 数据库主备不同步
3	数据库文件损坏或丢失
4	MySQL 数据库目录文件系统使用率高
5	Prometheus 监控状态异常
6	MySQL5.5 主备切换
7	存储监控数据过大导致磁盘使用率高
8	MySQL5.7 米同步备库切换
9	MVSQL5.7 异北备库计划内切换
10	MySQL 容器 CPU、内存扩容

序号	MYSQL 数据库应急场景名称
11	MySQL 宿主机配置 SRIOV
12	MySQL 使用 CSI 存储扩容
13	分布式中间件进程状态异常
14	MySQL 数据库基于时间点恢复
15	MySQL 容器重建

灾备应急主要关注在遭遇重大灾难（如自然灾害、严重网络攻击、重大硬件故障等）后，如何恢复和重建信息系统。因此，在创建应急预案和灾备预案时，略有差异。应急预案关联各种场景，灾备切换关注整个系统或业务的切换过程。

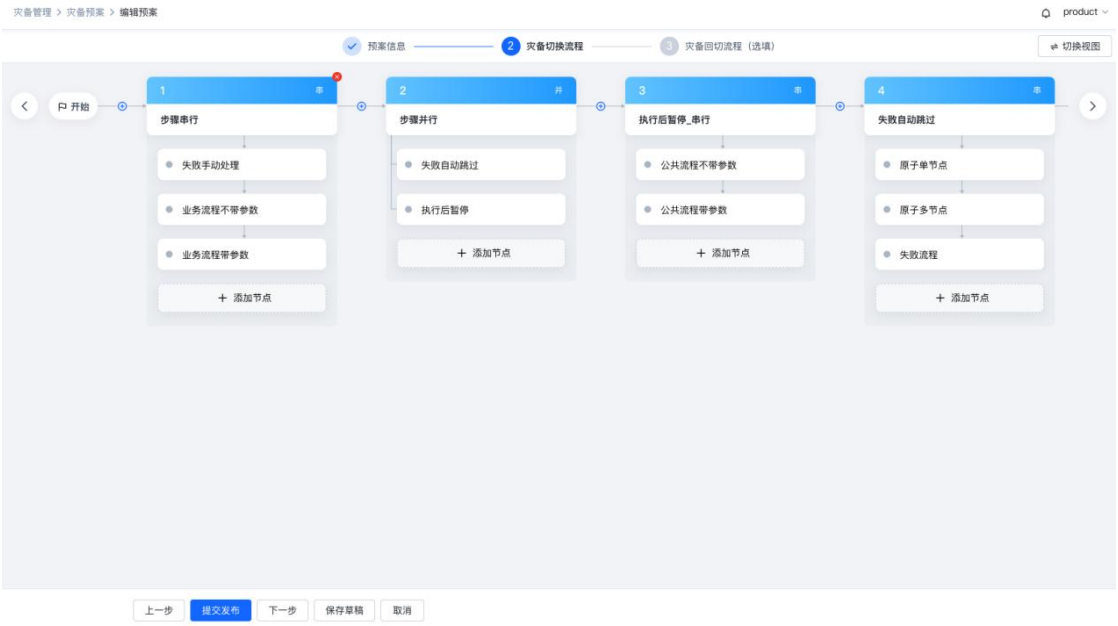
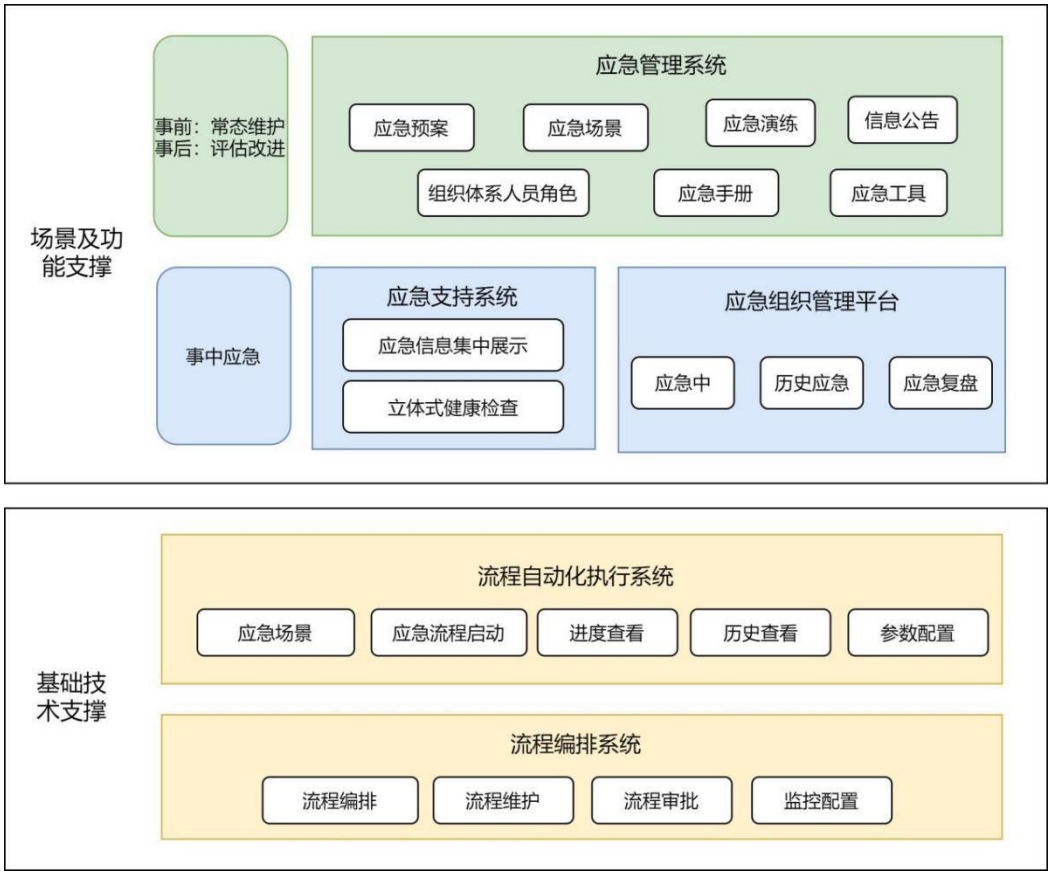


图.灾备预案

三个一体：技术一体、管理操作一体、数据一体

- "技术平台一体"实现底层能力平台 PaaS 化；

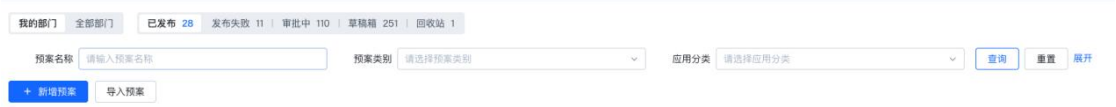
为有效管理繁多的应急场景及应急预案、快速组织开展各类演练、高效实现不同角色、不同专业人员的线上协同，该行研发了一整套支撑应急的工具平台，用于支撑日常事件应急管理实施工作的开展。可以看出，整个平台的底层基础技术支撑实现多种能力的融合和统一调用，包含流程编排、自动化任务、监控配置等。



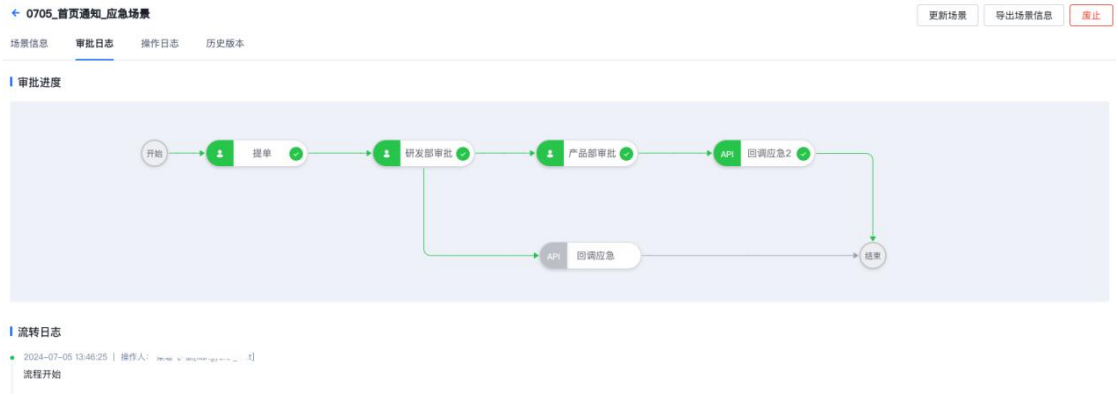
应急管理系统包括以下几个核心组件：

- 流程编排引擎，提供流程编排能力
- 流程执行系统，提供对流程执行的启动、控制、审计等能力
- 事前应急管理，提供应急预案、应急场景的定义，并管理日常应急演练工作
- 事中应急管理，提供应急启动后的组织协同、观测平面信息展示、应急故障分析决策和应急处置、应急验证和应急复盘能力，也包括应急事件的历史回溯、应急运营分析能力。
- "管理操作一体"，实现应急管理和自动化操作的一体化管理；

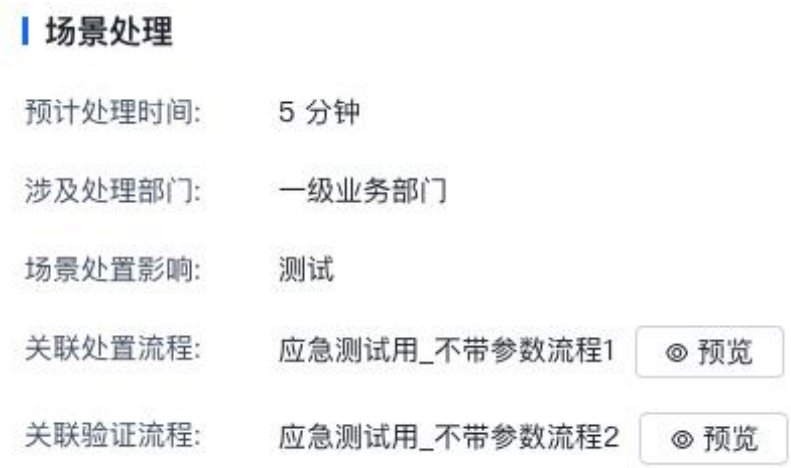
为了解决各系统应急处置组织无系统固化，事件应急预案管理不完善，对应急处置实操性不强的问题，在场景层建设了应急管理系统。通过不同 tab 状态页，管理不同组织范围下不同状态的应急预案、场景和演练计划等。



针对应急预案/场景的发布，应急演练计划的申请制定相应的审批流程，审批完成后即可生效。



针对应急场景预先制定好处置和业务验证的标准运维流程，与场景进行关联即可



灾备切换由于复杂性，需要制定好标准运维流程，并按照业务/系统切换和回切的步骤和子节点逻辑进行编排

- "数据融合一体化"，实现应急决策所需配置数据、执行数据、性能数据、变更数据的统一管理和展示。

在事中应急环节，需要一个统一的应急信息展示平台，展示以下信息：

– 故障业务的可观测链路数据（向上的故障影响分析和向下的故障溯源分析）

– 近期的变更单据数据

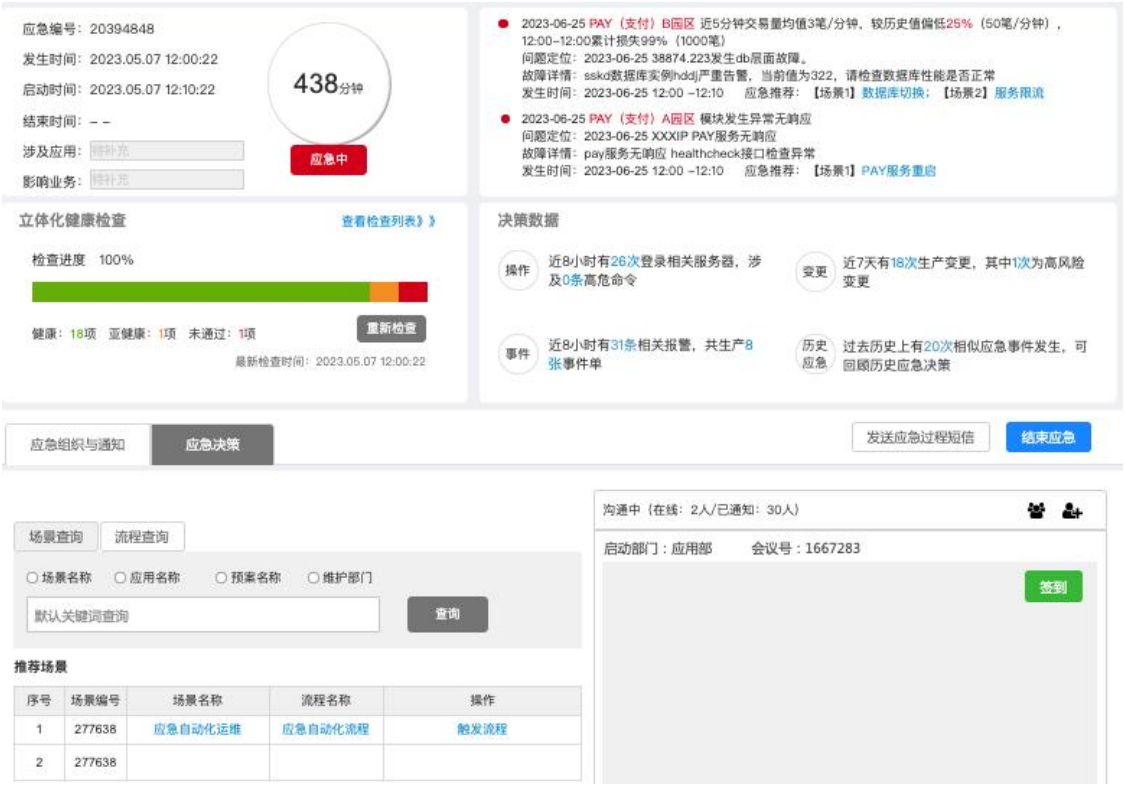
– 历史故障相似事件数据

– 健康检查数据

– 故障诊断决策分析数据

– 基于知识库的故障根因推荐和处置建议

– 应急小组在线协同信息



通过应急信息聚合，以便于应急组织能够更快的结合各类型数据进行处置动作的决策。

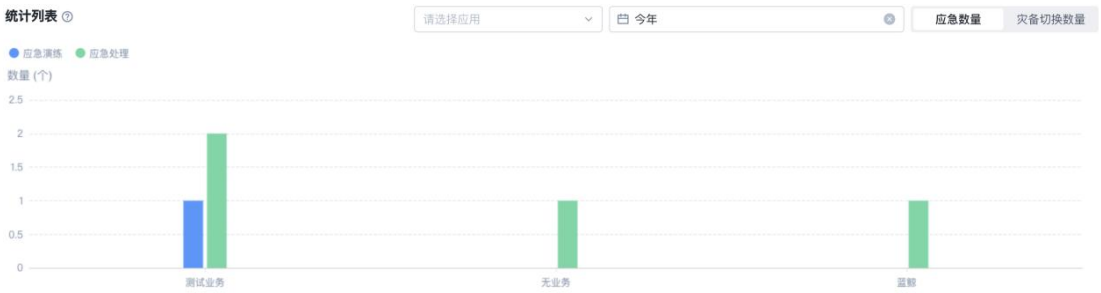
（三）总结及展望

对总结

应急平台记录应急演练和应急处置的任务记录，可详细的查看每个任务的执行结果与详情。

围绕应急场景进行的演练和处置都可以按照场景要素中记录的所属业务进行应急演练和应急处置的运营统计。

通过图表，可以看出哪些应用组织相关应急演练活动较多，代表着该业务应急人员的应急能力掌握程度越高；哪些应用应急处置的次数越多，说明该业务系统稳定性较差，且事后复盘没有进行彻底。



畅想展望

展望未来，在大模型和 AIOps 人工智能的能力加持下，真正实现根因快速定位，组织的应急能力能够更加直观地通过运营数据计算得出，从而进一步提升不同部门和专业间的协作和配合能力，以应对更复杂的问题和事件，使整个数据中心的应急能力更上一层楼，为业务提供更好的保驾护航能力。

5.3.5 美图故障管理体系搭建实践

SRE Elite 精选原因

美图在这个案例的分享中，展示了非常完整的 SRE 体系及故障管理体系，以故障生命周期管理为核心，引入了由人员、流程、技术和愿景构成的"PPTV 框架"；并强调数据驱动的决策，倡导定期复盘和持续改进，通过构建稳定性运营平台，实现对故障事前、事中及事后的全方位管理，全面且扎实，很值得大家进行研读。

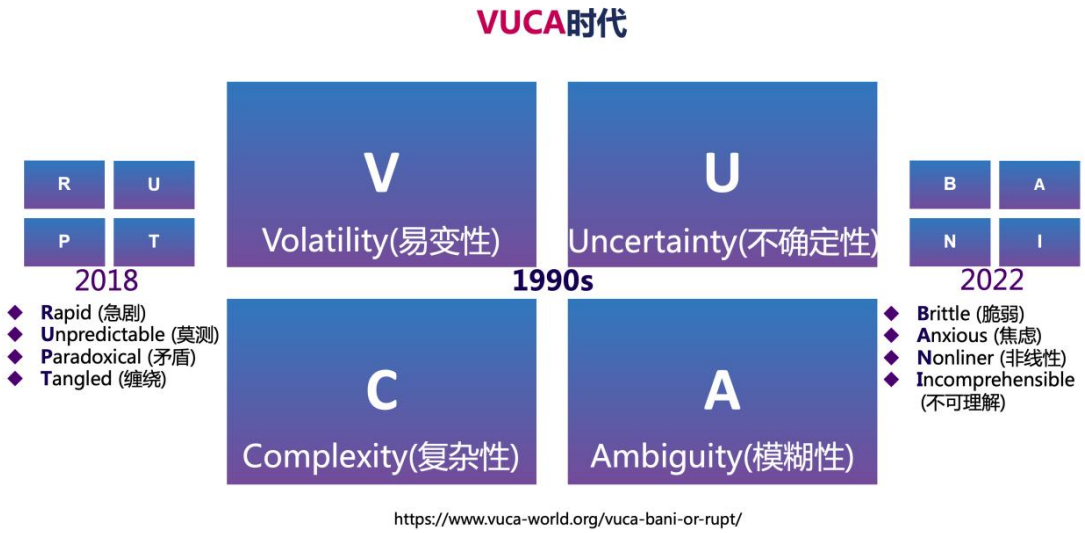
(一)背景及设计原则

案例说明：

本文由线下 MeetUp 分享的材料整理而来，因此在行文结构上跟常规的文章组织形式会有些差异。本文开篇先介绍了外部大环境的 VUCA(BANI)特性，作为引子，进而落脚到 SRE 的岗位来讲解 SRE 的岗位职责以及所面临的挑战。而后，针对「稳定性保障」的部分展开讲解，最终聚焦在「故障管理」相关的内容上。为了保证完整性，引子的部分虽略有删减，但整体还是保留的，各位读者在阅读的时候，按需索取即可。

此外，因为「故障管理 | 应急响应」在稳定性保障的工作中，类似于一个用于检验组织稳定性建设水平的考场，在上考场之前正常都是需要做大量的准备工作的，针对开展这些准备工作的框架和方法，在分享中也做了扩展、归纳，为了行文方便，对内容的先后顺序做了调整，将文章整体结构调整为总分、层层递进的形式。

引子：VUCA 时代 与 稳定性保障



VUCA 这个词起源于 20 世纪 90 年代的美国军方，指的是在冷战结束后出现的多边世界，其特征比以往任何时候都更加「复杂」以及「不确定」；因为这个特征是具有比较强的「普遍性的」，后来这个词的应用范围就越来越广泛。VUCA 中的几个特点在我们的 IT 环境、在稳定性保障过程中也无处不在。

甚至，我们可以说:"我们在做的稳定性保障工作，其实就是要和 VUCA 做对抗"。即在一个复杂和不确定的环境下，如何去追求一个确定性、稳定性的结果——保障服务安全稳定地运行。

那我们具体需要怎么做呢？

0，首先在心态上要保持开放和接纳，接受现实的 VUCA 特性；

1，理解和运用稳定性保障工作中，可能涉及的各种「因素」和「规律」；

2，为可能发生的各种情况尽可能地做好准备和预案；

3，做好稳定性的规划和设计，构建一个符合企业实际情况的整体框架，并持续更新迭代；

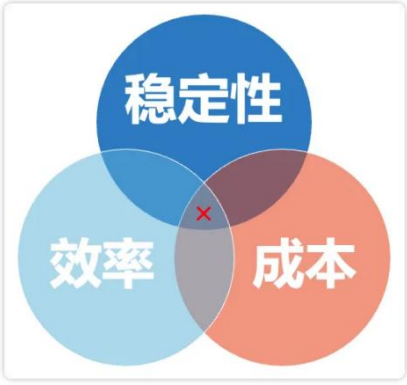
SRE 的岗位职责 与 面临的挑战

我们在公司内部是这么来定义 SRE 岗位的核心职责的，即：

- 保障线上服务的稳定性
- 建设工具、平台、基础设施来提升效率
- 用技术手段来控制、优化服务的运行成本

归结起来，其实就是几个关键字：稳定性、效率(包括职能支撑)、成本。其中「稳定性」是一个基础，也是 SRE 中的这个 R 的内涵。

当然，这三个职责也不仅仅是 SRE 或者运维同学需要承担的，诸多其他的岗位也有类似的职责目标。

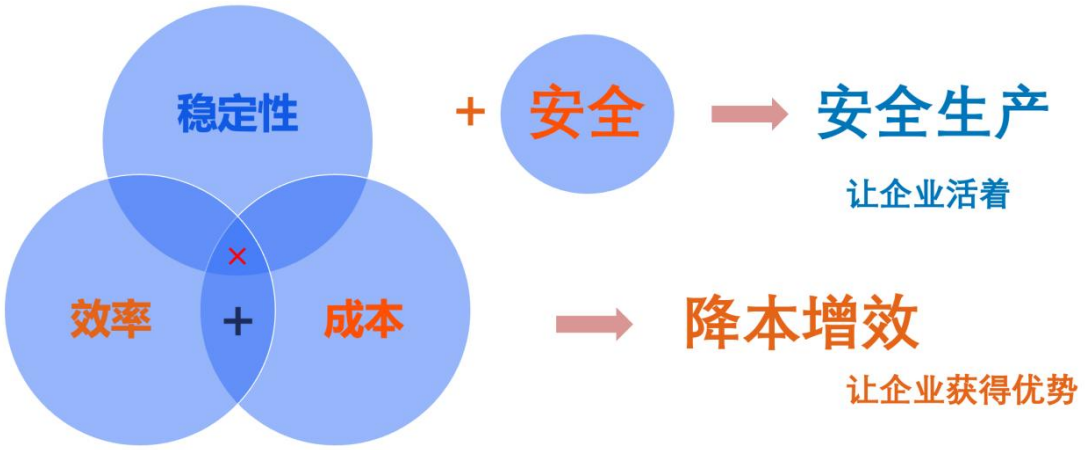


SRE 核心职责的不可能三角

愿望很美好，现实很"骨感"。因为如果你去细看，会发现这三个职责(或者称为目标)其实是一个「不可能三角」，我们无法同时在这三个方向上做到最优。

我们要做的、能做的事情其实是在这三者之间寻求一个平衡，一个动态的平衡。因为在公司、业务的不同发展阶段，在不同的场景下，我们的侧重点会有所不同。

那这三个核心职责，跟企业发展之间的关系又是什么呢？



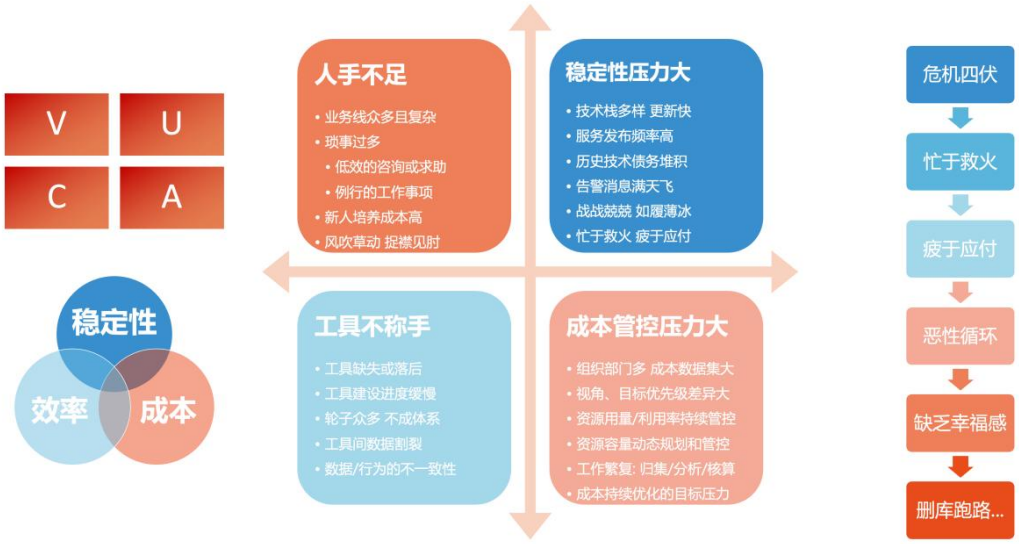
在我的理解下是这样的：

- 稳定性+安全＝安全生产，即质量方向
- 效率+成本＝降本增效，即效能方向

「安全生产」和「降本增效」这两者的主要目标分别是"让企业活着"和"让企业获得/保持优势"。

因为安全生产出问题，导致企业蒙受巨大损失甚至是倒闭的案例也有不少，这里我们就不做详细的展开。如果在相当的产品功能、质量、性能下，一个企业可以把单位成本控制得更低、可以有更高的交付效率，那么它就可以获得更高的竞争优势。

当然，这些事项或目标也是相辅相成的，这里我们只是探讨权重占比更高的部分。比如，如果一个企业的生产成本居高不下，在激烈的行业竞争中可能就会缺少或没有优势，最终也可能会影响到企业的生存问题。



SRE 的挑战和困境

那具体到 SRE 这个岗位，其面临的困境都有哪些呢？

除了前文提到的因几个岗位核心职责之间的相互制约所带来的困难之外，我总结了以下 4 个方面的困境：**1.人手不足**

在很多公司内部都会有"研发：测试比"、"研发：运维比"这样的数据，通常情况下运维/SRE 的占比不会太高。这里我们取另外的一个指标，"SRE 人数：公司全员人数"，通常情况下这个数值不会超过 1%，甚至不会超过 0.5%。也就是说在一个 1000 人左右的中小型企业中，SRE 的人数通常不会超过 10 个人，甚至不到 5 个人。

然而，随着公司业务的发展、持续地探索，公司的业务线会越来越多，并且日益复杂。同时 SRE 岗位还需要处理非常多的琐事，比如各种信息问询、问题排查等求助，还有很多例行的常规事项，比如服务巡检、OnCall 值守、响应&处理线上告警或故障、编写各种文档等等。最终会导致 SRE 的负载非常高，难以实现 Google 对 SRE 岗位的时间花费的建议：即有 50%的时间来做工程建设类的工作。

还有一个需要关注的点，考虑到 SRE 的岗位特点，需要对线上服务的稳定性负责，是一个责任非常重的角色，培养一个新同学的成本是非常高的。一位新同学要想顺利接手线上服务的稳定性保障工作，除了必备的基础技能，还有诸多的背景知识或信息需要掌握，一般都需要经过一段时间来建立真实的体感才能完全进入状态。

在这样的背景之下，SRE 的人手时常会比较窘迫，尤其是遇到一些人员的变动，会更加捉襟见肘。

2.稳定性压力大

随着行业和技术的快速发展，SRE 所面对的工作环境和管理对象也发生着巨大的变化。所管理的服务架构从单体到分层、分布式，再到微服务、云原生，一路快速甚至跨越式发展。这时候 SRE 就会面临技术栈多样且更新迭代快的现状，这些持续攀升的复杂度和急剧的变化都会对稳定性带来巨大压力。

此外，企业为了快速响应市场，服务发布、变更的频率也是越来越高，由此引发的稳定性问题也是层出不穷。与此同时，在业务快速增长阶段，为了快速地交付产品功能，在一些质量、稳定性方面的投入就会有所忽略或舍弃，最终会导致相关的技术债务不断堆积，进而给稳定性带来更大的威胁。

以上的几个问题，如果不能及时关注和处理，就会导致 SRE 同学每天面临着爆炸式的告警消息，忙于救火、疲于应付，也会导致 SRE 等同学对线上服务(操作)的畏惧，战战兢兢，如临深渊、如履薄冰。

3.工具不称手

与前两个点相对应的就是 SRE 在工具规划和建设上的投入了，因为人手不足、线上服务的稳定性压力大、日常工作负载高，就会导致在工具建设方面缺乏足够的投入，或者是为了快速地解决某些问题引入或建设了众多的轮子，这些工具(轮子)集缺乏顶层的规划和设计。

最终形成的局面可能就是：

- 工具的缺失或者没有做及时的更新适配
- 工具建设的进度缓慢，功能一再延期，无法交付
- 引入或制造了众多的轮子，各个轮子尺寸不一、不成体系，无法形成很好的合力
- 工具之间的数据相互割裂，每块数据都不够完整和新鲜，导致工具的易用性、可用性不足
- 严重的情况下可能会因为多个工具之间的数据、行为不一致带来灾难性的后果

以上 3 点，人手不足、稳定性压力大、工具不称手，也是存在相互依存、相互促进、相互制约的关系，如果协调不当可能就会进入上图所示的"死亡螺旋"：危机四伏 → 忙于救火 → 疲于应付 → 恶性循环 → 缺乏幸福感 → 删库跑路(开个玩笑，但愿不会这样)。

此外，还有第 4 点，成本管控相关的压力。

4.成本管控压力大

随着企业数字化转型的持续深化，IT 成本在企业成本支出中的占比也越来越高。在"降本增效"的大背景下，企业对于 IT 成本的管控和优化的诉求也就愈发迫切。

因为 SRE 岗位需要对业务的 IT 资源交付负责(这个点可能会因公司而异)，因此也就自然需要承担对 IT 资源管控的职责。比如对业务的资源需求合理性做评估，对资源用量、资源利用率、资源容量做管控，还需要对业务的 IT 成本数据去做归集、统计、核算分析，最后还需要识别出资源使用的不合理之处，并推动优化。

成本管控工作的难点有：

- 工作内容是非常繁杂的，琐碎且注重细节；
- 这是一个长期工作，需要周期性持续跟进和投入；
- 很多工作是需要多个部门、组织来相互协作才能达成的，会涉及跨部门沟通中的一些阻力或困难。

以上就是我司 SRE 岗位的核心职责，以及我所理解和归纳的 SRE 的挑战和困境。

稳定性运营的一些实施准则

要走出上述困境，我们是需要多管齐下，协同发力的。聚焦于 SRE 最核心的岗位价值，「稳定性」相关工作的优先级需要提高。为了确保系统的稳定性运营，建议 SRE 团队遵循以下几项准则：

1. 宏观框架与理论支撑：

在进行任何操作时，必须有一个宏观的框架和理论指导。这并不是要我们拘泥于条文或产生教条主义，而是这些理论和框架是经过大量实践总结出来的宝贵经验，能够帮助我们规避许多弯路。因此，理论支撑是非常重要的，当然，也需要我们结合实际情况灵活应用。

2. 数据思维与度量先行：

所有决策和操作都必须有数据支撑。只有通过足够的数据，才能做出科学的决策。比如，在应急响应时，需要依赖监控数据和大盘，了解系统状况和问题严重程度。因此，数据思维和度量体系的建设是关键。

3. 定期复盘与流程闭环：

定期的复盘和持续改进是稳定性运营中的重要环节。通过复盘，发现问题，总结经验，改进流程，形成闭环管理，从而不断提升系统的稳定性。

4. 紧扣价值与持续输出：

所有的工作都要明确其核心价值所在，尤其是要提升岗位的核心价值，如稳定性、效率、成本和安全。任何行动都应该有明确的目标感和价值感，这样才能确保工作的高效性和针对性。

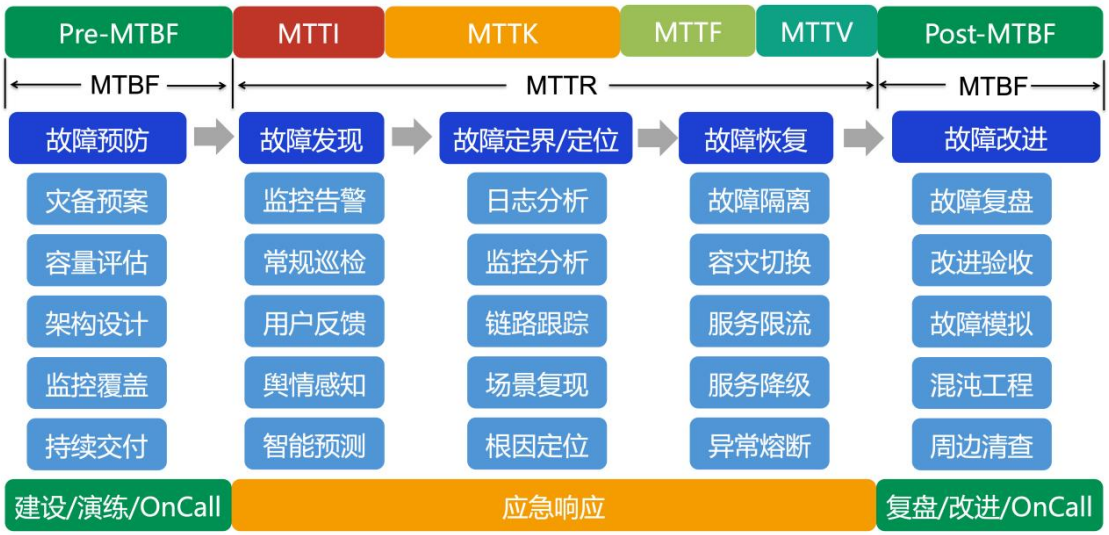
在推进稳定性建设的过程中，遇到职责划分(模糊)问题时，要主动出击，推动正确的事情取得进展，而不是因边界条件限制而放弃。这不仅是提升系统稳定性的关键，也是个人成长的重要途径。同时，这也是一名称职 SRE 的必备特质--「责任心」的体现。

通过以上准则，SRE 团队可以更好地保障系统的稳定性，实现高质量的运维管理。

(二)体系设计详情

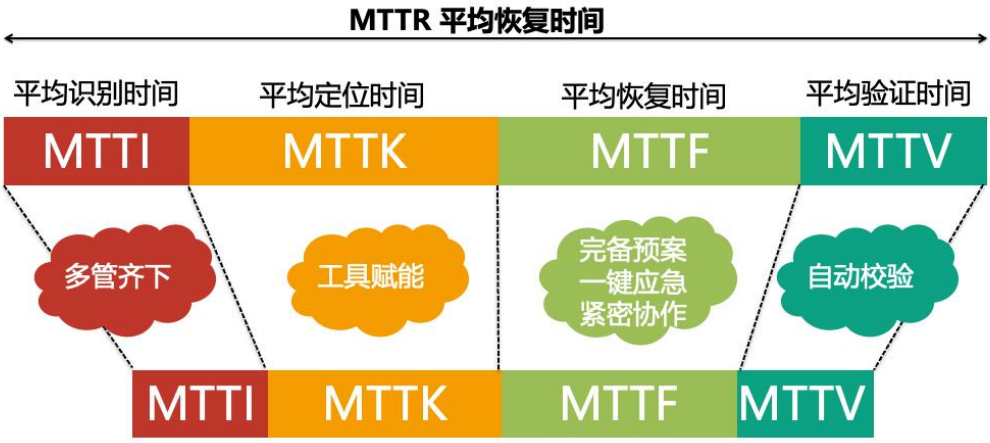
SRE 体系建设

稳定性建设全景设计



这张图应该是国内 SRE 同学比较熟悉的，即基于"故障生命周期管理"的思路来盘点和审视需要做的稳定性建设工作，核心目标是保障和提升服务的稳定性，框架中涉及的内容还是非常丰富的。

我们稳定性保障的目标，就是要尽可能缩短 MTTR，进而提高 MTBF。其中 MTTR 的指标是可以被细化拆解的，在细化之后，我们就可以分而治之、各个击破了。



我们可用的手段，总结如上图：

- MTTI 阶段：多管齐下；尽可能用更多的手段来覆盖可以发现、暴露问题的通道，包括完善监控点位的覆盖，建设体系化的观测能力和系统。
- MTTK 阶段：工具赋能；这个过程中需要更多地借助工具、平台的能力，建设健全运维系统，比如监控平台、日志平台、链路跟踪平台等，如果能力允许也可以去尝试建设"根因自动诊断定位"系统。
- MTTF 阶段：完备预案、一键应急、紧密协作；这个是在故障处理过程中最核心的部分，是真正可以让服务恢复正常的阶段；这个过程有比较多的前置依赖，也就是需要我们在平时做好储备的，比如建立健全预案体系，将预案的执行手段尽可能沉淀到工具平台中，应急动作尽可能做到一键直达，缩短预案执行的路径。其次，这个过程中可能还会涉及部门内部、部门之间的协作，尤其是在处理重大故障的场景中；这时候就需要有一套可以让大家紧密协作的流程或共识，让大家可以信息互通、各司其职、有条不紊、高效协同。
- MTTV 阶段：自动校验；这个过程也是需要尽可能依赖自动化的手段去实现，相关能力通常可以内置在「可观测相关平台」或「故障快恢系统」中。

关于持续交付能力的重要性

「持续交付」能力这个部分，我要稍微展开讲一下，美图公司目前在大力推进 AIGC 相关的业务，在 AIGC 类服务的保障工作中，持续交付能力显得尤为重要。以下是一些具体原因及相关细节的简要说明：

首先，AIGC 类业务具有传播速度快，容易形成热点、业务流量增速迅猛等特点，对资源的用量、性能、弹性效率都有非常高的要求。

其次，AIGC 高度依赖 GPU 算力资源(虽然也有其他解决方案，但 GPU 目前还是更通用的资源)，而 GPU 服务器的初始化时间远长于常规的 CPU 业务场景。初始化过程包括多个阶段：机器初始化、GPU 卡的初始化、驱动加载、容器镜像拉取、算法模型加载等，这些步骤复杂且耗时。有些场景下的算法模型和容器镜像体积可达到几十 GB，这将进一步增加网络和存储的吞吐压力，进而增加资源交付的耗时。相比于传统纯 CPU 算力场景下的秒级容器扩容，在当前的技术条件下，即便做了极致性能优化，GPU 算力资源也很难实现跟纯 CPU 同样的交付效率。

同时，GPU 资源是非常昂贵和稀缺的，尤其是受限于多种因素，相关资源非唾手可得，且企业通常较难容忍大量 GPU 资源的长期闲置，因此需要提前规划和准备。

我们目前采用多云战略，涵盖了国内主流云厂商和线下机房资源，并利用统一的「容器管理平台」整合纳管这些资源，可以在很大程度上解决仅依靠单一厂商无法满足资源需求的问题。不过，在资源调度过程中，也会涉及许多细节问题，例如：

- 不同 AI 算法所适用的 GPU 卡类型不同；
- 不同 AI 算法所依赖的周边资源(如网络、存储等)、周边服务会有差异；
- 同一个 AI 算法，在不同的 GPU 卡上的性能表现、性价比可能会有巨大差异；
- AI 算法/业务玩法，迭代速度快、丰富多变；
- 不同厂商之间的基础设施会存在差异；
- 在成本管控的压力下，我们还要保证算力成本最优；

这些现实细节都对我们的资源交付平台(容器管理平台)提出了高要求，必须具备强大的能力来支持高效的资源调度和优化，这也是我们近期重点建设的方向。

当然，除了上述 AIGC 的业务场景，在常规业务的稳定性保障过程中，「持续交付」能力也非常关键。比如在大量请求短时涌入的应对场景，在服务单点(服务组、AZ、Region 等)异常需要做快速容灾切换的场景等。

PPTV 框架

"PPT"应该是大家相对熟悉的一个概念，包含了人员、流程和技术，这是 ITIL 中的 IT 管理三要素，SRE 体系建设同样需要依赖这个模型。要做好故障管理，必须有适当的组织结构和流程保障；仅有组织和流程还不够，还需要强有力的技术支撑。

在此基础上，我增加了一个词，Vision（愿景、目标），与 PPT 结合，构成一个"PPTV"的框架。只有在稳定性建设上有共同的目标期许，才能做好 SRE。不同的参与部门应对服务稳定性有一致的愿景，不论是出于行政压力还是自身对服务稳定性有追求，大家都需要有统一的目标。

"PPTV"这个框架模型，涵盖了目标对齐、组织流程保障和技术支撑等几个重要方面，是一个相对完整的框架，在做 SRE 体系落地的过程中可作为参考。

稳定性运营平台

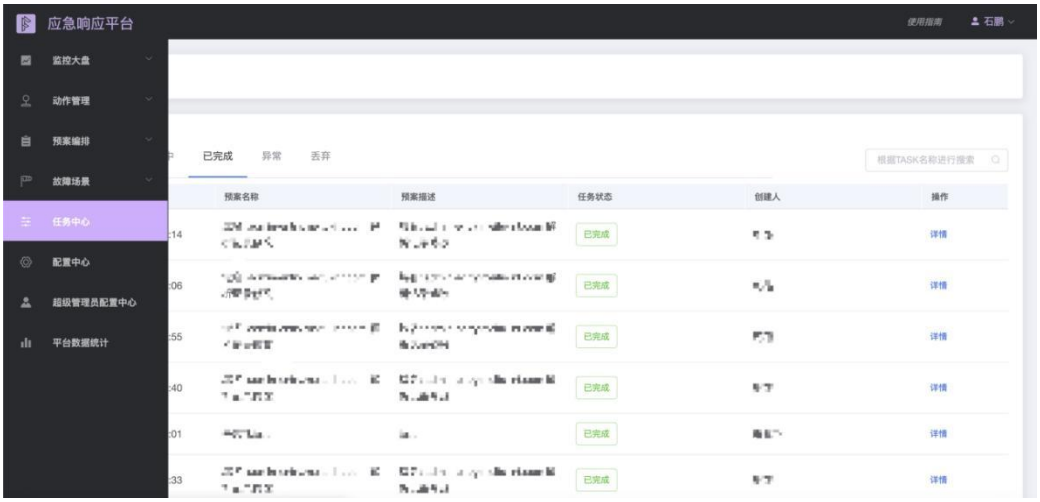
除了上述大家可以用来参考规划工作内容的「稳定性建设全景图」、用于推进稳定性建设工作的宏观框架「PPTV」，我们还需要建设一个可靠的平台来承载和实现相关能力。

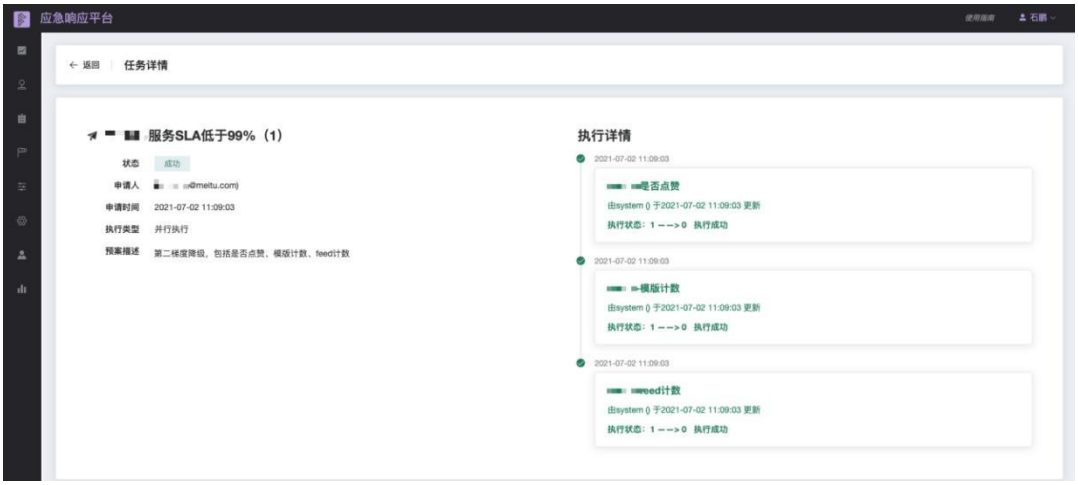
我们团队目前建设了一个「稳定性运营平台」，主要实现了对一些日常稳定性运营工作的管理，包含 OnCall 轮值、自动化巡检、事件管理、稳定性运营报告、业务关键指标数据管理、历史数据分析等；还有一个独立的「应急响应平台」，主要覆盖针对线上服务异常、故障的应急处理。目前我们正在将这两个平台做功能合并和刷新，下面是规划和实现中的「新版稳定性运营平台」的主体框架。

平台的主体功能按照故障生命周期的三大阶段进行拆解设计，以期覆盖故障事前、事中及事后的完整过程。整个平台的功能模块可分为三个层：由仪表盘构成的「呈现层」、由事件管理、故障管理、应急响应构成的「核心能力层」，以及平台的「基础能力层」。

我们期望以此平台作为 SRE 日常工作的底座，完成稳定性运营、应急响应、稳定性建设工作推进等核心工作。

针对故障应急的部分，我们现有的「应急响应平台」已经实现了必要需求的覆盖，在此简单讲解。





平台的核心逻辑是：

①将对服务干预的手段抽象为可执行的原子「动作」，如脚本或 HTTP 接口调用等；

②将这些原子动作进行组合编排，形成有序的应急处理「预案」；预案中的动作执行可以是带有依赖关系的，支持串行执行、并行执行、逻辑判断等；

③将预案绑定到预设的各个故障「场景」中，一个场景中可以绑定多个/多级预案；

当故障发生的时候，如果在对应的场景下有匹配的预案，我们就可以快速执行以实现止损；如果没有现成的预案，但是可以通过临时组合已有动作来覆盖的场景，也可以在平台上快速完成编排和执行的操作。

故障管理体系

通过前面的段落我们了解了 SRE 体系建设的主体内容，下面我们聚焦在「故障管理」的部分，聊一聊如何搭建故障管理体系。

我们前面一直在提"故障的生命周期"，我们的故障管理体系也要围绕故障的生命周期来展开：服务稳定运行、故障发生、故障恢复、故障复盘、故障改进、改进验收，这个过程会是周而复始的一个「环」，我们服务的稳定性就会在这一个又一个「环」的迭代下变得"更健壮"或者"更脆弱"。

为了做好故障管理工作，我们需要从指标体系、流程规范、考核体系、组织支撑等几个方面展开建设。

1、可用性体系

上述所提及的即为标准化的 SLIs/SLO/SLA 体系，该体系融入了 VALET 方法，通过五个核心维度——容量（V）、可用率（A）、延迟（L）、错误（E），以及人工提单与保底分类（T）——来系统地梳理 SLI。这一步骤是构建系统可用性的基石，若无此体系，则无法有效地衡量系统可用性，后续的工作亦将缺乏坚实的基础。

2、故障定级、定性、定责

我们对故障的判定，通常包含 3 个方面：定级、定性、定责。

- 定级：故障的严重程度；
- 定性：故障属于什么类型；
- 定责：故障的责任要如何划分；

这几个方面的判定，都依赖于我们在组织内部建立标准规范、达成共识、形成制度。

- 定级：通用标准

美图故障等级衡量指标说明			
故障对服务功能的影响	**权重占比xx%**	**备注**	**级别分值(阶梯式)**
1级	主要功能完全不可用		100
2级	主要功能部分不可用		75
3级	次要功能完全 不可用		50
4级	次要功能部分不可用		25
故障影响时长	**权重占比xx%**	**备注**	**级别分值(阶梯式)**
1级	xx分钟以上		100
2级	xx分钟~xx分钟		75
3级	xx分钟~xx分钟		50
4级	xx分钟~xx分钟		25
故障发生所处时段	**权重占比xx%**	**备注**	**级别分值(阶梯式)**
1级	最活跃时段		100
2级	次活跃时段		75
3级	非活跃时段		50
4级	最不活跃时段		25
对用户的影响范围	**权重占比xx%**	**备注**	**级别分值(阶梯式)**
1级	影响用户占比xx%以上		100
2级	影响用户占比xx%~xx%		75
3级	影响用户占比xx%~xx%		50
4级	影响用户占比xx%以下		25

针对故障定级，建议形成"可以量化"的规则。

上图是美图定义的故障等级的衡量指标说明，图中列出四点都是常规的考核项，主要看对服务功能的影响、影响时长、故障所发生的时段、对用户的影响范围。

我们会根据相关考核项的具体影响程度进行量化打分，最终加权计算得出一个故障评分，并对故障进行定级（不同分数对应不同级别，具体可参考下文「故障预算」的段落）。

2. 定级：个性化标准

除了通用定级标准中的四个考核项，在一些业务场景下可能还会有其他的考核项。比如某些电商类、商业化、广告类或金融类的业务，会重点考核资产损失；比如有些社区类应用，可能会重点考核舆情、公关事件等。因此，不同的业务部门会有不同的故障定级策略。

因为这些个性化故障定级的诉求是由业务真实的特性所决定的，因此是合理地存在，并且是无法直接简单粗暴地用通用规则所覆盖的。不过，为了维持公司全局的故障管理工作的公平、公正，我们需要把所有业务部门的故障定级标准拉齐。我们的做法是跟有个性化故障定级诉求的业务部门做定级标准的映射，各业务部门对故障的考核项、考核项的权重可以不同，但是故障级别的定义需要跟通用规则保持一致，并且定级规则需要公示并通过「故障管理委员会」的审议。

3. 定性：有效分类

除了对故障定级，我们还需要对故障做定性的分析和归类，由此可以发现一些故障的共性、特点或规律，并指导我们的稳定性建设工作。以下是美图的部分分类示例：

- 代码质量
- 测试质量
- 流程规范
- 变更操作
- 容量规划
- 产品逻辑
- 硬件设备
- 局方故障

- 云厂故障
- 第三方
- ……

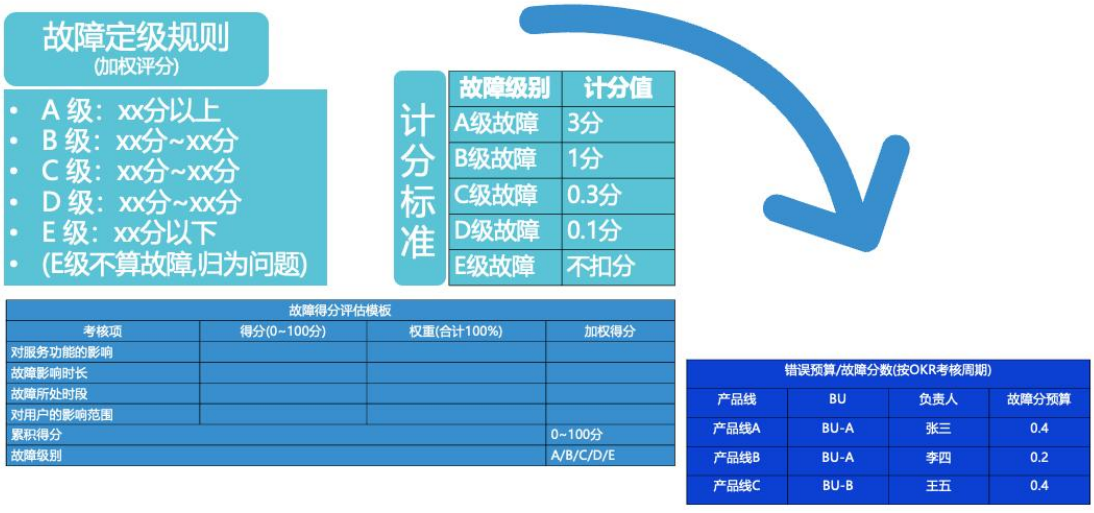
4. 定责：判定原则

定责任并不意味着处罚，更多的是要承担故障整改的责任。

我们整体推崇和践行的故障文化是"不指责"，但不指责不代表着可以持续犯错误，尤其是一些低级的错误。我们在做故障定责的时候，有一些比较重要的判定原则，简述如下：

- 高压线原则：一些企业内部的红线(比如数据安全)不能触碰；不能犯一些低级的错误；不能持续犯某些错误。
- 上下游原则/健壮性原则：需要根据服务的上下游依赖集及健壮性判定。每个部件自身要具备一定的自愈能力，比如主备、集群、限流、降级和重试等；对依赖的管理方面，原则上核心应用对非核心应用的依赖必须有降级和限流手段。
- 第三方默认无责：主要指的对内做故障判定时，默认第三方无责；稳定性责任一定是内部角色承担。这条规则是为了避免各种问题都甩给第三方、云厂，久而久之会失去责任心。当然，对于第三方该追责、索赔的还是要正常进行。
- 分段判定原则：一个故障的链路可能会比较长，原因也可能不止一个。因此我们要分段做分析，这样更有利于让故障问题更聚焦，改进措施也会更有针对性。
- 自由裁量原则：具体问题具体分析，Case by Case。同时针对一些边界定义尚模糊的场景，要灵活把握、留有合理的空间。

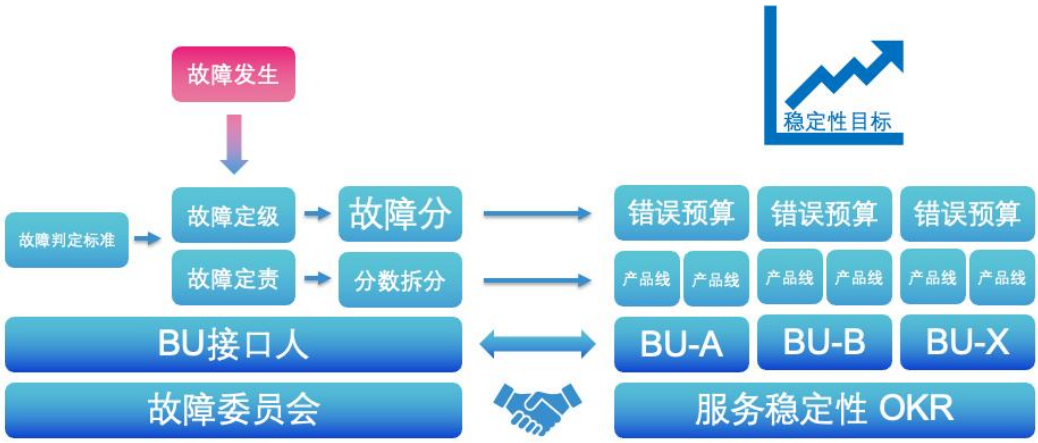
3、错误预算



我们在 SRE 相关的工作中会经常听到「错误预算」这个词，那故障预算的机制要如何落地呢？在不同的企业里会有不同的实践思路，下面我简单讲一下我们的做法。

根据前面讲的故障定级规则，不同的故障会基于最终的故障评分被定为 ABCDE 几个级别，不同的故障级别会被扣除不同的分数，这个用于抵扣的分数则是「故障预算」。我们每年为一个 OKR 考核周期(每季度、半年会做期中/期末的核算)，在一个考核周期里面每个 BU 会被分配一定的"故障分"，如果发生故障，则要根据规则抵扣相应的分数，如果故障分被扣完了则意味着错误预算耗尽，会对应着相关的限制策略。同 Google SRE 的做法类似，即可用性达到多少标准，发布将受到限制。我们将错误预算和 OKR 做了绑定，让大家在目标驱动之下达成这个目标。

4、组织结构支撑



故障管理特别依赖于组织的支撑，因为故障管理不仅仅是运营部门或者技术保障部门能单独完成的事情，需要拉通不同的团队进行协作。为此，我们成立了故障管理委员会。故障委员会是一个虚拟的组织，各 BU 指定接口人参与故障管理委员会，或参与故障管理委员会的一些日常工作，比如参与对故障判定的讨论和审议、传达相关信息给 BU 内部。

我们可以根据上图完整串一遍这个故障管理工作的流程。当故障发生了，基于判定规则给故障定级、定责。定级意味着每个级别产生对应的故障分；定责会涉及故障的拆分，比如发生了一个故障，它由 A、B 两个部门共同承担；比如，根据定责发现责任上 A、B 部门实际是五五开，因此 A、B 平分故障分，从 A、B 部门各自的错误预算里扣除。

通过这种方式就能约束好 BU 吗？当然不是，还要辅以行政管理的手段来保障，在我们内部就是 OKR。在各 BU 的 OKR 里需要有服务稳定性的目标项才行，当把服务稳定性的目标写进 OKR 之后，各 BU 才会更有压力和动力在故障管理及稳定性保障上做投入。

故障管理 之「三段式拆解」

根据故障发生的时间顺序，我们把故障管理分为故障前、故障中和故障后，在每个环节都有一些核心的工作内容和目标。

故障前：故障预防、灾备预案；目标是尽可能地做足准备工作，毕竟有背方可无患；

故障中：发现、定位、解决故障；目标是尽可能的提高效率，缩短故障恢复的时间；

故障后：故障复盘、故障改进；目标是尽可能多的从故障中吸取教训，反思和学习，完善和修复故障中暴露出来的问题。

1、故障前

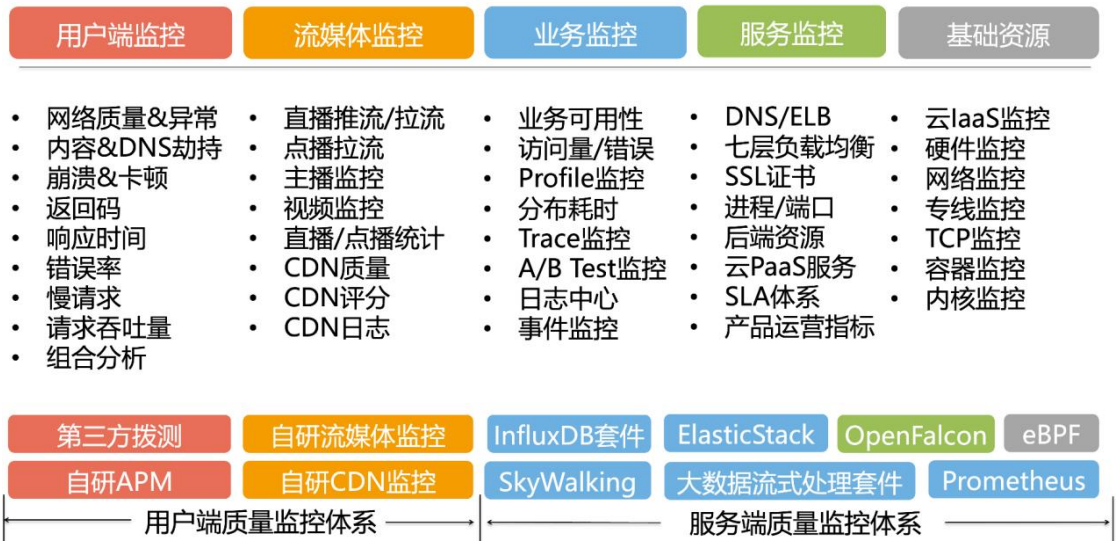
故障前的故障预防和灾备预案，我们应该怎么做呢？

这里列出了这么几个比较关键工作内容：

1. 监控覆盖

监控覆盖的话比较容易理解，服务上线后，只有拥有足够的监控手段，并且尽可能多的覆盖服务的各个环节，才有可能在后面发生问题的时候，快速地感知到。也就是说，我

们的目标是尽可能有更多的监控手段去覆盖我们服务，保障各个场景。下图是我们之前用到的一些监控组件：



我们把监控体系分为两块：客户端质量监控和服务端质量监控。

①客户端质量监控

我们在客户端质量监控里边去感知客户端的状态。

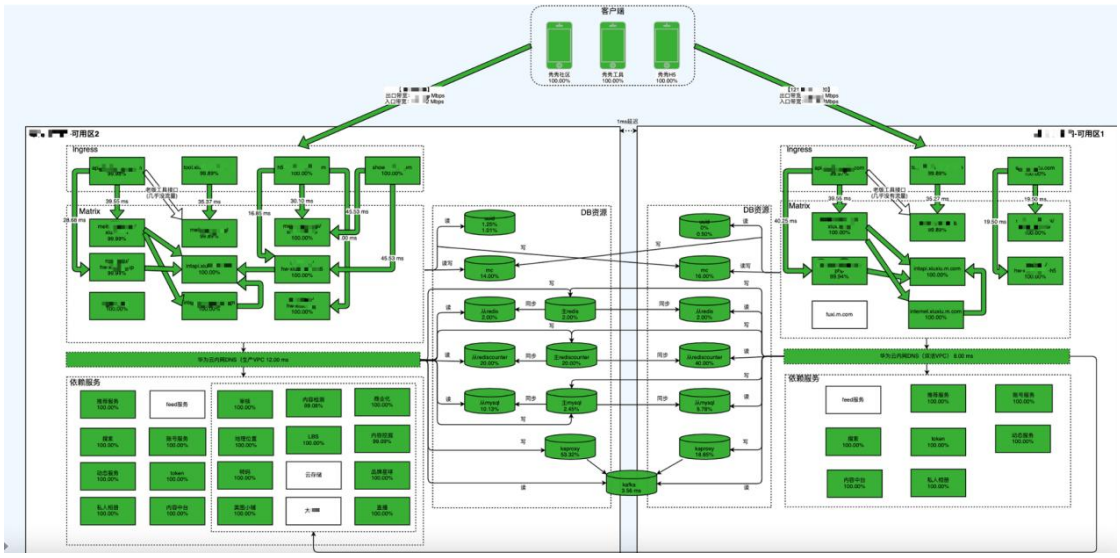
客户端质量监控我们都使用了哪些手段呢？

常规的有一些第三方的拨测、第三方的 APM。除了一些商用的产品，当然也有一些免费的工具是可以用的。

应对一些流媒体的应用，我们自研了一套融合 CDN 的监控还有流媒体的监控。这个其实也比较关键，因为现在业务体量大了之后，CDN 是不可避免地会用到，但如果没有这部分监控的话，CDN 质量就要依赖于用户反馈，链路就会比较长。所以说要有手段去主动监测 CDN 的质量。我们建设了这样的 CDN 的质量监控，然后会去给不同的 CDN 厂商去打分，再去做智能的调度。

②服务端质量监控

服务端的质量体系是用到了一些比较通用的，像 InfluxDB 套件、ELK、Prometheus、Open-Falcon，Skywalking 等等。除此之外，我们还建设了一些基于大数据流式计算的系统，主要是用来做我们 SLA 的体系。



通过可观测平台，构建了一个全景的链路拓扑下图是我们偏运维侧的一张数据流向图，主要是说明一个业务包含了哪些域名，大概经过了什么链路，经过了哪些中间组件，最终到达了服务端。但是这个拓扑偏向静态的关系构建，目前有点不是太符合现在的要求。

因此，目前在研发动态的拓扑展示。我们将服务的链路拓扑存储在图数据库中，以实现高效的全局链路信息管理。这不仅为上层业务提供支持，还在问题排查时能够快速定位和分析。数据的来源和整合过程如下：

首先，数据来源包括 CMDB（配置管理数据库）、资源层数据和业务层数据。这些数据从 CMDB 中提取并存储到图数据库中。同时，我们对机器、容器的节点（Node）、Pod 以及各种网络连接进行抓取和筛选，并将结果存入图数据库。此外，还从日志中提取相关信息，最终汇总到图数据库中。

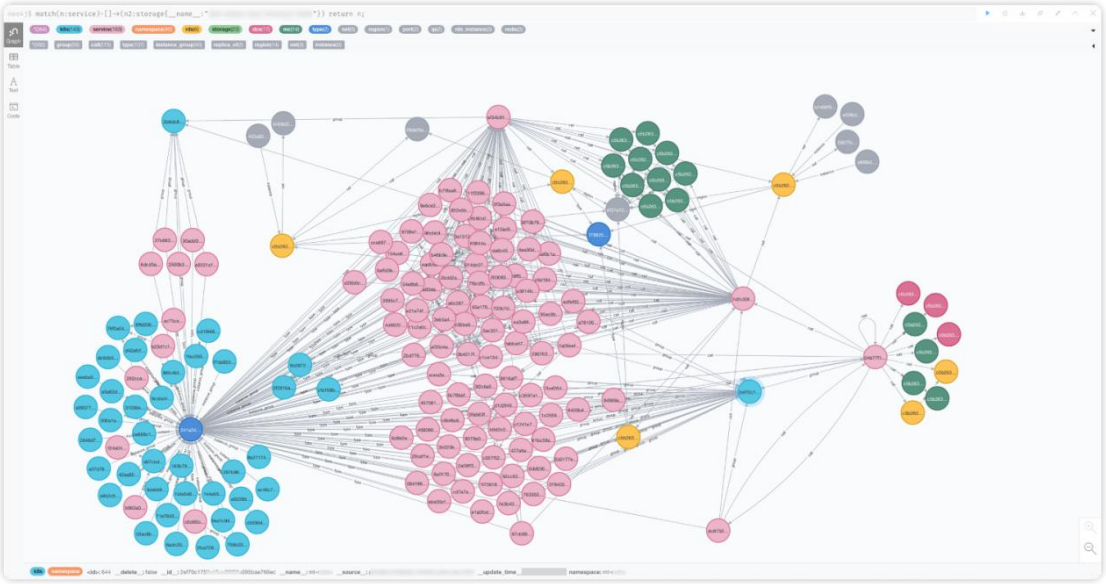
在建设过程中，需要注意以下几点：

数据多样性：链路信息复杂，单一手段无法获取全局信息，因此我们采用了多种数据来源的组合。

容器化环境：由于服务已全面容器化，扩缩频繁，数据量巨大，传统手段难以捕捉所有信息。因此，我们选择基于 eBPF（扩展的伯克利包过滤器）的网络连接数据抓取方法。这种方法尽管会产生大量数据并具有时效性，但能有效捕捉动态环境中的链路信息。

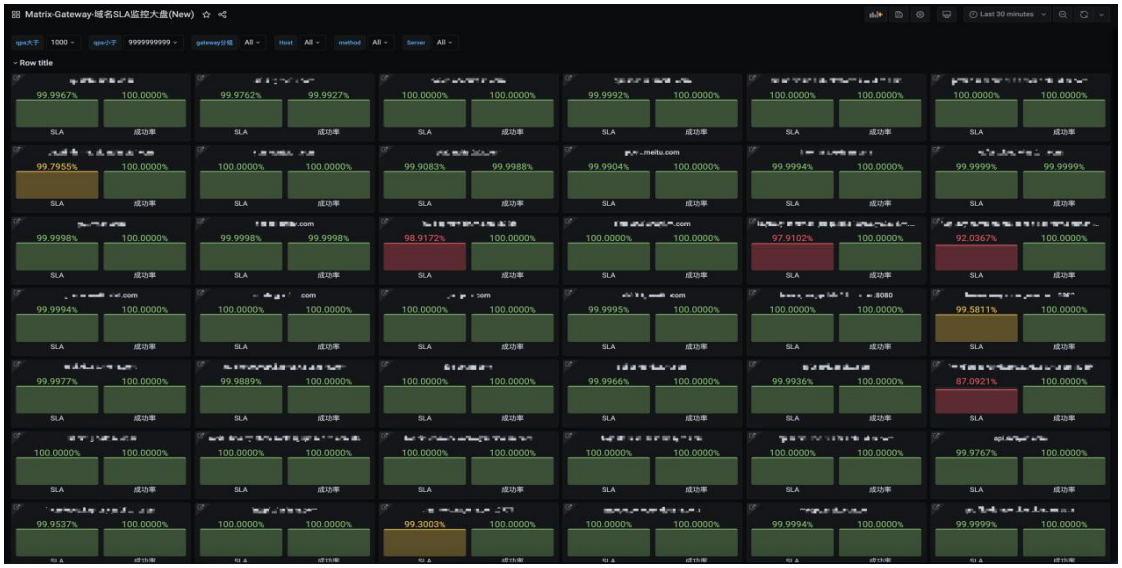
时效性管理：容器（如 Pod）的生命周期较短，可能仅存活 10 分钟。因此，在链路排查时，需要设计机制以查看特定时间段的数据，并确保过期数据不再展示。

通过这些方法，我们初步实现了对复杂链路信息的高效管理和利用，为业务支持和问题排查提供了强有力的工具。

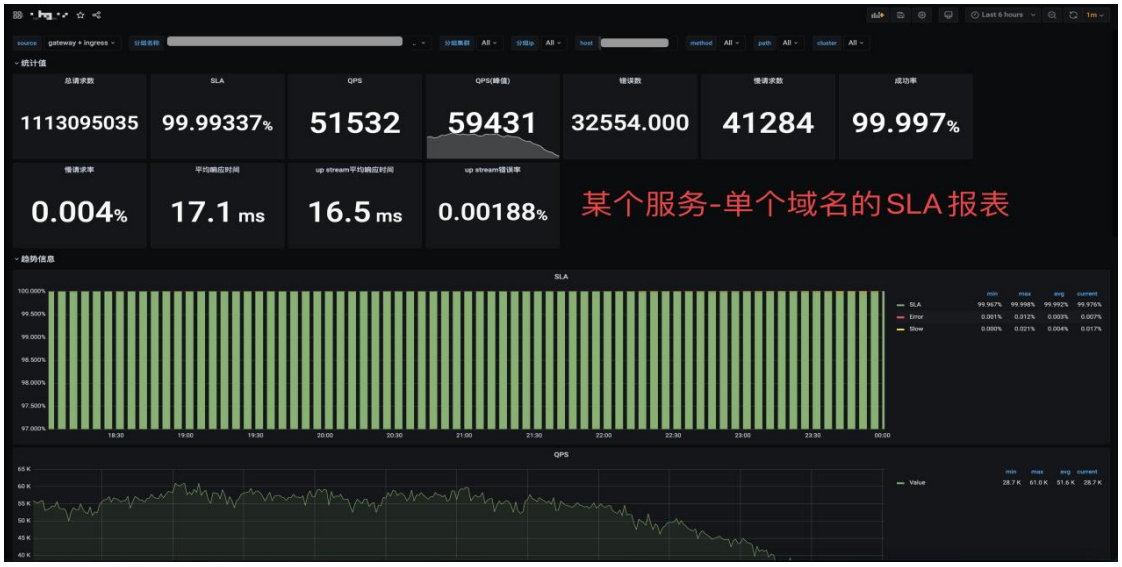


接下来展示一些我们现在正在使用的监控大盘案例，都是需要我们在日常工作中完善的：

下面这张图是我们的域名 SLA 监控大盘，每个域名一张卡片，我们可以在一个 Dashboard 里看到全局的信息。这个大盘在大面积服务故障的时候会非常有效，尤其是在故障恢复阶段，可能有个别服务会因为种种原因存在长尾问题、迟迟没有恢复正常，通过这个大盘我们就可以快速发现异常点。



下图是我们 SLA 的监控：



这是我们服务端质量体系里边最重要的一张报表，请求数、错误数、慢请求有多少、服务整体平均响应时间、后端响应时间、成功率等 SLA 的指标都会在这张报表里边体现。

下图就是前文提到的客户端的质量监控：

其实在这之前我们有用过一些商业产品，但后来发现商业产品不能很好地满足我们的监控需求，所以后来就决定自己研发了「哈勃监控」。

在这里面，我们就可以对客户端的状态有一个感知能力。比如说有时候客户端的网络质量不好，或者说因为一些网络错误、SSL 证书的错误，连接到 CDN 之类的失败了，导致最终请求没有到达服务端。

此时，你看 SLA 报表里面显示一直是 OK 的，但其实这时候用户真实的体验并不是这样的，他对服务的感受其实已经是很糟糕了。所以我们就需要有客户端的监控体系，在这个报表里，就可以用一些指标，将用户真实的用户体验反馈出来。

2. 架构设计

架构设计可能会更偏业务侧一些。我们在故障之前，要尽可能做好服务的架构设计，同时在做一些预案之前，也要把服务架构做好梳理。只有当我们把服务的架构了然于胸，才更有可能在故障发生的时候从容不迫，更好地定位问题。此外，要更多去加入柔性设计，也就是说你的服务要具备一些像降级熔断、故障隔离这些手段，要有这样的柔性设计在里边。这样架构可以提供这些能力，后面才能更好地去做服务的保障。再有一点就是，要尽可能地去规避常规的风险点，比如说单点之类的；同时还要尽可能地去规避架构里面常见的坑。

3. 容量评估

下图是我们压力测试的一个页面：

以这个图为例，容量评估是指在服务上线前，通过压测来评估服务的承载能力，从而更好地了解其上线后的状态。基于这些评估结果，我们可以根据业务量级来规划服务的容量。(此外，目前我们已经建设了能力更强的自动化压测平台来覆盖日常的压测需求。)

然而，容量评估并不能完全依赖压测。在压测之前，有没有其他方法可以辅助我们更科学地进行容量评估呢？答案是肯定的，但这往往被忽略，即数学计算。

在进行容量评估时，不能完全依赖压测。我们需要基于对服务的深刻理解，包括每条请求大致消耗多少资源等，进行基础认识的建立。然后，运用这些认识来评估所需的后端资源、CPU 和内存等。这些评估可以通过数学计算的方法来实现。虽然非常精确地计算可

能比较困难，但这种方法不可忽略。我们应先进行粗略计算，再结合压测，最终得出一个科学的容量评估。

4. 灾备预案和灾备演练

这两点是比较关键的，跟故障会更强相关。下图是我所整理的关于怎么做灾备预案和灾备演练的架构图：

我们把这个过程分为了五个环节，基本按照这五个环节来做，灾备预案就差不多可以完全覆盖了。下面来具体地说一下：

服务梳理：在服务梳理阶段，要基于前面的架构图等来梳理请求链路，要分段分层地梳理请求都经历了哪些层次、有哪些阶段、经过了哪些设备、周边有哪些依赖（包括内部的服务依赖、后端的资源依赖、第三方的依赖等），然后还要梳理架构目前是不是有什么风险、流量有没有经过一个单点、有没有哪个点可能是存在瓶颈的...这些都是我们在服务梳理阶段需要去梳理清楚的。

预案梳理：了解各种情况后，就可以梳理相应的预案了。仔细分析应该用什么样的手段来覆盖，将风险各个击破；然后要尽可能地做多级预案，因为预案如果只有一级的话，容灾能力是不够柔性的；此外，还要借助一些智能调度的手段；最后就是柔性设计，前面也有提到，是更偏业务侧一些，也就是说在服务里要有尽可能多的手段来做自己的一些failover，可以去做一些降级、熔断等。

沙盘推演：梳理出来预案后，并不是直接做演练。要了解预案是不是真的有效，不应该也不需要直接做演练，而应该是先想清楚。我的建议是去做沙盘推演。在这个过程里，我们要尽可能去发动多的部门协作起来，来审视我们的预案是不是有效。毕竟人多力量大，并且不同团队的人对业务可能有不同的理解，在这种头脑风暴下，就有可能碰撞出更多的

可能性。然后在这个过程里，会基于一些可能的故障场景、case 等来做推演，当故障场景 A 出现了，梳理出了预案 A'，那 A'能不能把故障 A 完全解决掉？在解决故障场景的时候，有没有引入一些其他的风险点或问题？在这个过程里面，我们都要想清楚，当得出一个大家都认可的结果后，预案才推演完成。

预案落地：这一步是需要做落地，包括文档输出、功能实现、架构适配、工具建设。

预案演练：落地之后，要通过演练的方法来验证预案是不是真的有效。在这里，我大概列了几种。比如无损演练和轻损演练：我们做故障演练要尽可能地做到对业务无损，但有些预案本身应对的故障场景就是会损害业务的，那这种时候我们要尽可能降低这种演练带来的损失，比如说选不同的时间段、流量的控制、灰度之类的，尽量去做轻损的演练，既然我们是通过故障演练的方式来确保预案有效，那肯定不能因为故障演练而演练出一个大的故障，这就有些得不偿失了；还有就是单点演练跟组合演练：你的演练到底是要一次演练某个模块，还是说要把一个大故障场景里所有涉及的点都演练一遍？这个也是我们在预案演练里需要考虑的。

下图是美图内部灾备预案和故障演练的例子：

大家可以看到，我们是分了几级来做的预案，包括数据的冷备、同城的双 AZ、异地灾备、热备等。我们在做故障梳理的时候，大致也是按照前面讲到的环节来做：先梳理（分为研发侧和运维侧梳理），然后盘点（盘点这个过程里有哪些故障点），再做 case 推演，而后是预案的输出，最后是演练。

5. 持续交付

持续交付也是需要我们在日常建设中就做好的事情。

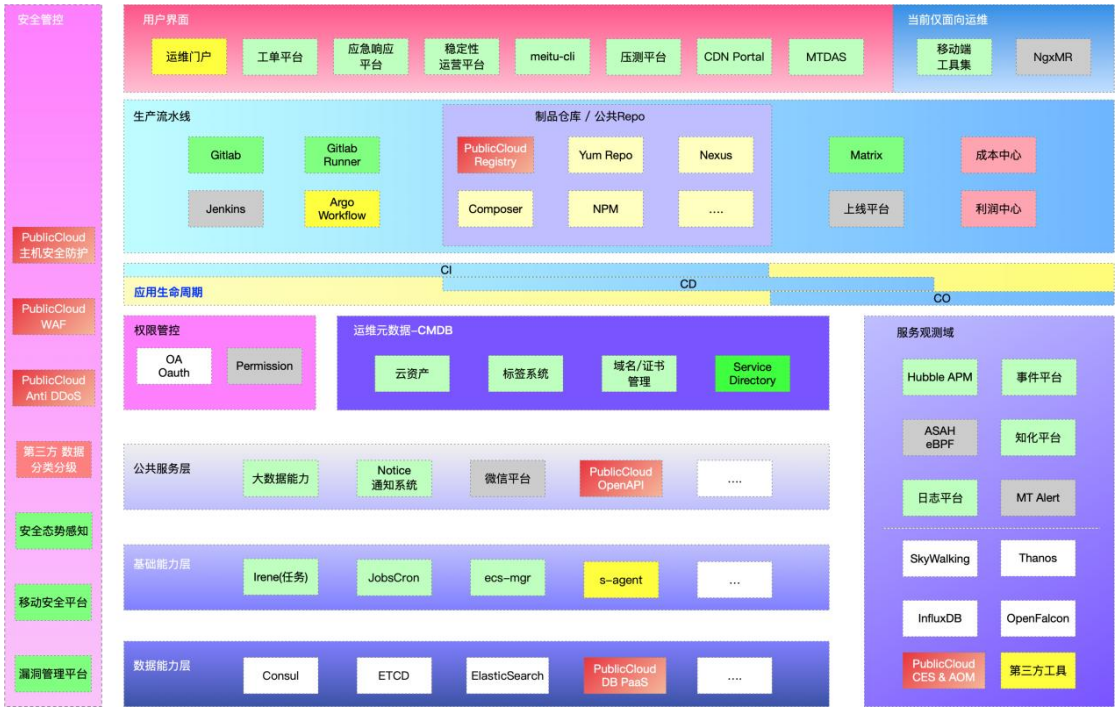
十几年前，我们主要使用物理机。当线上发生故障，原因是大流量导致性能瓶颈时，首要的应对策略是什么？是扩容。只有通过扩容才能完全承载流量，减少损失。否则，只能进行有损的降级，即丢弃请求。这就涉及到持续交付能力。

现在，随着云计算和容器技术的发展，基础设施架构具备了弹性伸缩的能力，能够更快速地进行扩容。那么问题是否解决了呢？并不完全。持续交付相关的场景还有很多，如果发生故障时，持续交付能力无法跟上，故障恢复时间仍会被拖长。

上图展示了我们公司目前使用的一些持续交付体系组件，大家可能比较熟悉。右上角是我们公司内部开发的基于 K8s 的云管平台，支持多集群管控，内部代号为 Matrix。与其他外部云管平台类似，但我们开发得较早。基于这些基础设施，我们能够保持良好的持续交付能力。

上述内容讲述了我们在故障管理之前需要做的事情：在故障来临之前，如何做好预防和准备，以及应对能力的储备。

- **工具体系建设**



除了前文有提到的几个跟稳定性保障强相关的几个平台或系统，还有非常多的工具平台对于稳定性保障、安全生产都是不可缺取的，我们需要构建一整套工具体系来实现相关需求并相互协作，最终共同保障服务的稳定性。

故障中

当故障真的发生了，我们应该怎么做？去发现定位和恢复。这里列举几个比较核心的手段：监控告警、日志分析、链路跟踪、故障隔离、容灾切换、降级熔断。只看字面意思也很好理解。下面我将以图结合案例分享具体做法。

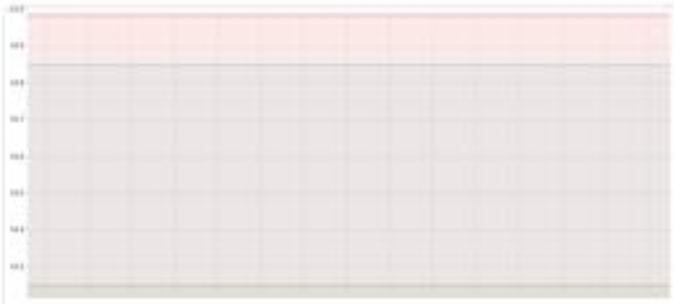
1. 监控告警

健康大盘 BOT

E健康大盘

服务30分钟内RT状态 的prom监控

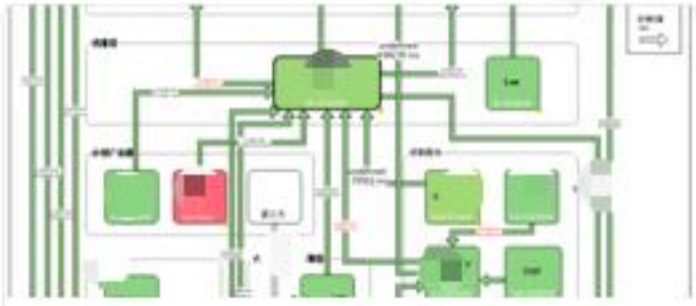
健康大盘 BOT



健康大盘

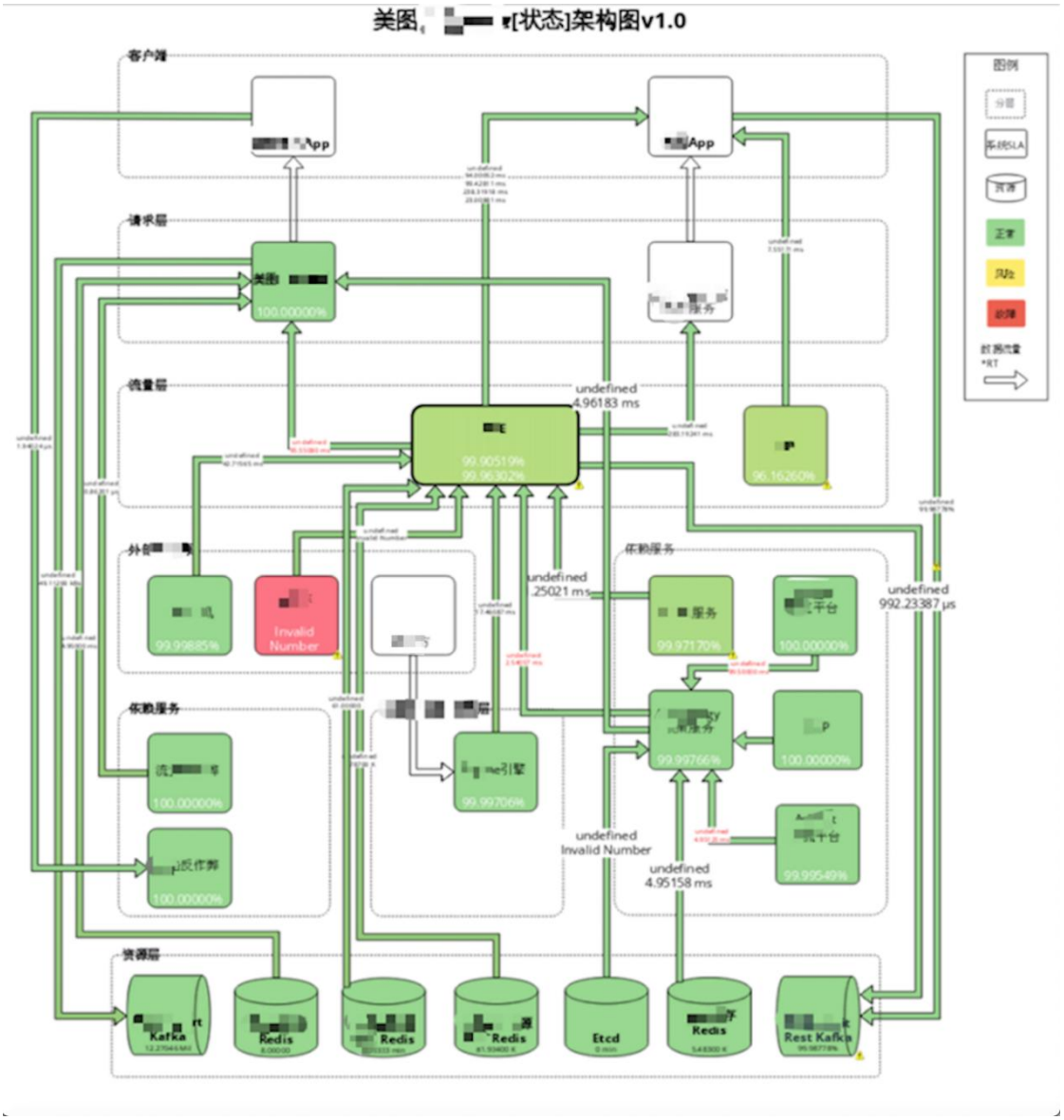
服务30分钟内RT状态 服务内的prom监控

健康大盘 BOT



健康大盘

E请求 服务接口RT大于2ms



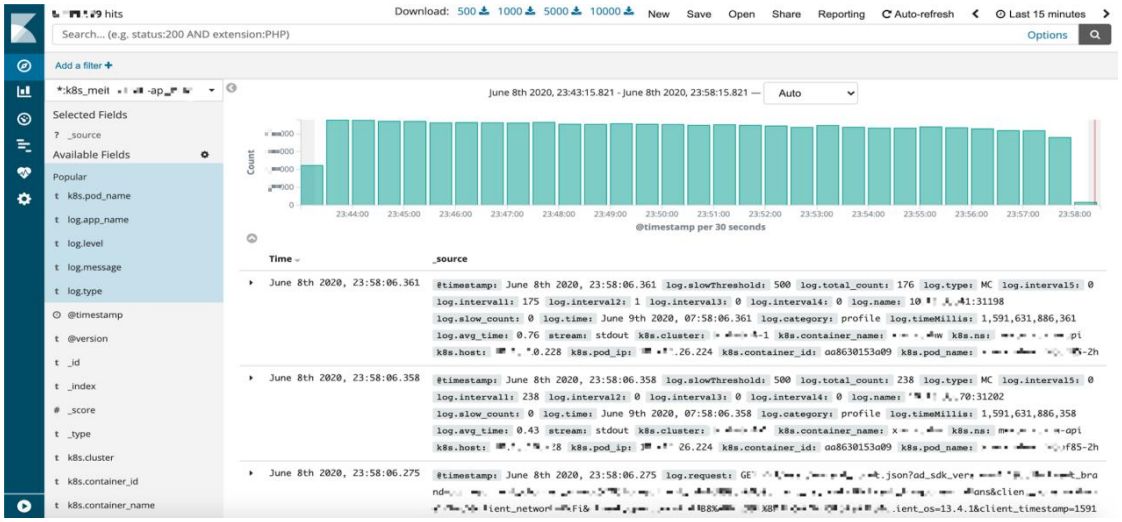
左图是一个流量突变的告警，可能是大家常见到的一种，通过这种文字的手段把报警发给你。这里有一个点需要注意，突降 30%，它其实是不同于常规那种告警，说目前的某个指标超出某个值或者低于某个值，或达到什么水平线那种阈值的告警。两者区别在于，突降是通过对比值得出的结论。这个示例说明在做监控报警时，可能需要考虑的建设点，想想是否有这类需求，有没有这类场景。

最右面的图看起来比较酷，它是怎么实现的呢？它的意义是什么？对比前面这两张文字图，文字图只是告知流量突降 30%，这是一种告警方式。中间的图就是由名为"监控大盘"的机器人发送出来的，也是我们服务的架构图。放大图看，我们的业务模块、交互链路已经呈现出来了；图中有一块是红色的，红色代表模块异常，与异常模块关联的某些组件的 SLA 降低到有个阈值，就变成淡绿色。

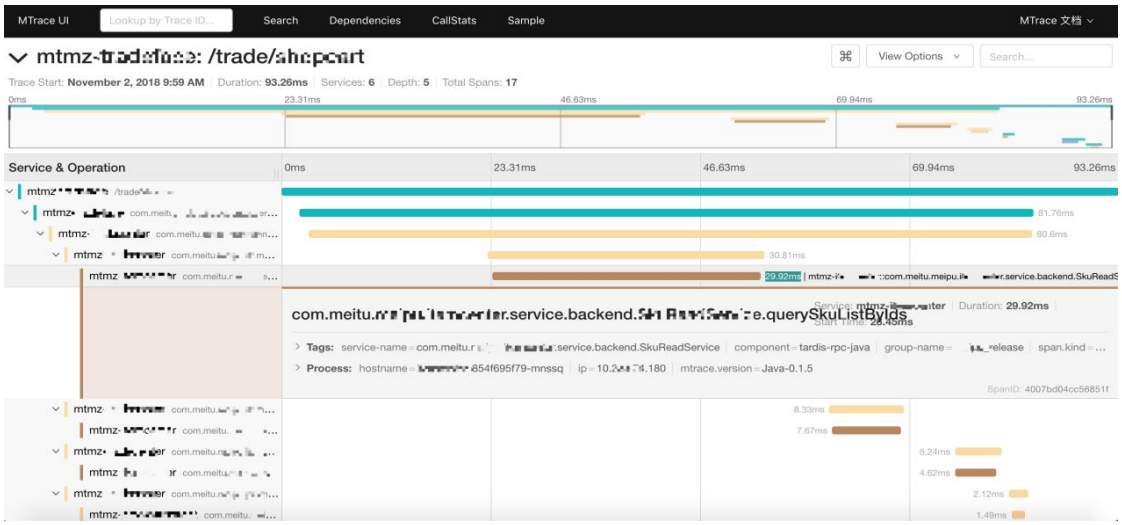
由于红色这块故障引发了另外一个异常，接连引发了一个又一个的异常。通过这张告警图，收获了更多的信息：首先知道系统挂了，或者系统异常了，然后这次异常大概是由什么引起的。一张图就可以让你非常快速地感知。现在有了这张图，不再需要文字告警来通知我组件异常，也不用翻找过往经验做排查。只要看图就能结合已有的监控系统直接做分析、排查，最终得出结论。后者这种监控手段就是让故障背后的原因甚至根因更快触达到用户。

这是基于 grafana 的插件 flowcharting 基于 draw.io 的一个绘图，结合我们的数据填充和告警配置来实现的。

2. 日志分析



3. 故障中链路跟踪



4. 预案执行

故障降级执行方案

程度	故障层次	故障波及范围	降级操作	降级对业务影响功能	恢复操作
严重 ↓ 轻微	区域 Region	区域整体不可用	1. 域名解析：移除 x 内外网域名对于 京 的解析 2. 后端资源： a. MySQL & Redis 写切集 b. 停止 数据同步 3. 集群配置： 全部组件移除 集群配置（kubectl 操作） 4. 镜像库配置：将镜像信息获取源切至 （目前为局部有容灾需求业务直接同步）	1. 群全部发布行为冻结，已有发布不会被修改 2. 独立部署 群的容灾服务操作不受影响 3. 对于多区域部署的容灾服务 a. 配置修改会仅对 集群生效，两区域集群配置会不一致 b. 会将所有实例份额调配 群，且不影响 集群	1. 后端资源： a. 开启 数据同步 b. 同步完成后切 c. 开启 数据同步 2. 集群配置： atrix 全部组件加回, 集群配置 3. 镜像库配置：将镜像信息获取源 4. 域名解析：恢复 Matrix 内外网域名的 解析（Gitlab CI 重新运行 release environment 的最后一次 job） 5. 确认故障期间业务配置差异恢复
		区域整体不可用	1. 集群配置： 部组件移 集群配置（kubectl 操作）	1. 群全部发布行为冻结，已有发布不会被修改 2. 独立部署 的容灾服务操作不受影响 3. 对于多区域部署的容灾服务 a. 配置修改会仅 群生效，两区域集群配置会不一致 b. 会将 实例份额调配到 群，且不影响 集群	1. 后端资源：确认 易数据同步情况 2. 集群配置： 全部组件加回 置（Gitlab CI 重新运行 release environment 的最后一次 job） 3. 确认故障期间业务配置差异恢复
	可用区 Available Zone	单个可用区故障 （ 单个 k8s 集群）	1. 移除 的故障集群配置	1. 故障集群全部发布行为冻结 2. 对于多区域部署的容灾服务，涉及故障集群： a. 配置修改会仅对正常集群生效，区域配置会有差异 b. 会将故障集群实例份额调配到百分比最高集群，且不影响故障机群	1. 集群配置： 全部组件加 群配置（Gitlab CI 重新运行 release environment 的最后一次 job） 2. 确认故障期间业务配置差异恢复
	节点池 Pool / 节点 Node	部分节点故障	1. 运维下线异常节点（与 ix 无关） 2. 根据实际故障情况限制全体或部分操作，修改 Gitlab CI/CD Variables 中维护相关的环境变量，运行对应环境 environment 的最后一次 Job，具体环境变量配置详见 维护模式 MR	1. 按照维护模式的配置限制全体或部分（按照接口或用户权限等规则设定）的操作	1. 将 Gitlab CI/CD Variables 中维护相关的环境变量置为 ""，运行对应环境 environment 的最后一次 Job 恢复

日志分析、链路跟踪、预案执行都是定位手段，在已经定位问题之后，并且匹配了相应的预案，要求去执行预案。当然前提是有相应的预案应对故障场景。然后依据操作手册，分别按不同的故障层次去执行处理。

恢复之后还要确认执行结果，看服务是否恢复正常。这是我们在故障处理中实际案例的截图。下面进入非常关键的故障后环节。

5. 故障处理的一些原则/建议

- 统一目标：恢复优先，问题定界 > 根因定位
- 稳定心态：SRE 一定要冷静，不要慌
- 流程机制：故障升级、War Room
- 故障现场：组织协调约定、信息通报机制

故障后

大家要重视一点，在故障发生和解决以后，以为问题真的结束了吗？其实并没有。在故障后，我们应该做好以下环节：故障复盘、故障改进、预案完善、容量压测、故障模拟、周边清查。我解释下其中几个概念：

- 预案完善：故障发生之前有做过预案，这些预案是不是完善的？在故障处理的过程中，有没有暴露出问题？是否要完善之前的预案中做得不完善的地方等等；
- 故障模拟：意思是用某种手段模拟某些故障，提前知道该如何解决这些问题；
- 周边清查：应该具备举一反三的能力，比如 A 服务出故障，那么 A 服务周边有一些相关联的服务有可能会受到影响。举个例子，比如集群里面有 N 台机器，收到机器 A 磁盘接近 100%的一条告警，只需处理机器 A 就可以了吗？当然不是，在处理完该异常之后，肯定要查看同集群的其他机器是否存在类似情况。

1. 故障复盘

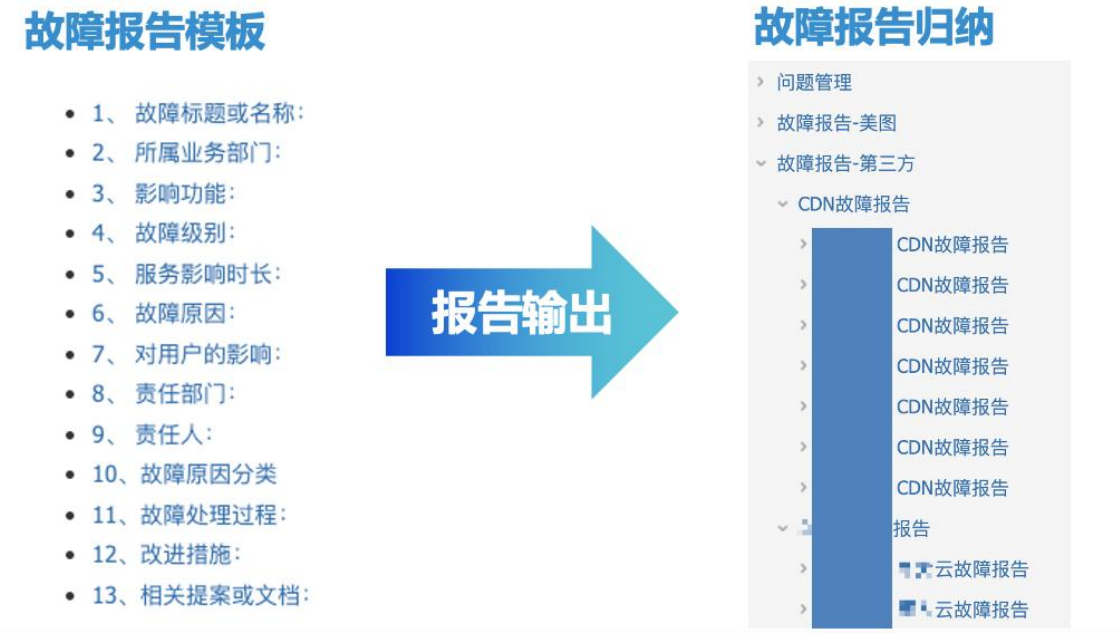
在故障复盘阶段，最核心的思想就是要回顾关键的时间点：故障是从什么时间发生的？从什么时间到什么时间结束的？在这个过程里面有哪些非常关键的操作？有哪些关键的信息？从什么时间点定位到问题？在什么时候执行怎样的操作？这些时间点都是非常重要的。

在故障复盘的过程中，要把这些操作的时间点一一还原回去，才能完整地回看操作是否合理有效。看在某个时间点执行了某个预案，思考某个时刻做出这样的操作是否恰当，此时有没有其他更好的手段能够恢复服务，某个时间段为什么没有及时处理而导致故障时间变长等等。

做好故障复盘，我们有黄金三问法：

- 我们应该怎么做，才能更快地恢复业务？
- 我们应该怎么做，才能避免再次出现类似问题？
- 我们有哪些好的经验可以总结、提炼，并固化？

基于以上三个问题，通过自问自答的形式弄清故障复盘。

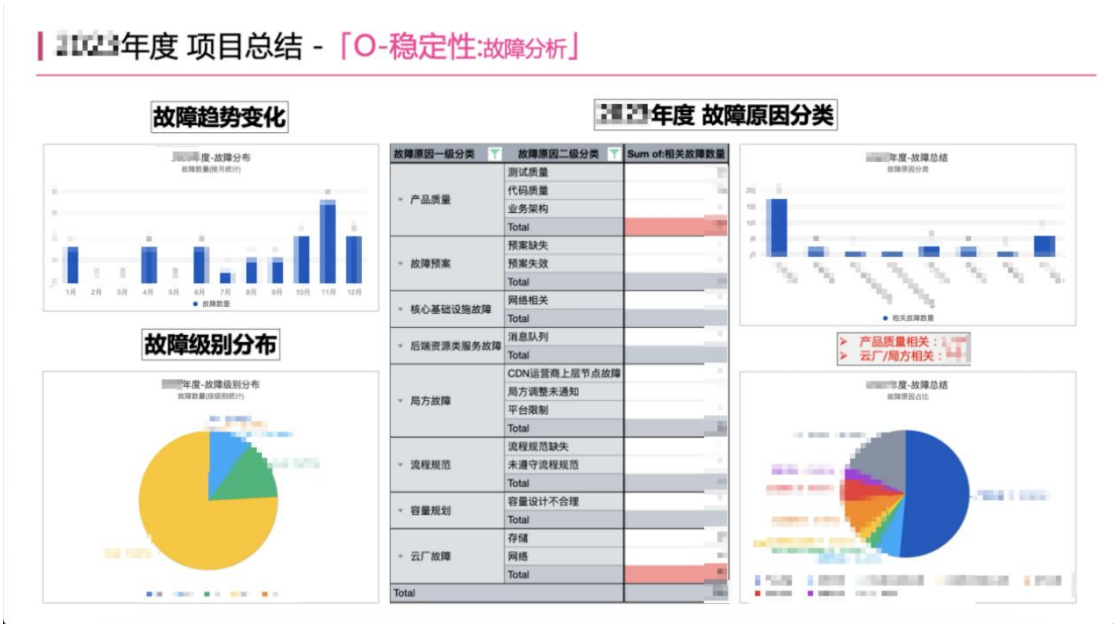


这是故障解决后绕不开的一点，在故障发生后都会做一个报告，上图就是我们内部的一个故障报告模板。在报告里面写清楚故障级别、责任人、处理过程、改进措施，将这些关键点归纳成故障报告的文档。

- 周期回顾

如前文所述，故障管理的工作本质上是持续迭代的、稳定性保障能力也是螺旋式上升的。

除了做好故障前中后的各块工作，我们还需要周期性地对稳定性保障工作做汇总分析、复盘总结，下面就是我们做 OKR 复盘的一个实例。



(三)总结及展望

我所看到的几个技术趋势

根据当前行业的发展趋势，我们可以看到以下几个重要方向：

1. 云原生：云原生技术将会持续发展和深化。越来越多的企业在采用云原生架构，云原生的优势在于它的灵活性和可扩展性，这些特性将帮助企业更好地应对不断变化的市场需求和技术挑战。

2. 可观测性：可观测性工具和实践正在被更多企业认同和应用。通过提高系统的可观测性，运维团队能够更迅速地发现和解决问题，提升系统的可靠性和性能。

3. 混沌工程：混沌工程在越来越多的场景中落地实施。通过故意制造故障并观察系统的响应，企业可以提高系统的抗压能力和容错性，从而提升整体服务的稳定性。

4. Dev-X-Ops：各类 Ops（例如 GitOps、ChatOps、AIOps 等）融合类的实践越来越多，不论是什么 Ops，实用性才是最重要的。只要这些 Ops 实践能解决实际问题、提升工作效率，它们就值得被采用。

5. AIOps：AIOps 正在持续进化，并且其应用范围也在不断扩大。随着大模型和人工智能技术的快速发展，AIOps 可以帮助企业更好地处理复杂的运维任务，实现更智能化的运维管理。

综上所述，SRE 未来的展望主要集中在持续创新和应用实用性强的新技术和新方法上，以提升系统的可靠性、可维护性和智能化水平。

最后的话：如何面对汹涌的技术浪潮

面对汹涌而来的技术浪潮，作为 SRE 从业人员，如何应对这一变化迅速的环境？以下几点可以帮助我们在技术变革中保持镇静和从容：

1. 看清本质：理解每种技术的核心价值，不被表面的复杂性所迷惑，找到其中的本质，并以此为基础进行实践。

2. 拥抱变化：接纳并适应新技术和新趋势，例如 AI 技术的广泛应用。顺应技术发展的趋势，积极学习和应用新技术，以保持竞争力。

3. 顺势而为：在理解和接受变化的同时，合理运用这些新技术来提升工作效率和质量。在技术变革中找到适合自己的位置，并发挥最大的价值。

4. 做好定位：明确自身的角色和价值，知道自己能提供什么样的价值，应该提供什么样的价值。持续改进和提升自己的技能，以确保能够稳定地、高效地提供高质量的服务。

通过这四个方面的努力，我们可以在技术浪潮中泰然自若，迎接未来的挑战，而不用担心被新技术所取代。保持积极的态度，拥抱变化，不断提升自己的能力，这样才能在不断变化的技术环境中立于不败之地。

本章节根据 2025 年 04 年 10 日 Qcon 上演讲，补充如下;

补充总结及展望

故障管理框架可概括为一个闭环的持续改进流程：系统在稳定运行阶段进入常态监测；当故障发生后迅速启动应急处置并恢复故障恢复；随后开展故障复盘与周期回顾，梳理成因与改进措施；最后落实改进验证并重新回到稳定态，形成周而复始的演进路径。该流程由四大治理支柱支撑：

- 可用性体系（SLIs/SLO/SLA）为目标设定与衡量提供量化基线；
- 故障分级、定性与责任划分确保响应优先级和问责清晰
- 错误预算与故障分通过量化容错与激励约束平衡创新与稳定性
- 组织结构支撑及故障委员会为跨团队协同、资源调度与策略制定提供制度保障。

多年来的实践表明，利用这一循环与支柱相结合，可系统化提炼 SLI、SLO、SLA 等可用性标准，贯通预算管理与经济责任分配，实现从预防到复盘的全链路故障治理。

我所看到的几个技术趋势

趋势	关键词	关键价值	典型演进方向
1. 云原生	基石	持续解耦 & 弹性扩展	K8s 全生命周期管理、Serverless、无状态优先架构
2. 可观测性	一体化	度量+追踪+日志 融合，数据闭环驱动优化	OpenTelemetry 标准化、AIOps 异常检测、体验指标（UXI）
3. LLM Ops	智能辅助	SRE 引入 LLM 做根因分析、智能决策	· 事件语义解析 · 自然语言变更评审 · Prompt 版本管理
4. AI Agent	自愈闭环	SRE Agent 主动故障响应、自适应扩容	· 多 Agent 协作协议 · 经验知识图谱 · 终态自愈策略
5. MCP / ANP	协议化	"上下一致"运维控制协议 & 多租户纳管	· MCP：层次化闭环指挥 · ANP：跨租户策略编排
6. AI 可信系统	安全透明	可解释、合规、伦理可审计	· 责任链追溯 · 攻击面评估 · 数字水印与模型偏差监测

云原生 → 可观测性：弹性基础奠定了统一数据采集与实时洞察的可能。

可观测性 → LLM Ops：高质量可观测数据为 LLM 提供上下文，驱动智能诊断。

LLM Ops → AI Agent：LLM 赋能的知识推理成为 Agent 自愈的"决策大脑"。

AI Agent → MCP/ANP：多 Agent 协同需要协议化编排，形成运维"控制面"。

MCP/ANP → AI 可信系统：协议化闭环与合规治理共同确保 AI 可靠、安全、可解释。

一句话总结：这一套路径以云原生基础设施为根，以可观测数据为血液，以 LLM 与多 Agent 为智能中枢，通过 MCP/ANP 协议实现跨环境协同，最终落到可解释、安全、合规的 AI 可信体系，实现运维与智能的深度融合与持续演进。

最后的话：如何面对汹涌的技术浪潮

看清本质　拥抱变化　顺势而为 做好定位　葆有价值　泰然自若**认清浪潮：云原生、微服务、Serverless、服务网格、可观测性、低/零代码、平台工程、AI Agent、Git-Chat-Biz-Dev-Sec-Fin-Green-AI Ops……新名词层出不穷，本质仍是「用更高抽象的工程体系，把业务价值交付得更快、更稳。**

价值优先：别被概念牵着走，始终围绕企业与客户的核心痛点评估 ROI——能提效、降本、控风险，才值得投入。

定位清晰：问自己"我的独特位置是什么？我能在这波技术中提供怎样不可替代的价值？"——找准专长，深耕垂类，而非全盘追逐。

小步试错：先做可验证的小实验（PoC / MVP），用数据说话；成功即滚动扩展，失败则快速止损，保持技术投入的弹性。

沉淀能力：把一次次试验中验证有效的方法论、工具链、最佳实践固化成平台化能力，让创新成为可复用、可迭代的"流水线"。

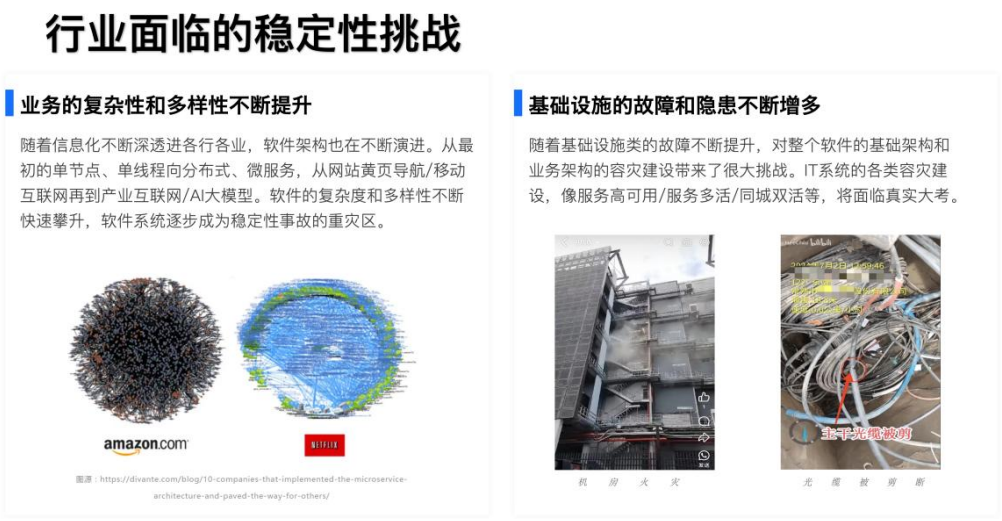
心态从容：技术迭代永无止境。保持学习曲线的同时，也要守住交付与运营的基本盘，既在浪尖上冲浪，也能随时回到稳健的航道。

5.3.6 B 站轻量级容灾演练体系构建

SRE Elite 精选原因

B 站的轻量级容灾演练体系构建聚焦于高效、低成本地确保系统稳定性。首先，通过建立原子级故障库和特化场景，演练对象和故障类型变得标准化，易于操作。演练体系涵盖技术支撑、跨领域赋能、组织文化及流程优化，以实现常态化、自动化的演练环境。其核心亮点包括：简化故障注入流程、丰富的观测能力、自动化参数计算、标准化的演练协议，以及跨部门的协同机制。演练的目标是通过自动化工具支持，降低操作复杂度，确保演练可以在没有 SRE 人员的情况下自主管理，最终推动全员参与的容灾演练文化，增强业务系统的抗压能力和恢复能力。此实践考虑充分，方案完整，工程化实践扎实，在业界中处于较为领先的地位，具备很强的参考意义。

（一）背景及设计原则



首先，当前新形势下的稳定性挑战日益严峻。随着 IT 的发展，信息化不断渗透到各个行业。从几十年前的单节点巨石服务，到如今的分布式、微服务和云原生架构，软件的复杂度和多样性都在几何倍数地攀升。这导致软件系统逐渐成为稳定性事故的重灾区。不单软件，硬件问题也变得更加复杂和严重，基础设施的故障和隐患也在不断增加，并且显露出更大规模的影响。

例如，前两个月新加坡某机房发生火灾，虽然仅一层受损，但对国内部分出海业务仍造成了重大影响。再如，7 月 2 日常熟光纤被剪事件，尽管光纤只被部分切断，但不巧阿里云和我们公司都在受影响的那一段，导致严重后果。

这些事件表明，当前基础设施故障带来的挑战已不再局限于单台设备的问题，而是可能导致整个机房或网络的瘫痪，进而引发大范围服务甚至整个站点的不可用。

此外，行业内对高可用服务的建设要求越来越高。平时可以吹嘘高可用服务的能力，但当真正面临外部考验时，能否经受住考验才是关键。

对于我们 B 站来说，面临的核心挑战包括：

B 站热搜体质，小故障大流量

由于微博有热搜机制，而 B 站的主要用户群体是年轻人，导致很多时候我们虽然遇到的故障并不大，但很快就会被推上微博的热搜。这种情况给我们负责稳定性和技术的同事带来了巨大的压力。

以下是一些相关案例：

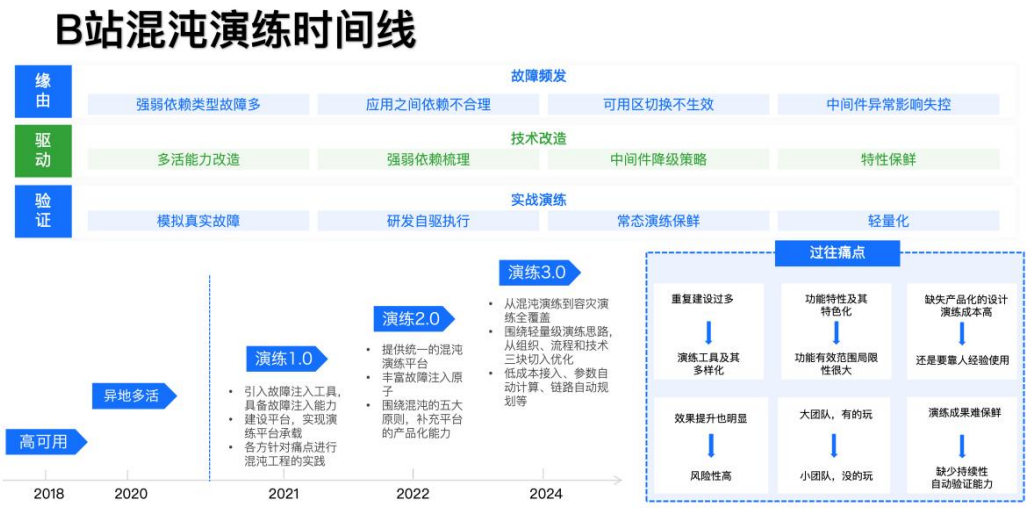
- 商业 CDN 故障引发的级联效应影响全链路的图片服务；
- IDC 网络设备故障导致专线异常，影响办公区网络；
- 业务代码变更导致数据库问题；
- 光缆被剪断导致可用区脱网等。

持续多活建设，有效性急需验收

在过去的几年中，众所周知，B 站遭遇了一次严重的系统故障——即所谓的"713 事件"。随后，我们内部启动了一系列专项工作，针对用户端、接入层、服务层、中间件层以及基础设施的各个层面，实施了多活技术支撑建设和业务架构的全面改造。

尽管改造工作已经进行了多年，但其成效如何，我们仍感到不确定。即便是负责此项改造的团队，可能也只是依据内部的多活白皮书和技术架构指导方案进行了部分改造工作，而这些改造是否真正达到了预期效果，尚需进一步验证。为了确保这些改造措施的有效性，我们迫切需要一个专门的平台来进行测试。这也是我们启动改造的初衷——频繁发生的故障迫使我们进行必要的改造，并在改造完成后，需要一个平台来验证其功能是否达标。

因此，我们需要对强弱依赖故障、应用依赖不当的故障、可用区切换的有效性、技术改造的推动效果、多活改造的推动效果，以及降级策略和兜底策略的实施效果进行严格的测试和验证。



为了评估这些改进措施的实际效果，我们有必要开展实战演练。在这些活动中，我们将在真实的工作环境中模拟实际问题，并鼓励研发团队自主驱动演练。鉴于保障团队的人力资源有限，若仅在故障发生后进行专项验收，可能无法实现持久的效果。因此，我们期望能够将演练常态化，以降低费用并维持效果。这便引出了轻量化演练的议题。

自 2018 年起，B 站便开始着手高可用性建设，而到了 2024 年，我们进一步推进了异地多活和双活架构的建设。

在 2021 年的演练 1.0 阶段，

我们采纳了多种开源混沌工程工具，包括来自阿里和 Pingcap 的工具，并在此基础上构建了一个简易平台，用于故障注入和验证。

然而，由于各部门独立运作，我们面临了一系列问题。首先是重复建设，不同部门开发了多种功能相似的工具。其次，由于各业务部门的关注点不同，工具的设计和使用缺乏统一标准。此外，技术团队开发的平台虽然比手动执行命令有所进步，但其效果并不突出。

尽管混沌演练的引入提升了效果，但同时也增加了风险。研发人员在功能验证时可能会使用这些工具进行测试，但工具的局限性有时会引发真实的线上故障。混沌工程并非随意注入故障，而是需要遵循逻辑和方法论。

大团队拥有充足的人力资源进行演练保障，而小团队则资源匮乏，难以获得同样的关注。通常，大家只在重大活动前进行专项演练，而日常演练缺乏持续性，难以实现常态化。

为了解决这些问题，**我们在 2022 年的演练 2.0 阶段**

，内部发起了一项统一行动，整合了大型事业部的资源，建立了一个统一的平台。我们增强了故障注入的能力，简化了公共部分，并根据混沌工程的五大原则指导产品化设计。以下是关于 Netflix 提出的五大原则：

- 建立稳定状态的假设；
- 用多样的现实世界事件做验证；
- 在生产环境中进行实验；
- 自动化实验以持续运行；
- 控制最小化爆炸半径。

演练 3.0 的目标

是希望在公司内部实现演练的常态化和真实有效的运行。为了达成这一目标，我们计划将演练从混沌演练逐步扩展到容灾演练，最终实现全面覆盖。

在前期工作的基础上，我们从去年开始推进演练 3.0，旨在使演练在公司内**实现常态化、真实有效地运行**。为此，我们提出了从混沌演练迈向容灾演练的目标，实现全覆盖。围绕常态化演练，我们引入了轻量级演练的思路，从**组织、流程、技术**三方面入手，全面优化。

我们不仅对平台进行改造，还关注以下关键点：

- **低成本覆盖与接入：**

如何以更低的成本实现广泛的演练覆盖和系统接入。

- **自动化参数计算与链路选择：**

在演练过程中，针对大量的输入参数，如何实现自动计算；如何自动圈选演练链路等功能。

这些需求引出了演练 3.0 阶段的目标——**轻量化演练**，这是我们持续一年多并仍在推进的工作。

为实现轻量化目标，我们在流程方面进行了优化。由于需要在全公司范围内推广，涉及的部门和角色众多，**如何高效协作成为关键**。这也呼应了之前提到的团队协作问题：如果团队协同不到位、职责不清，可能导致各自为政，影响整体进度。因此，我们需要**固化组织的演练流程和章程**，明确需要开展的重大演练，倒排计划，确定各部门在不同阶段的参与方式（后续会有详细的图示）。

其次，**技术流程**

也需要标准化和固化，建立**标准化的演练过程**

。在具体的技术支撑方面，我们需要强化产品能力。作为平台工程部门，我们思考如何减少人员的认知负担。只有将演练能力完全交付出去，**例如无需 SRE 人员陪同，业务方可以自行开展演练，这才是我们的最终目标。**

围绕演练过程，我们：

- **丰富观测指标能力**

：拓展可监测的指标，增强对演练效果的评估。

- **扩展演练对象和范围：**

增加可注入故障的演练对象，丰富演练场景。

- **实现自动化：**

在脱离人工操作的情况下，对演练的预期结果进行指标量化定义，使演练过程能够自动化地收集结果。

- **建立护栏机制：**

当出现问题时，自动防护，确保演练不会引入新的故障。

第三，在组织协同方面，我们建立了各职能、各业务的协同机制，提升内部对演练的认知。

演练并非"没事找事"，而是有实际收益的工作，需要大家重视。

此外，我们还需要进行培训，降低学习成本。即使产品体验再好，复杂的系统仍需一定的学习，因此相关的培训是必要的。

基于以上，我们提出了几个重要目标：

- 平台角色定位：希望新版本的平台能够扮演演练专家的角色，极大降低用户的使用成本。
- 规范演练行为：基于标准的流程，规范演练的行为和操作。
- 强化安全性：严格控制影响半径，确保演练足够可靠、可控。
- 推进常态化演练：这是我们的最终目标，如果无法实现常态化演练，前面的工作将失去意义。

（二）体系设计详情

轻量级容灾演练体系

轻量级容灾演练体系

接下来，我们引出了**轻量级容灾演练体系**

，整个体系分为四大块。

一、技术支撑

1. 原子级故障库：在底层，我们建立了一系列原子级的故障库，包括模拟 CPU 异常、网络延迟、丢包等简单故障注入。

2. 场景封装

- 一般场景：封装常见的故障场景，如 CPU 高负载、容器异常、网络丢包等，方便用户直接选择和使用。
- 特化场景：针对公司内部业务特性，如双活、多活架构和自研中间件，定向封装故障能力，供内部团队使用。

3. 混沌实验构建：在上述场景之上，我们构建混沌实验，包括实验的创建、分发以及故障注入到具体载体上。

4. 观测能力：引入观测机制，评估故障注入效果，验证系统恢复情况，并有效监测业务稳态指标，及时发现业务异常。

5. 经验沉淀

- 故障组合与复杂场景模拟：通过组合多个故障，模拟复杂场景，如网关故障、SLB 故障、跨网机房级别的故障等。

- 演练模板：将上述经验沉淀为演练模板，服务团队在演练时只需选择对象和范围，配置业务稳态指标，无需关心故障设计细节，降低了演练门槛。

6. 产品层面的演练管理：在日常使用中，通过产品化的演练管理来编排故障，并生成演练报告。整体路线是从工具到平台，再到一站式服务，最终实现轻量化演练的目标。

二、技术落地

1. 研发过程融合：将故障注入能力融入 CI 流程和测试回归流程，特别是对于强弱依赖的演练，应在制品库出包前的准出阶段自动模拟故障，检测是否存在问题。

2. 传统混沌演练：继续开展系统故障、网络故障以及内部特定场景的演练。

3. 大型容灾演练：针对跨机房、多活架构的容灾演练，以及降级和兜底方案的验证。

三、跨领域赋能

1. 应急响应：在应急响应过程中，利用平台进行演练溯源和故障定位。

2. 攻防演练：模拟故障链路，随机注入故障，提升系统的抗压能力。

3. 大模型赋能：借助内部大模型团队的能力，对演练内容进行复盘和梳理，基于历史应急问题，识别薄弱环节，建议针对性演练。

四、组织文化

组织架构支撑：调整组织架构，建立虚拟项目团队，确保演练协同目标的实现。

制度建设与文化宣贯：制定演练制度，日常宣导，提高全体技术团队的安全容灾意识、演练防范意识和应急响应意识。

我们基于组织、流程、技术三个方面展开工作。认识到组织的重要性，如果组织协同不到位、责任不清，工作将难以推进。通过明确负责人与职责，确保每个人都积极参与并承担相应责任，从而提高演练的效果和效率。

组织支撑

在组织支撑方面，我们建立了**安全生产委员会。**

该委员会由一线领导组成，在组织中担任重要角色，负责公司全局性的稳定性决策和重大稳定性演练路线的规划。他们还承担立法职责，制定惩罚规则，建立基线和红线。此外，委员会致力于提升公司整体的应急协同能力和文化意识水平，指导全局的管控、容灾演练系统的规划管理，以及各个子系统之间的协调。制度建设方面，则包括编写相关的白皮书。

在**制度建设部分**，我们规范了容灾架构设计，制定了容灾演练章程和故障演练规范。

在**文化宣导方面**，每年都会举办为期一个月的"安全生产月"活动。在每个楼层设置宣传展板，开展一天的公司专场活动。各部门，如机架部门和各业务线，会分享他们在稳定性改进方面的工作。此外，我们通过学院培训，录制视频，方便大家随时学习。对于校招和新入职的同事，入职培训中也会特别关注这些内容。

"孤岛行动"这是我们近期开展并将持续推进的工作，即进行机房级别的断网测试。例如，本周二我们就进行了断网演练。通过这种真实的专项活动，推动公司各业务线主动检查并改进自身的问题，确保保障措施到位。这是一种基于目标牵引的方式，促进工作的开展。

此外，我们还通过**技术分享渠道**

，如哔哩哔哩技术公众号，每周发布两篇文章，分享各技术线的实践内容。内部也有相应的平台发布技术实践，供大家学习和参考。

流程支撑

在组织支撑之外，我们还需要流程支撑，主要分为两部分：**演练执行流程和演练全生命周期流程**。梳理这些流程的目的是指导业务更高效地开展日常实验演练，并帮助平台更有效地设计功能，避免因主观设计导致各部门使用不便。

演练执行流程

演练执行流程旨在指导我们如何开展一次实验演练，分为演练前、演练中和演练后。

演练前

- 业务梳理：确定业务场景，明确哪些场景需要演练，制定验证方案，评估演练可能带来的影响。
- 方案确认：决定演练方式，如采用染色流量演练、线上不切流演练，或先切流再注入故障进行验证。

演练中

- 制定演练范围和环境：确定涉及的集群和机器范围。
- 配置规则：模拟故障的业务场景，选择需要观测的指标，设计护栏机制，确保出现问题时能够及时拦截并恢复故障。

- 验证场景：通过指标和断言，验证演练是否符合预期。这部分主要支撑自动化演练。

演练恢复

- 确保故障恢复后，业务状态符合预期。关注故障撤销后业务是否仍存在异常，确保其真正有效恢复。

演练后

- 常态化演练计划：评估本次演练场景能否常态化，制定未来演练计划，包括突袭演练等。
- 待办事项推进：针对演练过程中产生的待办事项，安排时间推进，并通知相关负责人。

演练全生命周期流程

演练执行流程只是一次性的，但我们的目标是根据"五大原则"，在真实环境中模拟真实场景。因此，通常先在非生产环境进行演练，再逐步过渡到生产环境。

非生产环境演练

- 在非生产环境（如 UAT）中执行演练流程，发现并修复重大故障和隐患。
    - 流量处理较为简单，可进行跨网环境的演练，可能不需要染色。

生产环境演练

- 需考虑是否进行染色，以及染色的粒度（实例级别或请求级别）。
    - 是否需要提前预切流量，切流后在一侧注入故障进行验证。
    - 验证与恢复阶段：需要联动应急场景，处理可能引发的故障，并按照相应流程进行恢复。

推进常态化与突袭演练

- 针对服务的各个重要业务场景，开展全生命周期的演练。
- 制定周期性的突袭演练计划，确保业务在各种情况下都能稳定运行。

技术支撑

技术支撑

在有了组织支撑和流程标准的支撑之后，我们需要落实技术手段，我们整体梳理了六个"化"，以支持组织协同和流程标准的有效落地，确保轻量化演练目标能够保质、保量、保效地开展。这六个"化"包括：**演练协议的标准化：**

使用统一的演练协议，保障所有的混沌操作统一控制**故障引擎的自研化：**

自行研发故障注入引擎，以满足特定需求，提高灵活性和可控性。

实验参数的精简化：

简化实验参数，降低复杂度，使演练过程更高效、更易于执行。

演练覆盖的全面化：

扩大演练的覆盖范围，确保所有关键业务场景都能被演练到，提升整体系统的可靠性。

演练过程和护栏的可视化：

对演练过程和保护机制进行可视化展示，方便实时监控和管理，确保在演练过程中能够及时发现和处理问题。

演练流程的自动化：

最终实现演练流程的自动化，减少人为干预，提高效率和准确性，保障演练工作的持续推进。

演练协议标准化

首先，我们强调了流程的标准化，进而提出演练协议也需要标准化。最初，我们的项目基于开源工具，主要使用了阿里巴巴的 ChaosBlade，这在前期为我们节省了大量的故障原子的开发成本。然而，在与行业内的交流中，我们发现一些公司声称其故障引擎完全自研，但实际上可能也是在他人代码的基础上修改而来。

与此同时，我们内部各部门也已经开发并封装了一些演练工具。这导致了一个问题：我们并非从零开始，而是需要整合许多内外部的演练工具。然而，由于团队不同，大家在技术设计上各有想法，导致工具的命令行参数、接口协议的抽象（如 Proto 文件）都不一致，尽管功能类似。

为了解决这一问题，我们需要将这些工具进行整合和统一，主要通过模型抽象。具体而言，我们提出了两个标准：**1. 标准的演练描述模型 演练对象：**

明确要演练的目标是什么。

演练范围：

确定演练涉及的范围。

载体类型：

演练范围内的载体类型，例如是否包含 IP、端口等信息。

演练参数：

例如在网络故障演练中，需要哪些参数，这些参数的格式和内容需要统一抽象。

演练模式：

包括是否有预检要求，场景是否符合故障注入的条件等。

2. 标准的演练交互模型 实验的注入和注册：

统一实验的启动和登记过程。

实验的执行和恢复：

标准化执行故障和恢复故障的流程。

审计和查询：

建立统一的审计和查询机制，记录指令转换、执行结果等信息，方便监控和排查问题。

通过使用类似 Golang 的接口进行行为抽象，我们确保了系统的统一性和兼容性。这样，我们就能够在这套系统中快速兼容原有的多种演练工具，逐步引导它们过渡到我们新的标准化工具。这种标准化不仅有助于提高演练的效率和可靠性，还为未来的扩展和维护奠定了坚实的基础。

故障引擎自研化

在完成标准化之后，我们进一步推进了故障引擎的自研化，主要目在于分享我们在混沌实验过程中使用开源工具时遇到的问题，以及自研所带来的改进。之前，我们大量使用了开源工具，但在实际应用中遇到了不少挑战，因此决定进行自研。

我们选择**自研的主要原因**

包括：

1. 提高演练成功率：开源故障注入工具的可靠性不如预期，降低了我们的演练成功率。虽然代码实现质量一般可以接受，但成功率的下降对我们影响较大，因为团队需要对演练指标负责。

以下是一些具体的问题案例：**DNS 干扰演练的问题：**

故障注入逻辑无效，导致故障逃逸：开源工具在进行 DNS 干扰演练时，通常直接在 /etc/hosts 文件的尾部添加一条新的域名解析记录来覆盖目标域名。然而，这种方式在现代操作系统和应用中并不可靠。

原因分析：

许多应用程序在进行 DNS 解析时，会按照顺序查找 /etc/hosts 文件中的记录。如果前面已经有一条匹配的域名解析记录，后面新增的记录可能不会被应用程序使用。这导致故障注入逻辑失效，故障被逃避。

影响：

演练过程中预期的 DNS 故障没有发生，无法验证系统在 DNS 解析异常情况下的稳健性。

网络不可用故障的问题：

注入命令过长，导致执行失败：当需要对大量（如数百个）实例注入网络不可用故障时，开源工具生成的命令行参数会过长，超过了系统对命令长度的限制，导致命令无法执行。

原因分析：

开源工具在注入网络故障时，可能会为每个目标实例生成一条命令，当目标实例数量过多时，命令行参数累积，导致命令长度超出系统限制。

影响：

无法对大规模的实例集群进行网络不可用故障的注入，限制了演练的覆盖范围和规模。

通过 TC（Traffic Control）实现的网络故障的问题：

使用 TC 工具来实现网络不可用故障时，故障恢复后网络状态可能仍不稳定，或者恢复过程出现异常。

原因分析：

TC 工具在网络故障注入和恢复的过程中，可能未能完全清除或还原网络配置，导致残留的规则影响网络性能。

影响：

即使故障注入结束，系统网络可能仍然受到影响，导致演练对生产环境造成不必要的干扰，增加了演练的风险。

CPU 负载注入的问题：

生效步长和节奏不匹配，导致负载未达预期：在模拟 CPU 高负载时，开源工具的故障注入逻辑在负载递增的步长和节奏上与我们的预期不一致。

原因分析：

可能是开源工具在实现 CPU 负载递增时，使用了固定的线程或进程数量，未能根据实际 CPU 核心数和系统性能进行动态调整。

影响：

无法准确模拟预期的 CPU 高负载情况，导致演练结果不准确，影响对系统性能和稳定性的评估。

- **改进故障注入链路的可靠性：**

开源混沌平台对故障注入的编排较为简单，类似于 HTTP 无状态请求。故障注入链路中每个节点之间缺乏状态管理，如果中间某个节点失败，整个链路会中断，难以回溯和排查问题。另外缺少状态上报和控制，在故障编排过程中，如果某个步骤出现问题，系统无法有效地上报状态，也无法进行控制和补救，导致演练过程断层。

3. 加强服务/资源鉴权，增加演练安全性：

开源项目无法满足我们对服务和资源鉴权的需求，注入对象过于裸露，无法与企业内部的服务对象、集群对象匹配，导致演练注入范围的权限控制存在问题。

安全风险：缺乏对演练注入范围的严格权限控制，可能导致对非预期目标的故障注入，带来安全和稳定性风险。

4. 增强可观测性，方便问题排查：

我们无法在未知情况下进行故障注入，因此需要对注入过程进行全面观测。

问题难以排查：故障注入的组件链路较长，包括用户请求、故障调度、故障注入代理，以及可能的 Kubernetes 调度。缺少对指令转换、执行结果的记录和审计，一旦出现问题，难以及时定位和解决。

缺乏指标埋点和状态上报：在演练过程中，针对整体实验、单个故障、注入对象等，缺乏必要的指标埋点和状态上报，无法实时监控演练状态和效果。

基于开源 ChaosBlade 的二次开发和改造

为了解决上述问题，我们对开源的阿里 ChaosBlade 进行了二次开发和改造，主要目标包括：

提升演练注入和过程的控制力度与稳定性：

改进故障注入逻辑：针对 DNS 干扰演练，我们采用更可靠的方法来干扰 DNS 解析，例如修改 DNS 配置或拦截 DNS 请求，确保故障注入有效。

优化命令生成和执行方式：针对网络不可用故障的命令过长问题，我们重新设计了命令生成机制，支持批量注入，避免了系统命令长度限制。

完善故障恢复机制：深入研究 TC 工具的工作原理，改进网络故障的恢复逻辑，确保在故障恢复后，网络配置能够完全还原，不影响系统的正常运行。

动态调整 CPU 负载注入参数：根据实际 CPU 核心数和系统性能，动态调整负载递增的步长和节奏，确保模拟的 CPU 负载符合预期。

扩展演练故障的性能和注入范围的规模化：

支持大规模实例的故障注入：通过优化资源调度和故障注入策略，能够稳定、高效地对大规模实例集群进行故障注入，提升演练的覆盖范围。

提高故障注入的可靠性：减少故障逃逸和注入失败的情况，确保演练成功率。

提升代码可维护性，加速开发迭代效率：

符合内部技术标准：按照公司的技术框架和研发标准进行开发，并接入内部的可观测平台，提高了代码的维护性和开发效率。

建立标准的演练描述模型和交互模型：

演练描述模型：统一描述故障注入的对象、范围、载体类型、参数和模式，确保不同故障类型的定义和扩展具有一致性。

演练交互模型：规范演练过程中的交互方式，包括指令的发送、状态的上报、结果的收集等，确保各组件之间的通信和协作顺畅。

加强服务/资源鉴权：在故障注入过程中，严格控制注入对象和范围，确保只有授权的服务和资源才能被注入故障，提升演练的安全性。

增强可观测性和审计：在故障注入链路的各个环节增加日志记录和指标埋点，实时监控指令转换、执行结果等信息，方便问题的定位和排查。

通过自研，我们不仅解决了开源工具在实际应用中遇到的问题，还为演练体系的持续优化和扩展奠定了坚实基础。自研的故障引擎使我们能够更好地控制演练过程，提高演练成功率，确保演练的安全性和可靠性。

演练覆盖全面化

针对公司内的各种服务，包括小型部门和海外业务，我们致力于实现演练覆盖的全面化。为此，我们采取了以下措施：**1. 对象类型的抽象与管理 抽象载体类型：**

尽管我们的服务架构复杂，拥有数十种类型的集群，但所有服务最终都运行在容器或主机上（无论是物理机还是虚拟机）。因此，我们将演练的载体类型抽象为容器和主机两类，满足大部分故障注入的需求。

依托 CMDB 系统：

利用配置管理数据库（CMDB），我们可以方便地选择目标集群和组件。例如，要对某个大数据集群下的组件进行演练，只需在 CMDB 中勾选相应的组件，便可获取其实例范围。

2. 自动化故障注入与回收 解决接入成本与安全隐患：

面对数十万容器的规模（如 Kubernetes 集群），如果在所有集群中长期驻留故障注入组件，既增加了接入成本，又存在安全风险（如代码 Bug 导致全公司故障）。

动态打标与撤标机制：

我们实现了自动化的打标和撤标机制。故障注入组件仅在演练时自动部署到目标集群（通过 DaemonSet），演练结束后自动回收并撤销标记。这样，即使有命令下发，未被打标的集群也不会执行，确保安全性。

3. 快速部署与性能优化 自动化部署工具：

采用 Helm Chart，实现集群的快速自动化部署，简化了部署流程。

流程编排优化：

基于前期的自研工作，对故障注入的性能进行了优化，重新设计了流程编排，解决了规模化下的广度问题，提高了演练效率。

4. 多环境支持与跨网络调度 支持多种环境：

我们的业务涉及国内、国外、机房内部和外部等多种环境。对于机房内部（有专线连接）的演练相对容易，挑战主要在于机房外和海外业务的演练。

区域信息映射与调度：

通过内部统一的 Zone 信息映射，方便地进行区域调度，支持在公有云环境和海外业务中开展演练。

Proxy 管理：

建立了 Proxy 管理机制，方便实现跨网络的指令调度，确保演练指令能够安全、可靠地传递到目标环境。

通过以上措施，我们成功地扩大了演练的覆盖范围，确保各类服务和环境都能参与到容灾演练中，提高了整体系统的可靠性和稳定性。

演练过程可视化

第五部分讨论的是演练过程的可视化。这与之前提到的标准模型类似。当需要演练的对象较多时，除了接入我们主要的可观测团队（他们的标准化程度较高，基本都采用了 OpenTelemetry ）外，还需要考虑一些尚未采用标准的小型事业部的业务服务。如果不将这些服务接入标准，在演练时他们将无法查看相关指标。为此，我们制定了一个监控指标的标准，相较于前者，这个标准更为简化，指标也较为精炼。

在抽象层面，大家的思路基本一致，因此我们抽象出了一个"获取行为"。首先，需要明确你的对象是什么；围绕这个对象，确定它有哪些指标；对于每个指标，如何获取其实时数据。这意味着，在演练过程中，当选择了服务对象后，系统会将该服务对象的指标列出，供用户自行勾选。

勾选完指标后，在演练过程中就可以实时查看这些指标的情况。另一方面，我们的演练架构图也在持续迭代中。通过围绕演练对象，基于其 Trace 动态调用链和 CMDB 的静态拓扑关系，我们能够了解在某个点注入故障后，对其上下游以及自身会产生哪些影响。

演练流程自动化

第六部分讨论的是演练流程的自动化。为了实现演练流程的自动化，我们主要关注以下三个方面：**1. 演练场景能否自动验证：**

即我们的演练场景是否能够实现自动化的验证。

2. 演练能否自动执行：

即演练过程是否能够实现自动化的执行。

3. 演练预期能否自动验证：

即演练的预期结果是否能够实现自动化的验证。

对于场景的自动验证，我们主要借助了专业的测试平台。他们已经开发了许多功能强大、易用的平台工具，特别是在 UI 自动化测试方面。因此，我们直接与他们对接，复用了他们的 UI 自动化测试能力。由于他们的测试也关注场景，包括多场景结果测试和单应用结果测试，这使得我们在演练特定业务场景时，能够完全实现自动化验证。

在演练的自动化执行方面，主要包括以下三部分：**定时调度器：**

这一部分相对简单，类似于常见的代码定时执行机制。

流程编排的自动推进：

我们能够自动推进演练流程的逐步滚动，确保演练过程的顺利进行。

护栏机制的支持：

在演练过程中，设置护栏机制，确保在出现问题时能够及时预防和处理。这一点实际上与前面提到的可视化和可观测性密切相关。只要能够获取可观测指标，添加断言和规则逻辑引擎，就能实现这部分功能。

对于预期结果的自动验证，实际上与场景验证类似，但更偏重于指标层面。也就是说，我们需要判断在特定场景下，预期结果是正向的还是反向的。例如，在某些场景中，我们期望在演练后必须出现异常。然而，在实际演练中，我们发现有些预期的问题并未出现，链路未按照预期运行。这种情况下，我们需要查明结果被代偿的原因，这是不应发生的。

业务场景演练实践

第三部分是关于业务场景的演练实践，我们将举几个实例进行说明。之前主要讨论了平台技术和组织方面的内容。通过将平台实验与演练协同一体化，我们希望将过去需要在小会议室进行线下协同的演练方式，转变为线上协同。借助演练的面板，我们可以将演练人员、相关系统、业务服务以及演练过程中涉及的信息集中在一起。

在演练开始的配置阶段，我们可以在同一页面上完成演练对象的设定范围圈选、流程编排、原子实验的具体配置，以及观测护栏的配置。随后，在演练执行阶段，我们可以在同一页面推进执行流。最后，生成演练报告并进行验收。通过这一流程，实现了演练全过程的线上化和一体化管理。

可用区断网容灾演练

此前我们介绍了服务级别的演练，接下来讨论可用区（AZ）级别的断网演练。当前，这类演练并非通过我们的平台注入故障，而是采用其他方式实施。这里详细说明我们内部是如何开展这项工作的。

当服务级别的跨可用区演练基本就绪后，我们会进一步进行公司级别的大规模可用区演练。具体而言，我们直接在交换机上配置策略，切断机房之间的专线连接和公网入口。这样做的总体目的是确保当可用区级别的故障发生时，我们的业务系统和基础架构服务能够具备故障逃生和容灾的能力。

大型活动事件——S 赛演练

首先，给不太了解游戏的朋友科普一下，S 赛指的是英雄联盟全球总决赛。全球总决赛从 S1 到 S13（"S" 即 Season，意为赛季）已举办了十三届比赛，因此被称为 S 赛。由于 LOL 的玩家众多，且 S 赛在广大电竞爱好者心目中有非常高的地位，因此每次 S 赛都是对我们的一次考验，因此需要进行详细的规划，应对业务挑战。

总体规划与执行步骤：**1. 年度目标设定与规划：**

制定大目标：每年设定一个主要目标，围绕该目标进行详细的方案规划。

关键节点拆解：将目标分解为全年需要完成的关键节点和任务。

组建团队（拉阵营）：确定在不同环节中需要哪些角色和事业部的参与，明确每个人的职责和推进事项。

2. 现状摸排与演练执行：

摸排现状：在演练前，对当前的系统和环境进行全面的摸底排查。

处理白名单等特殊情况：在摸排过程中，处理需要排除的白名单等特殊配置。

开始演练：按照计划进行演练，模拟各种故障和异常情况。

复盘总结：演练结束后，对整个过程进行复盘，总结经验和发现的问题。

3. 演练思路分为三大块： 断专线测试：

目的：先切断两个机房之间的专线，验证在实施双活或多活架构后，机房之间是否仍存在不合理的依赖。

断公网入口测试（南北向流量）：

目的：切断机房的公网出口流量，测试各个机房是否能独立承载进入的公网流量。

全断测试：

目的：同时断开所有连接，模拟一个机房完全脱离网络的情况，观察对整体系统的影响。

4. 具体计划与执行细节： 治理多活风险：

从最初开始，识别并处理多活架构可能带来的风险。

公网流量切换：

尝试在两个机房之间切换公网流量，测试各自的承载能力。

逐步断开专线：

保留 1% 流量测试：在断开专线时，保留 1% 的流量进行测试，确保真实流量下的可靠性。

模拟断公网场景：

非主机房公网断开：断开非主机房的公网连接，观察系统反应。

再次保留 1% 流量测试：在全面断网前，保留少量流量进行最终测试。

完整脱网验证：最终实现机房的完全脱网，评估对业务的影响和系统的稳定性。

通过上述步骤，团队能够全面评估在大型活动（如 S 赛）期间，系统在面对各种故障场景时的应对能力

集成测试演练

1. 集成测试演练的核心场景：

故障注入能力的迁移：将故障注入能力移至测试环境，是日常测试团队经常进行的工作。

平台提供的 Open API：测试平台提供了开放的 API，供 QA 测试使用。

测试用例的编写与集成：QA 人员编写测试用例，并将其集成到 CI（持续集成）构建流程中。

自动化测试流程：在代码提交后，测试用例会自动运行，检测是否引入了不合理的依赖等问题。

2. 快速验证场景的重要性：

选择适合 CI 的测试场景：关键是寻找能够快速验证的场景，方便在流水线中运行。

不适合 CI 的场景：如模拟断网的测试不适合在 CI 流水线中执行。

充分利用测试功能：利用 UI 自动化测试、多场景接口测试等功能，进行有效的验证。

3. S 赛（英雄联盟全球总决赛）演练：

年度赛事准备：每年都会针对英雄联盟赛事进行专项演练。

核心流程：

场景分级：按照内部的分级规则对赛事场景进行分级（具体规则将在 PPT 中提供）。

形成场景地图：根据分级结果，构建具体的赛事保障场景地图。

能力应用与摸排：利用前述的各种测试能力，对场景地图进行全面的摸排和演练。

专项演练的效果：

问题发现：通过专项演练，发现了大量问题，主要集中在代码层面的强弱依赖和不合理的降级方案上。

提升系统可靠性：这些演练有效提高了系统的稳定性和对大型活动的保障能力。

（三）总结及展望

成效总结

效率提升：

通过引入多种手段，演练从规划到落地的时间由周维度缩短为天维度，降低了成本，提高了效率。

全面覆盖：

以前只有核心部门能充分利用演练工具（"吃得饱"），现在所有部门都能参与演练，使用这些工具，实现了全面覆盖。

演练频次增加：

过去需要领导推动才能进行演练，频次可能是每季度一次。现在演练频次提升到了每周一次，业务同事因为演练不会引入过多新负载，变得更加积极，不再抗拒参与。

演练有效性验收成本：

以前需要人工介入，成本较高，现在我们实现了自动化验收，并能够复用测试中的多个场景，从而实现了技术建设的复利效应。这种自动化与复用机制不仅提升了工作效率，也大大减少了人为错误的发生。

实践反思

- 演练之前要有明确的目的，不然演练结束会觉得没有效果
- 大规模推广时候，需要结合业务团队特性来量身定做方案，主动推动

- 强弱依赖演练、预案演练、多活演练，具备高密度、高关注度、高收益的特征
- 演练自动化的关键还是在于过程护栏的有效性和演练效果验证的自动化
- 需要有面向演练场景的自动化用例，可能需要专门去补充
- 接口类的自动化用例，可以考虑用流量录制的流量集去转化
- 不同系统处于不同阶段，需要以不同的混沌实验帮助提升稳定性建设

5.3.7 蚂蚁故障应急全流程体系构建及应用实践

SRE Elite 精选原因

本案例来自蚂蚁集团的平台工程与技术风险部行业一线的实践总结构建了完善的故障应急全流程体系，主要亮点包括：明确故障定义和分级，聚焦关键业务的 GOC 场景，专注保障最重要的服务接口和结果点，驱动业务稳定性建设；还建立极具特色的应急值班长体系，明确应急角色和职责，提升组织协同能力；开发技术风险管理平台（TRM），实现故障全生命周期的数字化运营，相关工作令人。

在展望未来时，此次分享向 SRE 领域的同仁们展示了人工智能与 SRE Agent 如何增强应急响应能力，并在智能化故障定位、决策和处理方面展现出巨大潜力，极具启发性。

（一）背景及设计原则

备注：

本文档内容基于 2024 年 Qcon 上海站，蚂蚁集团的 SRE 技术专家刘凯宁演讲实录。为提升可读性，本部分将采用第一人称进行叙述，并包含更多的口语表达。

我想与大家分享一句我非常喜欢的话："Fix the problem, not the blame." 在稳定性工作中，我们经常会遇到问题，可能会问："这是谁的责任？不是我的问题，该怎么办？" 但我们必须专注于解决问题，而非相互指责，这样的指责没有意义。如果我们以解决问题为导向，而不是追究责任，那么稳定性工作一定会做得更好。所以，我想先将这句话分享给大家。

在正式开始之前，我想提出几个问题，请大家思考一下：线上故障究竟是什么？谁应该为线上稳定性负责？以及如何评价应急处理的效果？接下来，我将为大家介绍蚂蚁故障应急的整体流程体系，并尝试回答这三个问题。

稳定性灵魂三问

■线上故障到底是什么?

■谁应该为线上稳定性负责?

■如何评价应急到底做的好不好?

我希望此次分享能够给大家带来一些启发。虽然蚂蚁的实践并非最完美，但希望能够为大家提供一些借鉴和参考。

本次分享主要包括四个部分：

1. 蚂蚁的故障体系介绍：我们将从故障的定义开始，全面介绍我们在故障管理中所使用的平台。

2. 蚂蚁的应急体系：由于每个公司都在讨论故障，应急响应也有相应的标准和平台。我将介绍我们常用的平台能力，以及我们最重要的组织结构。这是本次分享的重点之一。

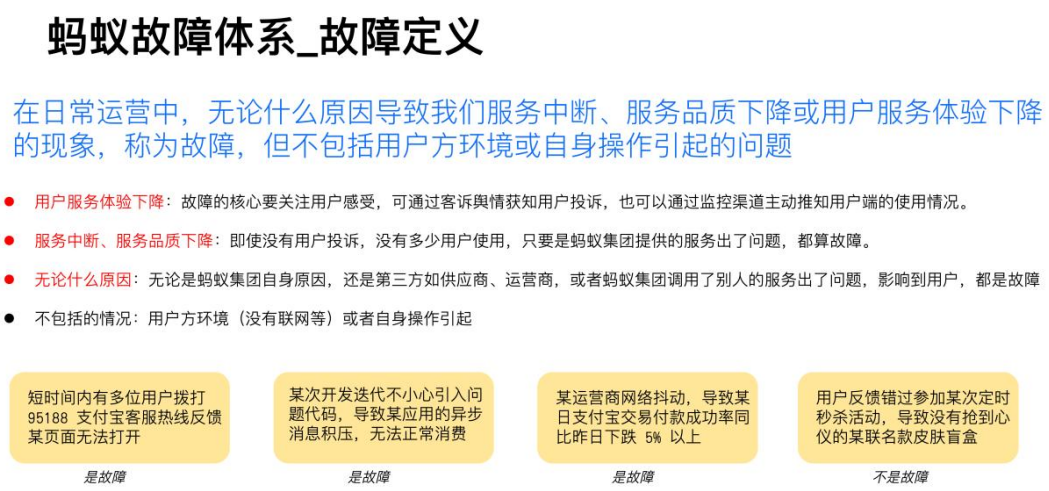
3. 故障生命周期：我将介绍一个完整的故障案例，详细说明其中包含的各类信息。

4. 未来展望：鉴于在本次 QCon 大会上，许多公司都在讨论 AI 技术。我将分享一些关于 AI 如何在我们的 SRE 领域中应用的初步想法。

（二）体系设计详情

蚂蚁故障体系介绍

故障定义



首先，我们进入蚂蚁故障体系的介绍。事实上，许多公司都在讨论他们遇到的各种故障，但究竟什么是故障？这一点必须明确定义。如果不加以清晰界定，可能会出现各种模糊的说法，比如"这是一个线上问题"或"这是一个重大故障"，导致无法准确识别和处理。因此，我们首先需要明确什么是故障。

在蚂蚁集团，故障的定义如下：在日常运营中，无论任何原因导致我们的服务中断、服务品质下降或用户服务体验下降，都被视为故障。当然，这不包括用户自身的问题。

这里有几个关键点需要关注：

第一，用户服务体验下降：这指的是什么？每家公司都有不同的业务，对于蚂蚁而言，涉及支付交易、互联网业务、财富保险等。我们故障定义的核心是关注用户的感受，必须具备手段去感知用户是否遇到异常情况，并及时获取相关信息。因此，我们需要通过客户投诉、舆情监控等多种方式来感知用户的反馈。

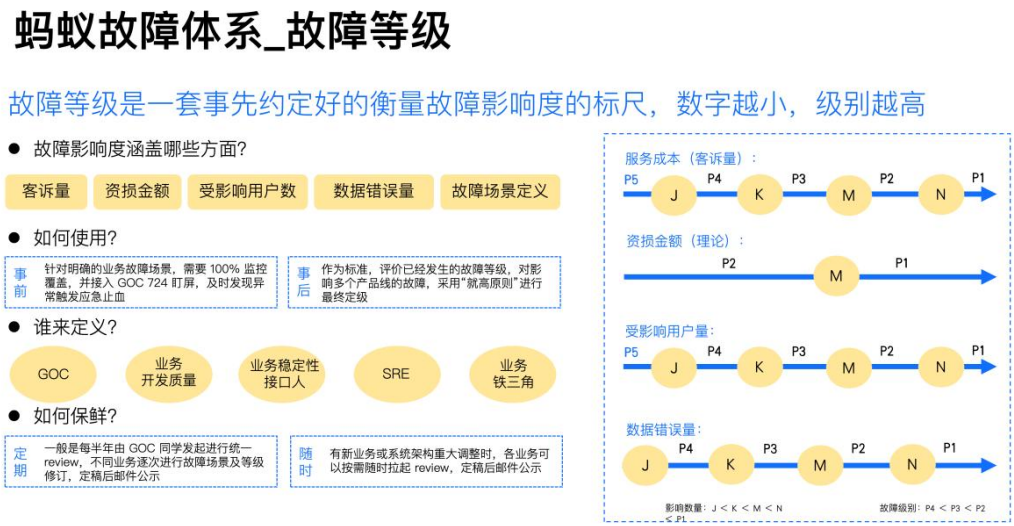
第二，服务中断和服务品质下降：正如之前提到的，可能存在一些用户无法感知的场景。如果用户感知不到，如何发现问题？为此，我们建立了可观测性系统，拥有大量服务层面的稳定性监控。即使用户没有直接感知，但蚂蚁集团提供的服务确实出现了问题，这也被视为故障。

第三，无论何种原因导致的故障：不论是我们自身的原因，还是依赖方的问题，例如运营商、第三方服务提供商，或我们调用的其他服务出现问题，都被纳入故障范围。由此可见，我们对故障的定义非常严格，正是由于这种严格性，我们更加追求在故障稳定性和应急领域的能力建设。

同时，也有一些情况不包括在故障定义中。例如，用户自身的操作问题。举个例子，如果客户在短时间内打电话反映某个页面打不开，这是典型的故障。另外，如果开发同事发布了代码，导致异步消息积压，这也是故障。再比如，运营商的网络抖动导致我们某些功能下跌，这同样被视为故障。

但是，像用户反馈自己错过了时间、没有抢到某商品等情况，就不算故障，因为这不属于我们的责任范围。明确了故障的定义后，接下来我们来看一下故障的等级划分。

故障等级



那么，什么是故障等级？故障等级是一套事先约定的衡量故障影响度的标准，数字越小，级别越高。通过故障影响度，我们才能评估故障的严重性。

故障影响度涵盖哪些方面？在蚂蚁集团，我们主要参考客户投诉量、资金损失金额、受影响用户数等指标。不同公司可能有不同的标准，以上仅供参考。如右图所示，我们可以先解释一下其中的其中 J K M N，这些字母其实代表脱敏后的量级的阈值，如在服务成本里头（客诉量）J 可能代表 10 个投诉，K 是 100 个投诉（假设），具体是哪个数量不重要，这个和具体公司的情况相关.

而其中 P 级别从左到右数字逐渐减小，但故障等级逐渐升高，对应的影响也越来越大。我们目前的等级划分从最低的 P5 到最高的 P1，大家可以参考。但这里存在一个问题：这些 P 级（P1、P2 等）具体意味着什么？稍后我会详细介绍。我们需要理解的是背后的思路。

定义故障等级的作用是什么？在故障发生之前，我们可以基于这些定义，开展可观测性建设和平台能力构建，目的是提前识别潜在问题，进行预防和保障。当故障发生后，我们需要对故障进行评估。我们采用"就高原则"，即如果一个案例符合多条故障定义，我们以最高等级为准。

那么，故障等级由谁来定义？一些部门可能有自己的质量部、技术风险部或运维部门。在蚂蚁集团，由 GOC（全球运营中心）、业务开发质量团队、业务 SRE 以及业务稳定性部门的同事共同制定故障等级。制定完成后，需要提交给业务负责人审批。由于业务随着时间发展，公司的战略也可能调整，如何保持故障等级的有效性？通常，GOC 同事每半年会统一进行一次审核，例如针对支付宝、财富、保险等业务域，统一审核故障定义和等级，并予以公示，大家按照统一标准执行。

大家可以思考一下，自己的公司是否有一个明确的机制，能够清晰地了解自身业务可能出现的故障类型，以及可能达到的最高等级。

故障序列

当公司出现新业务或重大架构调整时，我们应及时启动故障等级的审查，确保必要的工作和保障措施到位。此前提到了等级和 P 级，那么在蚂蚁内部，所谓的"序列"是如何定义的？"序列"究竟代表着什么？

故障序列是根据故障的根本原因和实际影响对故障进行分类的方法。关键在于区分"故障根因"和"实际影响"。有时，高稳定性团队可能会感到委屈，因为故障并非源于自身的技术问题，可能是由于产品运营的配置失误、网络运营商的抖动，或其他外部原因所致。因此，我们有必要将故障进行分类，明确技术故障与外部原因导致的故障之间的区别，并对其进行清晰的定义。

在定义故障序列时，主要参考以下几点：

1. 技术主要责任：首先也是最重要的一点，判断故障是否源于技术上的主要责任。如果故障确实因技术原因导致，那么可能需要扣减我们的故障预算。因此，我们主要关注 P1 到 P5 的故障等级，它们通常涉及技术主要责任，并对业务产生实际影响。

2. 故障记录与配额：另一个关键点是是否记录故障，以及如何通过故障配额（Quota）来评价我们每一年的稳定性表现。许多公司都有故障配额的规定，例如，一年内最多只能出现多少次故障，超出后将受到处罚，绩效可能会受到影响。因此，故障配额非常重要。

然而，有些故障并非技术原因引起，可能是业务操作失误，因此不需要扣减故障预算。对于外部原因导致的故障，我们定义为"W"；而由业务产品操作引起的故障，我们定义为"S"。每个公司都可以参考这些分类，制定自身的序列定义。

为便于在同序列故障在日后发生时能力进行更有效的故障应急，我们需要将部分序列所对应的指标纳入能力考察范围，并逐步提升相关能力，具体有以下四个指标：**响应原则**

例如，我们常提到的"1-5-10"原则，即"一分钟发现，五分钟响应，十分钟应急止血"，尽管许多公司都在倡导"1-5-10"，但真正能够实现的公司并不多。因此，我们将目标适当延长，提出"30 分钟恢复"，为应急处理留出一些缓冲时间。

监控发现率

关于监控发现率，大家可以回想一下线上故障的情况，有多少故障是通过监控发现的？有客户投诉或外部反馈，但真正通过监控发现的故障有多少？根据我们的实际数据，监控发现率可能只有 60% 到 70%，这已经相当不错了。因此，我们将监控发现数量和监控发现率视为非常重要的衡量指标。

根因定位能力

那么，什么是根因定位能力呢？这指的是我们具备一种应急的平台能力，当线上出现故障时，能够快速且准确地找到问题的根本原因。许多公司都在开发根因定位的产品，蚂蚁内部也有一个专门的研发团队，负责根因定位平台的建设，他们的主要任务就是确保根因定位的准确性。

如何评价他们的工作效果？关键在于当线上出现故障时，根因定位能力是否能一击即中，准确地定位问题的根源，从而及时解决故障。

Action 完结率

Action 完结率，指的是在故障发生后，如何快速推进问题的解决，并将相应的行动项彻底完成。当出现故障时，能够迅速推进并完成修复措施，对于恢复系统的正常运行至关重要。

GOC 故障点

在这里，大家可能会有疑惑：我们公司有如此众多的业务，究竟应该保障哪些？我们必须优先保障最重要的业务，而不是面面俱到地试图保障所有业务。因此，我们定义了最重要、最关键的服务接口和结果点，将其称为 GOC（Global Operation Center）场景。只要保障好这些 GOC 场景，我们的公司就不会出现重大的问题。并不是所有业务都需要全面保障，我们只需确保 GOC 的稳定性即可。

那么，GOC 是什么？我们通过三个领域来定义 GOC 场景：**1. 场景描述：**

明确说明需要保障的是什么。我们需要对所要保护的业务场景进行清晰的定义，明确其功能和作用。

2. 场景影响评价：

确定需要达到什么标准才能证明工作效果良好。我们需要设定衡量指标，例如成功量、成功率、响应时间等，以评估场景的稳定性和性能。通过这些指标，我们可以清晰地了解自己的工作如何被评价。

3. 可触达的故障等级：

明确该场景在出现故障时可能涉及的最高故障等级。如果某个场景即使完全失效，最高只会触发 P4 级别的故障，那么我们可能不会投入大量精力去保障它。但如果某个业务非常重要，稍有不慎就可能触发 P1 级别的故障，那么我们就必须重点保障。

实例说明

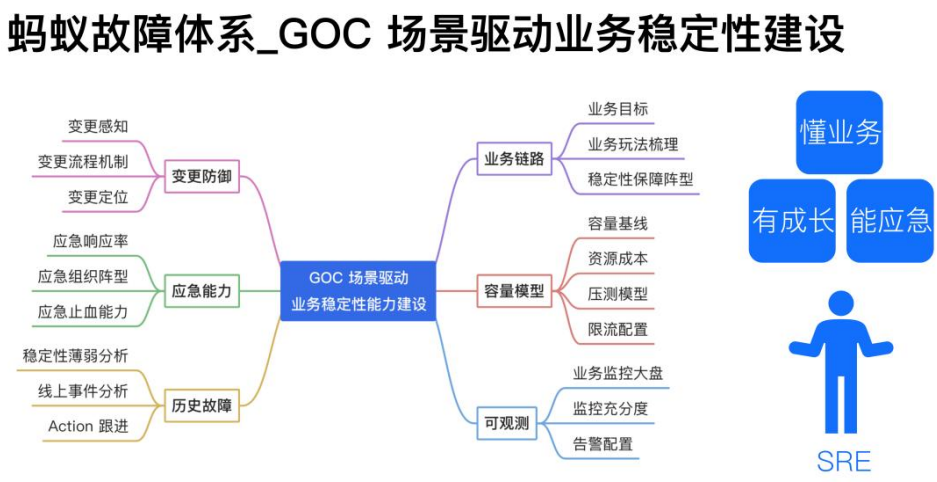
以登录功能为例，每个公司可能都有登录这一功能。我们需要思考在自己的业务中，是否对登录功能进行了明确的定义和保障。

免密登录：我们将成功量作为免密登录功能的稳定性衡量指标。将成功量视为评估登录功能稳定性的依据。如果成功量下降了某个百分比，就可能触发相应的故障等级。这些触发条件和对应的故障等级都可以被明确定义。

账号密码登录：对于账号密码登录场景，我们可能使用成功率来进行衡量。不同的场景采用不同的指标，这取决于其特性和重要性。

通过这些具体的数字和指标，我们可以清晰地定义每个业务场景的稳定性标准。尽管不同场景的衡量方式可能不同，但只要理解了核心理念，我们就能有效地保障关键业务的

GOC 业务场景驱动业务稳定性建设



GOC 场景定义的目的：

通过定义 GOC（Global Operation Center）场景，驱动业务稳定性能力的建设。

聚焦最重要、最关键的服务接口和结果点，确保公司整体稳定性。

明确业务链路和目标：

- 清晰了解所保障业务的核心目标和完整的业务链路。
- 明确稳定性保障团队，当问题发生时，能够迅速召集相关人员共同解决。

绘制容量模型：

- 从接入层到中间件，再到下游数据库，全面了解系统架构。
- 明确系统可承受的最大 QPS、日常流量和峰值流量。
- 描绘流量模型，识别不同时间段的流量特征（如早高峰、午高峰、晚高峰）。

全链路压测：

- 准备好压测所需的测试脚本和资源。
- 定期进行全链路压测，评估系统容量，确保在高负载下的稳定性。

配置限流措施：

- 根据容量模型，设置合理的限流策略。
- 防止突发流量超出系统承受范围，保障服务稳定。

完善监控体系：

- 明确业务的关键监控点和指标（成功量、成功率、响应时间等）。
- 建立实时监控和预警机制，及时发现和处理异常情况。

严格变更管理：

- 对可能触发 P1、P2 级别故障的核心业务，实施严格的变更管控。
- 建立完善的变更流程机制，具备变更感知和定位能力，确保变更安全可控。

加强应急响应能力：

- 制定应急预案，明确应急流程和责任人。
- 提升团队快速定位和解决问题的能力，确保故障发生时能够迅速恢复服务。

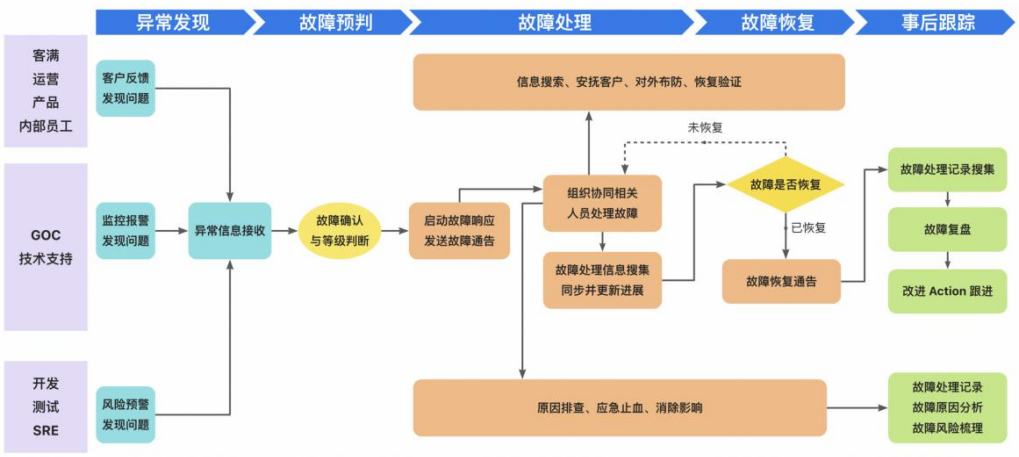
业务 SRE 的要求与成长：

- 深入理解业务：SRE 需要真正懂得所保障的业务，清晰其重要性和运行机制。
- 多领域能力提升：在压测、限流、容量管理、变更控制、防御措施、应急响应、监控等领域不断学习和成长。
- 具备全面的稳定性保障技能：能够统筹各项工作，全面提升业务的稳定性。

通过 GOC 场景实施稳定性保障工作：

将理论应用于实践，持续优化和完善稳定性保障体系。

蚂蚁故障体系_故障处理流程



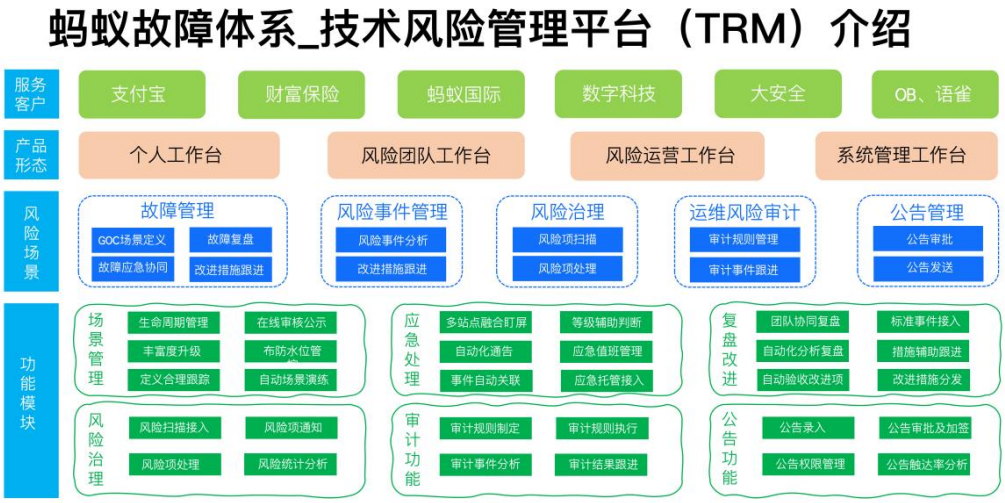
当线上出现故障时，需要明确的处理流程和团队协作，SRE 无法单独解决所有问题。故障处理流程分为以下几个阶段：异常发现、故障预判、故障处理、故障恢复和后续改进。

故障处理的主要阶段和步骤

1. 异常发现：
    - 通过内部客户投诉、监控告警异常、内部开发人员主动发现等方式，收集所有异常信息。
    - GOC（Global Operation Center）设立异常信息接收机制，汇总并记录所有异常情况。
2. 故障预判：
    - 基于预先定义的故障等级，GOC 对异常进行初步判断，确认是否属于故障。
    - 确定故障的严重程度，决定发布何种级别的故障通告。
    - 一旦故障确认并完成等级判定，正式启动故障应急流程。
3. 故障处理：
    - 产品运营：关注客户受损情况，评估故障对客户的影响。
    - GOC：组织各方人员参与应急响应，协调故障处理工作。
    - 开发、测试、SRE：快速恢复线上服务，查找问题根因，制定并实施应急预案或止血方案。
4. 故障恢复和后续工作：

- 故障验证：产品运营验证服务是否已恢复正常，客户体验是否恢复；GOC 确认各项指标恢复正常后，发布故障恢复通告。
- 问题根因分析：开发团队深入排查故障根因，总结问题产生的原因。
- 后续行动：制定改进措施和行动计划，跟踪落实，防止类似故障再次发生。
- 故障复盘：全面回顾故障处理过程，总结经验教训，完善故障处理流程。

平台能力支持



在完整的故障处理流程基础上，需要借助平台能力提升处理效率。平台工具和机制包括：

- 提供统一的异常信息接收和管理。
- 支持故障等级判断和通告的快速发布。
- 协助各团队高效协同，监控故障处理进展。
- 记录和跟踪后续行动项的落实情况。

我想向大家强调一点：在构建平台能力时，必须基于特定的场景。我们不是为了平台而做平台，而是为了解决具体的问题才去构建平台。当存在技术风险需要时，再去开发平台，而不是先搭建一个平台，然后再询问 SRE 或质量同学是否需要。因此，这一理念值得大家思考。

在蚂蚁这边，我们的技术风险管理平台（TRM）面向各个业务部门，客户基本涵盖了蚂蚁的所有 BU，如支付宝、国际大安全等，甚至包括已经独立的 OB、语雀等。他们都可以在这个平台上进行操作。产品形态主要服务于普通的 SRE 同学以及业务的接口人。稍后

我会介绍我们的整体应急阵型。针对不同的角色，我们提供不同形式的工作台。一些 GOC 的同学可能会在系统中录入 GOC 故障定义的场景，因此我们有风险运营的机制。

这个平台具备哪些功能？最重要的功能包括：**故障管理：**

主要是指前面所提及的故障场景、故障定义等信息用平台沉淀起来，同时通过故障复盘、Action 跟进之类的的管理能力整体形成功能闭环。

风险事件管理：

风险事件是指未被定义为故障、影响较小的业务事件，可以归类为风险事件。

风险治理：

主要是集中处理一些风险巡检系统的风险产出物，业务稳定性同学可以基于风险项信息进行风险的跟踪与治理。

运维风险审计：

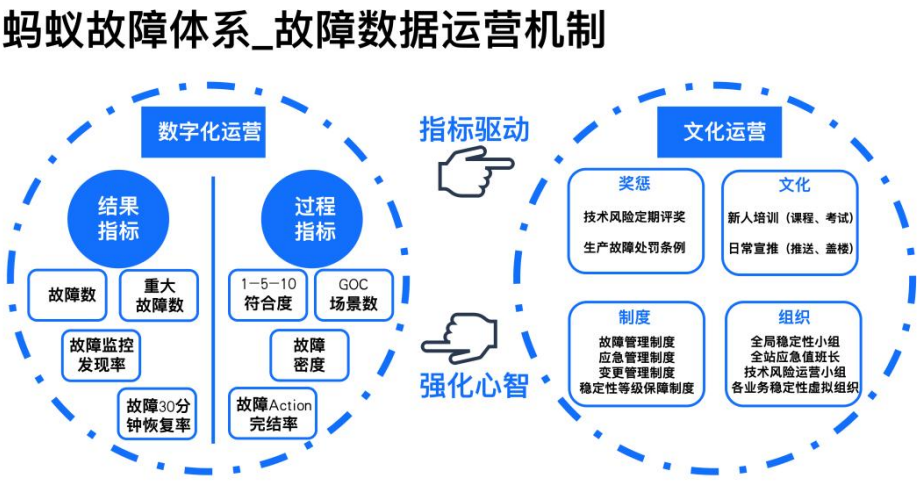
蚂蚁历史上曾出现过由于运维操作不合规导致的线上问题，这在各家公司都可能发生。因此，我们需要收紧重要权限，如果出现违规操作，需要进行审计。

公告管理：

平台提供公告功能，用于发布重要的稳定性信息。

其他功能模块，如场景管理、审计功能、复盘改进等，都是为整个故障生命周期和各种风险场景提供服务。

故障数据运营机制



那么，我们制定这些工作的目的究竟是什么？累积故障数据的意义何在？在此，我向大家介绍一下：我们希望通过数字化运营，将故障水平提升至一个相对较好的状态。为此，我们会定义一些结果性指标和过程性指标。

结果性指标方面，例如故障数量，这是非常直观的。今年，我们必须控制故障数量在一定范围内，若出现过多故障，就可能存在问题。对于重大故障而言，如从 P1 到 P5，我们更关注 P1、P2 级别的故障数量。由于 P1、P2 级别故障的影响面较大，我们需要设定指标，将其控制在合理范围内。

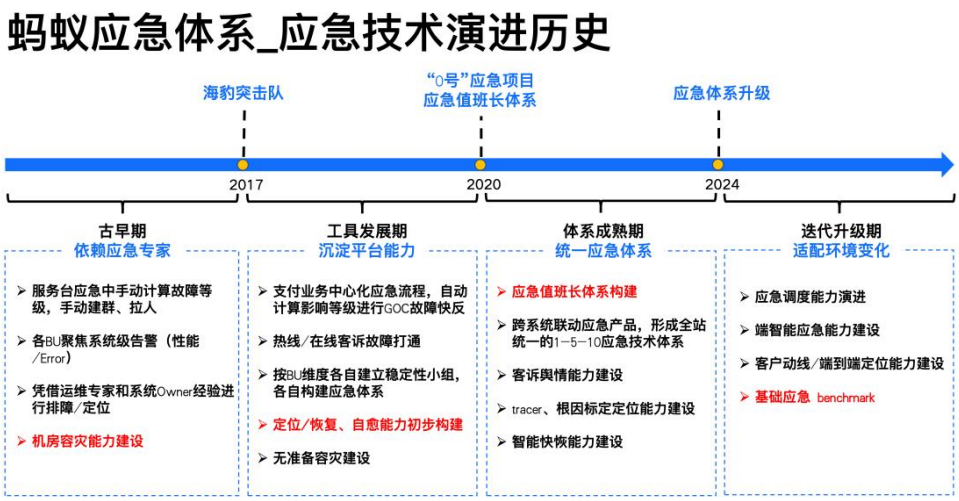
此外，还有故障监控率、监控发现率、30 分钟恢复率以及平均修复时间（MTTR）等指标。这些结果性指标直接影响我们对稳定性的评价和绩效考核，是整体评价体系的重要组成部分。

过程性指标方面，当我们定义好这些故障场景且线上出现问题时，我们的 MTTR 是否满足要求？故障处理措施的完结率达到了怎样的水准？例如，去年我们有许多故障处理措施超期，今年能否及时完成？通过数字化运营，结合过程指标和结果指标，我们可以生成一份全局稳定性的故障数据报表。利用这份报表，我们能够在月度或半年度开展各类评审，对我们的稳定性工作进行评价。

在文化运营方面，我们制定了相关的奖惩措施、文化理念和制度等。原本下一页是要介绍我们的惩罚制度的，但来之前老板提醒我不要提及，担心会影响招聘。因此，这一部分就省略了。总之，我们的奖励机制非常完善，但若触犯红线，必然会受到相应的惩罚。毕竟，从事稳定性工作的同仁，谁没经历过几次故障呢？

然而，我们通过数据与文化的相互促进，利用数字化运营拓展出各类指标，定义好指标后，让我们的文化起到驱动作用。同时，借助奖惩措施、文化制度等一系列手段，加强稳定性团队的意识，使他们真正认识到稳定性工作的重大意义，明确这是值得付出和投入的事业。

蚂蚁故障应急体系



在之前的内容中，我向大家介绍了故障的相关概念，接下来，我将讲解故障应急体系。

首先，让我们来看一下应急工作。蚂蚁集团的应急体系已经建立了很多年，从 2017 年开始至今，每个阶段的重点各不相同。

最初期，我们称之为"古早期"。在这个阶段，我们主要依靠一些非常优秀的应急专家。当线上出现问题时，他们会第一时间进行处理，没有明确的职责边界，也不存在推诿的情况，他们的目标就是迅速解决问题。然而，这种方式工作量巨大，压力也非常大，因此我们意识到需要建立一个梯队化的体系。

当时，这些应急专家最重要的任务是完成机房级别的容灾建设，之后线上出现的问题，基本都由他们负责处理。

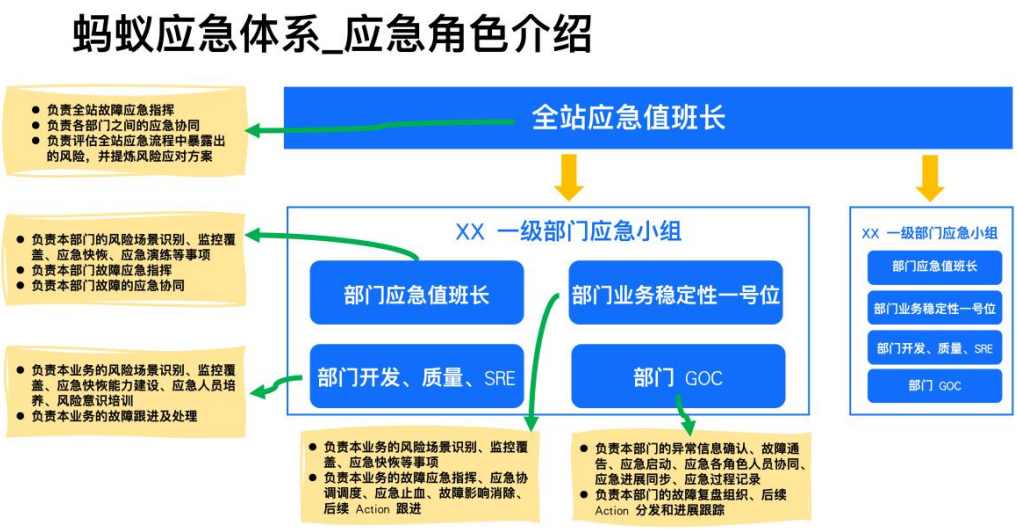
从 2017 年到 2020 年，我们开始构建各种平台。我们意识到，有些问题不能仅靠人力解决，必须依托平台。因此，我们沉淀了多种平台能力，推出了第一版的定位恢复和自愈功能。这一时期被我们称为"工具发展期"。

在 2020 年，我们正式成立了应急值班长组织，稍后我会重点介绍。从那时起，随着应急值班长体系的构建，线上问题和故障出现了断崖式的下降。原因在于我们明确了谁应对线上稳定性负责，谁需要采取哪些措施以快速恢复问题，并设立了相应的评价指标。

因此，我们在那个阶段实现了跨系统的应急产品联动，形成了全栈的"1-5-10"标准。同时，我们也拓展了其他能力，例如客户投诉、舆情监控，以及智能会议功能。

到了 2024 年，也就是今年年初，我们的团队对 GOC 和稳定性组织进行了升级。为了适应整体的变化和发展需求，我们演进了一些精细化的应急能力，例如应急调度能力、终端智能的应急能力，以及基础应急的基准测试。这意味着，单有平台是不够的，我们还需要评估工作实际效果，确保应急体系的有效性。

蚂蚁应急体系应急角色



前面提到应急角色的重要性，那么具体来说，蚂蚁集团的应急体系有哪些角色？我们主要分为两个级别，五种角色。

第一个级别是全站的应急值班长，

这是一个至关重要的角色，负责全站的整体应急指挥。全站应急值班长的职责是协调各部门在线上出现问题时的应急协同，及时揭示重大风险，保持全局掌控，并推动应急流程的执行。他位于应急指挥体系的最高层。

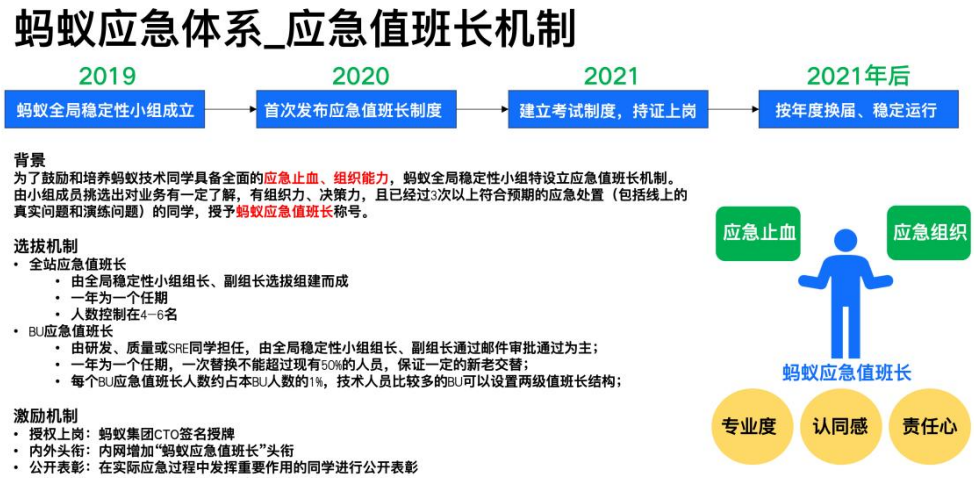
第二个级别是部门级，

不同部门有各自的稳定性保障人员，包括部门的应急值班长、开发质量同事、以及各业务线的应急人员。具体而言，部门内会设立一个统一的应急小组，由部门应急值班长负责。在每个业务线下，配置专门的稳定性一号位。部门应急值班长的任务是识别并监控部门的风险场景，构建监控覆盖、告警触发、应急响应等能力。稳定性一号位则专注于其所负责的业务，与部门应急值班长形成上下游的合作关系，不是上下级，而是互为支持的关系。

在每个部门中，GOC 同事（定义部门级 GOC 场景的技术支持人员）负责在现场出现问题时快速响应并激活应急措施，确保部门的应急流程顺利进行。

这样，每个部门的应急小组与全站应急值班长紧密联动，形成了一个完整的应急体系。回答"谁为线上稳定性负责？"的问题，这些角色和他们的协作体系就是蚂蚁线上稳定性的保障核心。

应急值班长体系的建立经历了一段曲折的发展。早在 2019 年，蚂蚁集团就成立了全局稳定性小组，目标是避免重大线上故障。然而，当时的组织结构是虚拟的，没有正式的值班长角色。这种虚拟组织虽然能够提供初步的稳定性支持，但尚未达到体系化的水平。



过去提到，虚拟组织在明确"谁负责线上稳定性"方面存在一些弊端，因为各部门都有各自的能力，但职责不够清晰。为了解决这一问题，2020 年，蚂蚁集团正式向全公司公开了应急值班长制度。我们的目标是培养蚂蚁的技术人员具备全面的应急能力，包括"应急止血"和"应急组织"两大核心能力。因此，我们建立了全局稳定性的应急值班长机制，但并不是每个人都可以担任此职，需具备以下几点：

- 组织力与决策力：应急值班长必须具备出色的组织和决策能力，这意味着新人通常难以承担这一角色。
- 实战经验：值班长需有丰富的线上应急实战经验，并在处理线上故障时展示出优异的效果。

在选拔上，我们有一套标准流程（具体细节略），但必须提到的是激励机制。被评选为应急值班长的员工将获得蚂蚁 CTO 的授权和象征性的荣誉牌，并在全公司进行宣传。获

得这一称号不仅意味着表彰，也要求承担更大的责任。例如，值班长需对线上问题保持 24 小时的响应准备——责任越大，激励也相应更大。

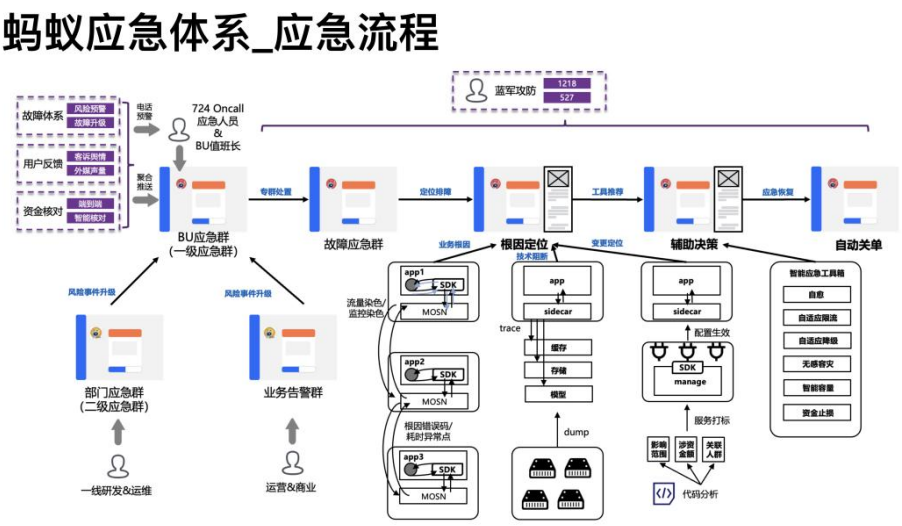
应急值班长的职责十分严峻，需要具备线上快速止血和组织应急的能力。线上问题通常涉及多个角色和部门，不仅是值班长的个人职责，而是一个全面的协同任务，涉及产品运营、GOC、开发质量、SRE，甚至公关、风控和安全等多个团队。因此，全局值班长的应急组织能力至关重要。

值班长需具备三项关键素质：

- 专业度：在稳定性工作中展示卓越的专业能力；
- 认同感：对稳定性工作有高度的认同，愿意为其付出努力；
- 责任心：始终如一地承担责任，不能因个人情绪忽视告警响应。

这些素质确保应急值班长能够高效推动线上问题的快速解决，并协调各方力量维护系统稳定。

应急流程

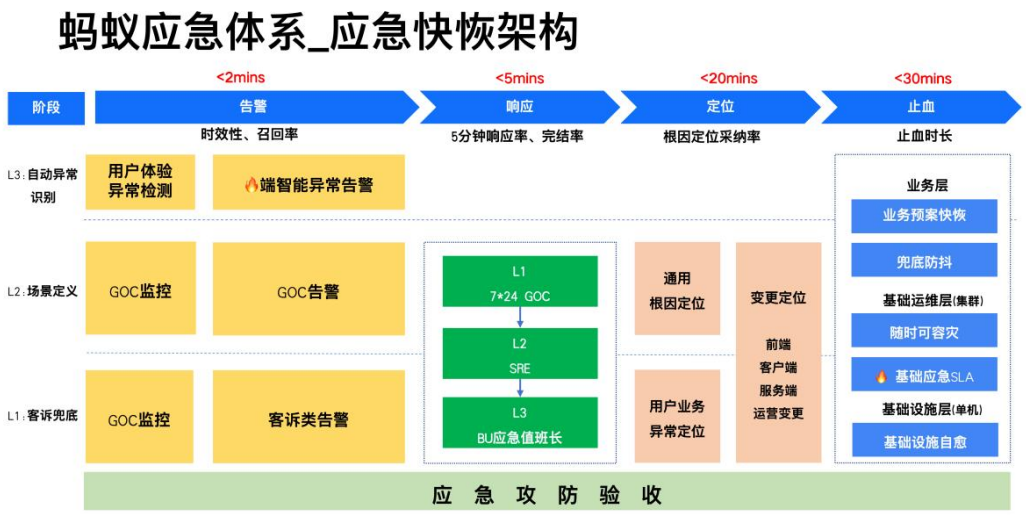


接下来具体讲解一下蚂蚁集团的应急流程。虽然前面已经介绍了故障处理流程，但这里特别提出应急流程，目的是展示我们在应急响应中的平台能力。

在应急启动之前，我们已经建立了多种感知渠道和能力。例如，线上会有电话预警、预案推送，或是各部门的应急群开始处理预警信息。这些预警阶段尚未被定义为故障，但我们会预先组建好业务单元的应急群，以便在真正的故障发生时迅速拉起全局的故障应急群，确保统一协调。

故障应急群建立后，我们依赖根因定位团队第一时间锁定故障根因，虽然定位可能不完全精准，但我们有明确的要求和标准来提升根因定位的准确度。同时，我们还提供辅助决策的能力，以确保在关键决策时有足够的数据支持。

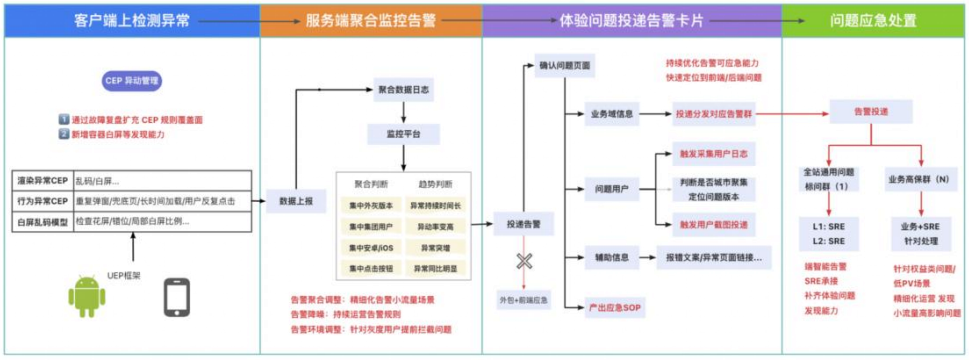
在一些情况下，系统会自动完成工单关闭，这样的自动化处理进一步优化了应急流程。通过这些平台能力和流程，应急处理流程变得更加高效和可控，实现了从预警到故障处理的全流程闭环管理。



关于应急快速恢复，蚂蚁集团的关键能力聚焦在不同阶段的核心指标。每家公司都会设立类似的应急时间节点：例如"一分钟响应，五分钟监控响应，三十分钟止血恢复"。在这些阶段，我们需要明确具体的能力评价指标。

这些评价指标正是前面提到的各种运营指标，确保每一项能力都能被有效衡量。若没有清晰的指标，能力的执行效果将无法得到客观评价。换句话说，每一项应急能力必须有与之对应的指标，以便明确判断工作的完成质量和效果。这种指标体系不仅验证了应急能力的有效性，也为进一步的优化提供了依据。

蚂蚁应急体系_🔥端智能异常告警

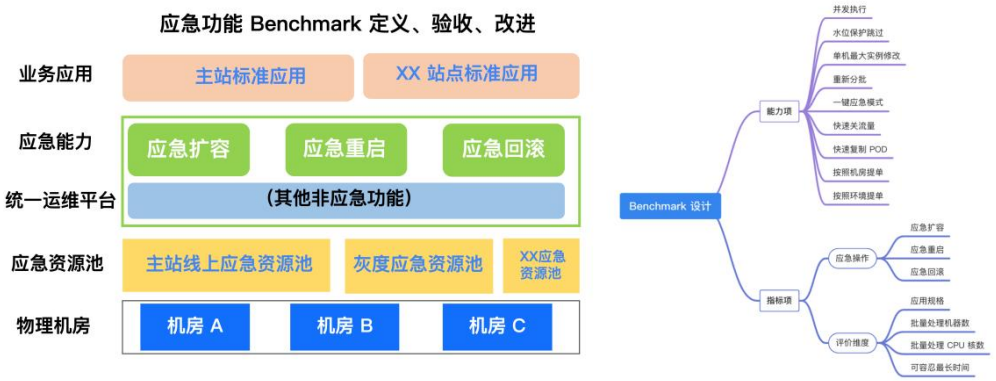


我们最近新增了一项关键能力——**端智能异常告警和应急 SLO**。这一端智能功能能够及时感知客户端的异常状况，例如白屏、黑屏、重复弹框等异常情况。客户端一旦出现这些异常，我们的系统可以立即感知并迅速将异常聚合到服务端进行分析和决策，判断是否属于实际问题。

如果确认存在问题，系统会自动拉起故障处理群组，并将该事件接入故障处理和应急流程，形成关联。客户端上传异常后，服务端聚合告警并生成问题卡片，从而启动应急响应。目前，这项能力已经帮助我们发现并响应了许多线上问题，极大提升了故障的感知和快速响应能力。

蚂蚁应急体系_🔥基础应急 SLA

通过定义应急操作的能力指标 Benchmark，定期验证达标情况，驱动应急能力演进，助力应急时效提升



在基础应急管理中，我们需要定义和构建多种应急能力，以确保在面对故障时能够快速响应和处理。例如，我们会设定应急扩容、应急重启和应急回滚等能力，确保在平台出现问题时能够有效应对。

为此，我们构建了不同的应急池，赋予这些应急能力具体的执行要求。这些要求包括：

- 并发能力：确保系统能够处理多个请求，以应对突发流量。
- 重新分批能力：能够灵活调整请求处理策略，确保服务不中断。
- 一键切换应急模式：在发现问题时，能够迅速切换到应急模式，降低潜在影响。

我们还会设定具体的指标来衡量这些能力的有效性，并进行定期验证，以确保这些应急能力在实际使用中能够发挥作用，确保系统的稳定性和可靠性。通过这样的措施，我们能够有效降低故障对业务的影响，提升整体应急响应能力。

在评估应急管理效果时，我们可以使用标准化的评价指标。这些指标包括：

- 发现时间：故障是在多长时间内被发现的，是否通过监控系统发现。
- 响应时间：从发现故障到采取行动的时间（5 分钟响应）。
- 止血时间：止血（初步解决问题）的时长。
- 定位信息：是否有定位卡片及相关信息，帮助识别故障根因。
- 告警情况：是否触发了端智能告警等。

我们要求 SRE 团队在故障发生后 48 小时内完成应急复盘，这个复盘专注于评估应急响应的有效性，而非故障的全面分析。通过复盘，我们可以明确在应急过程中做得好的地方和需要改进的环节。

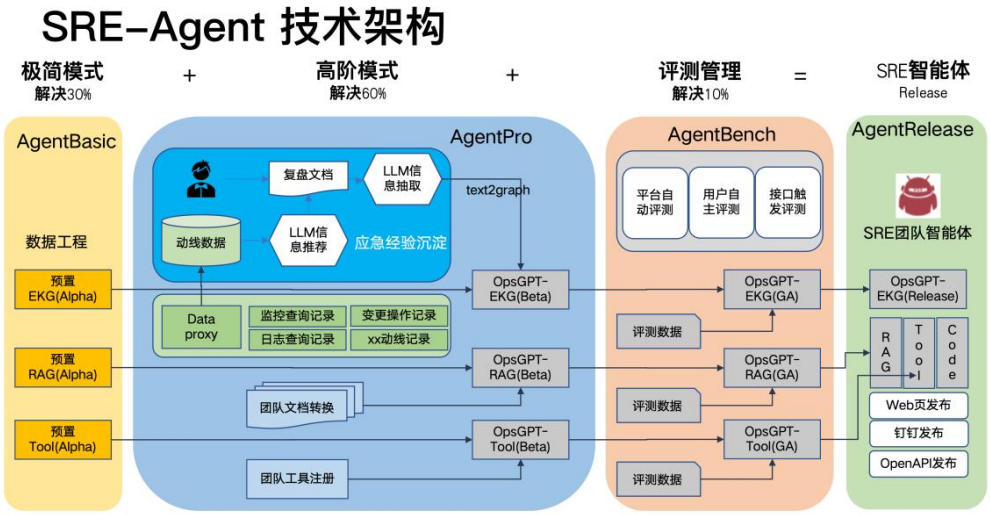
应急止血的方式可以因公司而异，常见的手段包括：

- 业务层面的操作：快速调整业务策略以减轻影响。
- 基础操作：如重启服务或调整资源配置。
- 容灾切流：在出现问题时快速切换到备份系统。
- 变更管理：快速回滚至稳定版本。

这些平台能力支持我们在出现故障时快速响应，并减少业务中断的影响。通过系统化的评估和复盘，我们能够不断优化应急管理流程，提高整体的稳定性。

（三）总结及展望

AI 助力，未来已来



未来应急能力与 SRE Agent

1. 平台能力的演变

未来的应急体系将依赖于平台能力的提升。

SRE Agent 的协作：

- 提供预置知识，支持基本问题解答（如应用调用量、机房部署等）。

2. 高阶能力的融合

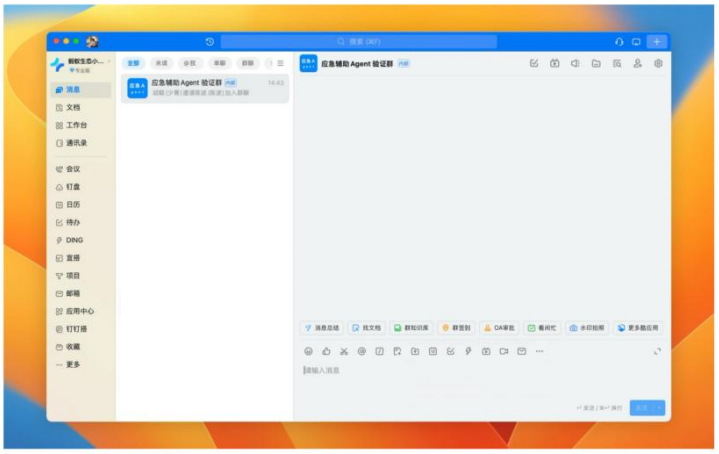
- 应急复盘和相关文档的整合：
    - 通过 OPS、GPT 等技术提升应急复盘能力。
    - 发展为 Agent Pro 模式，能够解决约 60%的问题。
- 剩余问题的处理：
    - 需要将 Agent 交付给 SRE 团队使用，收集反馈并持续优化，以解决剩余的约 10%问题。

3. Agent 模块整合
    - 包括 Agent Basic、Agent Pro 和 Agent Bench 三个模块。
    - 最终形成 Agent Release，整合 SRE 团队的智能体能力。
    - 当前版本已在内部使用，具备多种平台及应急能力。

SRE-Agent 页面演示

SRE-Agent 构建演示

SRE-Agent 使用演示



1. 知识导入

利用知识图谱系统，导入相关的知识和信息。

将导入的知识整理为可视化数据，便于后续处理。

2. 知识可视化

生成可视化图形，展示导入知识的结构与关系。

知识经过编排后，形成逻辑关联，便于理解和应用。

3. 机器人关联

将编排后的知识信息与机器人系统进行关联。

机器人能够调用这些知识进行实际操作与决策。

4. 决策过程

机器人根据用户输入的信息进行初步决策：

若结论正确：

维持当前决策，执行相应的操作。

若结论错误：

触发学习机制，重新分析相关知识。

调整决策逻辑，更新知识库，以优化未来的决策。

5. 大模型能力体验

当前正在进行大模型能力的实际使用，收集反馈以改进功能。

鼓励团队成员关注和参与 SRE Agent 团队的进展与反馈。

6. 集成到钉钉

将 Agent 系统集成到钉钉平台：

生成应急卡片时，机器人自动分析潜在问题。

识别异常情况，并提供根因分析。

输出相关数据库和监控指标的实时反馈，便于团队快速响应和处理。

7. 应用反馈

在实际使用中不断收集反馈信息，优化机器人决策流程。

根据用户体验和问题解决情况，进行持续的功能迭代和更新。

5.3.8 xx 银行应急提升实践：深耕 1-5-10 工程

SRE Elite 精选原因：

该案例以"1510"工程为牵引，从组织、流程、工具三维度提升金融级故障应急韧性。通过横纵三层技术支持、一分钟监控感知、五分钟节点级定界、十分钟应急六板斧处置，实现端到端闭环；配套早例会、复盘、治理、体验等机制沉淀知识并驱动持续改进；值班经理八大修养与跨部门协作文化保障指挥效率；最终形成可预见故障场景及一键应急平台，将专家隐性知识自动化，故障诊断秒级完成，为大中型股份制银行构建高效、可复制的稳定性保障范式。

（一）背景及设计原则

金融业的稳定性保障所面临的挑战正日益严峻，影响稳定性的核心挑战可以概括为三个相互关联的方面：

监管新常态：合规性和稳定性要求不断收紧，不仅要求严格、具体还有演练实操。例如，针对高级别故障需要向监管机构汇报并闭环落实改进措施落地情况、现场和非现场监管检查以及组织各金融机构开展的"不打招呼、令下即切"极限压测演练。

业务高要求：随着科技与业务的深度融合，不仅业务部门，科技部门也深刻感受到系统稳定性已成为保障一线业务展业和触达客户的基础，对系统停业时间（无论在计划内，还是计划外）的容忍度都不断降低。

技术（大变革）：以 XC、容器、大模型等为代表的新技术架构和应用层出不穷，叠加金融业固有的技术架构多样化痛点，运维规模、复杂度急剧增长，配套的管理要求也越来越高，这与现有运维人员的数量、能力增长速度是不匹配的，进一步增加了稳定性保障的难度。

在上述三重挑战下，任何内部微小的跑冒滴漏都可能迅速放大，导致稳定性天平失衡，引发严重问题。

为了积极应对这些挑战，并争取更具韧性的业务运营环境，我们集结了数据中心、研发中心和科技部门的力量，共同启动了"1510"工程。这项工程包括"主动预防"和"快速应急恢复"两个分支，核心目标是：1 分钟发现、5 分钟定界、10 分钟应急，并以此牵引数据中心整体运维能力的全面提升。接下来要介绍的是 "1510 工程"关于快速应急恢复这个分支。

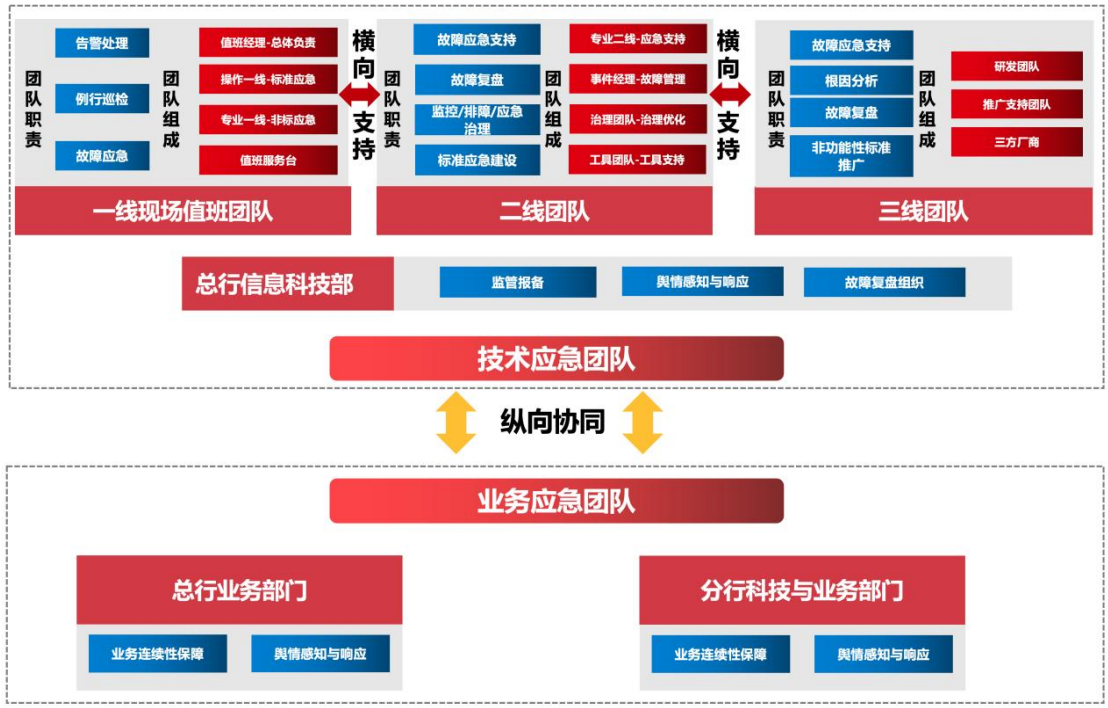
注：1510 代表 1 分钟故障感知能力、5 分钟故障定界能力（包含故障感知时间）、10 分钟完成应急处置，总体应急恢复目标为 15 分钟。

（二）解决方案

组织能力提升

"组织能力"的提升，这不仅是我们迈向持续优化运维体系的第一步，更是应对未知挑战、确保系统稳定运行的基石。

应急组织架构



核心目标：构建高效应急响应体系，力求"1 分钟发现、5 分钟定界、10 分钟恢复"

多层级横向技术支持： 我们的组织架构采用 "横纵" 结构，构建了三级技术支持体系：

一线团队：

作为应急处置的主力，负责告警处理、例行巡检及故障应急。其组成为：**值班经理：**

作为应急的总体负责人，全面负责应急指挥。

操作一线：

负责标准应急操作的执行，例如，可预见故障场景的应急。

专业一线：

针对非标准化故障，进行专业处置。

二线团队：

为一线团队提供支持与增援，协同处理一线团队无法独立解决的复杂故障。

三线团队：

提供更深层次的技术支持，通常包含研发、推广支持及三方厂商等资源，应对疑难杂症。二、三线团队除了应急支持，日常还承担标准应急流程建设、治理、应急演练落地以及非功能性需求的落地等职责。

背景说明：通常情况下，银行的生产环境和办公环境、研发测试环境之间是严格限制网络访问的，因此，所有生产环境的运维操作，都只能生产区终端执行。通常，银行会设置生产技术支持区、ECC（企业总控中心），其中，ECC 是现场值班应急的场所。

横纵应急协同： 构建技术与业务的紧密协同机制，包括技术应急团队与业务应急团队的联动，总行数据中心、研发中心、信息科技部与总行业务部门、分行科技与业务部门的协同，确保在应急事件发生时，技术与业务能够快速对齐目标，形成处置合力。

应急四大原则

为了支撑高效的应急响应，我们遵循以下四大核心原则：**①第一时间通知：**

对客服务的联机交易类故障必须第一时间口头通知值班经理和专业组长，然后再处理；重大故障值班经理电话通知处室经理、中心负责人。

②优先恢复业务：

故障应急应以第一时间恢复业务服务为首要目标，业务恢复后再处置账务调整、数据恢复等问题。

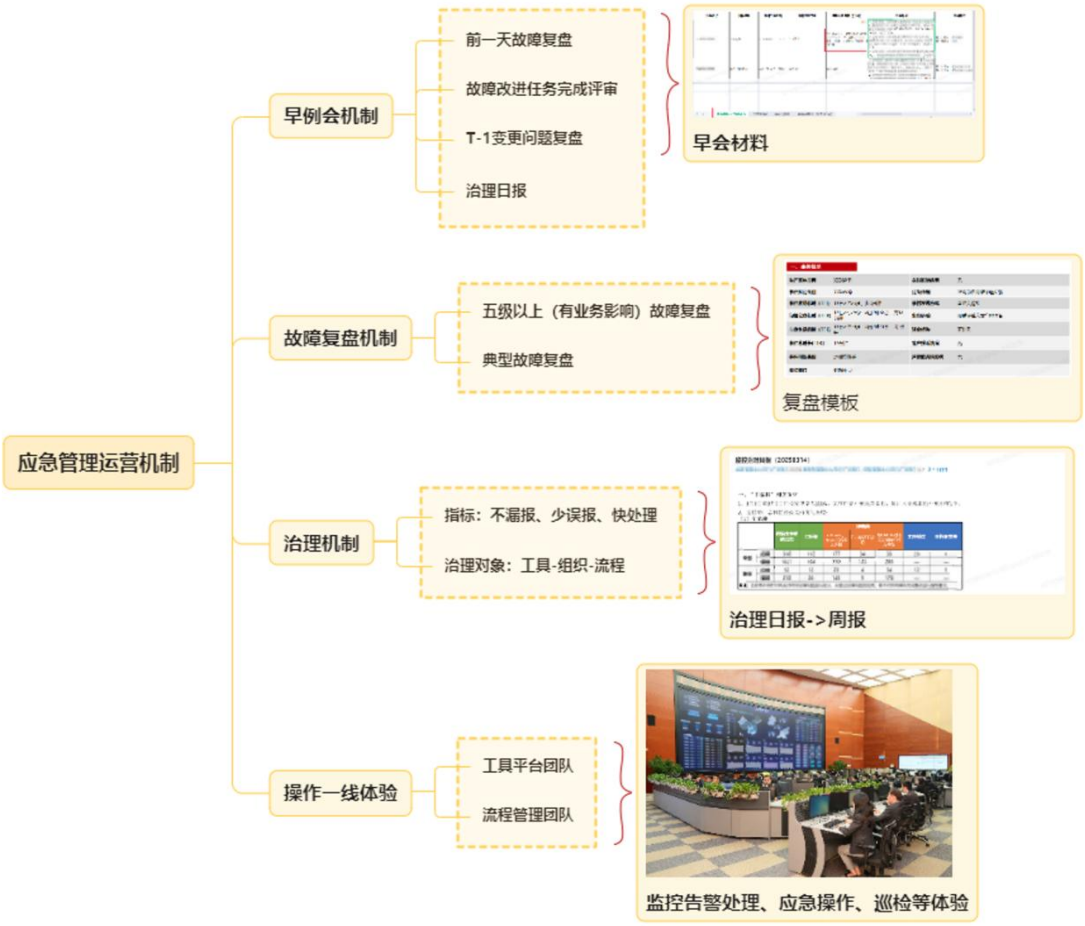
③并行现场应急：

值班经理决策启动应急程序，组织值班人员、相关人员现场应急；专业组长第一时间到 ECC 现场应急支援；相关专业条线并行现场应急，相互联动，必要时与业务并行应急。

④30 分钟恢复（15 分钟恢复为牵引目标）：

联机交易类事件必须力争在 30 分钟内完成应急处置，恢复业务服务，同时提出了 15 分钟的牵引目标，鼓励团队持续提升应急响应与恢复的速度。

应急管理运营机制



应急管理运营机制：重复、持续地通过早例会、故障复盘、治理、体验等机制落地应急管理要求，同时，统一每个人的故障应急认知、意识和行为。

对应急管理的持续改进与落地，我们构建了一套系统的运营机制，核心目标是让应急管理的要求能够渗透到日常工作中，并以此统一团队对于故障应急的认知、意识与行动。这里介绍其中 4 个机制：

早例会机制：

由数据中心、研发中心和科技部门共同参与，会议核心内容包括前一天的故障、变更问题、治理问题复盘，以及对已完成的故障整改任务进行评审。会议由数据中心的各级负责人轮流主持，谁主持、谁发问，例如，监控是否做到了比监管、银联更快发现，故障定界是否精准，应急是否使用了工具，有没有规范没落地的地方，每个团队在应急中都做了什么。这种跨部门协同与负责人轮动主持的方式，除了复盘和评审，还有 3 个作用：一是，通过主持的形式，可以强化各层级的负责人对应急要求的主动思考和反复训练；二是，有句话说的好，别人的失败，才是自己真正的成功之母，当各层级负责人对别人进行灵魂拷

问的时候，他就会下意识地主动思考自己团队内是否也存在类似"灯下黑"的问题，有利于真正做到问题的举一反三；三是，早例会还是针对治理问题快速决策、拍板整改方案的高效平台，显著提升了治理工作的落地效率。

故障复盘机制：

针对五级以上（有业务影响的）故障以及其他具有典型意义的故障，由科技部门牵头组织进行深入复盘。通过标准化的复盘模板，系统性地分析故障发生的根本原因、影响范围，并制定详细的改进措施。这确保我们将每一次故障都转化为宝贵的经验，避免类似问题再次发生。在复盘章节将对此机制进行更详细的阐述。

治理机制：

我们组建了全职的治理团队，同时，每个专业团队配备了对口的治理整改负责人。团队围绕"不漏报、少误报、快处理"的核心目标，持续优化工具、组织和流程。目前，我们的故障监控发现率稳定保持在 93%至 96%之间，日均告警量控制在 400 以内，平均每小时约 16 条，分摊到各专业条线后，每条线每小时约处理 3-5 条告警，基本实现了故障不漏、人能处理。治理日报和周报作为重要的过程总结和汇报形式，推动了治理工作的持续演进。

操作一线体验：

这是一个颇具特色且深受欢迎的机制，我们要求工具平台团队和流程管理团队每周安排一人到操作一线体验一天，亲身体验监控告警的处理、应急操作的流程以及日常巡检等工作。通过这种方式，工具和流程团队能够直观感受一线同学在故障应急过程中的痛点与需求，收集最直接的反馈，为工具和流程的优化提供生动、真实的素材，从而更好地支撑一线应急工作的开展。这种接地气的体验，极大地促进了跨团队的理解与协作，确保了交付的工具和流程真正符合实际操作场景的需求，最终提升整体应急响应效率和体验。

一线现场值班机制

一线值班优化：

标准化一线应急响应标准及流程

首先，在**值班协同**层面，我们着重对人员组织、工作职责及流程进行了精细化调整。我们对值班角色及其工作职责进行了重新定义，并配套建立了相应的绩效评估指标，例如，一分钟告警响应率、一线故障应急处理比例，以此牵引团队行为。同时，我们构建了一套面向生产环境的标准化应急响应流程，并特别优化了一线与二线、三线团队之间的协同工作流程，强调一线应急协同问题的闭环管理，确保问题得到彻底解决。为了支撑这些改进，我们从各专业团队挑选了专业人员，补充加入到一线团队。

其次，从**技术平台**的角度来看，我们构建一系列技术工具来辅助值班与应急响应工作。我们建设了应急协同平台，以加速故障发生时的指挥调度效率。此外，我们还提供了故障定界工具，使得团队能够快速准确地定位问题根源。更为重要的是，我们建立了通用的"应急六板斧"，为一线团队提供快速实现故障恢复的有效手段。

最后，**值班赋能**是确保一线团队具备足够应急能力的关键环节。我们建立了一套行之有效的应急移交机制，将经过验证的故障场景应急处理能力系统性地传递给一线值班团队。一方面，可以提升一线直接应急比例；另一方面，也解放二、三线节假日、夜间应急压力，二、三线在标准应急方面的投入，也转变成了收益。

值班经理能力培养-值班经理的八大修养

作为故障应急的核心人物，值班经理是应急指挥中的C位担当，是接棒第一人、坐镇指挥官，也是信息枢纽站、关键决策统筹人。我们根据应急指挥实战经验，**总结出了值班经理必须具备的八大修养**，这些修养同样重要，没有先后之分：

要有**有担当，有责任心**。在故障发生时，要主动响应，亲力敢为；

要具备**把控全局的能力**。能够在短时间内调度、协调多方，特别是场面混乱的时候，要能及时综合统筹，发现问题及时纠偏，要能控场、也要有多线程处置能力；

要具备良好的**沟通协调能力**

。能够在紧急情况下有效沟通，能快速提炼和精准传达故障信息，确保各方理解一致；

拥有**果断决策能力**

。以恢复业务为第一目标，盯紧应急时间线，关键时刻拍板决策应急处置方案。

要做到**知过去，通过程**

。值班经理应熟悉过去几年的重大和经典故障的处理过程及经验教训；

具备一定的**技术判断力。**

了解生产环境的技术架构，有各个技术栈的基础判断能力；

熟悉当前环境。熟悉当天值班人员、各领域的核心人员和关键角色，还要熟悉事、熟悉工具；

最后，要有强大的**压力管理能力**

。在面对重大故障的时候，如果 CIO 就站在你的身后，值班经理是否能够依然有序、有效地组织故障应急。

值班经理像一位技艺精湛的杂技演员，需要同时掌控八个球，并确保它们都能平稳运行。

应急文化建设

应急文化塑造：

让各团队精诚合作、聚焦目标，做难而正确的事。在前面已经详细探讨了各种应急机制与流程，正是这些具体的规定与要求，逐步统一了我们对应急的认知、意识与行动方式，我们的应急文化也是通过这些机制去逐步塑造的。

我们的应急文化被构建为清晰的四个层级：

担当履责

- 要求每一位 SRE 工程师都拥有"守土有责，自扫门前雪"的主人翁精神。

- 具体而言，每个人对自己所负责保障的范围，要能做到监控、应急不跑冒滴漏，守好自己的"责任田"。

知行合一

- 在 SRE 领域，仅仅"知道"是远远不够的，真正难得的是将知识转化为行动，并做到极致。
- 不仅需要掌握广泛的技术知识和领域专家知识，更要对内部的规范、标准了然于心，要有作为兜底把关人的思维，做了还要确保做到、做好，不要在复盘的时候拍大腿，例如，有监控，但监控是否完全落地了标准监控；架构是双活的，但业务是否对称双活；有应急场景，但应急是否演练验证过、并移交一线。

自我驱动

- 鼓励自我驱动的 SRE 文化。
- 每个团队的骨干，应该总有一两个痛点、难点挂在心头，日思、晚想、夜梦如何解决。
- 鼓励塑造个人的技术品牌与影响力，并运用这种影响力去推动和解决那些看似棘手的难题，形成一种自发的、持续的改进动力。

团队协作

- 在组织层面，消除了部门、团队之间的隐形墙，集中三个部门的力量，做成"应急"这一件事。
- 这是一种共同体意识，正如"成时举杯相庆，败时拼死相救"所描绘的那样，无论是在取得成绩时共同庆祝，还是在遭遇挑战时相互扶持，所有人都并肩作战。
- 在故障发生时，我们不仅要求各团队能够快速自证清白，排除自身嫌疑，更要拥有一种"拼死相救"的强烈主动意识和全局观，站在更高的维度上思考如何协助其他团队，共同渡过难关。

借用一句话总结，对于稳定性保障这项系统性工程，**组织能力是放大技术杠杆的关键生产力。**

分钟级响应能力

我们在应对系统故障时所追求的核心目标：高效、迅速地发现、定位并处理问题。在 SRE 实践中，实现分钟级的响应并非易事，它需要一套完整的体系来支撑。

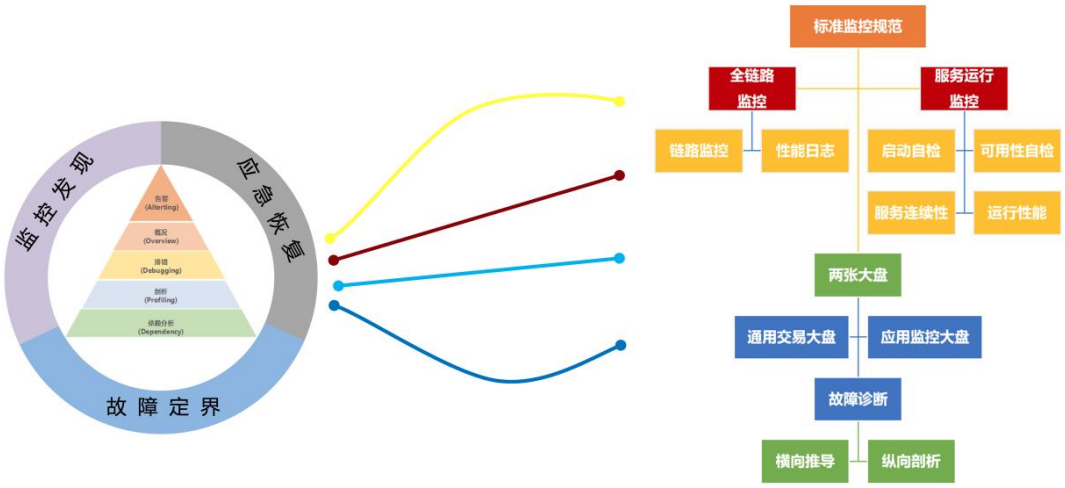
如何实现 "1"

首先，是感知快。确保所有关键、重要的服务都能在一分钟内被准确感知到其状态或故障。

其次，是响应快与协同快。这两者相辅相成，工具平台和治理工作要确保能支撑一线快速聚焦重要告警、能为一线提供线上线下协同工具。

最后，是量可控。要在不误报、不漏报的前提下，确保告警的数量是人有能力响应、处理得过来的。如果告警泛滥、事件堆积，即使具备了快速感知和响应能力，也会因为工作量过载而无法实现"1 分钟"的目标。最后这一项是治理工作范畴，这次不展开讲。

发现更快-1 分钟感知

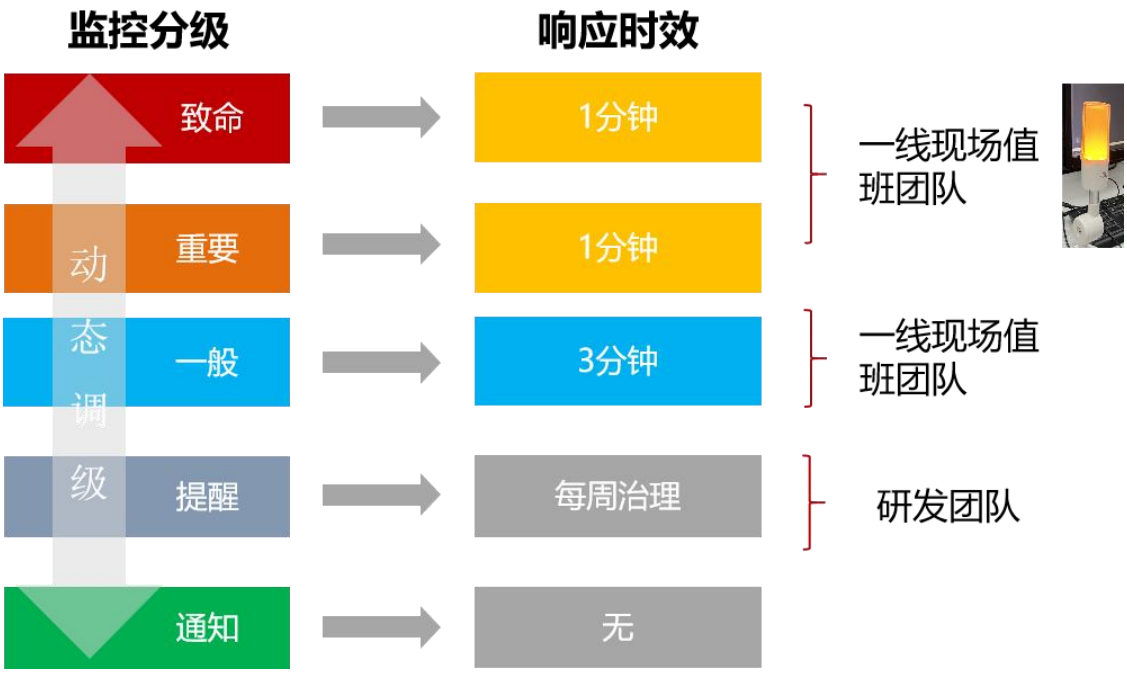


核心目标：在 1 分钟内感知故障，通过标准化的可观测性体系，实现快速发现与精准定位。

整体框架：围绕"全链路监控"和"服务运行监控"两大标准，构建统一、通用的监控策略。

基于自研的 RTP 开发框架，以日志+报文的形式打造应用的可观测性，通过全链路和服务运行两个标准监控确保关键、重要的交易、服务都有监控、且监控时效符合要求：

- 业务侧全链路监控-基于日志：
- 实现交易量、技术成功率、耗时等业务交易黄金监控指标
- 两个关键传播能力，实现技术失败、业务失败的快速定位：业务场景信息传播、技术失败信息传播
- 服务侧运行监控-基于应用主动上送的报文：
- 服务启动自检：在服务启动时，主动上送一次启动自检报文，防止进程起来，服务不可用、假死的情况
- 服务可用性：每分钟上送一次服务可用性自检报文，自检内容是可以自定义的，通用的包括到服务网关、缓存中心、消息中心等连通性自检，防止服务不可用情况
- 服务连续性：类似心跳机制，是通过可用性报文计算得出
- 服务运行性能：线程池、连接池等服务运行性能####### 响应更快-1 分钟响应



感知问题通过标准监控解决，但如果日常告警数量非常多，则会影响应急启动的判断，容易错失良机，也容易狼来了。为了能够快速聚焦重要的告警，做出准确的预判。我们细化了监控告警的级别、分工和处理时效要求，同时，通过告警的动态升级、置顶规则、合并规则，使一线可以快速聚焦重要告警，达到一分钟响应目标：

- **告警分级与响应时效：**

我们将告警按严重程度和影响范围细分为：

- **致命 (Critical):**

对应 1 分钟响应时效，由一线现场值班团队负责。这类故障被定义为"生死告警"，实施严格的准入/准出控制，彻底杜绝过去"致命不致命、致命需要二次确认"的模糊状况，确保告警的真实性和紧迫性。在值班现场，我们甚至设置了**物理的告警灯**

，专门用于强化致命和重要告警的提醒效果，确保即时关注。

- **重要 (Important):**

对应 1 分钟响应时效，由一线现场值班团队处理。

- **一般 (General):**

对应 3 分钟响应时效，由一线现场值班团队负责。

- **提醒 (Alert):**

对应轻微隐患，需进行每周治理，由研发团队负责。

- **通知 (Notification):**

对应信息知会，无需特定响应时效，由研发团队后续处理。

- **动态调级、置顶、合并机制：**

支持按渠道、对象、监控策略、关键字、告警状态、持续时长等维度设置告警的动态调级、置顶、合并规则。

- **动态升级/降级：**

告警等级可根据特定条件进行动态调整，确保告警的当前状态能准确反映问题的严重性。例如，某些"一般"告警在特定条件下可以升级为"重要"。

- **自动置顶：**

致命告警、特定类型的重要告警，能够自动在告警列表中置顶，确保其优先级和可见性。

- **告警合并：**

支持特定时间窗口内，相似、相关、同故障源（errSrc）的告警自动合并，帮助一线快速减少告警处理量，聚焦重要告警。

协同更快-线上线下应急协同

线上线下应急协同

：通过有效的流程和工具，实现现场值班人员、线上二、三线支持人员以及业务人员的快速集结与协同作战。

- 我们根据告警级别、业务影响情况制定了 4 个级别的**应急响应模型**

：一级和二级力求在 15 分钟内解决，三级故障要求在 30 分钟内解决，没有业务影响的四级故障则可以在一个小时内解决。应急响应模型可以帮助值班经理快速掌握应急调度人员要求、进展跟进要求、升级要求等。

- 当一线现场无法独立解决故障时，便会通过**应急协同工具**

启动线上线下多方协同。该工具能够**一键建群、自动拉人、启动会议以及自动对支持人员进入生产中心（ECC)进行来访授权、临时用户授权**

，快速集结二、三线技术支持和业务人员。我们提供了 **PC 端和移动端**

两种接入方式，方便不同场景下的协同。

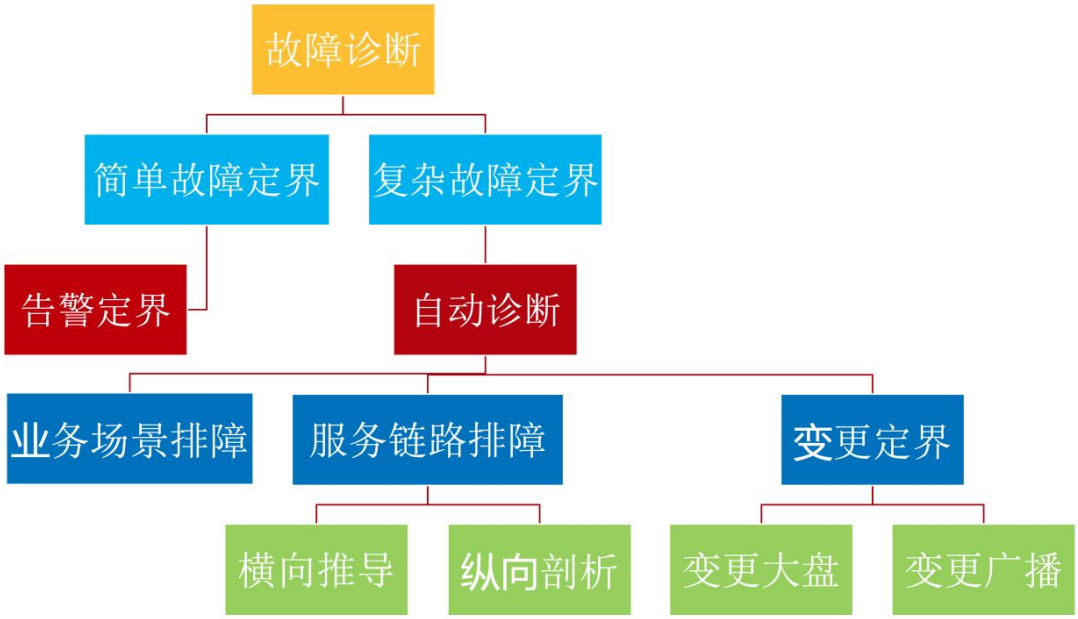
- 在此过程中，应急协同工具还提供了重要的应急协同辅助能力，例如，应急过程中的**信息共享、进度通报、协同任务分配以及决策指挥**

。

节点级定界能力

如何实现 “5”

故障应急启动后，最关键的是精准的节点级定界能力。



如何实现"5"，即一切围绕如何在最短时间内定界故障节点。

针对不同类型的故障有不同的定界方法。总体思路是：简单故障力求通过告警就能直接定界，复杂故障，我们先搭建了基础排障能力，包括横向业务场景排障能力、服务调用链排障能力和变更定位能力，再实现基于专家规则的自动诊断。

1. 简单故障定界：依赖清晰的告警

- 对于那些简单、直接且有明确告警的故障，我们力求通过告警信息直接进行定界，这就像是故障在以预设的信号直接告诉我们问题所在。

2. 复杂故障定界：构建基础的排障能力

- 面临复杂的、表现不明确的故障时，我们打造了以下基础排障能力：
- **业务场景排障能力：**

通过业务场景排障大盘，按照"业务场景-渠道应用-渠道接口/服务-服务链路 -异常应用定界"的层级关系，逐层观测业务运行情况。

- **服务链路排障能力：**

基于服务调用链监控以及应用输出的性能日志，实现技术失败、交易耗时的定界以及秒级性能剖析。

- **变更定界能力：**

通过"变更大盘"快速筛选近期所有变更信息，并利用"变更广播"机制及时知会相关人员，从而快速定位到可能导致故障的变更。

3. 自动诊断：实现专家规则自动诊断

- 在构建了业务场景、服务链路和变更定界这些基础能力之上，我们进一步实现了基于专家规则的自动诊断能力，将过往的故障处理经验和知识转化为自动化规则，当新的故障发生时，系统能够依据这些规则自动进行诊断和定界，大大提升效率和准确性。

简单故障定界—告警定界

简单故障：告警直接定界

我们高度重视告警的直接定位能力，也是早会、故障复盘会、治理必问的问题。

其核心在于：**1. 告警对象信息丰富：**

- 提供告警对象本身的属性，例如 CI 名称和关键属性，直接指向故障发生的具体组件。
- 向上关联其归属信息，包括业务模块、所属应用等，帮助理解故障在业务架构中的位置。
- 向下延伸至运行环境信息，如 OS/POD、宿主机、物理机，清晰地展示故障对象所处的底层基础设施，例如告警涉及的节点归属于哪个 POD、哪个宿主机、哪台物理机。

2. 告警内容信息丰富：

- 技术失败告警：基于全链路传播能力，告警内容包括了交易量、耗时、技术成功率三大黄金指标，还可直接展示错误源信息-errSrc (故障源应用)，快速识别技术根源。对于单一故障源引发的大量告警（即告警风暴），监控平台的告警合并规则能够自动将其聚合成一条告警，这极大地简化了告警列表，显著提升了告警定界的效率。
- 业务失败告警：自动关联告警影响的业务信息，阐明其影响的业务场景，方便从业务视角评估故障范围和影响用户的情况。

备注：errSrc（故障源）：一笔涉及多个服务调用的交易，在请求下一跳服务时，如果服务提供方报错，则由服务提供方在链路日志中将 errSrc 标记为自己，同时，输出具体错误信息（errMsg），并往回传播；如服务提供方超时，则由请求方将 errSrc 标记为服务方，并往回传播。

复杂故障定界—业务场景排障

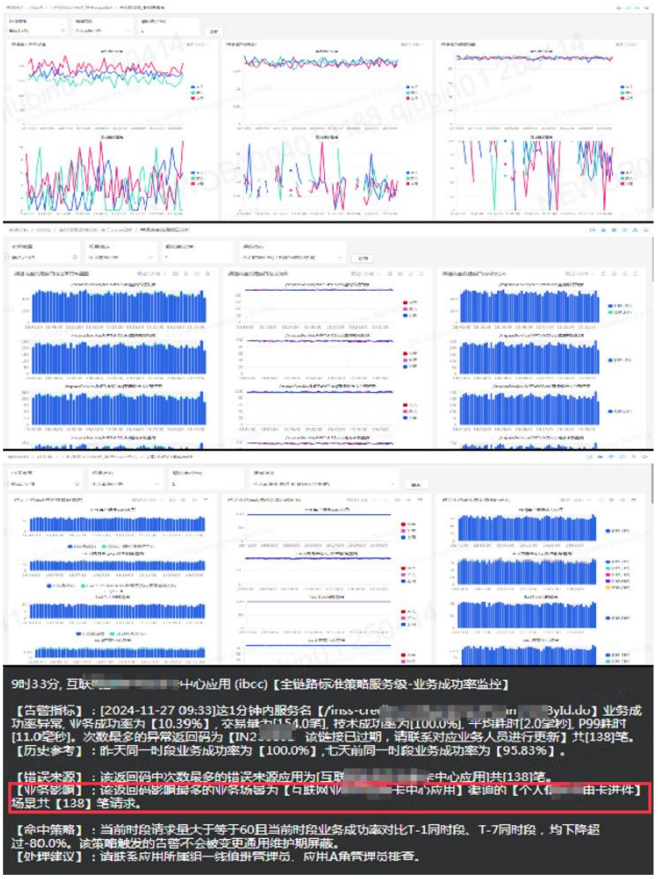


图1 基于业务场景查看各办理渠道的运行指标

图2 基于业务场景+渠道，查看该渠道接口运行情况

图3 基于业务场景+渠道，查看该渠道办理该业务场景所依赖的中后端应用运行情况

图4 全链路监控的告警信息增强，告警内容提示业务影响。

对于复杂故障的定界能力，我们是基于全链路监控平台去搭建的。通过将业务场景信息注入链路日志，利用全链路的信息传播能力，从前往后看，可以看到业务场景涉及的中后端关联应用有哪些，业务依赖的主流程接口有哪些，快速圈定排查范围，在一个页面内查看完所有相关接口的监控指标；从后往前看，可以快速知道 errSrc(错误源)可能影响了哪些业务场景、哪些上游的应用，从而快速评估故障的扩散范围。

基于业务场景的排障方法遵循一个清晰的层级关系进行逐层观测与定位：从最上层的业务场景，向下穿透到渠道应用，再到渠道接口/服务，进入服务链路，最终聚焦到具体的异常应用。

我们以一个具体的例子来阐述这一过程：考虑全行范围的"个人转账"业务场景。个人转账可在多个渠道进行，涉及用户登录状态检查、密码校验、扣款账户状态检查、余额判断、反洗钱等服务请求。假设某一天，个人转账的成功交易量突然下降，但不为零，而转账相关的接口技术成功率、耗时却表现正常。这可能是什么原因呢？一种可能性是某个渠道调用的余额查询接口出现了问题，导致转账业务到不了转账提交那一步。

那么，我们如何快速了解或判断个人转账业务会依赖于余额查询接口，并进而定位到该接口的具体问题呢？这就是业务场景排障发挥作用的地方。

复杂故障定界—服务链路排障

服务调用链排障能力，在全链路 trace 能力的基础上增加了传播能力，并基于"应用+业务模块"为核心的 CMDB 数据，将应用的链路日志和性能日志以及纵向技术栈指标建立关联关系，形成横到端、竖到底的可观测能力。

全链路可视与定界：

利用链路追踪技术，将服务间的调用关系可视化为链路聚合拓扑图，这使我们能够直观地识别出交易链路中的瓶颈或失败节点。

结合告警信息，例如技术失败或耗时异常的告警，可以直接导航到相关的链路视图，快速锁定可能存在问题的应用服务。

多维分析与精准定位：

在定位到问题应用后，通过应用多维分析大盘，可以从应用站点级、服务级、节点级分析本应用的交易情况，并支持按渠道、模块、交易类型、返回码等多维统计分析，帮助我们聚焦于有问题的交易或场景。

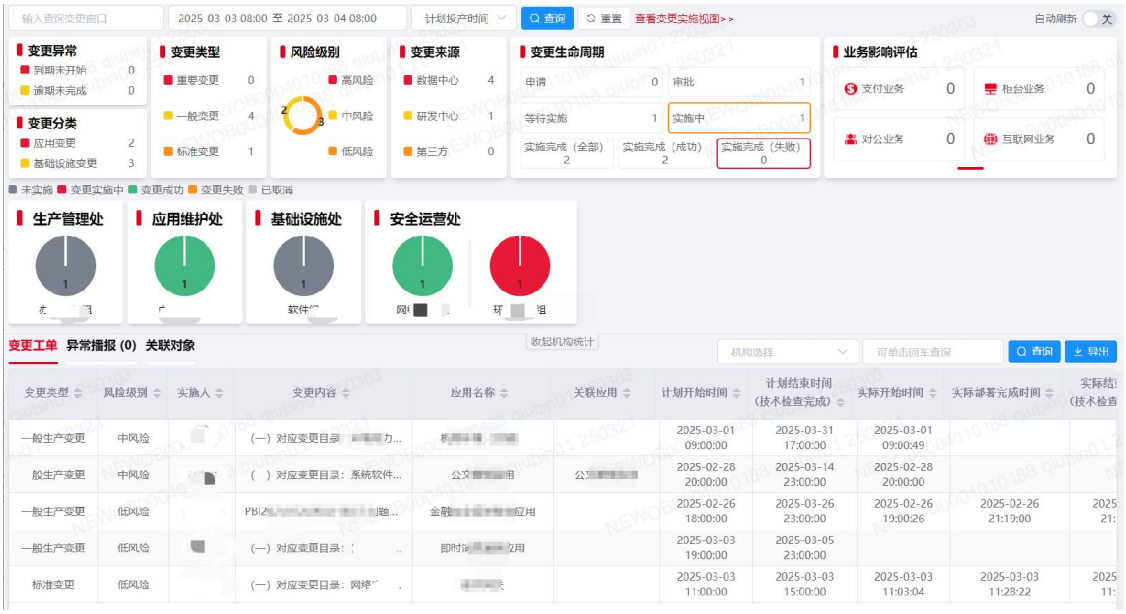
选择有问题的交易记录，我们可以进行单笔跟踪，精确定位到链路上的失败节点或耗时异常的节点，从而完成故障的初步定界。

性能剖析与根因分析：

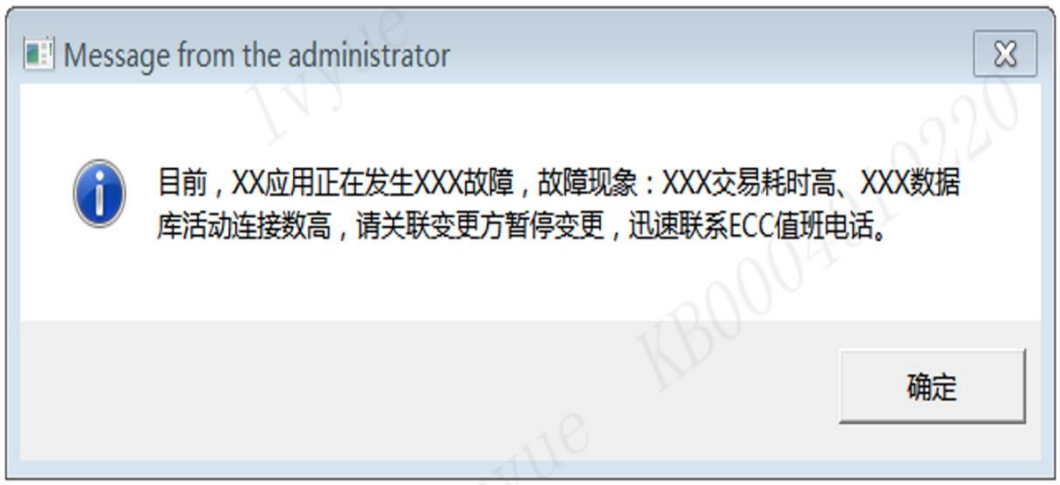
在完成定界后，如果需要深入分析性能问题或定位故障的根本原因，平台提供了秒级的应用性能分析能力。

应用性能分析大盘，支持对关键性能指标的剖析，例如，RPC 耗时、sql 耗时、druid 耗时、请求 TPS、堆栈日志分析、gc 日志分析等，帮助我们更细粒度地了解应用内部的运行状况。

变更定位



变更大盘：通过变更视图，快速查找应用本身、关联影响的变更信息，对于关键业务有影响的变更，提供快速筛选功能。这个视图支持多种维度的筛选，让我们可以根据变更的类型、风险等级、来源、当前的生命周期状态，以及对重要业务的影响程度，快速定位到特定应用相关的变更。



变更广播：

终端变更寻播对于变更窗口人多、变更多的场景，可以非常便捷地将故障信息实时推送到变更实施人员的终端设备上，及时提醒他们关注当前发生的故障，并评估自己的变更是否可能是故障源，以便及时终止那些潜在的、可能加剧故障的变更。

故障自动诊断

前面几个是我们的基础故障定界能力，从上面，我们可以看到，故障定界时要检查的指标很多，个人完全掌握所有指标相关的排查方法，基本不可能，跨专业领域的协同分析，成本很高，很难满足 5 分钟故障定界的要求。因此，我们建立了基于专家经验的故障自动诊断模块，实现模拟人工的故障排查过程。

在全链路监控的基础上，基于 CMDB 数据，将应用的链路日志和性能日志以及纵向技术栈指标建立关联关系，形成横向推导和纵向剖析两大能力，对全栈数据进行自动分析：

交易链路层的横向推导：

包括业务场景的故障源推导、服务级技术失败推导、耗时推导等，通过定界故障源系统的站点级、服务级、节点级情况，分别对应站点流量切换、服务降级/限流、进程重启/隔离等应急措施。

纵向技术栈剖析：

包括应用资源、软件资源、系统资源、基础设施，重点检查生死指标和性能指标。

横纵诊断结果以列表的形式展示，查看诊断返回详情就能获取异常节点信息：

点击"详情"还可以进一步查看该诊断项的详细诊断过程，按模块失败总数从高到低排序或按时间从开始到最近检查结果。

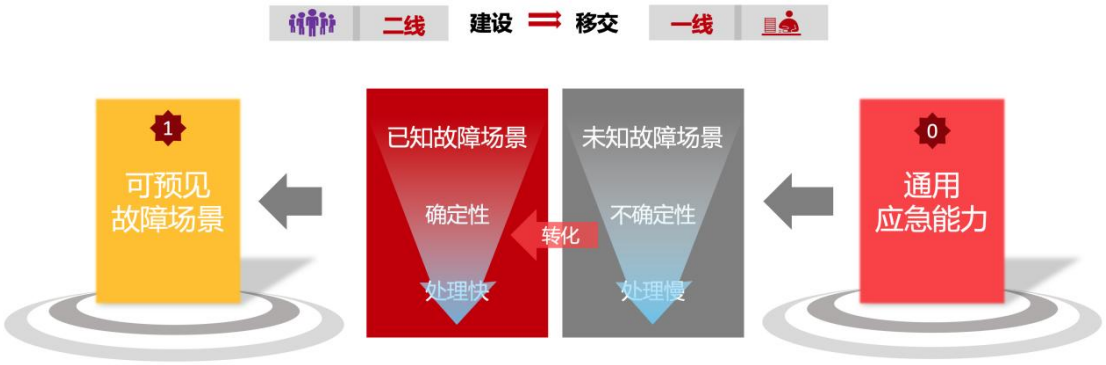
落地价值

- 从"发现"到"定位"——将原本依赖少数资深专家的隐性知识，转化为可持续的自动化能力；
- 在典型故障场景中，诊断时间从数分钟缩短至数十秒；
- 解放团队在繁杂指标间的排障定界投入，将更多精力投入到修复与优化上。

打造可预见故障场景

如何实现“10”：从通用应急能力向可预见故障场景的持续转化

在应急处置方面，未知的故障场景不确定性高，应急慢，已知的故障场景确定性高，处理快，因此，应急恢复的建设思路是先建基础通用应急能力，再逐步打造可预见故障场景，实现快速应急。



######## 打造通用应急能力-应急六板斧

我们构建了一套通用的应急能力体系，也就是我们常说的"应急六板斧"：切流、重启、扩容、隔离、降级限流以及回切。

这六板斧分别对应着不同的故障场景和应急需求，旨在帮助我们快速、有效地响应和处置各类系统异常。

- **切流：**

主要应用于站点级或单元级的故障。当某个数据中心或区域发生问题时，我们可以迅速通过工具（比如"同城双活应用快速切流"、"防火墙一键逃生"）将流量切换到健康的单元，确保业务连续性。

- **重启：**

适用于服务出现 Full GC、OOM 或服务状态异常等场景。通过"一键启停"工具，我们可以便捷地对应用服务进行快速现场保留、重启，重启操作往往能有效恢复系统功能。

- **扩容：**

是解决资源紧张类故障的利器。当系统面临预期的或突发的流量洪峰、或者资源（CPU、内存、文件系统、数据库表空间等）成为瓶颈时，我们可以通过相应的工具快速增加资源供给，提升系统承载能力。

- **隔离：**

适用于集群内部出现单个节点故障的情况。在这种场景下，通过负载均衡的"节点隔离"或 服务运营管理平台的"服务隔离"工具，我们可以将故障节点快速从集群中剔除，防止故障蔓延影响整体服务，同时还能保留故障现场，便于后续分析原因。

- **降级/限流：**

主要适用于链路弱依赖、避免影响其他业务的故障。在出现上游依赖服务不可用或自身资源不足以应对当前流量时，我们可以通过例如"互联网渠道交易限流"或控制"银联支付贷中开关"等手段，对非核心功能进行降级或限制流量，从而优先保障核心业务的可用性。

- **回切：**

适用于解决由变更操作（如应用部署、配置更新）引入的问题。当发现变更导致系统异常时，我们可以快速通过自动化部署平台实现应用版本的自动回退。

总而言之，这"应急六板斧"，配以相应的自动化工具，构成了我们应对复杂分布式系统故障的通用应急能力基石，能够覆盖绝大多数常见的应急场景。

可预见故障场景建设

所谓的可预见故障场景，是告警出来就可以直接应急的，这中间无需耗费时间排查，直接进入应急处置流程。

可预见故障场景，主要通过"历史故障分析"、"故障复盘"、"经验总结"、"头脑风暴"等方式，将成熟的故障定界及处置经验，形成可预见故障应急场景清单，实现确定的故障场景有"确定的告警"、"确定的应急处置方案"、"确定的应急动作"：

确定的告警： 清晰定义故障发生的监控告警，确保能第一时间被发现。

确定的应急处置方案： 针对特定场景，有预先设计好的、标准化的应急方案。

确定的应急动作： 方案中的具体操作步骤必须细化、可执行，避免临场决策与判断。

一键应急能力

为了应对复杂的系统挑战并有效处置突发事件，我们构建了一套集报警、知识和应急处置于一体的"一键应急能力"平台，旨在将应急流程的各个环节无缝衔接，从而实现端到端的快速响应。

复盘改进

进入故障复盘环节前，不妨先思考两个核心问题：关于故障复盘，我们最害怕和最期待的是什么？或许有人会说，最害怕的是复盘会变成问责会，哪里还敢有什么期待呢？

复盘会的重点是发现问题并改进，我们内部一直倡导这样一个信念：**能在内部复盘的故障，那都是上天给我们的一次改正机会，**

如果你们的领导是抱着讨论的态度和你复盘的，相信能让大家放心把问题讲出来，把改进的要点讲清楚。

XXX 事件分析报告

一、事件现象

二、事件情况简述

北极星指标统计：应急处置时长 xx 分钟

1. 故障发现时长：xx 分钟
2. 故障定位时长：xx 分钟
3. 应急恢复时长：xx 分钟

三、事件处理情况

四、事件影响

【网络舆情情况】：

【客户投诉情况】：

【资金损失情况】：

【交易影响情况】：

【业务影响情况】：

【其他情况】：

五、系统、业务处理流程/场景

六、事件原因分析

直接原因：

根本原因：

管理原因：

七、 事件责任认定

责任系统：

主要责任人：

原由：

八、待改进点分析

一）故障为什么会发生

关键过程	具体分析点	改进点分析情况（具体是什么，如何规避问题再次发生）
设计环节	是否方案有设计问题，如有规避问题再次发生	
开发环节	是否为程序设计问题，是否留改进	
测试环节	测试是否已发现，为何未发现	
变更环节	是否为变更发布导致，如有规避问题再次发生	

二）如何更快应急恢复

关键过程	具体分析点	改进点分析情况（具体是什么，如何规避问题再次发生）
监控环节	监控是否第一时间发现	
故障定界	故障定界工具是否可以发挥作用	
应急处置	应急方法是否做到一键操作或降本化	
现场协同	现场协同是否有序，是否留改进	

九、后续改进措施

十、举一反三（可选）

以上是一份标准化的"XXX 事件分析报告"模板，涵盖了从事件现象到举一反三等十个章节，力求全面、深入地分析故障。这份模板详细包含了以下关键要素：**事件现象：**

对故障的初步观察和描述。

事件情况简述：

精确记录故障发生的关键时间点，包括故障发现、定位、恢复的时间，以及核心北极星指标。

事件处理情况：

详细记录故障处理过程中的每一步具体操作和决策。

事件影响：

全方位评估故障的影响，包括但不限于网络舆情、客户投诉、直接或间接的资金损失、对交易和核心业务功能的影响，以及其他次生影响。

系统、业务处理流程/场景：

回顾并分析故障发生的具体系统、相关的业务处理流程以及涉及的场景，为后续分析提供上下文。

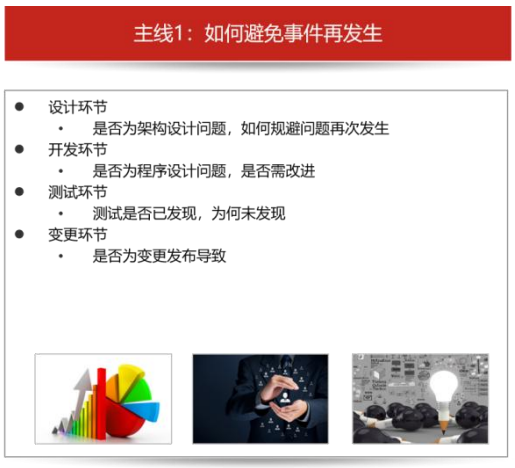
事件原因分析：

系统性地从直接原因、根本原因和管理原因三个维度深度挖掘故障发生的深层逻辑和诱因。

事件责任认定：

明确在故障中哪些系统和人员负有主要责任，并分析其理由。

待改进点分析：



- 故障发生的原因：从设计、开发、测试、变更等研发和发布流程的关键环节入手，深入剖析是否存在问题，例如架构设计是否合理、程序是否存在缺陷、测试方案是否充分、变更流程是否规范等，并思考如何从源头避免同类问题再次发生。
- 应急恢复过程中存在的问题：聚焦故障应急响应、处置效率，分析监控系统是否能第一时间准确报警、故障定界工具是否有效、应急处置方法是否最优、现场多团队之间的协同是否顺畅等，识别响应流程中的不足之处。
- 后续改进措施：基于"待改进点分析"的结果，制定具体的、可执行的、有明确责任人和完成时间的改进计划，确保复盘成果真正落地。
- 举一反三 (可选)：将本次故障中获得的宝贵经验教训进行提炼和推广，识别在其他系统或领域可能存在的类似风险，并主动进行排查和预防。

（三）总结及展望

XX 银行通过"1510"工程牵引整体运维能力的提升，构建业务连续性保障能力。本案例聚焦应急组织流程、故障发现、故障响应、故障定界、快速恢复、复盘等方面的改进经验，从应急组织架构、应急管理机制、一线应急机制、组织文化营造、分钟级响应能力、节点级定界能力、可预见故障场景建设等方面阐述从组织到工具的应用应急能力提升方法。为规模体量相似的中小商业银行、同业提供可借鉴的故障应急能力提升思路：

1、了解组织在故障应急提升方面的作用、应急组织机制如何实现应急能力提升的闭环优化、值班经理应当具备哪些修为

2、了解分钟级响应、节点级定界、可预见故障场景、一键应急等能力的建设思路

5.3.9 腾讯 IEG SRE 应急响应实践

SRE Elite 精选原因：

方案亮点在于依托蓝鲸基座,以平台工程的方式整合监控、CMDB、权限等能力，形成"告警→响应→诊断→恢复→复盘"全链闭环。标准化告警接入配合巡检确保观测确定性，自动升级流程打通跨团队协作；APM+eBPF 零侵入全栈观测结合 LLM Agent 智能根因定位，将诊断压缩至分钟级；混沌工程与每日过载验证保障过载保护有效，常态化 On-Call 与演练提升团队实战熟练度，使多数故障实现分钟级自愈，真正把应急从人治升级为体系化工程。

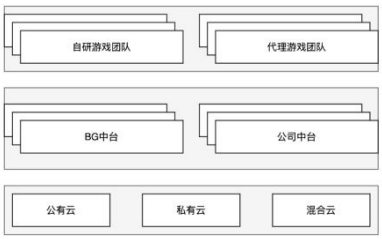
（一）背景及设计原则

游戏应急响应的目标和痛点

游戏应急响应的目标和痛点

我将首先基于腾讯游戏的特点，与大家探讨在应急响应领域我们所遇到的困难和痛点。随后，我将针对这些痛点，详细分享我们如何基于"蓝鲸基座"构建一个通用的应急响应方案。最后会简要分享下我在执行平台业务应急响应上的一些思考和经验。

腾讯游戏的特点



现状：
- 研发团队：自研和代理
- 游戏架构：模块多差异大，发行区域：国内和海外，全球发行
- 支撑平台：BG内、BG外平台服务
- 规模大：单游戏体量和整体体量

特点：
- 上下游涉及服务、人员多（多组织，多角色）
- 异构性非常强
- 运营环境复杂

在讨论腾讯游戏的特点及其 SRE 稳定性面临的挑战时，需要认识到其核心特性：**业务模式与产品多样性:**

腾讯游戏应急响应的特点

腾讯游戏涵盖自研与代理模式，同时进行国内外的全球发行，新生代游戏更是普遍面向全球市场。

多品类（如动作类）、多玩法、多运营需求以及不同工作室的存在，导致其架构差异巨大、模块较多。

中台体系的支撑:

- 为支持游戏的快速上线与高效运营，IEG（互动娱乐事业群）构建了 BG（业务群）级的中台，同时公司层面也有统一的中台能力作为补充。

庞大的体量规模:

- 腾讯游戏的整体体量及单个游戏的规模都相当庞大，这使得许多潜在问题（例如效能瓶颈）的影响会被显著放大。

复杂的协作与架构:

- 由于涉及多样的业务模式、产品及中台系统，整个生态系统的上下游参与者众多，形成了多组织、多角色的复杂协作关系。
- 核心架构具备显著的异构性特点。

- 运营环境复杂，不同游戏特性在环境部署需求上有明显差异。（比如延迟敏感的游戏会要求就近部署、多接入层动态调度等）

这些特点共同构成了腾讯游戏在 SRE 稳定性实践中必须面对和解决的独特挑战，尤其体现在大规模、多变的环境下保障服务的连续性和可靠性。

腾讯游戏应急响应的目标

腾讯游戏应急响应的目标

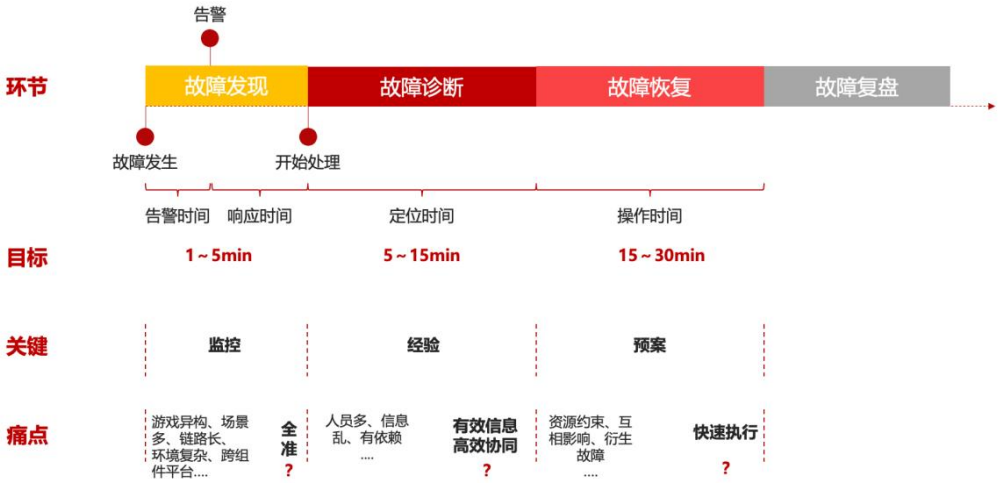
游戏作为一个对**用户体验要求极高**的产品，即使是微小的网络抖动或连接中断，都可能导致大量玩家投诉甚至流失。鉴于此，在**支付、登录、注册等关键游戏业务场景下**，我们的稳定性保障设定了严格的服务级别指标，旨在：

- 确保能够在故障发生后一分钟内完成发现；
- 实现故障在五分钟内完成诊断；
- 目标是在十五分钟内完成系统的恢复，从而最大限度地减少对玩家体验的影响。

腾讯游戏应急响应的痛点

腾讯游戏应急响应的痛点

腾讯游戏应急响应的痛点



1. 故障发现（1~5min）

在这个阶段，目标是尽快发现故障。关键在于实时监控，确保能够及时捕捉到系统中的异常。由于游戏架构异构、场景多且链路长（涉及多组件平台）、游戏部署环境复杂，能否做到监控的全和准是该阶段的主要痛点。

2. 故障诊断（5~15min）

此阶段的目标是精确诊断故障原因，依赖于过往经验和技术积累。有效的信息收集对故障的快速定位至关重要。由于游戏自身模块较多且单一功能场景会涉及多组件多平台，链路长，这会导致无法快速的收集信息，此外模块与模块之间、服务与服务之间会存在相互影响，一个故障的产生可能不是单一链路的问题，需要整体协同定位，如何高效协同找到根因也是当前环节的痛点。

3. 故障恢复（15~30min）

目标是在这个阶段快速恢复服务，确保系统尽快恢复正常运行。关键在于快速执行预案，并确保不会丢失重要数据。痛点集中在资源的有限性以及不同系统之间的相互依赖，可能导致恢复时间较长。而且，故障之间往往存在连锁反应，需要谨慎操作以避免扩大影响。

基本止损优化策略尝试，有用，但不够

实际上，在着手构建专业的应急响应系统之前，我们团队进行了多方面的探索与实践。这其中包括了：

1. 告警优化**基础优化：**

对 CPU、内存、磁盘、IO、网络等基础性能进行监控，确保系统的健康状态。

业务服务优化：

关注业务服务的调用量、成功率等关键指标，确保服务的稳定性。

用户体验优化：

监控用户操作过程中的登录、下载、支付、单品等环节，提升用户体验。

综合 SLI/SLO：

综合考虑服务质量指标，通过 SLI/SLO 来衡量系统的可靠性。

2. 架构优化**冗余、容错：**

通过增加系统的冗余设计，提高系统容错能力，确保在故障发生时能够快速恢复。

负载均衡：

通过负载均衡技术，分担系统的压力，避免单点故障。

在线更新与升级：

系统能够在线更新与升级，确保在不停机的情况下进行功能扩展和优化。

3. 部署优化

跨区域部署：通过跨区域部署服务，提高系统的可用性和容灾能力。

网络优化：确保数据的快速传输，优化网络架构，减少延迟。

服务接入与数据层优化：优化服务接入方式，简化数据传输流程，减少瓶颈。

4. 预案执行

自动化预案：构建自动化的应急预案系统，实现故障的自动响应与修复，减少人为干预，提高恢复效率。

资源调度与混沌工程：通过资源调度和混沌工程来提升系统的弹性和抗压能力，确保在高负载时仍能保持稳定运行。

这些工作带来了显著的效果，它们有效解决了我们所面临的大部分稳定性问题，极大地改善了系统的整体可靠性。

然而，尽管取得了这些进展，我们仍然面对着一些棘手的问题，这些问题是上述单一或局部的优化手段无法彻底解决的。这正是促使我们进一步思考和构建一套更为全面、系统化的应急体系的根本原因。

我们意识到，需要一种更具穿透力的方法来应对那些复杂、突发的未知风险，而不仅仅依赖于对已知问题的改进。

建设应急系统之前的必要性阐述

在阐述建设的必要性之前，我们先来看两个案例。

案例一：游戏更新导致存储层代理模块 CPU 满载，无法及时触发告警引发故障 背景：

游戏 A 11/7 日发布了新版本，客户端版本新增了协议，会放大存储层压力，11/13 日 A 业务活动放量，导致存储层 proxy 模块 CPU 持续 100%，引发故障。

事故过程：

- 19:56 游戏 A 运维收到告警，显示"用户登录失败超过阀值"，遂联系负责登录后台的开发同事第一时间核查和解决。
- 19:57-20:30 游戏 A 运维、研发协同存储层负责同事一起排查，期间尝试重启游戏后台服务、对服务进行降级等措施，暂时恢复了登录，但是部分游戏功能依然受到影响。
- 20:30-20:35 存储层服务团队定位到一台 proxy cpu100%，遂立即进行扩容。
- 20:40 扩容完毕，业务 A 逐步恢复……..**事故原因：**

……..

proxy 模块 CPU100%却无告警核查结果：告警策略配置了但未生效，导致没有触发告警。

……

故障影响大于 40 分钟**案例二：热更新引发登录失败问题，误判为版本问题导致故障排查延迟 背景：**

游戏 A 6/11 日发布客户端资源热更新，更新后出现登录成功率下降。

事故过程：

- 6/10 15:00~15:30 登录组件运维按计划执行某一模块升级，灰度和全量期间观测无异常。
- 6/11 16:00 游戏 A 发布热更。
- 6/11 16:08~16:14 游戏 A 运维收到登录异常告警，期间登录组件运维也收到游戏 A 登录异常告警，经排查发现自身服务成功率无异常，同时周知游戏 A 时，了解到 A 刚发了热更，遂认为是版本问题，于是等待 A 业务修复。
- 6/11 16:15~16:40 游戏 A 运维和研发联合排查，期间排除了新功能的影响，定位到用户的异常均在登录模块中的 lua 脚本，而该 lua 脚本这次没有改动，且和本次版本更新内容没有关联，并且只是部分玩家出现异常。
- 6/11 16:41 游戏 A 寻求登录组件帮助，组件运维联系研发进行排查。

- 6/11 16:42~16:50 登录组件研发经排查，发现后台存在某些异常的 ID 有一直在发起登录请求，也成功返回了数据，但是却没有登录成功，确认是由于 A 游戏本次更新触发了 6/10 组件升级带来的 BUG 导致。
- 6/11 16:51~16:56 登录组件运维紧急回滚版本。
- 6/11 17:00 游戏 A 登录异常逐步恢复。

故障影响近 1 小时

案例分析

案例分析

第一个案例揭示了告警覆盖的盲区与决策失误的代价，一个关键模块告警的缺失直接导致故障诊断出现偏差，进而将故障影响扩大到超过 40 分钟。这凸显了完善的可观测性、尤其是关键指标的全面监控与告警机制的重要性。

第二个案例则指出了跨部门或跨岗位协同的潜在漏洞，诊断过程中关键人员的遗漏参与，使得问题解决耗时近一个小时。这强调了在应急响应中建立清晰、敏捷的协同机制与信息共享渠道的必要性，确保所有关联方都能及时到位，共同参与故障处置。

这两个案例虽然简单，但都指向了稳定性保障中的核心痛点：一方面是信息获取的完整性和准确性（告警缺失、可观测性不足），另一方面是故障处置的人员协同效率。后续我们将探讨如何通过系统性的方法和工具来弥补这些不足，提升整体的 SRE 稳定性保障能力。

同时，我们深入分析了以往耗时较久的故障，通过复盘发现，问题主要集中在以下三个核心环节：告警、协同与处置。

告警痛点

首先来谈谈告警环节。这里面的痛点多种多样，可能表现为：

告警缺失：

本应触发的告警未能如期发出，导致问题发现不及时。

告警延迟：

告警未能实时送达，延误了响应的最佳时机。

告警泛滥：

系统产生过多无效或低优先级告警，使得真正关键的问题告警被淹没在信息的海洋中，难以被有效识别和处理。

接下来是协同环节。高效的协同是应急响应的关键，其挑战在于：

人员未及时到位：

故障发生时，未能迅速集结所有必需的响应人员，导致处置力量不足。

关键角色缺失：

在响应过程中，发现缺少对故障类型至关重要的专业人员，影响问题分析和决策效率。

最后，也是最考验 **SRE 硬实力的处置环节，问题处置。**

快速有效地解决故障极具挑战性，因为问题的根因复杂多样，而且对 SRE 的个人能力、经验积累，以及整个工程体系的成熟度提出了非常高的要求。故障处置的速度和成功率在很大程度上取决于工程师的能力、自动化工具的支持程度和预案的完善度，是一个概率与能力综合影响的过程。

在这三个环节中，告警问题可能并非总是最直观的痛点，但它却是整个应急响应流程的起点。下面我们将重点展开讨论告警方面的问题。

告警痛点：游戏业务的监控告警配置痛点困难分析

游戏业务有其显著特点，尤其体现在监控指标的复杂性和多样性上。我们发现不同类型的游戏所需的监控指标截然不同。例如，对于 FPS（第一人称射击游戏）类游戏，其基础监控项就已数量庞大。

类别	监控项	说明
下载	下载量/成功率/下载速度	关键项
安装	安装量/成功率/耗时	关键项
登录	域名劫持	域名解析异常(劫持)/跨网
登录	登录成功率	各个步骤的转化率
登录	登录时长	玩家平均登录时长
登录	游戏内登录	辅助判断掉线以及网络异常
登录	游戏内登出	辅助判断掉线以及网络异常
在线	大厅在线/掉线/重连	判断大厅基础体验
在线	gamesvr 网络连接数	辅助判断大厅掉线
在线	gamesvr 当前房间数量	辅助判断大厅掉线
单局	网络延迟/loading 耗时/FPS	单局体验判断情况
单局	单局在线/房间数/掉线	单局基础运行情况判断

类别	监控项	说明
单局	流量监控	辅助判断游戏内异常/掉线异常/网络异常
单局	平均丢包率/平均延时	监控玩家网络情况/局内体验
注册	注册监控	渠道、平台注册
客户端崩溃	崩溃率和崩溃量	监控玩家游戏体验
FPS	FPS 明细	监控游戏流畅度
匹配	匹配时长/超时次数	监控玩家游戏体验/网络异常
角色创建	角色创建流水监控	关键项
金钱/道具	金钱消耗流水监控	关键项，道具分类监控
购买	购买失败量	监控玩家游戏体验
周边系统	周边系统的访问质量	关键项
后台逻辑异常	逻辑异常监控告警	常规项
服务器基础	CPU/内存/IO/流量/ping/…	常规项
后台基础	进程不存在/core/通道	常规项
容器平台	pod/node/apiserver/….	常规项
文件上传	录像上传量/成功率	监控玩家游戏体验

一个 FPS 类游戏的基础监控

除了 FPS 外，还有 RPG, STG, MOBA, SLG，卡牌，开放世界沙盒等各种游戏类型。

更具挑战性的是，这些指标会受到多重因素的影响，包括：**游戏类型差异：**

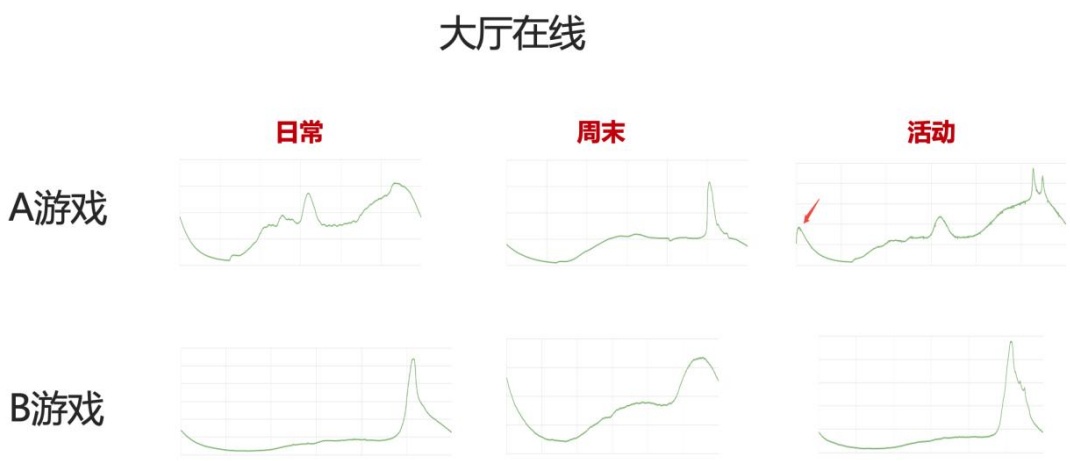
不同类型游戏的核心指标不同，例如射击类关注延迟和帧率，策略类关注对局完成率和匹配时长。

地域特性：

不同地域的网络环境、用户习惯会导致游戏监控指标表现出差异。

运营活动：

新版本上线、限时活动等会带来流量骤增和用户行为变化，直接影响游戏监控指标。



举例来说，即使是**同类型的两款游戏（上图 A 和 B 游戏）**

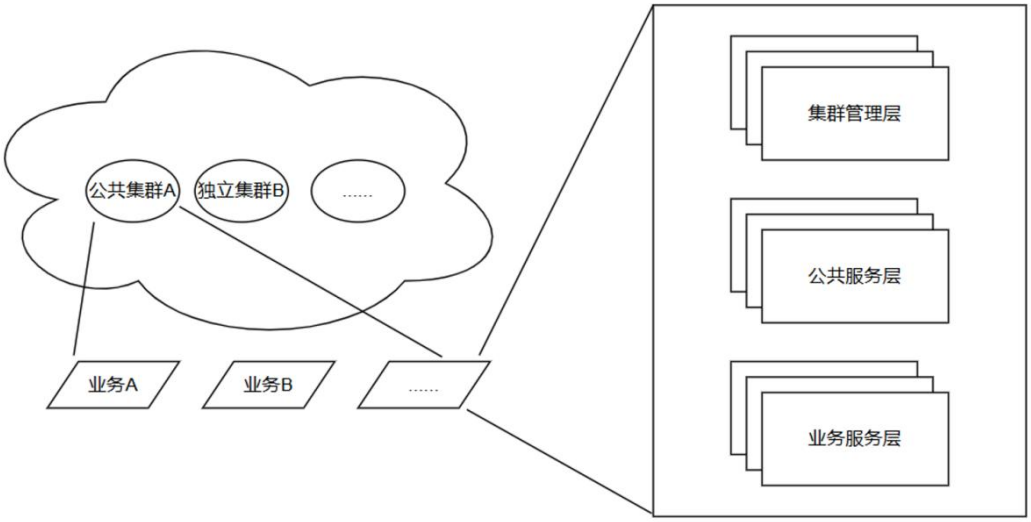
，由于用户基数、活跃时间分布或运营策略的不同，其**指标曲线会展现出非常显著的差异**

。此外，同一款游戏在不同时间段（如工作日与周末、白天与夜晚）的在线人数和行为模式变化，**也会导致监控指标波动巨大。**

这意味着我们无法简单地采用一套统一的标准或规则来对所有游戏进行有效的监控。如果试图为每个游戏的每个细分场景都配置详尽的监控规则并确保其精准性，监控项的数量将呈爆炸式增长。尽管我们可以尝试通过分级或降噪技术来管理海量告警，但这又极易导致重要异常被遗漏，从而影响应急响应的及时性。

这正是我们在游戏监控实践中遇到的一个现实的、迫切需要解决的问题。下一步，我们将探讨在面对平台业务时，情况又有何不同。

告警痛点：平台业务的独特挑战



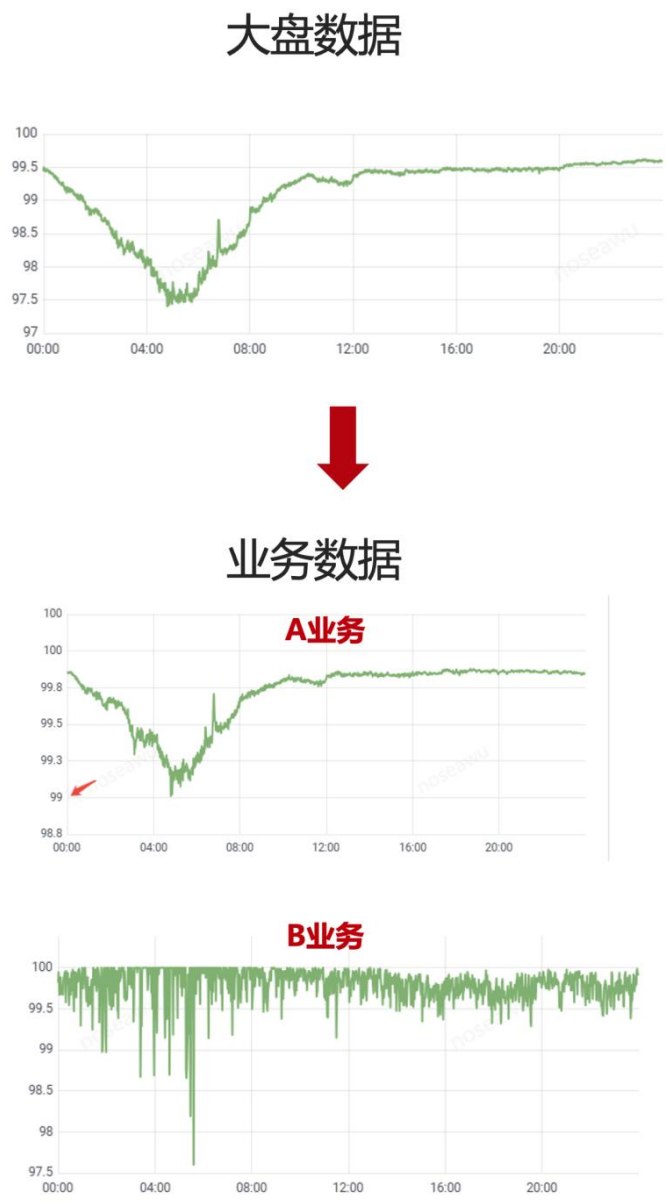
从上面的部署架构模式可以看到，对于平台业务，尤其是服务众多内部客户的平台业务，其特点是面向外部表现为一个统一服务，但在内部却支撑着可能上百个差异巨大的业务。正如云计算平台通常会遇到的情况，一旦核心平台出现问题，将可能波及大量下游客户。

在游戏业务体系中，我们的平台业务主要分为了 面向 c 端游戏平台业务，面向 c 端组件平台业务，面向后台组件平台业务。

类别	监控项	说明
集群维度	A 服务大盘请求量	基础项，可以发现端侧异常
集群维度	A 服务大盘请求成功量	基础项，可以发现自身或者下游异常
集群维度	A 服务大盘请求失败量	基础项，稳定且敏感业务需要此项数据监控
集群维度	A 服务大盘请求成功率	基础项，可以发现自身或者下游异常
集群维度	A 服务大盘错误码总量	基础项，稳定且敏感业务需要此项数据监控
集群维度	A 服务大盘细分错误码总量	基础项，用于针对特定错误码监控
集群维度	A 服务大盘请求延迟	基础项，用于服务能力的及时感知
集群维度	A 服务大盘容量水位	基础项，用于容量异常的及时感知
集群维度	A 服务大盘配置检查	基础项，用于部署容灾、过载保护等配置监控
集群维度	B 服务……	
业务维度	A 业务请求量	基础项，作用同大盘，但是细分到 A 业务

类别	监控项	说明
业务维度	A 业务请求成功量	基础项，作用同大盘，但是细分到 A 业务
业务维度	A 业务请求失败量	基础项，作用同大盘，但是细分到 A 业务
业务维度	A 业务请求成功率	基础项，作用同大盘，但是细分到 A 业务
业务维度	A 业务错误码总量	基础项，作用同大盘，但是细分到 A 业务
业务维度	A 业务细分错误码总量	基础项，作用同大盘，但是细分到 A 业务
业务维度	A 业务请求延迟	基础项，作用同大盘，但是细分到 A 业务
业务维度	A 业务容量水位	基础项，作用同大盘，但是细分到 A 业务
其他	基本同游戏业务	
.....	.....	.....

以上是一个面向后台组建平台业务的基础监控指标列表



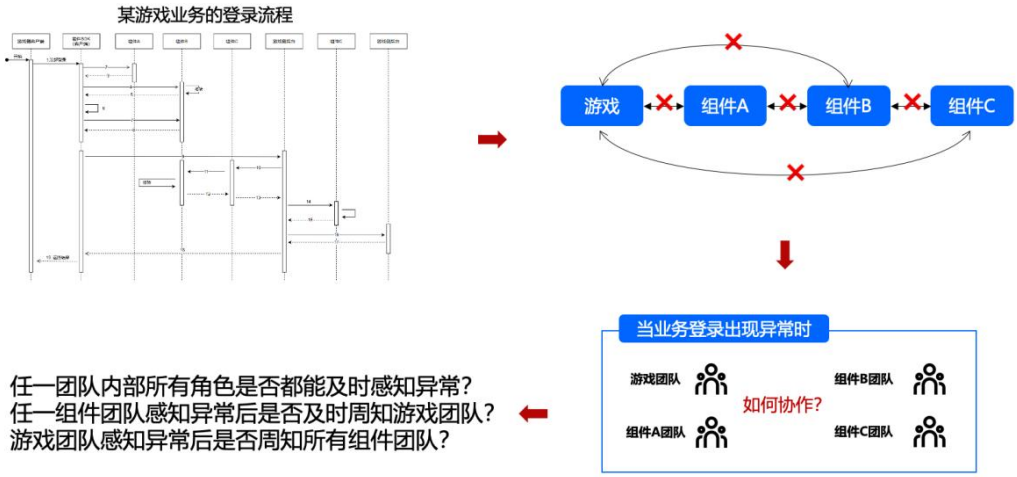
虽然平台业务相比游戏业务，SLI 项不多，但是由于接入游戏业务众多，且游戏业务的异构性、服务接入方式、量级、运营周期、运营活动不同，会导致业务维度监控差异较大。

面对这样的场景，传统的整体监控模式显然不足以应对，必须下沉至每个具体业务层面构建精细化监控体系。然而，游戏业务的独特性在于其高度的多样性和快速迭代。这导致在游戏行业中构建普适且全面的监控体系面临巨大挑战，往往陷入覆盖面与精细度难以兼得的困境，即难以同时实现 0 死角覆盖与全面监控。这暴露了监控层面的固有难题。协同问题也因游戏场景的复杂性而进一步凸显。

协同痛点

协同痛点：信息孤岛导致无法协同。

ß



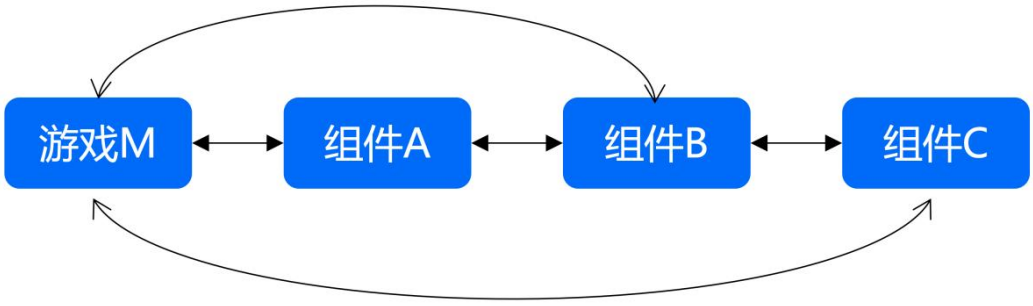
IEG 的业务场景极其复杂多样，与其说是一个简单的技术栈，不如说是一个由多个组件交织而成的庞大生态系统，例如，一个简单的登录流程就可能牵涉到多个组件，而组件与组件之间、游戏与组件之间的交互关系盘根错节。在这种高度复杂的环境下，一个微小的异常就可能迅速蔓延。

在过往的实践中，我们发现一个普遍存在的、也是非常耽误我们处置问题的情况：**当故障发生时，研发或运维团队往往倾向于先排查自身负责的系统，而不是快速判断是哪一个环节出了问题。**

这种"先看自己"的习惯虽然体现了对自身系统的负责，但在跨组件、跨团队协作定位和解决问题的场景下，却白白浪费了黄金的应急响应时间。这正是我们在实践中深刻体会到的痛点，也是推动我们不断优化应急响应流程的重要驱动力。

处置痛点

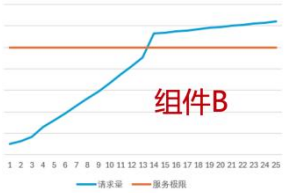
处置痛点：诊断和恢复时间过长 案例：



游戏 M 登陆流程及组件**背景：**

游戏 M 预计于周年庆当日 0 点举行活动，预计登录峰值增长 200%，且提前与组件 A、B、C 做了报备，组件 A、B、C 也充分评估容量满足报备要求。

组件B返回时延（单位：毫秒）	组件A单机器性能（单位：请求/秒）
100	42000
1000	24000
2000	18000



- A调用时延增大，性能下降
- A有效输出低于极限
- 游戏请求超时，重试

……

事故过程：

00:05 组件 A 运维收到告警电话，开始处理，发现从 00:02 开始游戏 M 登录请求成功率下降，并且有持续下降趋势，**此时登录服务开始产生大量告警，于是开始联系组件 A 研发紧急处理。**

00:08 组件 A 通过查看日志，发现大量组件 B 请求超时错误，于是紧急联系组件 B, 组件 B 回复服务 CPU 高，正在紧急扩容。此时组件 A 也发现，自身服务 CPU90%，通过查看请求量数据，发现在自身容量的 30%处出现异常，判断当前 CPU 90%应当是下游容量不足导致自身性能受影响（处理耗时增加+业务不断重试），于是并未做进一步的处理，

等待组件 B 恢复。

00:15 收到组件 B 同步，已经扩容完毕，但是组件 A 观测数据，发现登录请求成功率并未恢复，登录成功请求量恢复并逐步增长到自身极限性能，

自身 CPU 从 90%增长到 100%，此时请求量在持续上升。

00:18 组件 A 开始紧急扩容。

00:32 组件 A 扩容完毕，游戏 M 登录请求成功率逐步恢复。

右上角两个图分别展示了不同组件 B 调用延迟对组件 A 处理性能的影响。下面左右两张图展示了整个过程中，组件 B 和组件 A 请求量和有效处理能力的趋势，可以明显看到，

当组件 B 超过自身极限后，导致组件 A 调用延迟增大，组件 A 的有效输出断崖式下跌，远远达不到自身的处理极限。这也是组件 A 运维判断自身不需要扩容的决策依据。

案例分析

这个案例确实比较复杂，游戏的特殊性在于其受环境因素影响较大。本次故障并非单一事件，而是一个多米诺骨牌效应。最初是活动进行中，组件 B 出现问题，修复组件 B 后，却紧接着导致了组件 A 的问题。值得注意的是，组件 A 的问题也受到游戏活动自身特性的影响。整个故障处置过程因此变得非常复杂。

复杂性根源：游戏业务受外部环境因素影响大，故障往往不是独立出现而是层层传导。

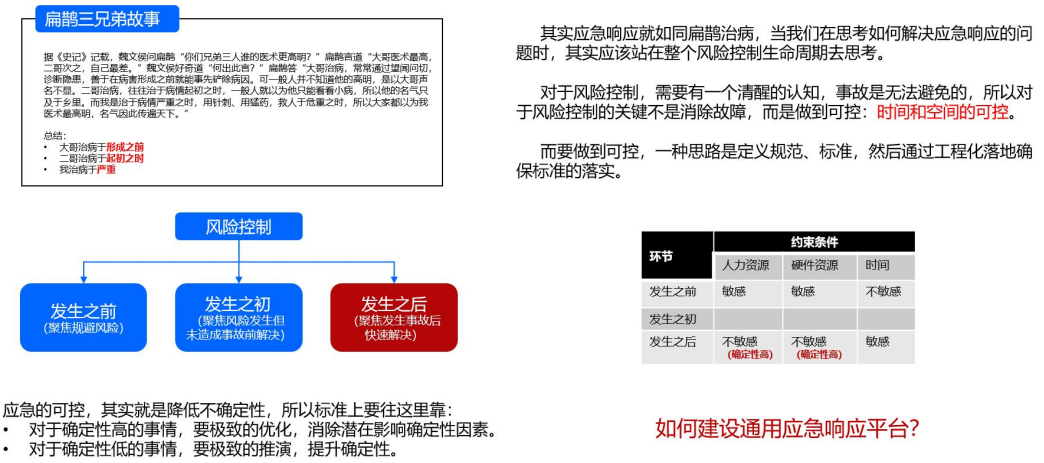
故障链条：活动触发 → 组件 B 故障 → 修复组件 B → 导致组件 A 故障。

活动影响：组件 A 的故障也与游戏活动本身的特性相关，进一步增加了复杂性。

在这种复杂场景下，我们发现单凭人工判断和处置能力，虽然能快速解决某些局部问题，但整个故障解决链条可能反而被拉长。这种不确定性的场景，我们认为更应该依赖工具化的手段来解决。

总体思路

总体思路



尽管本次分享的核心聚焦于"应急响应"，但我们必须强调一个根本理念：应急本身并非风险控制的全部，它更准确地说，是风险控制链条中的"最后一公里"。将整个风险控制体系视为一个流程，其核心环节理应涵盖故障或风险发生之前、发生之初以及最终的发生之后。我们的主要精力和资源，必须压倒性地投放在前两个阶段，而非仅仅局限于最后一个"发生之后"的应急处理。

通过真实数据能够最直观地印证这一理念的有效性：

- 在过去一个月中，某款游戏共发生 80 余起故障事件。
- 其中，有超过 70 起故障，得益于前置的故障自愈机制，得到了自动化解决。

这意味着，在风险预防和故障发生之初投入的资源与努力，能显著减少需要人工介入的情景。因为一旦进入应急响应阶段，本质上仍需依赖人员进行干预处理，而我们始终追求的目标是尽可能减少或消除对人工操作的依赖。前置的投入越多，自动化、自愈能力越强，最终需要团队进入"备战"状态去应对的故障就会越少，SRE 团队的核心价值也由此获得更深层次的体现。

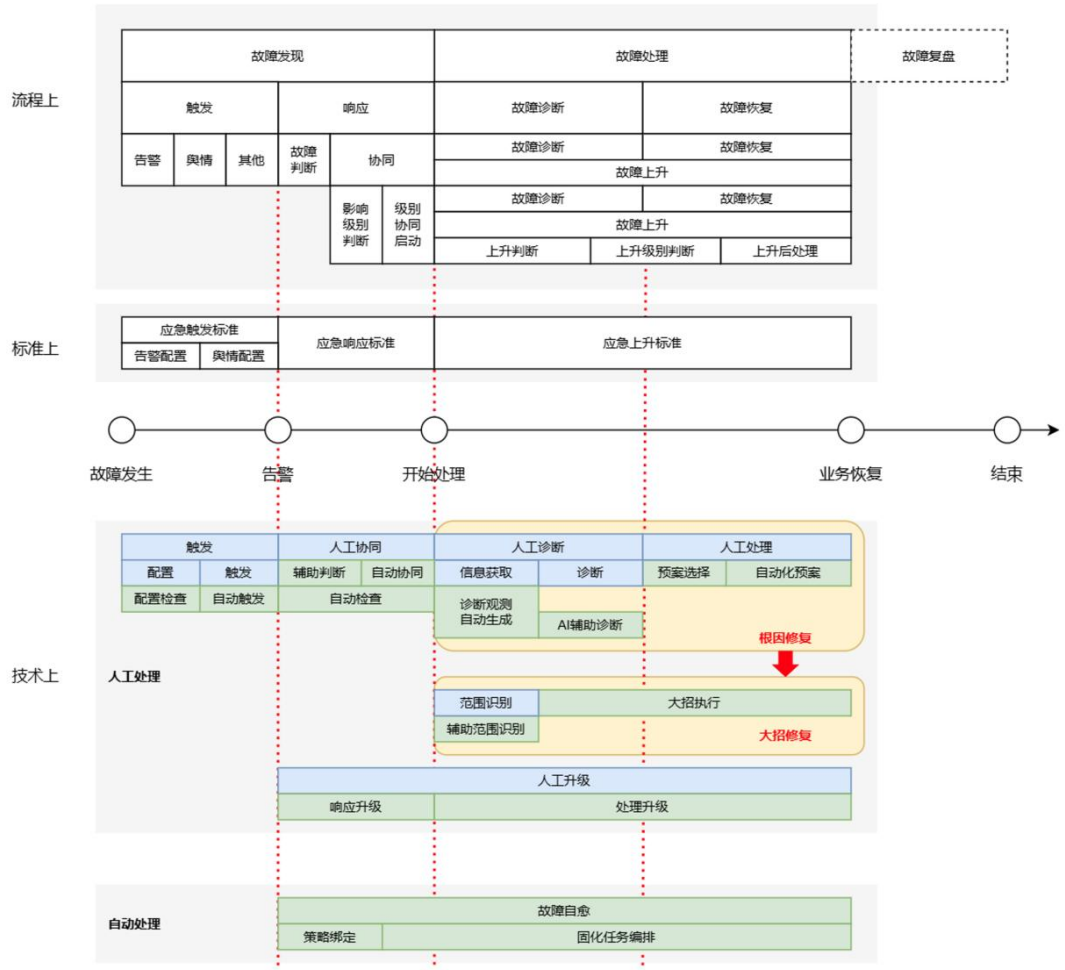
接下来，我们将深入探讨如何构建通用的应急体系，但这依然是对前述理念的延续和支撑，旨在即便在故障发生之后，也能通过结构化、体系化的建设，提高应急处理的效率和效果，最大化减少故障带来的影响。

（二）体系设计详情

基于蓝鲸底座的应急响应方案建设

基于蓝鲸底座的应急响应方案建设 方案思路

我们将从方案设计的视角切入，聚焦于如何利用统一的基座平台，构建一套通用的应急响应解决方案。



我们在构建这样一个通用的应急响应方案时，核心的理念在于追求极致的"确定性"保障。因为在应急场景下，快速有效地解决问题是首要目标。我们深信，在整个应急处置流程中，那些确定性高的环节应当避免任何人为失误的引入；而对于那些确定性较低的环节，则应充分使用工具和辅助手段，以提升整体的确定性水平。正是基于此思考，我们构建了一个涵盖"流程-标准-救障（即故障处置）"的完整链条。正如你从图上看到的那部分被标注为绿色的区域所示，这些都是我们经过仔细拆解后，识别出可以通过工具和技术手段有效解决的环节。

我们整个应急响应平台和体系的建设目标主要聚焦于三大方向，旨在解决我们前面提到的诸多痛点：

首先，我们立志于消除信息孤岛，实现信息的实时共享。这意味着所有相关的监控数据、告警信息、人员状态和处置进展都应该在第一时间触达需要的人员。

其次，我们将标准化的应急工作流程通过平台工程化落地，以此来避免多团队协同过程中可能出现的混乱和效率低下问题。

最后，我们致力于通过平台能力辅助信息的采集、决策的制定以及执行的落地，从而显著提升整个应急响应环节的执行效率。

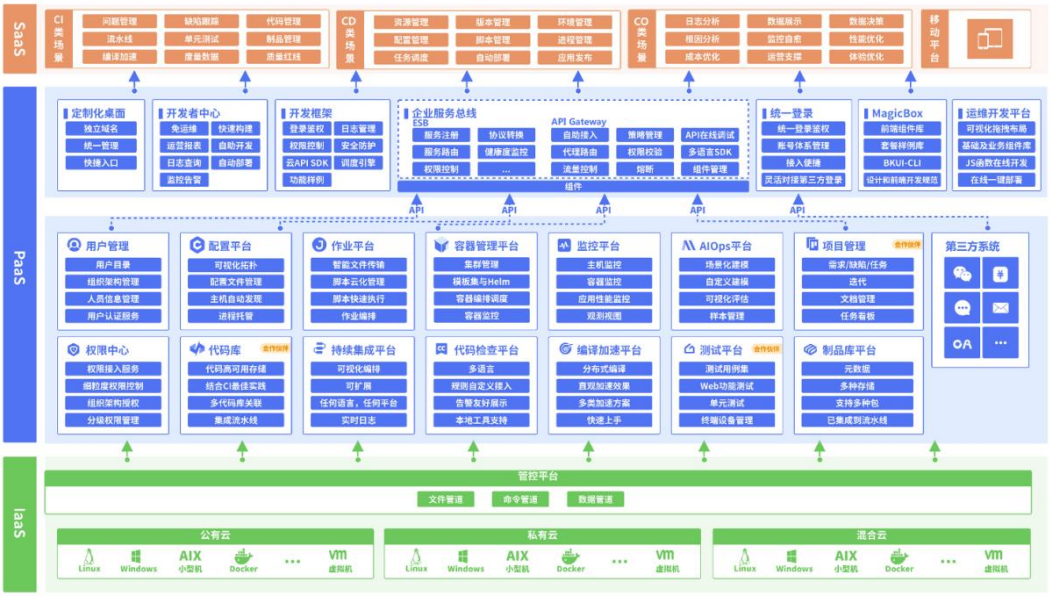
为实现这些目标，我们的平台需具备如下能力：

- 提供便捷的方式来获取监控信息、人员状态；
- 支持高效的观测手段；
- 并且能够执行策略驱动的任务。

其实蓝鲸平台天然满足这些需求。

通过蓝鲸基座便捷构建应急系统

通过蓝鲸基座便捷构建应急系统



在构建应急响应体系的基石层面，我们充分利用了公司内已有的成熟平台能力：**权限中心：**

提供了便捷、集中的全线权限管控能力，这是应急响应过程中进行权限调整和隔离的基础。

配置平台：

精确、完整地存储了包括人员信息和 拓扑结构在内的游戏业务关键配置信息，在故障定位和人员拉起方面至关重要。

监控平台：

能够提供从设备层到上层应用层全链路的、完整的监控信息，是故障发现和根因分析的眼睛。

正是基于这些已然具备的平台能力，我们得以更加顺畅、高效地设计并搭建了我们的应急响应体系。

方案架构

方案架构

这个应急系统架构展示了不同层次的组件和功能模块，它们协同工作以确保有效的应急管理。下面是对各个层次和功能模块的描述：**入口层**

-功能

：作为系统的消费入口，可以快速启动应急响应。

- **组成**

：

- **蓝鲸监控**

：通过监控告警启动应急响应。

- **BKChat**

：通过信息流启动应急响应。

- **接口和页面**

：通过系统页面启动应急响应。

能力层

-功能

：负责处理核心的应急管理和资源管理任务，包含多个重要模块，确保应急响应的能力。

- **组成**

：

- **预案管理**

：用于管理应急预案，包括演练计划、预案流程等内容。

- **灾备管理**

：管理设备的使用和调度，保障在应急事件中的设备需求。

- **应急演练**

：执行和规划应急响应的演练，以提升团队响应和处理的能力。

- **应急管理**

：在应急事件发生后，进行统一的管理和响应。

数据层

-功能

：数据存储和管理层，提供系统操作所需的数据支撑。

- **组成**

：

- **CMDB 数据**

：包括业务人员信息、业务拓扑信息、资产信息等数据，为应急管理提供必要的资源信息。

- **监控平台**

：提供实时监控和告警功能，用于识别和响应潜在的系统故障。

- **告警事件**

：实时处理告警信息和事件，并实施应急响应。

平台层

-功能

：为系统提供支持和管理服务，保证各个模块之间的协同工作。

- **组成**

：

- **监控平台**

：支持实时监控和数据流转。

- **AI 开发平台（AIDev）**

：集成 AI 相关功能， 包括 Agent 开发的配套能力

- **标准运维平台**

：为运维和操作提供标准化服务，确保应急操作的一致性。

- **BKVision**

：用于对业务数据进行大屏可视化

- **蓝鲸配置平台**

：负责系统配置和资源调度信息。

- **权限中心**

：管理系统的访问权限和安全控制。

在我们的应急系统架构中，底层依托的是蓝鲸平台所提供的各项核心能力，包括监控平台、标准运维平台、BKvison，信息流等。这些平台能力是整个架构能够正常运行的基础。

在数据层面，我们利用 CMDB 系统来集中管理和存储各种业务信息，包括业务人员信息、业务拓扑信息和资产信息等。CMDB 的数据层整合了各类业务相关数据，从而为应急响应提供了必要的信息支持。当然监控的数据更是关键。

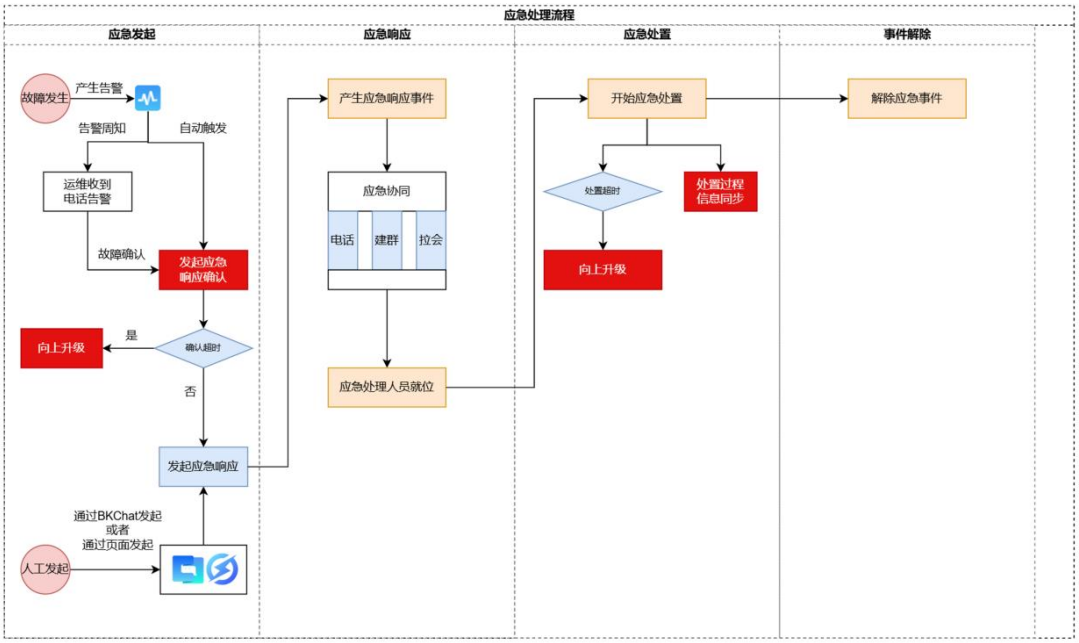
我们的主要研发工作量来自能力层。我们开发了包括预案管理、灾备管理、应急管理、应急演练等功能模块。

- 预案管理主要负责应急预案的创建、更新和管理，确保在紧急情况下可以迅速执行。
- 灾备管理是针对灾备预案的建设，在面对容灾类故障时可以快速一键恢复。
- 而应急管理则聚焦于对发生的各类应急事件进行追踪和处理。
- 应急演练聚焦通过既定的计划实施故障演习，提升人员对于故障处理的熟悉程度。

此外，这一系统是与我们的合作伙伴嘉为共同协作共建的。我们将腾讯游戏的经验与嘉为在行业应用方面的经验相结合，形成了此应急系统。虽然这一系统最初应用于腾讯游戏领域，但其设计理念和架构是通用的，可以适用于其他行业的应急管理需求。

应急管理流程概览

应急管理流程概览



应急管理流程的核心在于确保故障发生后，能够迅速响应并有效处理。具体来说，整个流程主要划分为发起、响应、处置与解除几个关键阶段，其中我们特别关注以下几个方面：

首先，确保故障发生后，运维能够**及时响应**。响应流程通过**监控触发告警**，发送确认通知启动，运维人员（如 SRE 或运维工程师）必须在规定时间内**及时处理告警**。如果未能响应，系统会自动将告警**向上升级**，通知部门经理。响应完成后，系统会自动启动**应急群聊**，促进**跨团队协同**，确保相关人员迅速加入并高效合作。

在故障的处置阶段，我们将**故障诊断与故障处置**进行了整合，因为在实际修复过程中，这两部分常常无法明确区分。整个过程中，系统自动收集并记录所有**关键信息**，并同步给所有相关人员，无需人工干预，直到故障被完全解决。

此外，为了确保**上级及时了解整个处理过程**，系统会实时同步故障处理进展。这种设计优化了**响应效率**、**跨团队协作**

以及**故障升级机制**

，使我们能够在故障发生后迅速启动应急响应，确保各项工作能够顺利进行并高效处理。

接下来，我将详细阐述如何通过这一流程提升系统的确定性，涵盖从告警的接入到故障复盘的整个过程。

告警配置

通过巡检确保告警正确接入

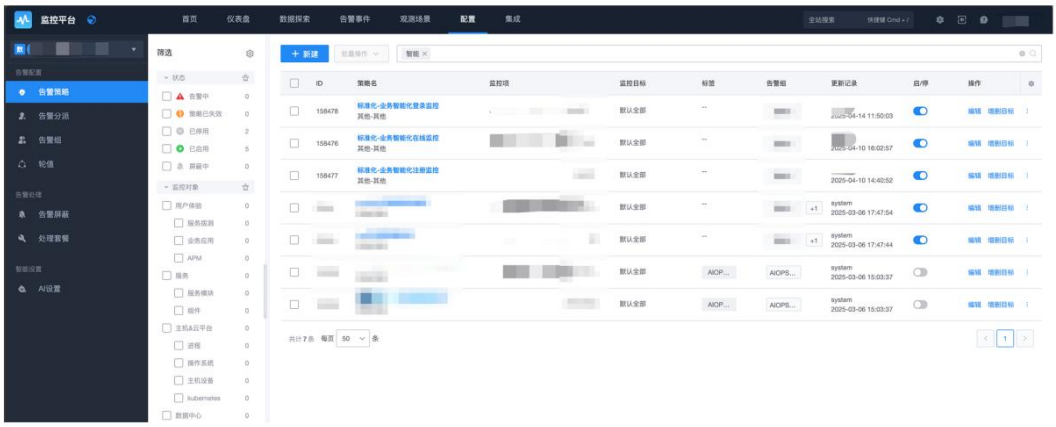
通过巡检确保告警正确接入

告警配置

面对告警信息爆炸的现实困境，团队务实地选择聚焦核心，通过建立一套标准化的告警接入机制，旨在有效识别并及时处理真正关键的告警，从而提升整体应急响应效率。

核心思路概括为：

- 聚焦重点，而非全面接入：鉴于告警数量庞大且良莠不齐，团队明确无法也不应将所有告警接入，而是优先筛选并接入那些能够反映核心业务问题的通用性告警指标。

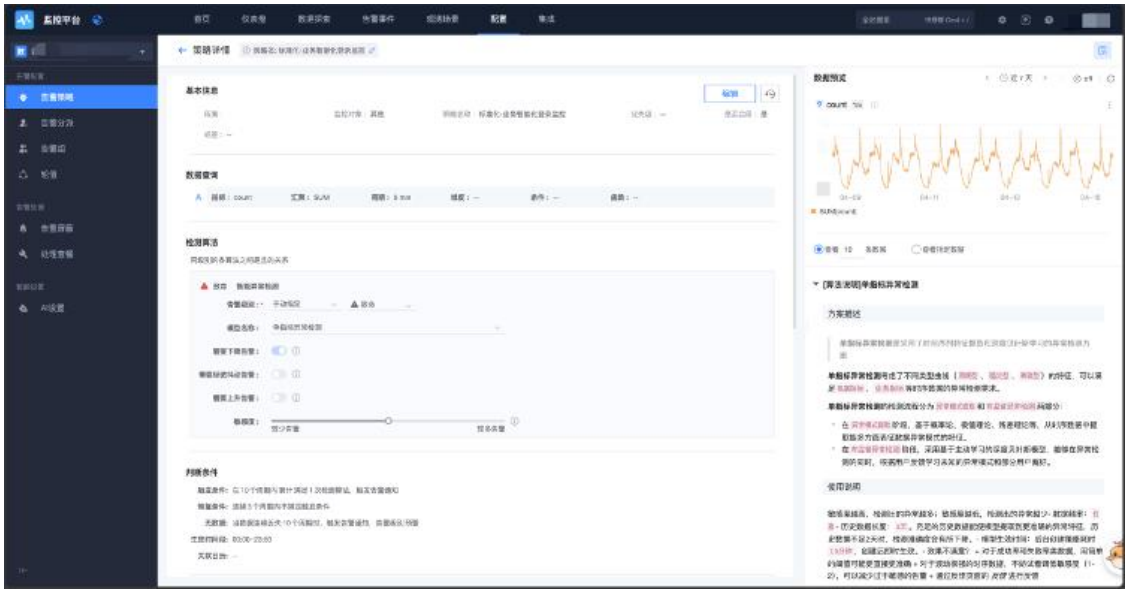


- 标准化告警接入规范：在蓝鲸监控平台上，制定了一套清晰的告警接入标准。业务团队遵循此标准进行告警配置，确保接入的告警具备有效性和代表性。（注意上图告警策略标准化字眼）
- 巡检确保告警正确接入：配套告警接入标准，构建了巡检系统。该系统定期扫描并检测业务团队的告警配置是否符合既定标准。
- 自动化跟进与升级机制：对于发现的不符合标准，例如未配置某项关键告警的情况，系统会自动生成工单提示业务团队限时整改修复。若业务团队未能在规定时间内完成修复，系统将自动将问题升级并通知到其上级管理者，形成强制性的闭环管理，确保关键告警的覆盖率与准确性。

通过这种规范化的配置与自动化的巡检及提醒机制，腾讯 IEG 团队有效地解决了告警接入阶段人为失误的潜在风险，确保了关键业务指标能够被及时有效地监控，并在发生异常时能够快速触发响应流程。这套流程不仅提升了告警的有效性，也通过自动化的方式减轻了人工检查和跟进的工作量，保障了应急响应体系的基石——准确且及时的告警信息源。

智能异常检测(无需配置阈值、无需配置同环比)

智能异常检测（无需配置阈值、无需配置同环比）



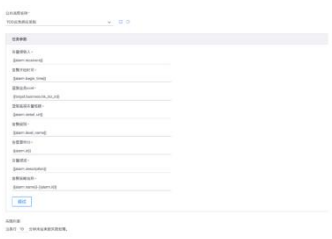
我们的监控平台支持单指标智能检测功能，这意味着一旦那些关键的核心指标接入监控系统后，系统将自动为这些指标配置智能检测策略。通过这一手段，我们成功地杜绝了因人工配置失误而引发的问题的可能。

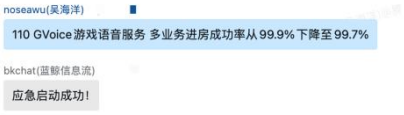
完成监控配置后，下一步便是将监控与应急响应紧密绑定，实现从发现问题到解决问题的快速联动。

应急机制

应急响应触发:监控平台自愈流程接入

应急机制 应急响应触发：监控平台自愈流程接入





- 针对关键指标，蓝鲸监控自动绑定故障应急
- 页面录入、BKChat启动应急后，无需确认
- 监控启动的应急，需要确认

应急响应的触发可以由人工报告，但更主要的手段是通过监控系统进行自动触发并介入处置。在腾讯 IEG 的实践中，我们广泛使用了蓝鲸监控的处理套餐作为触发机制。

具体来说，该套餐提供了标准化的公共配置，并且设计为无需人工干预即可自动生效。只要系统中的关键指标被接入并，系统会自动绑定到这个套餐，当这些指标出现异常时，系统将自动完成以下步骤：

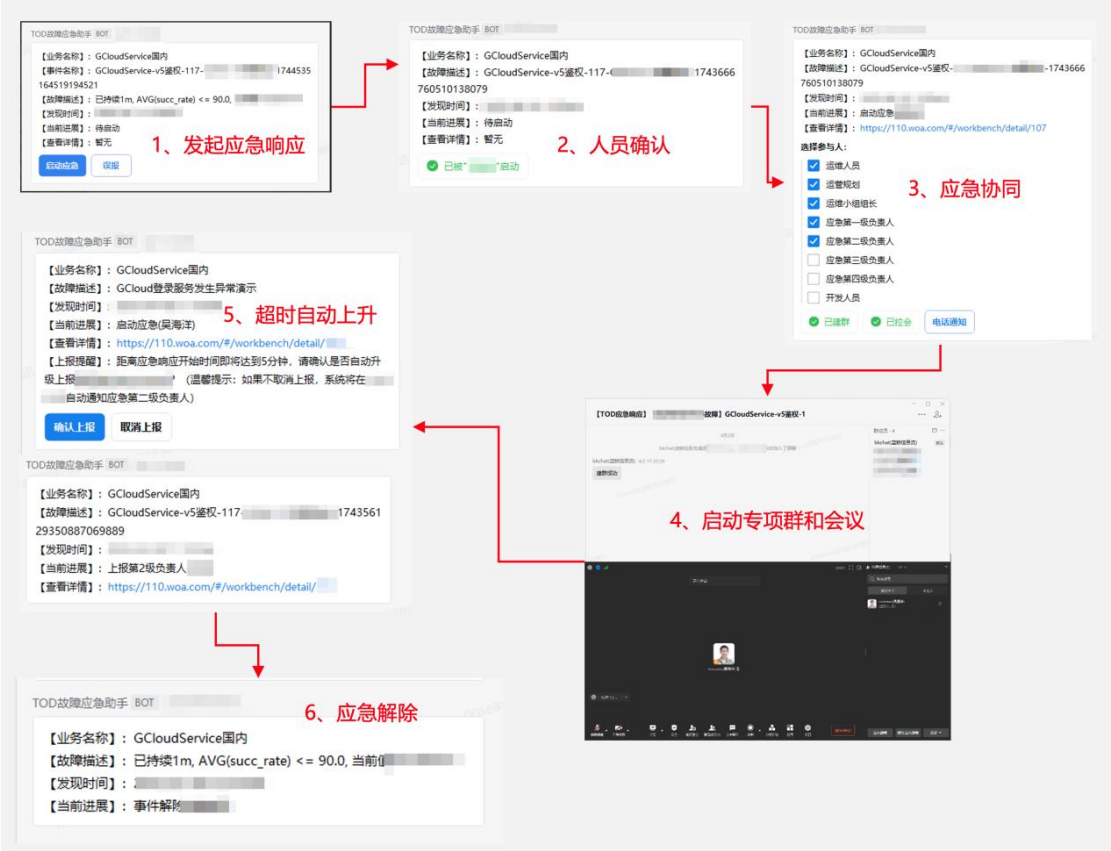
告警生成： 系统会自动根据预设规则生成告警。

套餐调用： 告警触发后会同时调用，例如自动执行自愈脚本、触发降级或限流策略等。

应急触发： 如果自愈失败，触发应急响应流程。

应急系统使用概览

应急系统使用概览



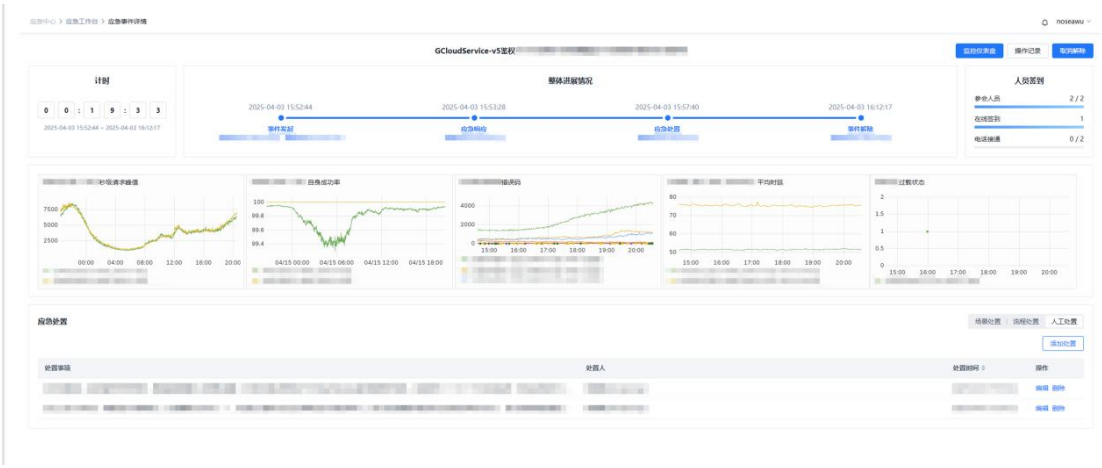
针对应急响应流程，我们首先从监控发起告警开始。

告警触发后，会启动一个确认环节，因为需要人工判断告警的真实性，排查误告的可能性。

确认告警有效后，应急响应人员随即开始协同，在这个阶段，系统支持根据组织结构配置和选择相关的协同人员。

应急响应人员确定后，1 分钟内启动紧急响应并触发相关人员拉起专用会议，当前主要以群聊和腾讯会议为载体，其中会议被认为是协同效率最高的方式。

在整个处置过程中，系统将全程收集处理信息，直至问题得以解决。此外，如果处置过程发生超时，系统会自动进行升级处理。



当前界面是整个应急响应流程的可视化载体，其设计重点在于突出关键信息，确保参与处置的各方能够快速、准确地掌握事态进展：

- 时间信息：界面清晰地标识应急事件的起始时间，这是判断事件持续时长和评估影响范围的基础。
- 处置人员信息：明确展示当前事件的处置负责人或团队，便于快速沟通和任务分配。
- 核心指标数据：实时呈现与事件相关的关键业务或系统指标，例如错误率、延迟、流量等，这些数据是判断异常状态、评估影响程度及验证恢复效果的重要依据。

这张幻灯片聚焦于应急协同的前置环节，即如何在事件发生初期通过界面聚合信息、提升协作效率。而后续的应急处置环节，其主要的耗时瓶颈往往集中在根因定位这个阶段。

上升机制:响应上升和处置上升

上升机制：响应上升和处置上升 响应上升：

•响应上升指运维人员未及时对应急事件进行响应

•如果运维人员电话未接通，会逐级上升，直到其中一个层级电话打通

•如果运维人员电话接通，但是未能对应急事件进行确认，会在设定时间超时后进行上升**处置上升：**

•处置上升指运维未在规定时间内对完成故障修复而进行上升

•按照启动应急开始计算，每到一个层级的处置超时时间，就会对该层级人员进行电话周知

这里，我们着重讲一下应急事件的"上升"机制。简单来说，如果一个事件被系统自动检测并触发告警后，在指定的时间窗口内未得到及时处理或确认，系统将自动启动逐级上升流程。上图看到，这个时间阈值是可配置的。例如：

- 如果在 3 分钟 内没有对告警进行确认响应，事件将自动上升到第一级处理人。
- 如果在接下来的 5 分钟 内仍未得到处理，事件将进一步上升到第二级，通常是你的直属主管（Leader）。

这种处置上升机制，核心在于确保任何异常事件都能在限定时间内被高效率地处理。如果在规定的处理时限内未能解决问题，则会自动向上级传递，直至足以能够协同足够资源或具备决策权限的上级介入处置，从而避免事件被遗漏或延误。

故障发现和故障诊断 蓝鲸监控：辅助诊断（通过故障自愈套餐触发基础环境状态检查）

故障发现和故障诊断

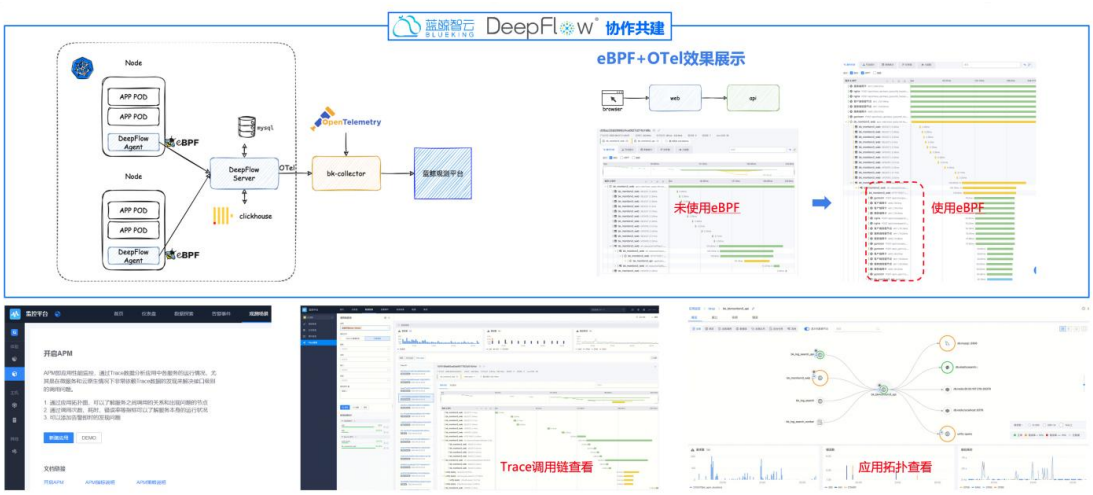
故障自愈除了可以修复明确处理流程的故障之外，也可以用于简单的信息提取来辅助故障诊断。

蓝鲸监控:辅助诊断(通过故障自愈套餐触发基础环境状态检查)

比如上述案例展示了通过故障自愈实现了对基础环境的健康检查，可以协助运维快速诊断是否故障为硬件、网络等问题导致。

简单的信息提取只能用于一些简单场景的故障诊断辅助，接下来会介绍更深层次的辅助诊断方案。

蓝鲸监控：辅助诊断（APM+eBPF）



蓝鲸监控:辅助诊断(APM+eBPF)

以上我们是与 DeepFlow 开源社区协作共建的一个解决方案,整体方案的特性如下;

特性	说明
真全栈覆盖	从应用层 → 系统调用层 → 网络层，做到全链路观测、无盲区
双视角融合	融合 开发者视角（OTel/APM） 与 SRE 视角（eBPF） 数据
零侵入部署	利用 eBPF 技术无需修改业务代码或加载 SDK，大幅降低接入成本
高扩展性支持	支持自定义协议解析（如 TRPC）和业务定制字段（通过 WASM 插件）
全链路数据关联	实现 trace + metric + log + profiling 全数据关联与跳转
云原生一键接入	借助蓝鲸平台，支持容器/K8S/物理机的自动化部署与插件管理

这个 1+1 >2 的方案的协同优势如下：

维度	APM / OTel	eBPF
观测视角	应用逻辑调用链，开发者视角	系统调用、网络流，SRE 视角
接入方式	SDK 插码，易受限（如发行游戏/第三方服务难以改动）	零插码，运行时通过内核抓取，无需业务配合
数据完整性	无法覆盖未接入 SDK 的组件（如 NGINX、代理网关等）	可捕捉所有系统级和网络级调用，包括"盲区"组件
数据粒度与覆盖	高层抽象，但有时缺少系统底层视角	细粒度、涵盖系统层级、底层性能、网络交互
融合方式	使用 TraceID、TCP 序列号、SyscallTraceID 实现两类数据自动关联	能补齐 OTel 缺失组件链路，增强拓扑完整性

基于此解决方案，若业务场景并非极为复杂，基本上已经能够很好地进行根因的定位。

蓝鲸 AIDev：智能辅助分析和根因诊断

蓝鲸 AIDev:智能辅助分析和根因诊断

LLM 模型服务层

包含"LLM 网关服务"和"LLM 模型集成"两部分。

"LLM 网关服务"负责对外暴露统一的 API 接口，接收上层请求并转发到具体模型。

"LLM 模型集成"负责将不同的大语言模型（比如 OpenAI、Anthropic、内部自研模型等）接入到平台中，并统一管理模型调用、鉴权、限流等细节。

LLM 工具构建层 核心由两个 Agent 组件构成：

LLM Workflow Agent：负责定义和执行多步骤的业务流程（workflow），例如串联一系列模型调用、工具调用和决策逻辑。

LLM Agent Framework：提供通用的 agent 运行时框架，支持插件化地加载各种功能模块。

底部支撑能力包括：

角色管理：定义对话或任务中不同"角色"（system、assistant、user 等）的权限和行为规范。

提示词管理：集中存储、版本化和调优各类 prompt 模板。

工具管理：管理可被 agent 调用的外部工具或服务（如检索、计算、数据库查询等）。

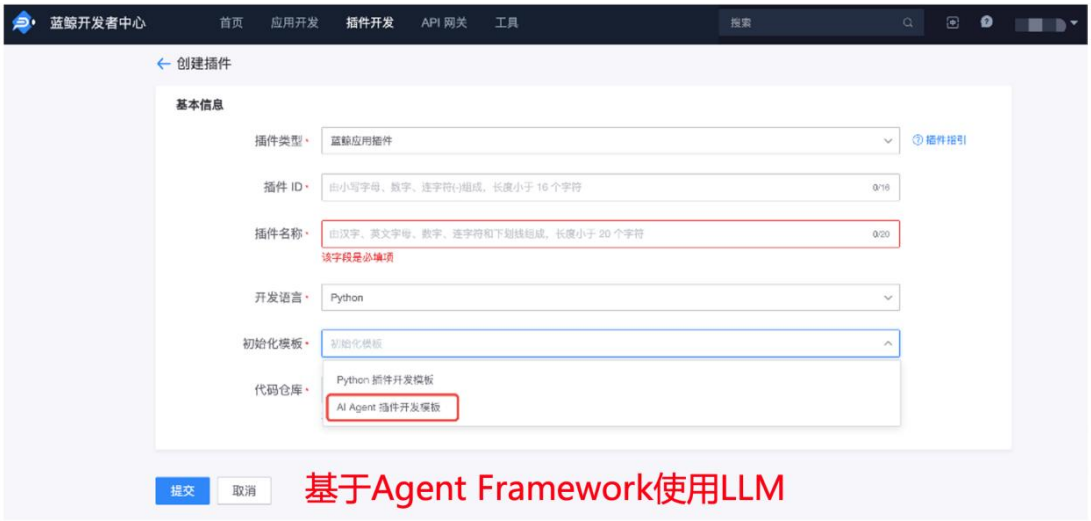
知识库：持久化存储结构化或半结构化的领域知识，供 agent 在执行时检索与引用。

LLM 场景服务层

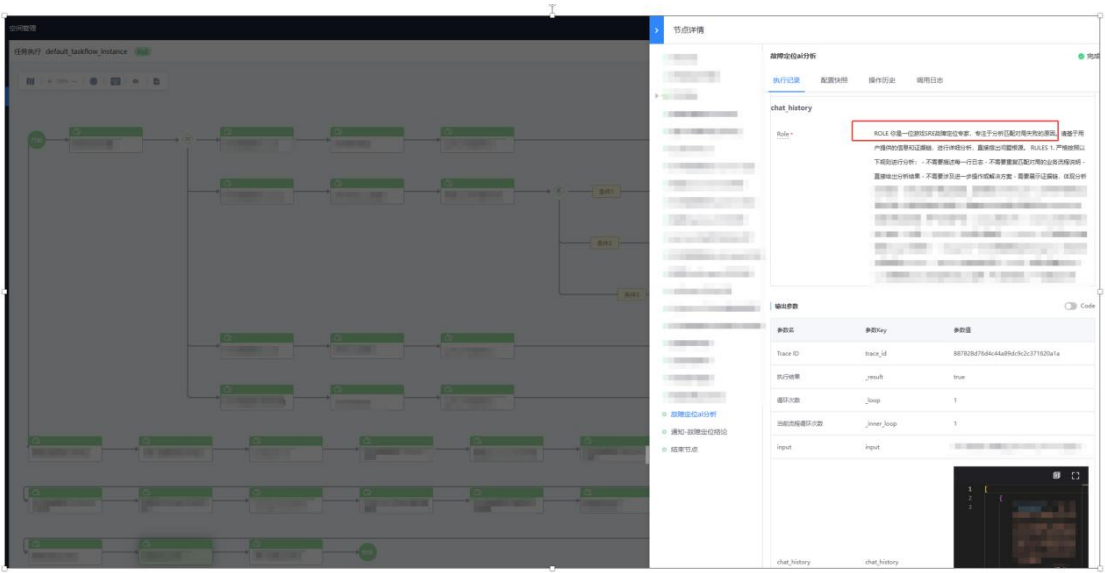
CI 智能场景（Continuous Integration）：在持续集成流程中提供代码审查、自动化测试建议等。

CD 智能场景（Continuous Deployment）：在持续交付/部署环节提供发布策略优化、回滚决策支持等。

CO 智能场景（Continuous Operations）：在持续运维环节提供故障诊断、应急响应辅助等。

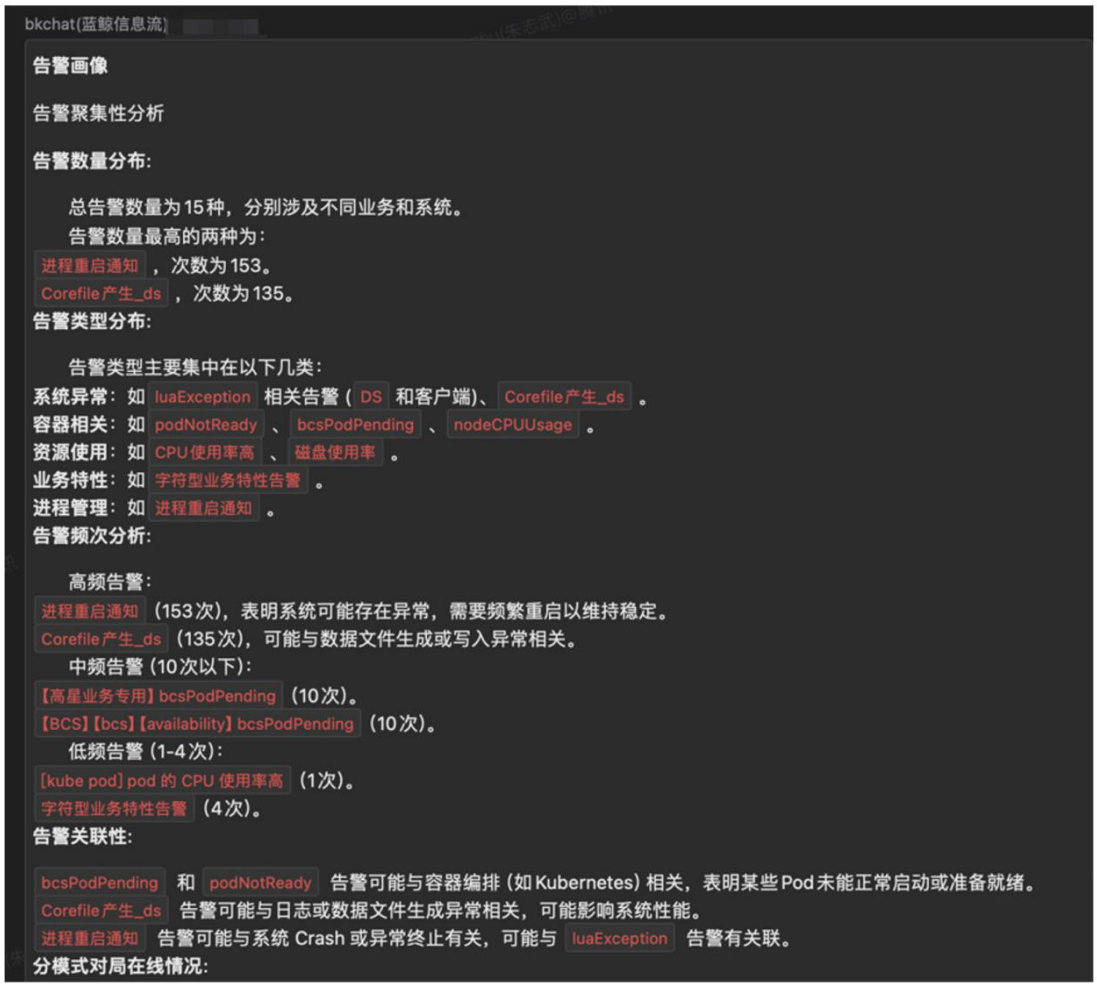


然后我们就分享一个 LLM Workflow 在 SRE 应急响应中的应用。Workflow 这种模式之所以被我们采用，很大程度上是因为它非常适合那些场景相对固定、变化不大的定位场景。



上图可以看到我们通过我们的 AIDev 平台编排了一个固定的 LLM workflow , 其中 一个 LLM 节点， 我们使用了一个 "你是一个游戏 SRE 故障定位专家，专注于分析匹配对局失败的原因。请基于用户提供的信息和证据链进行详细分析，直接指出问题根源....." 去进行辅助故障定位

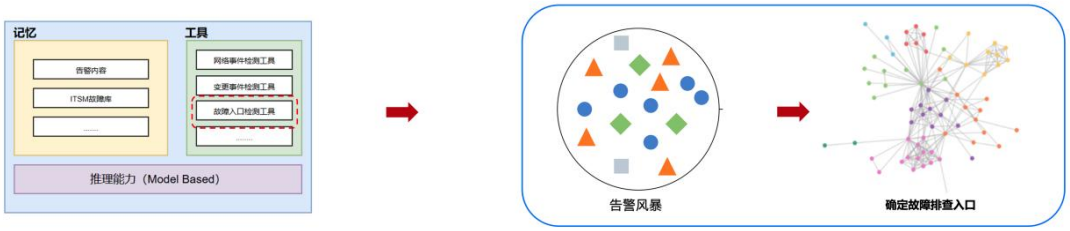
以下是定位的结果通过 bkchat 蓝鲸信息流反馈给用户。。



具体到生产环境，由于其节点众多且复杂，一个设计良好的 Workflow 能够帮助我们快速地进行一次完整、全面的事故分析。大家可以看到，它最终能够输出一个涵盖整体细节的分析结果，极大地提升了运维同学快速定位问题的效率。

基于 AgentFramework 实现根因诊断

基于 AgentFramework 实现根因诊断

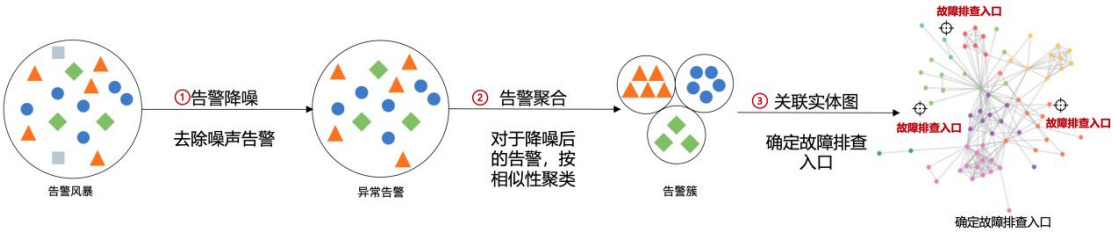


基于"Agent 化"应急方案，整个流程由系统智能触发，无需针对每个场景手动编排 Workflow。首先，Agent 会利用蓝鲸基座提供的入口检测工具对告警进行降噪、聚合并构建实体关系图，从中迅速识别出故障的"入口"服务或组件。确定入口后，系统会在整体网

络拓扑中截取与该入口关联的子图，再通过内置的根因定位算法对这段子图深度分析，直接输出故障根因节点。

在这个案例中，业务故障的上层表现最终追溯到 Redis 异常。Agent 方案首先在告警中识别出影响最广的入口节点，随着分析子图的不断收缩，系统自动锁定了 Redis 实例为故障根因。

故障发现：基于 Transformer 和对比学习的故障入口确定方案



核心任务：在大量线上告警事件中确定故障诊断的排查入口

依赖数据：告警数据、指标数据、图谱数据

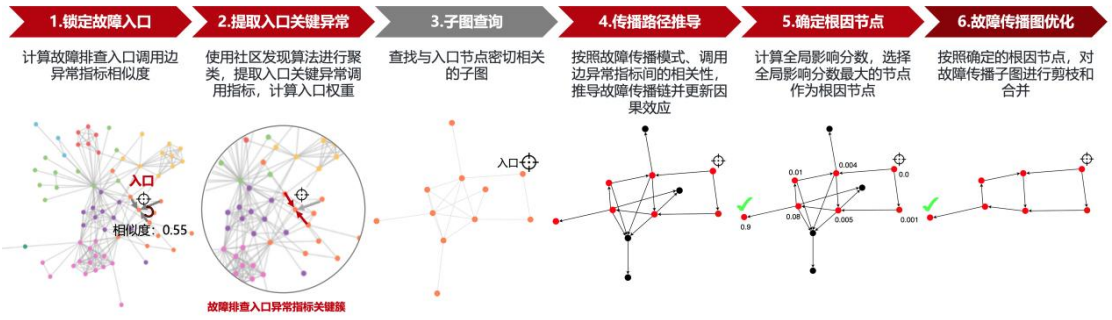
我们构建的应急体系中，关键一环在于提供智能化工具，快速定位问题根源。幻灯片展示的正是其中的一个核心组件，它专注于帮助运维人员高效识别故障的入口。

① 第一步是告警降噪，即去除噪声告警。告警降噪采用 Transformer 模型作为告警语义的理解模型，同时设计告警嵌入方案（指标嵌入+维度嵌入），将告警数据转化为告警嵌入。在训练阶段，以历史告警构建的告警三元组（<基准告警，相似告警，无关告警>）作为样本，通过对比学习的训练方式，让模型可以很好的区分相似告警和不同告警之间的差异。在预测阶段，训练好的降噪模型可以得到每个告警对应的嵌入，将告警嵌入和噪声嵌入计算相似度，相似度大于阈值就预测为噪声告警，就可以进行降噪去除。

② 第二步是告警聚合。首先会把降噪后的告警数量时序进行异常检测，拿到异常的时间点对应的告警。然后这些异常告警根据从第一步告警降噪模型输出的告警嵌入之间的相似度进行聚合。

③ 第三步是关联图谱。将聚合后的告警维度和实体关系图谱（CMDB+BCS）进行关联，从而确定告警实体对应的故障排查入口。

故障诊断：基于社区发现和图的动态规划的故障诊断方案



针对复杂业务系统，其网络拓扑结构往往是极度复杂的。我们依赖图谱数据（CMDB+BCS）及调用关系数据（APM / eBPF 调用指标），推导故障传播图，确定根因节点。流程如上图所示。

① 第一步是锁定「故障发现」确定的故障排查入口。通过计算故障排查入口调用边上的异常指标相似度，确定异常指标关键簇。

② 第二步是提取故障排查入口关键异常。通过使用社区发现算法对异常指标进行聚类，提取入口关键异常调用指标，计算入口权重。

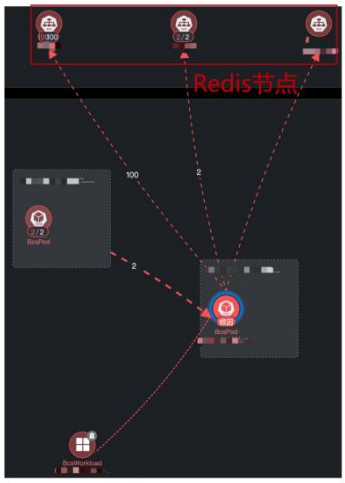
③ 第三步是子图查询。从故障入口出发，找到和入口节点密切相关的实体关系子图。

④ 第四步是传播路径推导。沿着调用边的方向去进行游走推理，如果满足预设的故障传播模式，就会将这些点标记为红色，代表异常。同时计算调用边和被调用边之间异常指标的相似度，来更新故障传播图调用边之间的因果效应。

⑤ 第五步是确定根因节点。经过上面的游走推理，整个故障传播子图已经拿到了，计算故障传播子图所有节点的全局故障影响分数，选择全局故障影响分数最大的作为故障传播子图的根因节点。

⑥ 第六步是故障传播图的优化，即故障传播子图的剪枝和合并。剪枝是指去掉故障子图里面没有异常或者不符合异常传播模式的边和节点；同时每一个入口都能拿到对应的一个故障传播子图，取全局异常分数最大的根因节点作为最终的根因节点，同时合并根因节点和最终根因节点一致的多个故障传播子图。

故障诊断案例：Redis 延迟引发的故障：



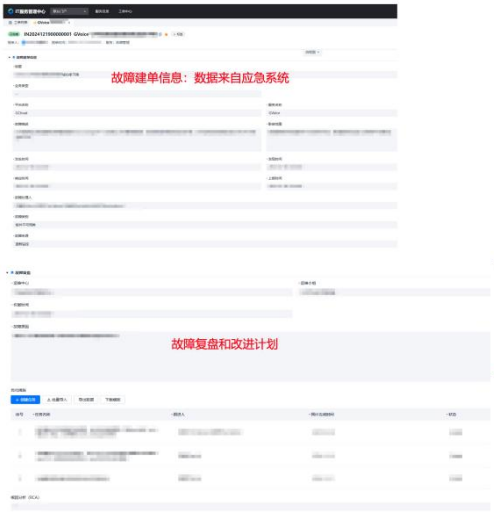
故障传播图

调用边异常指标（业务Pod请求Redis节点）

上图通过一个生产环境中的实际案例，来具体剖析前面提到的应急响应机制是如何发挥作用的。

这个案例是由 Redis 延迟引发的业务级故障，左图是故障传播图，右图点击业务 Pod 到 Redis 节点的调用边，可以看到对应的 Redis 延迟指标。正是通过前述的故障排查和分析方法，才得以快速定位到其深层原因：Redis 出现了异常。

故障复盘：通过 ITSM 系统实现复盘和改进



故障复盘

接下来我将分享我们在故障复盘方面的实践，这页幻灯片展示了我们在 ITSM 系统在复盘流程中扮演的关键角色。

流程化优势：

ITSM 系统最显著的优势在于，它能够方便地帮助我们固化并制定流程、能够让各业务团队在故障处理和复盘过程中遵循统一的标准。这确保了不同团队在应对故障时，拥有统一且规范的行为模式。

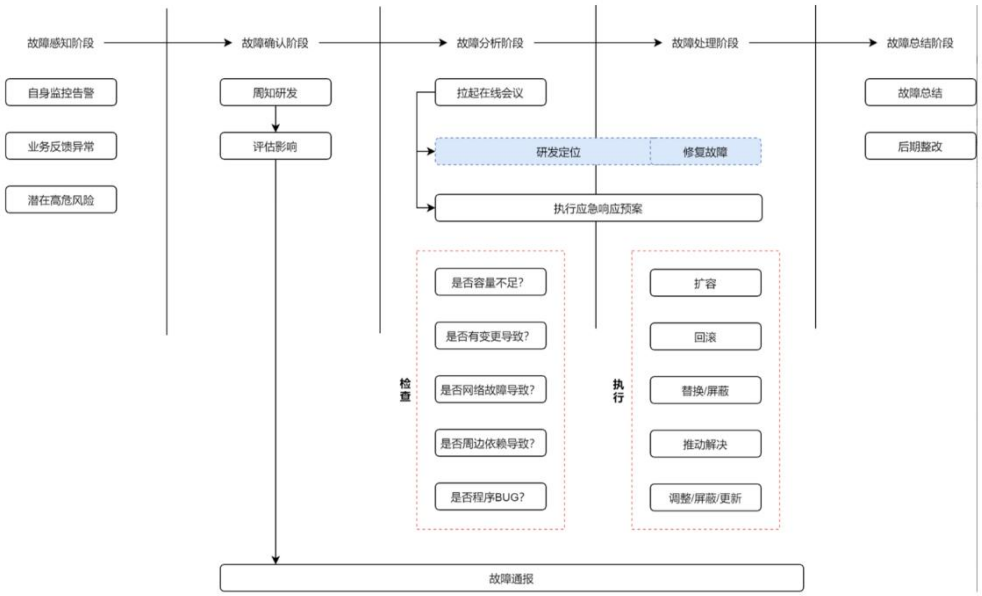
系统联动与人员协同：

ITSM 强大的联动能力使其能够与其他周边系统无缝对接，实现信息互通和流程自动化。同时，它也能有效连接所有相关人员，无论是协调跨团队响应，还是推动后续改进措施，都可在同一平台上完成。

通过将故障复盘与跟进集中到应急系统中，我们能够更便捷、高效地完成整个复盘过程，并确保后续改进措施得到切实落实。

执行应急响应的思考和经验

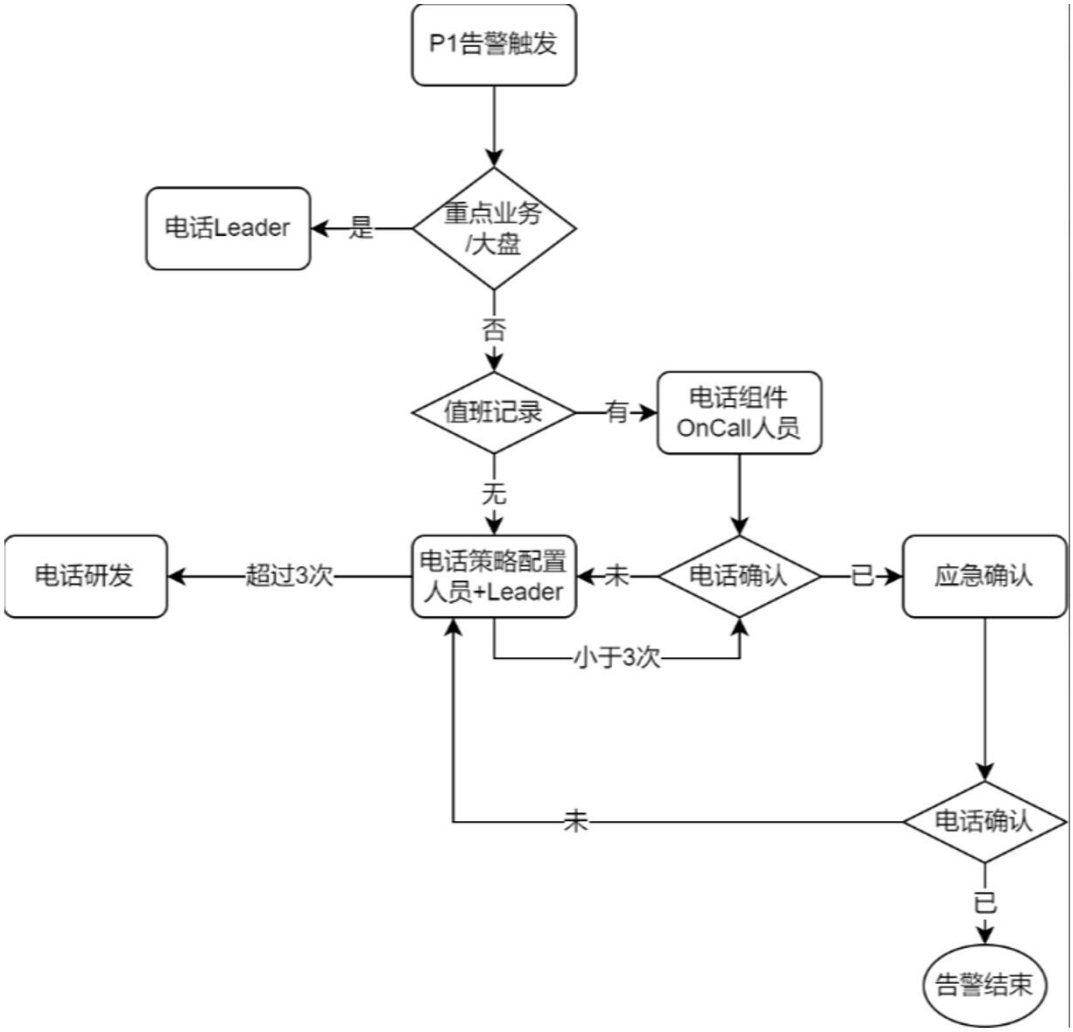
关于应急响应，我们的经验是是，必须高度重视流程建设，以下是我们一些流程规范的示例



分类	术语	定义
告警级别	P1	•该级别告警触发时，需要立即进行处理。 •通知方式：电话+企业微信
告警级别	P2	•该级别告警触发时，无需立即进行处理。 •通知方式：企业微信
响应级别优先级：一级>二级>三级。	一级	定义：关键路径异常，且持续时间大于 10 分钟，需跨团队关注与协作处理 • 该级别要求立即周知至总监以及部门重大故障群。 •该级别要求研发、运维立即共同响应。
响应级别优先级：	二级	定义：关键路径异常，且持续时间小于 10 分钟，需跨团队关注与协作处理该

分类	术语	定义
一级>二级>三级。		级别要求研发、运维共同响应。
响应级别优先级：一级>二级>三级。	三级	定义：非关键路径服务异常，且持续小于 10 分钟，无需跨团队协作处理，该级别要求运维响应。
业务级别	重点业务	包含收入活跃>xxx 的游戏，以及自定义添加的游戏。

角色	责任	候选人员	说明
事故总控	负责组建事故处理团队、分配任务给团队成员，协调周边和资源以便更快速的处理故障。	运维负责人研发负责人	运维负责人负责运维侧协调安排研发负责人负责研发侧协调安排
事故处理	处理故障	直属运维直属研发	运维作为现网唯一操作人员
发言人	本次事故对外发言人，职责包括：•向关注事故的人周期性的发送事故进展 •维护当前事故文档，保障其及时性和正确性	运维负责人服务经理	

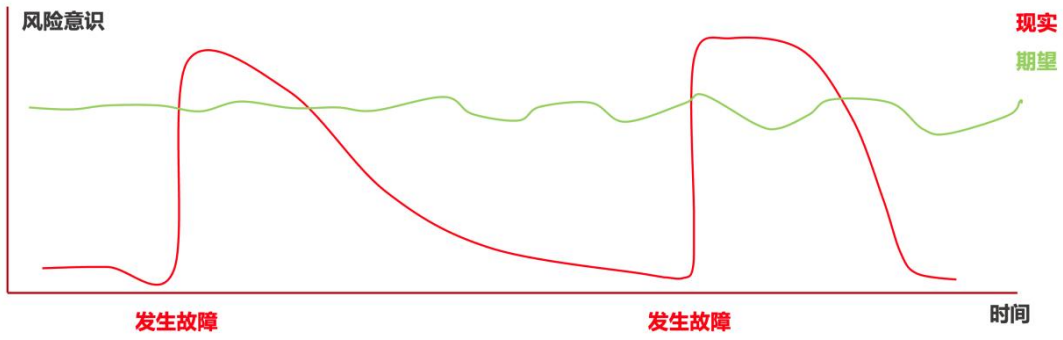


正是因为在整个组织层面，应急流程可能相对宽泛，对于单个业务，尤其是核心业务，流程的建设至关重要。前两位讲者也都特别强调了流程建设的重要性，而流程建设的核心目标在于：

- 确立全链路确定性：通过预先规定的步骤和规范，确保在应急场景下各环节的确定性执行。
- 明确协作边界：精确划分各团队和角色的职责边界，避免出现应急响应时责任不清、忙乱无序的情况。

常态化应急

常态化应急



常态化应急：以小博大，培养高风险意识与实战能力

在经历多场实战后，我们深刻认识到，常态化的应急至关重要。我们曾满心期望在应急发生时能如预设般高效运转，遵循那条理想的绿色曲线。然而现实往往是，在故障发生之前，团队可能处于一种松弛的状态，一旦真正的故障来临，却可能手忙脚乱，走上那条充满混乱与失序的红色曲线。

因此，不能只在发生大规模、轰动性故障时才启动应急体系。即使是体量较小、影响范围有限的故障，也应当按照既定的应急流程进行处置。

推行常态化、覆盖大小故障的应急机制，其核心目的在于：

固化流程，强化操作熟练度：通过反复实践，使应急响应流程深入人心，团队成员能够在压力下保持清晰的头脑和标准化的操作。

培养高风险意识：

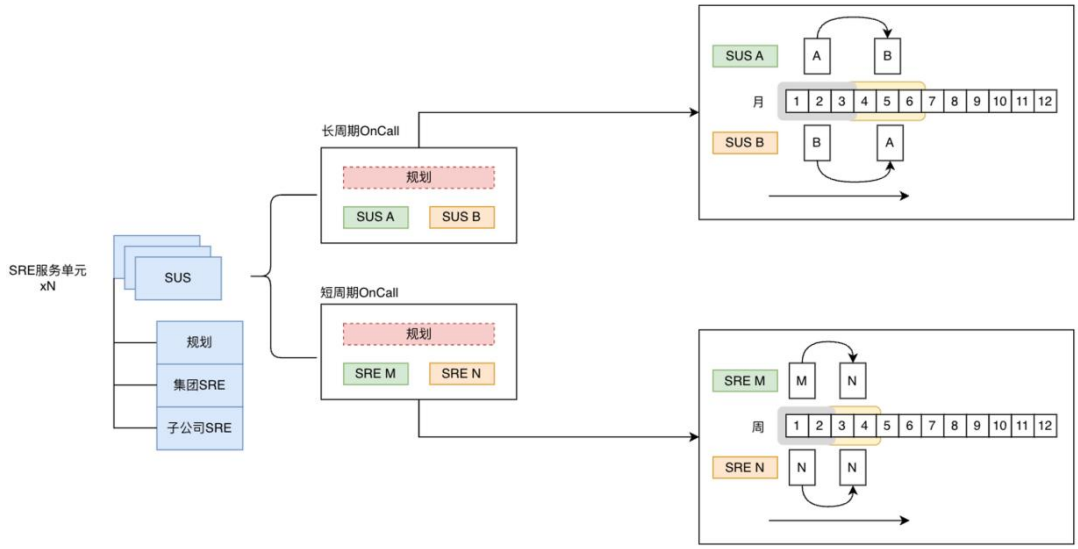
持续的应急训练能够不断提醒团队风险的存在，促使每个人都对潜在的问题保持警惕，从而在日常工作中更加注重稳定性和安全性。

提升团队协同效率：

在模拟或实际的小规模应急中磨合团队间的协作方式和沟通机制，确保在大故障发生时能够迅速进入"作战状态"。

通过将应急处置融入日常工作，即使是微小的故障也能成为学习和提升的契机，从而筑牢应对更大挑战的根基。

组织建设：OnCall

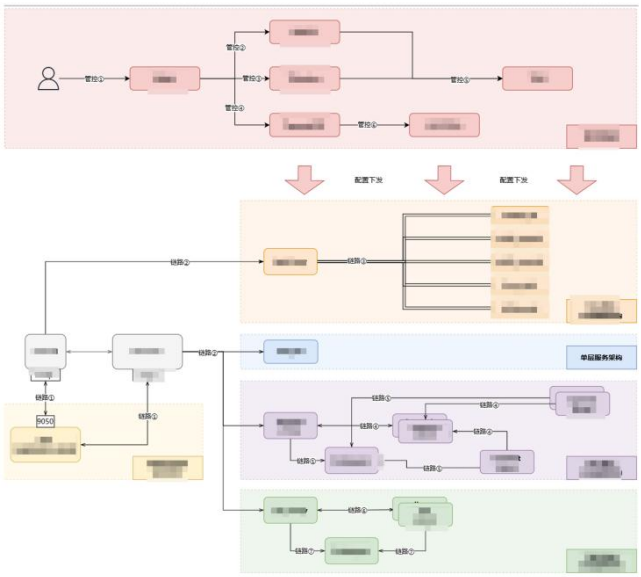


我们做"on-call"制度，其根本目的在于消除人员层面可能存在的单点依赖。毕竟，在实际工作中，我们确实遇到过这样的情况：当紧急事件发生 SRE 同学处理时，关键的负责人却因为各种原因（比如恰好不在服务区）而暂时失联，这并非个人责任问题。

而"on-call"机制的优势恰恰在于，一旦我们全面推行了这套体系，当某个服务的原负责人无法响应时，系统中会指定另一位熟悉此服务的 SRE 同学迅速介入，接手并解决问题，确保故障处理的及时性和有效性。

断路分析

断路分析

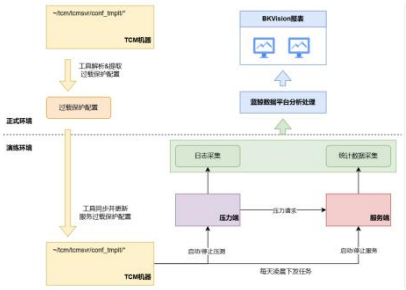


- 列出所有管控、服务链路
- 周期实施
- 拆分成原子链路
  - 说明链路功能
  - 是否具备容灾能力
  - 是否具备自愈能力
  - 链路故障影响
  - 链路故障恢复预案
  - 恢复预案自动化程度

断路分析，我们认为，是实施稳定性保障手段中尤为有效的一种方法。它要求在管控系统以及整个服务链路的每一个关键节点上，都详尽地标记出可能存在的问题点，并明确如何进行修复，以及是否具备相应的自愈能力。

确保过载保护配置的有效

确保过载保护配置的有效



针对频繁面临流量突发现象的业务系统，过载保护配置的有效性至关重要。配置变更、程序修改或基础设施环境调整都可能导致该保护失效，一旦突发流量超出系统容量且自动扩容（如 Kubernetes 弹性伸缩）无法在 2–3 分钟内完成，系统就存在崩溃风险。

解决方案

我们引入混沌工程，构建每日自动化验证流程：

- 自动触发各类过载场景
- 验证过载保护配置在真实压力下的弹性与鲁棒性
- 监控验证结果并生成报告，确保及时发现配置失效**价值与成果**

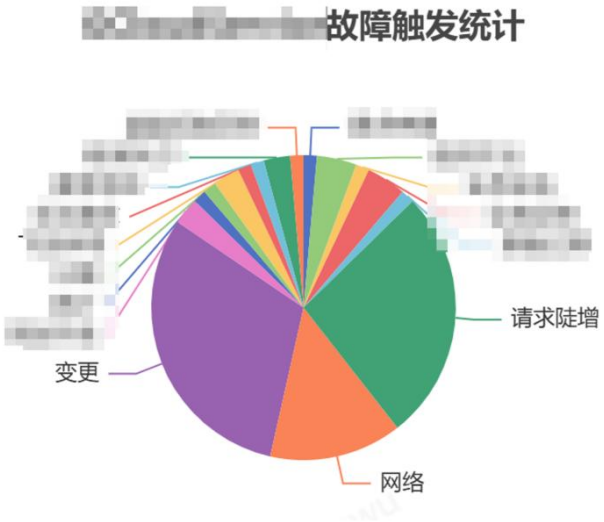
此验证机制可在极端流量突发时保证系统保持最低可用状态，避免因过载导致雪崩。它为构建高韧性系统提供了关键保障。

针对频繁面临流量突发现象的业务系统，确保过载保护配置的有效性至关重要。我们在实践中借助混沌工程，通过每日自动化的验证流程，持续检验系统过载配置的弹性与鲁棒性。

变更风险

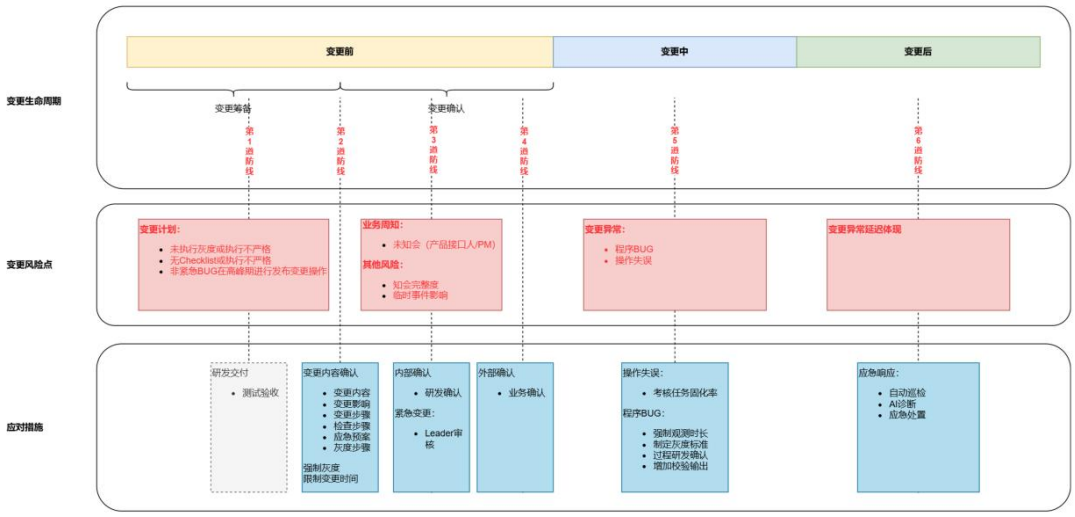
变更风险

某组件故障触发分析



某组件故障触发分析

在我们探讨如何构建高效的应急响应体系时，变更无疑是一个需要重点关注的关键点。正如之前的专家们所强调的，很大一部分生产环境故障都与变更操作紧密相关。根据我们某个组件的数据统计，大约有32%的故障直接或间接源于变更。



面对这一挑战，我们在变更管理流程中构建了多重防护防线，旨在最大限度地降低变更引入故障的风险。

上图展示了“变更防控”流程在生命周期各阶段的风险防护与应对措施部署。

整体流程分为“变更前”、“变更中”、“变更后”三个阶段，并在六个关键节点设置六道防线，以拦截不同环节的潜在风险。

在“变更前”阶段，第 1 、2 道防线通过变更策划和预实施核查拦截风险：包括未执行或不严格执行灰度、无或不严谨执行 Checklist、非紧急 BUG 在高峰期变更等。应对措施要求研发交付后进行测试验收（灰度演练），并在变更内容确认环节对变更内容、影响范围、步骤、应急预案及灰度方案强制审批、限制时间窗口。

进入“变更确认”节点（第 3 、4 道防线）时，内部（研发 Leader 审核）与外部（业务/产品 PM 确认）双重把关，紧急变更需额外复核，方可推进到“变更中”。

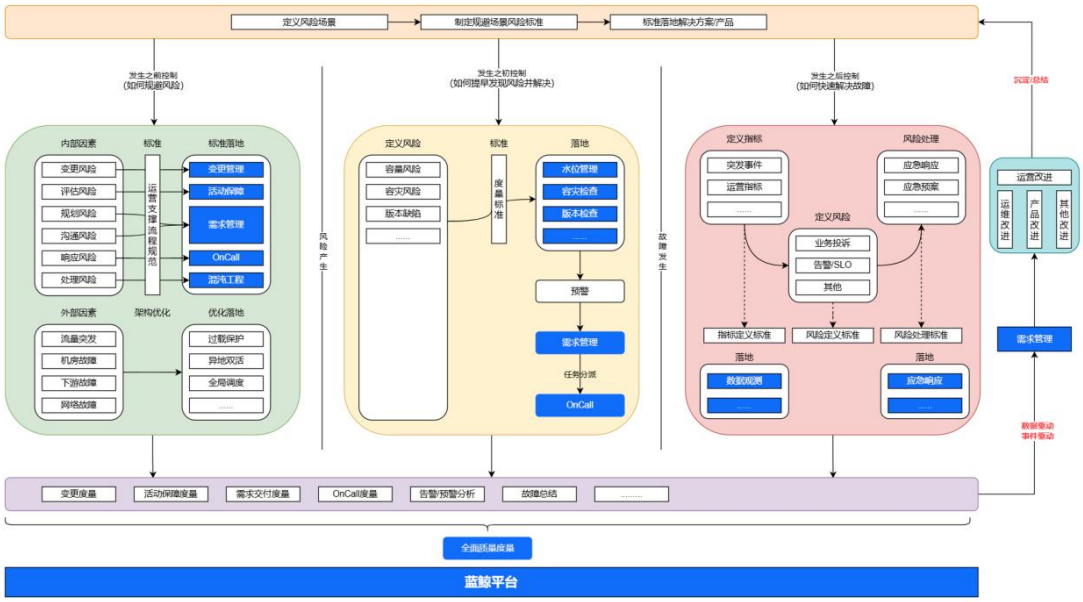
在“变更中”阶段，第 5 道防线重点防范操作失误与程序 BUG。操作失误通过考核任务固化率触发人工干预；程序 BUG 则通过延长观测时长、制定容量/性能验收标准、增加回滚出口等措施防止故障扩散。

变更完成进入“变更后”阶段，第 6 道防线通过自动巡检、AI 诊断与应急处置工具，及时发现并修复潜在异常，将延迟或残留问题反馈至变更管理台账，形成下一轮优化闭环。

更为重要的是，这些变更管控措施必须与整体的应急响应流程紧密集成。因为一旦不幸出现由变更引发的故障，最快速有效的应对手段通常就是立即执行回滚操作，或者进行流量切换，从而迅速恢复服务的正常运行。此时，迅速的行动远比深入分析故障原因更为紧迫和有效。

（三）总结和展望

苹果从 iPhone 3 到现在涨了 20 多倍，而 AI 这一波一定比移动互联网还要大"center">



应急只是风险控制链条中的一环。要想在事后应对时游刃有余，就必须把精力更多地投入到前端：基于业务特点设立可执行的标准和规范，依托平台能力打造相应工具，将这些规范真正落地。架构越健壮、平台能力越强大，后端发生故障时恢复就越轻松。采用"小步快跑"的方式，持续发现问题、持续改进，构建**故障的可控**

的 SRE 运作体系，方能在现网中不断降低风险。

5.3.10 小米米家故障应急保障体系实战：稳中求胜的构建之道

SRE Elite 精选原因

米家承载 8.6 亿设备、1 亿月活、日 PV 200 亿，一次失误即波及千万用户。团队以六层技术架构配合"三句话"多活容灾（层层防护、多机房、多版本快照回滚），并建立"流程前置-平台自动化-组织兜底"三维稳定性体系。研运一体化平台打通工单-灰度-监控，AI 因果 RCA 和 SRERobot 实现秒级定位与自愈。5-30 分钟分级应急及告警小助手，自动拉群、匹配预案、闭环复盘。

2022 跨机房专线中断与 2024 机房火灾两次实战，通过 79 项改进验证体系成效，极具参考意义，体现"稳中求胜、持续迭代"的 SRE 文化。

（一）背景及设计原则

米家应用是小米推出的一款智能家居管理平台。用户能够借助该平台，对米家以及第三方生态链中的各类智能设备进行统一连接、控制，并可自定义联动这些设备。除了实现对智能设备的统一管理之外，米家应用还具备支持场景联动以及自动化规则的功能。当前，米家应用在全球范围内处于领先地位，是全球领先的智能家居管理平台。

核心指标

- 设备接入量：8.61 亿，年均增长率 30%
- 月活跃用户：1 亿
- 累计开发者：1.3 万
- 六大区域，六大核心机房，涵盖 中国、亚太、欧洲、美洲等

以上数据既让人惊喜，又让人惊吓，因为每一次我们米家的 SRE 工程师去变更的时候，我们的一个失误都有可能造成千万的用户和设备不可用，这是一个极大的风险。

一、米家产品架构体系

(一) 设备层 覆盖"人"（手机、可穿戴）、"家"（家电、智能终端）与"车"（SU7 ULTRA 等车载终端），构成完整的人—车—家感知网络。

(二) 系统层 由座舱子系统、IoT 子系统和 OT 系统组成，分别支撑车载、家居/穿戴以及传统运营技术场景的核心能力。

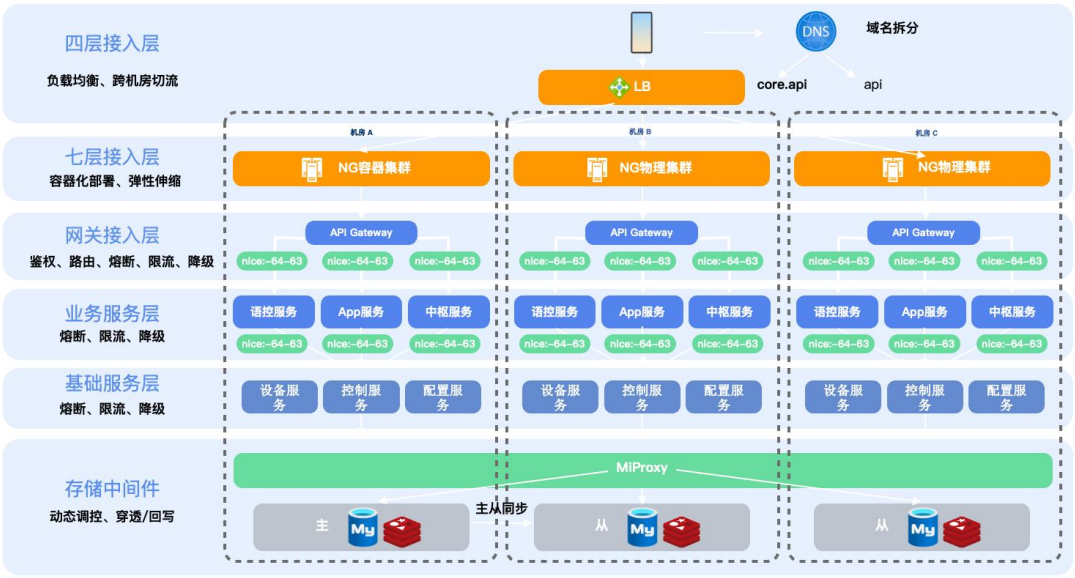
(三) 接入层 端侧通过域名+负载均衡引流，依托车接入网关、IoT 网关与 OT 网关完成协议转换，实现统一接入和初步隔离。

(四) 业务层 集中处理设备绑定、事件上报、控制指令和场景编排，支持 APP 中添加/查看/管理设备，以及开发者的自定义联动逻辑。

(五) 平台服务层 以"中台"形式抽象通用功能，提供设备管理、设备影子、设备控制，设备联动编排、SPEC 标准库及 SDK 标准接口，保证数据与业务协议一致。

(六) 基础设施层 数据层负责采集、清洗、分析、运营与配置；存储层采用 OLTP 架构、对象存储与分布式存储（MySQL/Redis/Pegasus），保障高可用与数据一致。

二、米家全链路服务治理多机房容灾架构



讲完了产品层的的架构体系，我们分享一下米家的容灾体系，可以归纳成三句话**第一句话：层层递进、层层保护**

在 四层/七层接入层、网关层、业务服务层 和 基础服务层 均配置限流、熔断、降级与故障转移机制；**任一组件异常时，可快速隔离并启动降级或切流，保障整体链路稳定。**

第二句话：多机房多活

至少在 机房 A/B/C（中国区示例为三机房）进行一致化部署；

结合 DNS 拆分 与 四层 LB，实现跨机房流量智能分发与自动切换。

第三句话：快照与事务回滚，保证数据一致性

在 存储中间件层**引入 MiProxy**

，动态路由至主 / 从 MySQL + Redis；

所有线上配置与数据操作均支持快照和回滚事务，确保一致性与完整性。

三、SRE 挑战

在如此复杂的情况下，米家的 SRE 团队面临压力过载、交付平衡双重困境**一、压力过载（Quality Pressure Overload）**

- 业务规模大:核心域名 QPS 峰值达 200 万，日 PV 200 亿
- 业务增长快：年均业务增速约 30%，还在持续增长中
- 技术架构复杂：IoT 产品 SKU 数千，设备品类与通信协议多样，用户可自由组合冰箱、电视、洗衣机、空调、灯具打造复杂的个性化场景
- 客户舆情压力大：米家 App 影响用户生活，故障后大量投诉，5–6 分钟内可冲上微博热搜**二、交付平衡困境（Delivery vs. Stability）**

技术层面挑战

- 故障定位与快速恢复：小流量 SKU 故障难以监测，如特定某款型号的门锁出故障，Root Cause Analysis 及快速恢复压力大
- 设备长尾效应：数千型号兼容性难题，证书更新、固件兼容需长期维护

流程层面挑战

- 测试覆盖不足：内部 IoT 测试平台难以实现对海量边界场景的 100% 自动化验证
- 变更管理压力：为了保持竞争力，需不断迭代新设备与功能，变更节奏与上线回滚能力需与时间赛跑

综上所述，米家 SRE 团队在应对日益攀升的核心域名 QPS、海量 IoT 设备与用户舆情压力的同时，又要保持快速迭代与严格的变更管控，不仅要在技术层面解决小流量故障定位与设备兼容性地狱，还要在流程层面填补测试与回滚的盲区。因此，小米的 SRE 团队从三个个层面的体系进行了建设。

四、稳定性体系设计原则

一、流程层：抓"两头"保障可靠性

- 研发入口：
- 技术方案评审、稳定性准入机制、架构韧性评估，从一开始就嵌入质量保障；
- 运维尾端：
- 上线前进行风险演练与准入审核
- 上线中实时监控、限流降级与故障转移
- 上线后快速响应、根因定位与复盘改进**二、工具层：让流程"跑到平台上"**
- 可观测性标准化：统一指标、日志与 Trace 采集格式，打通全链路视图；
- 自动化应急响应：基于规则引擎自动触发告警、自愈脚本，并将执行结果反馈回规则中心；
- 变更安全防控：灰度发布、限流熔断与一键回滚，减少人工操作盲区；
- 核心目标：以一条可复用的工具链承载流程，提升协作效率，杜绝人为失误。

三、组织层：超预案场景的"兜底"

- **AI 知识库联动：**

故障经验、SOP、监控规则等在知识库中沉淀，供应急时快速检索；

- **作战室机制：**

事故或演练触发后，跨团队迅速集结，统一指挥、协同处置；

- **使命：**

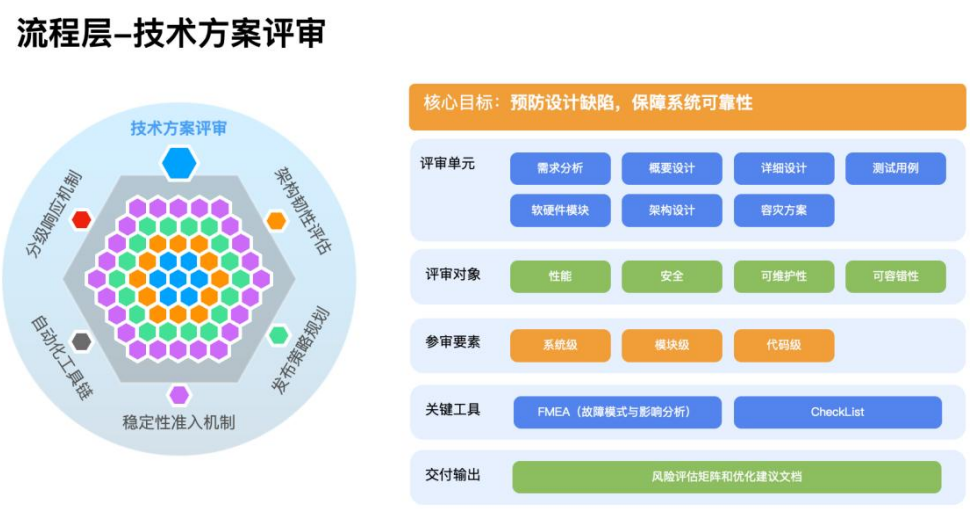
在流程与工具之外，用人力与组织协同保障未知场景的快速决策与落地。

正是在这样的双重困境下，团队构建了"前置防护–平台化工具–分级组织"三层运维体系，下一步我们将详细介绍这些策略如何帮助米家在亿级设备与千万用户规模下，实现从预防到响应、从发布到复盘的全链路高可用保障。

（二）体系设计详情

上面分享了我们总体的设计原则和分层逻辑，往下就针对每一层进行展开。

流程层·技术方案评审



核心目标：

预防设计缺陷，保障系统可靠性

我们贯彻"DevOps 前置介入、尾端闭环保障"的理念，具体做法如下：**前置介入·需求与架构评审**

-早期沟通

：从软件研发生命周期初期，即与产品及开发团队共同研讨产品需求与设计，及早发现设计缺陷与可靠性风险。

- **评审内容**

：

- **需求分析**

：功能边界与使用场景是否完备?

- **概要/详细设计**

：核心模块与软硬件规划是否合理?

- **高可用考量**

：是否内置限流、熔断、故障转移等机制?

- **测试策略**

：测试用例覆盖度与自动化程度是否充分?

分层深入·系统→模块→代码

-系统级评估

：对整体架构进行稳定性与容灾方案审核;

- **模块级评估**

：验证子系统接口、性能与运维可持续性;

- **代码级审查**

：细查关键依赖与安全漏洞（如发现 FastJSON 旧版本漏洞即时要求升级）。

聚焦四大评审维度

-性能

：资源利用率、最优算力申请、峰值 QPS 承载能力;

- **安全**

：漏洞隐患、权限隔离与数据保护;

- **可维护性**

：运维自动化、配置管理与监控可视化;

- **可容错性**

：熔断策略、回滚流程与多机房多活支持。

评审工具与交付物

-FMEA（故障模式与影响分析）

与 **Checklist**

表单:

- 将两者合并，形成**风险评估矩阵**

与**优化建议文档**

；

- 输出给架构师与团队 Leader，确保每轮评审都有"可落地"的改进方案。

流程层·架构韧性评估

核心目标：

构建抗故障的能力，保障业务的连续性**容灾层面**

-多活部署

：验证数据层、应用层和业务层是否在异地多活机房中并行运行；

- **数据一致性**

：检查跨机房的数据备份与回滚机制，确保出现故障后可快速回溯到安全状态；**故障恢复能力**

-自主健康检查

：每条业务链路都要具备实时自检能力，能及时发现异常；

- **自动故障转移**

：在发现服务不可用时，能够无缝切换到备用实例或机房；

- **限流降级**

：遇到大规模故障或流量洪峰，立即触发限流与降级策略，优先保留核心功能；**韧性验证**

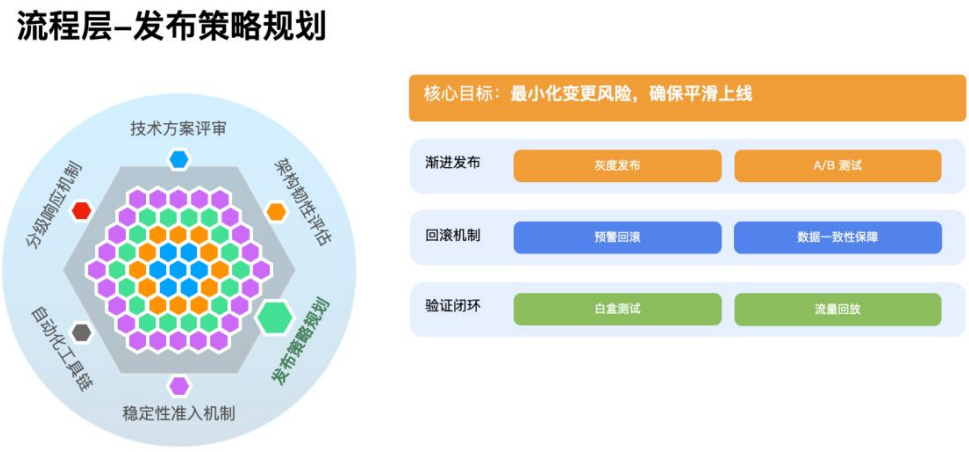
-故障注入与混沌演练

：通过自动化脚本或混沌工程，定期模拟网络抖动、节点宕机等场景；

- **红蓝对抗与安全攻防**

：邀请专业团队渗透测试，识别潜在漏洞与配置缺陷；

流程层·发布策略规划



在六大能力环中，"发布策略规划"是米家 SRE 在变更中投入最多精力的环节。其**核心目标**是：**最小化变更风险，确保平滑上线**。我们围绕这一目标，精炼出"三大法宝"：**渐进式发布**

- 按用户灰度、按区域灰度逐步放量；
- 启用 A/B 测试与蓝绿发布，验收新功能后再全量下发；
- **目的**

：将故障或回归风险限制在小范围内，避免大面积故障。

智能回滚机制

- 每个关键服务上线前绑定 SLO，发布过程监控 SLO 指标；
- 一旦发现异常（如错误率或延迟超阈），立即触发预警并自动回滚；
- 配置和数据变更均采用快照、备份与事务性回滚，确保回退后数据一致。

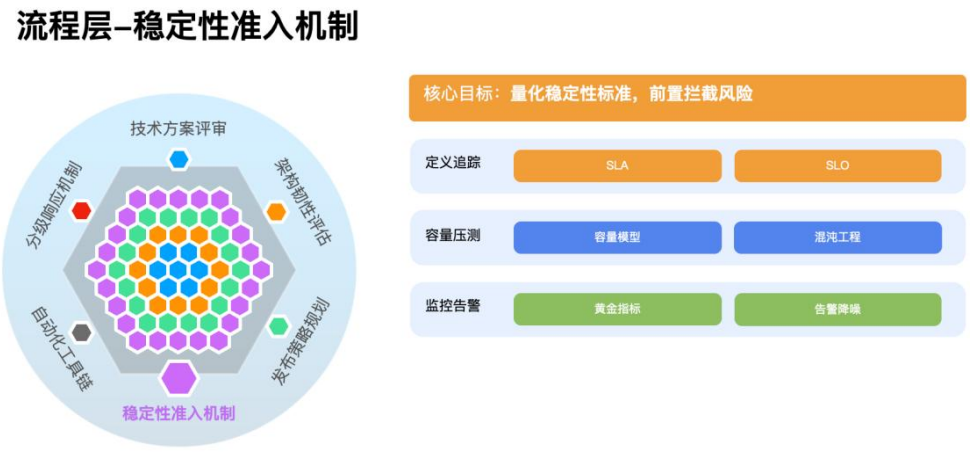
验证闭环

- 内部 BVT 自动化框架覆盖单元测试和接口测试；
- 构建 POC "影子环境"——线上全量流量回放与高危动作沙箱，验证隔离与权限策略；

- **收敛成果**

：每次发布都在与真实场景高度一致的环境中演练，缺陷提前暴露。

流程层·稳定性准入机制



核心目标：

量化稳定性标准前置拦截风险

稳定性准入三大关键要素：**定义与追踪（SLA & SLO）**

- 对外 SLA 设为 **99.99%**

，内部通过 SMDB 系统将 500+ 微服务分为 P0/P1/P2 等级；

- **P0 生死告警**

：重点关注可用性、容量承载及抖动频次——即使微量抖动也要触发告警。

容量评估与优化

-容量模型

：结合年均 30% 增长率，构建资源—业务增长的线性回归模型；

- **弹性伸缩**

：利用容器化技术按月监控 CPU、内存、QPS、IO 等指标，动态调整各可用区（AZ）配比，达到资源与成本的最优平衡；

- **混沌工程**

：在准入阶段注入故障，验证系统极限承载与自动恢复能力。

监控与告警

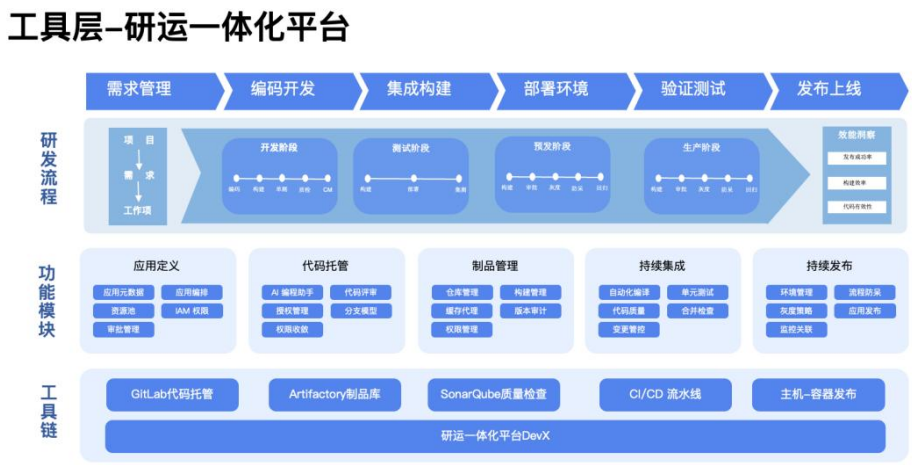
-黄金指标

：精炼少量关键指标进行全链路监控，实现链路级分析与快速根因定位；

- 告警降噪

：基于异常检测引擎过滤无效告警，并按时间窗口 / Endpoint 聚合，防止告警风暴、提升敏感度。

工具层·研运一体化平台



端到端研发流程

从需求管理

→编码开发

→集成构建

→部署环境

→验证测试

→发布上线

，全链路覆盖开发、测试、预发布与生产四个阶段，并配备效能洞察（发布成功率、构建效率、代码有效性）指标。

核心功能模块

-应用定义

：应用元数据、资源池分配、审批管理、应用编排与 IAM 权限

- 代码托管

：GitLab 托管、AI 编程助手、分支模型、代码评审与授权/权限收敛

- **制品管理**

：二进制仓库、缓存代理、版本审计、构建管理与权限控制

- **持续集成**

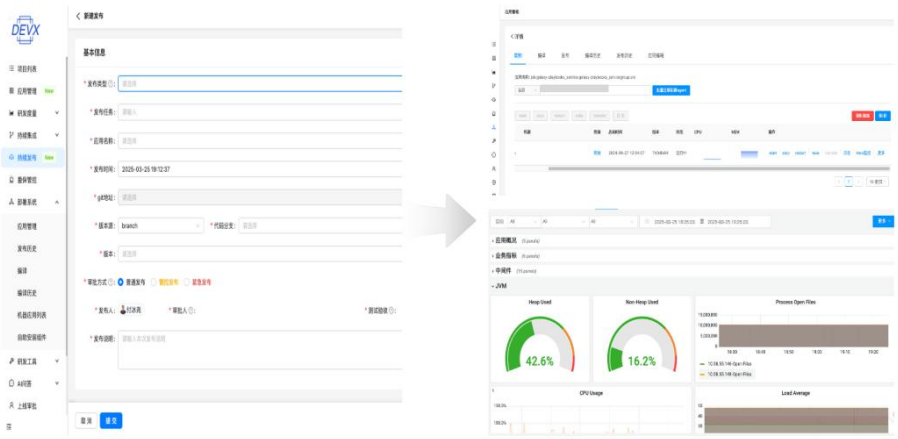
：自动化编译、单元测试、SonarQube 质量检查、合并检查、变更管控

- **持续发布**

：多环境管理、灰度策略、流程防呆、监控关联与一键应用发布**下游工具链**

底层集成 GitLab、Artifactory、SonarQube、CI/CD 流水线，以及主机/容器发布能力，共同支撑 平台的研运一体化。

工具层–研运一体化平台



以从"发布工单"到"实时监控"的一体化闭环为例子 发布前配置（左图）

-侧边导航

：项目列表→应用管理→持续发布；

- **填写"新建发布"表单**

：

- **需求工单**

：关联功能需求的工单编号与测试报告（接口 / 集成 / 压力测试）；

- **代码合入**

：选择 GitLab "release" 分支、分支授权、SonarQube 静态扫描通过；

- **发布策略**

：首次上线需从灰度模板中新建策略，老版本则直接提工单；

- **多级审批**

：依次由功能负责人（Leader）、SRE（稳定性评审）、QA（用例准备）三方在线审批。

灰度上线与实例管理（右图上半部）

-一键触发

：审批完成后自动构建并按策略推送第一个灰度实例；

- **实例面板**

：可对容器 / 虚拟机实例执行 start/stop/restart/nuke/transfer 操作，查看版本、状态、CPU & MEM 指标；**全景监控与验收（右图下半部）**

-多维监控面板

：Heap Used、Non-Heap Used、Open Files、CPU Usage、Load Average 等黄金指标可视化；

- **观察窗口**

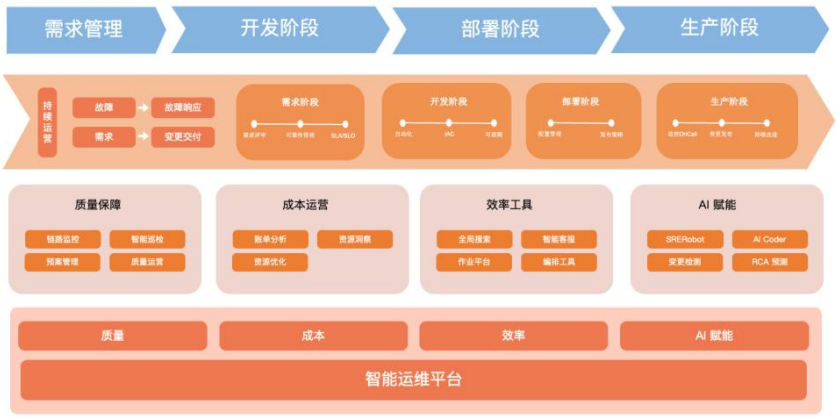
：在 30 分钟无异常后，研发工程师与 QA 共同确认回归通过，继续推进下一个灰度环或全量上线；

- **闭环反馈**

：上线结果自动归档至效能洞察模块，用于发布成功率与流程优化。

通过左侧的"发布工单→多级审批"流程与右侧的"灰度实例→实时监控"视图，研发一体化平台将发布操作与运维监控无缝衔接，实现了从需求到交付、从审批到验收的端到端可控发布场景。

工具层–智能运维平台

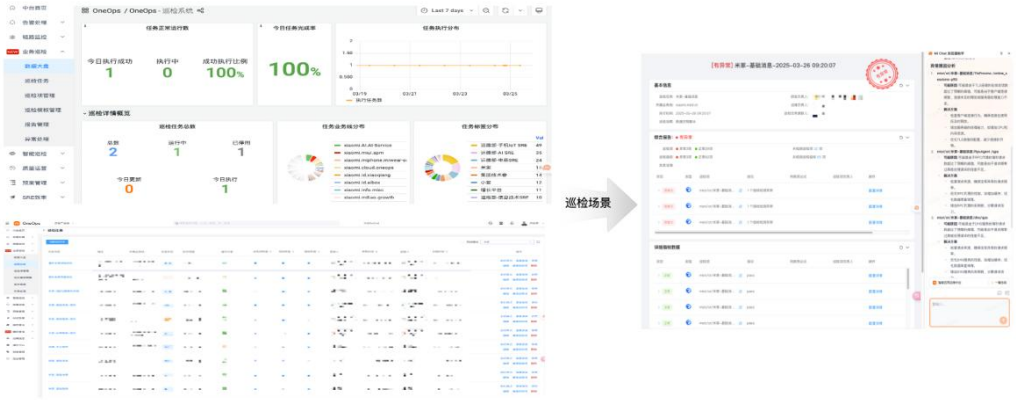


小米的智能运维平台，按照 SRE 角色场景设计规划了四个板块，质量、成本、效率、AI 赋能，为什么要这样设计呢？小米融合云平台具备完备的 CMDB、LVS、主机&容器管理平台、监控平台、预算成本平台等等，而 SRE 的工作职责主要面向质量、成本、效率，我们按照 SRE 高频的工作场景规划了相应的平台能力和功能，下面进行展开

- **质量保障**
- 链路监控、预案管理、智能巡检、质量运营
- 场景：故障告警触发后，运维工程师可在"链路监控"面板一键查看端到端调用，借助"预案管理"快速执行标准化故障脚本。
- **成本运营**
- 账单分析、资源洞察、资源优化
- 场景：每月 SRE 团队对比增长率与云资源账单报告，通过"资源洞察"界面拉取各 AZ 用量，驱动弹性伸缩与容器编排，动态调优资源配比。
- **效率工具**
- 全局搜索、作业平台、智能客服、编排工具
- 场景：当多个团队协同排查问题时，可通过"全局搜索"快速定位服务元数据，利用"作业平台"批量下发运维脚本，无需频繁切换终端。
- **AI 赋能**
- SRERobot、AI Coder、变更检测、RCA 预测

- 场景：SRERobot 在夜间自动分析最近的告警与日志，生成根因预测报告；AI Coder 辅助生成故障处理脚本，节省手动撰写时间。

工具层–智能运维平台–巡检



场景：日常业务巡检的端到端闭环

全局健康看板（左图）

每天早晨，运维工程师打开"业务巡检"基础看板，一键扫全：

- **SaaS 服务 & 自定义业务**

：展示今日执行成功数、执行中、成功率及趋势折线；

- **SLA/SLO 指标**

：汇总各服务的可用性与性能得分；

- **任务分布**

：饼图展示所有巡检任务在业务线与监控点的分布；

- **日/周清规则**

：高危异常（日清）、低危异常（周清）均在看板上标红提醒，确保无遗漏。

自定义巡检任务管理（中图）

基于 IAM 账号体系，运维工程师可：

- **按组织→产品线→资源**

层层映射，精准绑定每条巡检任务；

- **批量调度**

：针对千余垂直业务线（如手机 App、系统工具、家电中台等），灵活配置执行频率与负责人；

- **资产关联**

：账号、主机、容器与监控指标均可一键查阅，保障巡检覆盖面与可追溯性。

智能诊断与闭环（右图）

巡检任务发现异常后，系统自动生成 **AI 助手报告**

：

- **摘要分析**

：归纳本次巡检的主要异常项（如响应延迟、错误率上升）；

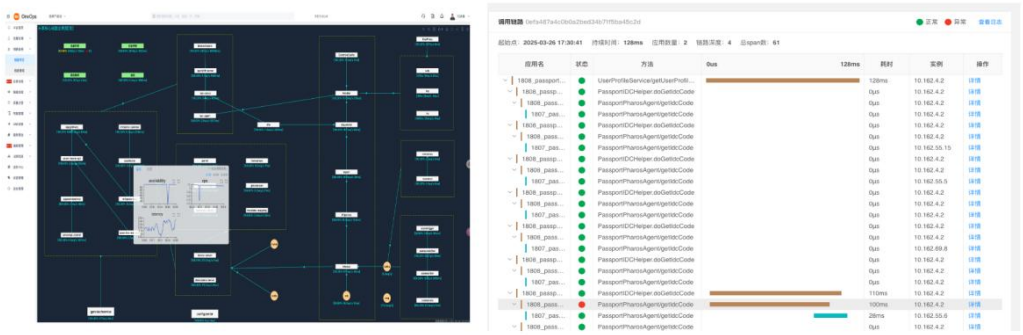
- **优化建议**

：结合历史 RCA 模型，提供脚本示例与操作步骤；

- **一键落地**

：可直接触发工单、下发自动化修复脚本或邀请 SRE 讨论。

工具层–智能运维平台–故障分析



场景：故障分析

-简化核心链路图（左图）

-手绘精华

：并非展示米家后台数万微服务，而是由资深 SRE 与架构师携手"手绘"出的 **500+ 核心功能节点**

及其调用关系；

- **清晰可读**

：图中每个方框代表一条关键业务线或功能服务，虚线区域则归并为上/下游整体；

- **故障闪烁**

：当某功能触发 SLO 异常，对应节点与连线自动闪烁高亮，瞬间显现"爆炸半径"——哪些上下游也将受影响，帮助快速锁定问题区域。

- **下转定位**

：点击异常点，可以透视到 黄金指标的关键信息，再次点击可以跳转到关键链路的指标监控，快速定位根因

- **调用链详情列表（右图）**

-Trace 概览

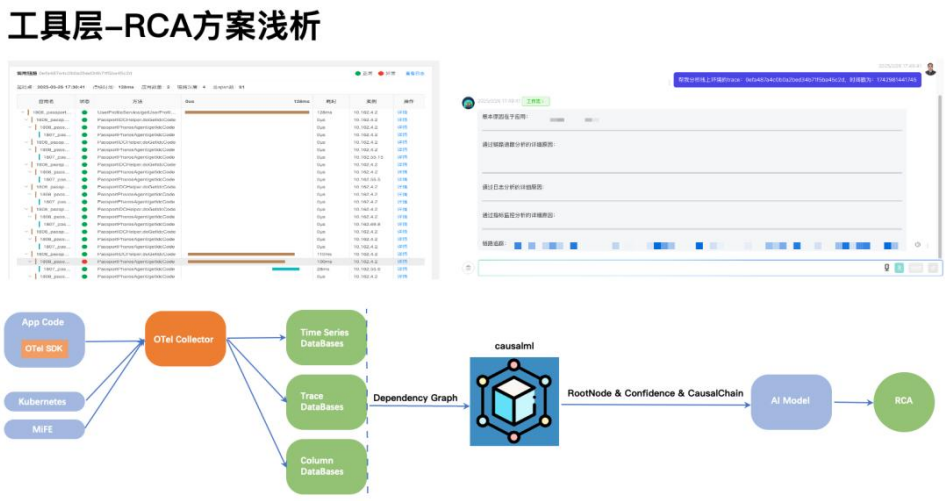
：顶部展示 Trace ID、起始时间、持续时长、链路深度与总 Span 数；

- **方法级条目**

：按调用顺序罗列各服务方法（应用名、接口名），以绿色/红色标识正常或异常调用，并突出高耗时行；

- **实例定位**

：展示每次调用对应的实例 IP 与耗时，支持"一键查看日志"深入排查。



场景：RCA 方案，告警触发 → 定位 → 预案匹配 → 执行闭环 告警触发 & Trace 列表（左上图）

- 当某条调用链超出 SLO 阈值，监控平台自动在"故障分析"面板中将高耗时或异常调用行高亮；
- 列表展示每个方法调用的应用名、接口、耗时、实例 IP，并支持"一键查看日志"快速锁定问题链路。

AI 小助手 & 预案匹配（右上图）

- SRE 输入："帮我分析该 Trace 的 root cause"；
- AI 助手基于通过滑动时间窗口拟合的方式，推动给开源的因果推断工具链，基于因果推断工具的故障权重打分，推送给大模型
- 平台自动从知识库与预案库检索对应 Runbook，呈现建议给 SRE。

人工确认 & 执行

-SRE 对 AI 建议进行确定性复核；

- 一键触发对应自动化预案或修复脚本完成故障修复；
- 修复过程与结果同步回知识库，持续优化 RCA 模型与预案体系。

实现流程：端到端的 RCA 数据管道

以下流程对应下图左半部，从"告警触发"开始，到"Root Cause 输出"结束：**埋点采集**

- 在 App 代码（OTel SDK）、Kubernetes Pod 与 MiFE 前端注入 OpenTelemetry 探针，自动收集调用链 Trace、关键性能指标与日志。

数据汇聚

- 各节点的采集数据统一发送至 OTel Collector，进行格式化与批量转发。

多路存储

- 根据数据类型分别写入：
- **时序数据库**

（指标数据）

- **Trace 数据库**

（调用链）

- **列存数据库**

（日志与上下文）**依赖图构建**

- 从 Trace DB 中提取所有服务调用关系，生成全局依赖图（Dependency Graph），作为因果推断的骨架。

因果评分

- 将依赖图结点、Trace 统计权重（调用时延、错误率、调用频度）输入 CausalML 引擎；
- 挑选得分最高的节点作为 RootNote，并将其连同置信度（Confidence），因果链（Causal Chain）传递给 AI 助手与 SRE，完成 RCA 闭环。

组织层·应急响应机制

在突发故障场景下，米家应急响应机制分为"重大"、"严重"、"一般"三层，结合**决策层级**

与**升级阈值**

，确保高效集结与闭环处置。

- 分级与决策（见上图"分级响应体系"）

故障级别	触发条件	决策层级	汇报 & 集结阈值
重大级别	收入受损或核心功能不可用	部门经理 + 专家组	5 分钟内全部成员集结
严重级别	关键功能异常影响业务体验	部门总监	15 分钟内完成流程闭环
一般级别	局部功能抖动或非核心服务降级	值班 OnCall 与团队决策	30 分钟内内部排查完毕

- **高 T 决策**

：重大问题需高层快速拍板；

- **阈值触发**

：触及"收入+核心功能"即升级至重大级；

二、协同处置流程（见上图"协同流程"）

- **触发告警**

：监控系统检测到异常并发出告警；

- **自动定级**

：系统依据 SLA/SLO 与业务规则自动判定故障级别；

- **启动作战室**

：对应级别的角色在线集结（部门经理/总监/OnCall）；

- **角色赋能**

：自动派发工单与权限，赋予各方所需工具与数据访问；

- **信息同步**

：共享链路拓扑、监控大盘和 AI RCA 报告；

- **预案执行**

：运维、开发、QA 按 Runbook 协同处置或触发自动化脚本；

- **状态反馈**

：实时更新故障进度，30 分钟—5 分钟内完成闭环，并同步复盘。

通过 **三层分级＋标准协同流程**，米家运维组织在 **5–30 分钟**

内完成从**告警**

到**闭环复盘**

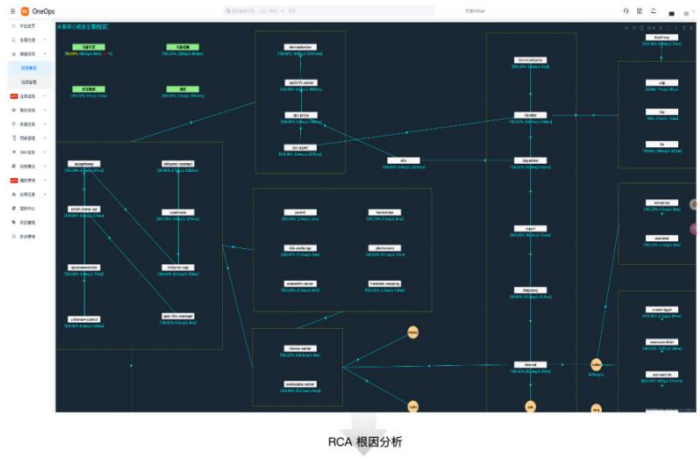
的全流程响应，确保千万级用户和亿级设备规模下的系统高可用与业务连续。

组织层·应急响应实例：P0 生死告警群 ＋ 自动化小程序

当可用性跌破 P0 阈值时，监控系统会在专门的 "IoT 米家生死告警群"（左图）中推送红色高危告警，并自动弹出右键菜单中的 "告警小助手"。SRE 工程师仅需以下几步，即可完成从"告警"到"信息聚合"再到"协同处置"的全流程：**告警小助手一键建群：**

点击"告警小助手"后，自动抓取本周 OnCall 值班人、研发负责人 与 平台负责人，新建临时作战群，同步该业务的 变更记录 与 相关告警历史到群里头

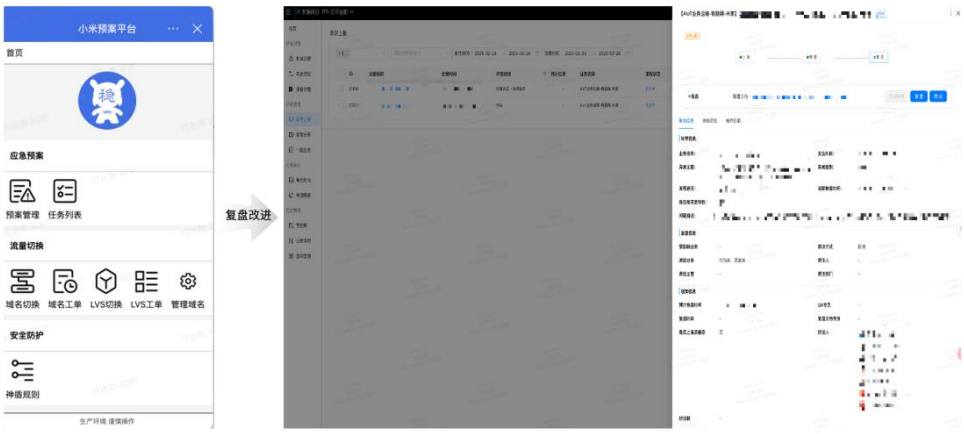
组织层–应急响应机制–案例



信息自动聚合：

所有被拉入的响应角色即可在群内一览全貌——包括调用链拓扑、告警语句与历史趋势。

组织层–应急响应机制–案例



然后进入预案平台，快速切流或执行业务层的降级预案，最后预案平台会联动 质量平台，生成复盘报告入口，将协同过程的飞书信息生成一个摘要文档，便于 SRE 和研发工程师进行信息补全最终形成改进 TODO。

案例一：2022 年跨机房专线中断

事件回顾

第一个案例发生在 2022 年的 4 月 30 号，由于网络的故障，第一次"喜提"微博热搜。

背景回顾

当日正值业务高峰，团队在多机房架构下对 POP 点专线的拨分设备进行替换升级评估；由于对兼容性风险评级不足，工程师误判为"低风险"后直接实施，导致主线路断连而备线路因带宽不足发生严重丢包。

触发与演进

- 17:30 SRE OnCall 工程师接到 C3 和 C4 两个核心机房出现丢包的监控告警，随即开始排查。
- 随后告警持续升级，至 19:15 故障已连续恶化超过 115 分钟，常规降级与切流未能恢复服务。
- 在超过 115 分钟后， SRE 这个时候已经没有其他任何办法了，在评估全量流量转移至单一机房的风险与可能性后（单机房扛不住，所有业务会雪崩，；多机房流量互串），SRE 团队果断决定"多机房切换至单机房模式"，但由于单机房带宽仍无法承载全部业务，初次切换依旧未能成功恢复。

- 20:05 硬件服务商定位问题根源波分设备兼容性缺陷，并立即推送新的固件版本；
- 20:35 固件升级完成后，跨机房专线恢复正常，业务可用性重回稳定。

复盘流程

事实还原

撰写多篇详实文档，按时间轴细化从 17:30 到 20:35 的每一次决策与系统状态，量化收入损失、用户投诉量与流量丢失数据。

原因分析

- 触发原因：在未完成兼容性验证的前提下对波分设备进行升级。
- 根本原因：设备替换流程缺乏严格的风险评级与可行的回滚预案。
- 复盘原则：所有参与者严格自我反思，仅剖析自身决策与执行偏差，杜绝相互指责。

改进建议

- 在设备替换与升级预案中新增"兼容性压力测试"与"回滚演练"步骤。
- 优化多机房流量切换脚本，提升备线带宽分配与自动切流能力。

执行与跟踪

- 在质量平台自动生成 Postmortem 事件，列出 79 项改进任务并分派给各责任团队；
- 持续跟踪任务完成情况，确保所有改进在一年内闭环。

案例二：云厂商火灾触发数据锁定

事件回顾

在 2024 年 9 月 10 日，小米位于新加坡的 IDC 机房发生火灾。该机房四楼所承载的米家摄像机服务 OSS 对象存储服务受到影响，数据被锁定，从相关图示可见其状况。

复盘流程

案例二：云厂商火灾触发数据锁定 — 应急处置全故事

下图以时间线形式展示 9 月 10 日新加坡机房火灾导致对象存储锁定时，米家 SRE 团队的从慌乱到沉着全过程。

一、响应定位（09-10 10:20 – 11:22.

- 10:20 可用区 C 首次 1 分钟级波动，SRE 接警，初步判断是云厂商底层网络抖动，在联系云厂商给出问题原因。
- 11:20 同一区域再度抖动，服务提供商依旧"毫无头绪"，并没有给出第一次抖动的反馈。
- 11:22 SRE 当机立断，立即"启动作战室"预案——缩容 C 区、扩容 A/B 区全量流量

二、预案处置（09-10 11:22 – 20:23.

初步切换（11:22 – 12:10. 在小米预案平台，一键触发 "缩容 C 区 + A/B 区灾切"

12:10 但是在操作完毕之后，业务基本恢复，但 OSS 读写异常，此时联系原厂商定位故障原因，云厂商答复："对不起，现在对象存储没法用了，能不能 new create 一个 Bucket？"，当时 OnCall 工程师的心"咯噔"一下，因为旧数据可能永远丢失；但目前有限还是先处理新业务的写入问题，SRE 团队立刻创建"New Bucket"任务，把所有流量都导入到新的 bucket 之中，业务写数据逻辑恢复

13:15 业务可用性恢复到 99.9%

业务恢复后，SRE 启动利用内部 HBase 源数据与备份进行详细数据恢复的方案，但是需要一天的时间，幸运的是 20:23 的时候云厂商告诉我们数据恢复了。

三、09-12 15:00 在质量平台完成第一轮复盘，输出六大模块：**还原事实，评估影响，原因剖析，问题反思，改进建议，跟踪反馈**

：

可以看到，与我们 22 年那次 4 月 30 号相比，形成的鲜明的对比。这次我们就很果断有信心该执行相关的预案，有一定的进步，但是还是有问题：

第一个就是为什么 多 AZ 的 OSS bucket 会被锁定呢？为什么监控发现不了

第二个问题是为什么数据恢复需要一天时间，预案还要商量

在此，我们仍有诸多提升空间，因此我们需持续进行复盘与改进。

（三）总结及展望

回顾米家产品这几年的发展历程，业务发展飞速，质量要求越来越高，SRE 围绕质量工作虽然有了一些积累和规划，但仍处于追赶业务的亡羊补牢阶段，如何通过体系化能力的建设及 AIOps 能力弯道超车是米家全体 SRE 同仁短期工作的重中之重，我们会持续带着敬畏心和专注力，持续打磨做好米家的质量保障工作。

展望未来，首先要做好当下，持续打磨好质量体系的工具平台能力，让运维保障类工作更加高效落地；其次，展望未来，在 AI 时代，运维体系和技术的变革不仅仅是挑战也是机遇，我们要拥抱 AI 技术能力带来的效能提升的同时，认清楚技术与体系运营的边界，用 AI 赋能的同时，深挖运维保障体系建设的灰色地带，和业务一起做好技术体系和服务体系的建设工作，确保业务的稳定、健康、高质量的发展。

5.3.11 广发证券数智化重构故障管理：构建主动防御新体系

SRE Elite 精选原因：

广发证券设计了通过应急实现全生命周期稳定: 运维左移通过架构韧性模型前置可靠性；变更管控用数字化平台与感知因子，达成事前防御、事中阻断、事后追溯；应急指挥借 ECC 与 ChatOps 机器人秒级集结，数字化预案保障快速恢复；大模型运维整合算法中心、知识库与智能体，智能定位根因、降噪、脚本合规；机器人把监控、变更、巡检等能力原子化嵌入 IM，释放专家价值，并探索多模态看板异常识别。

整体方案完整，落地，并具备一定的前瞻性及探索性， 是证券行业的 SRE 数智化标杆案例。

（一）背景及总体解决思路

本次分享将围绕几个核心方面展开：

- 全生命周期故障管理：探讨如何在应用程序和系统的整个生命周期中融入故障管理的理念和实践。
- 运维左移：阐述如何将运维职责和活动更早地融入到软件开发和测试阶段，从而在早期发现和解决潜在问题。
- 变更管控：详细说明如何对系统变更进行严格管理和控制，以降低因变更引入故障的风险。
- 应急指挥：介绍在故障发生时如何建立高效的应急响应和处理机制，确保能够快速恢复服务。
- 智能化与场景融合：探讨如何将智能化技术（如大模型、基于算法的异常识别等）与具体的故障管理场景相结合，实现人机协同的运维模型。

一、全生命周期故障管理

证券公司交易系统的运行具有每天"秒杀"（开盘业务量巨大）、应急时效性要求高、客户权益受损零容忍等特点。

与此同时，行情变幻莫测导致系统负载难以预测，而客户对系统可用性的要求更是到了极致，因为股票交易遵循时间优先、价格优先原则，哪怕一笔交易出现问题都可能导致客户投诉甚至产生资损。另外近年来，证券行业数字化转型、重要交易系统架构升级、容器云/多云/AI 等技术发展创新，都给证券公司运维带来较大挑战，行业内重大故障频发，公司对系统稳定性建设愈发重视。

以往，故障管理主要关注系统故障应急发现、定位、处置、恢复等能力建设。但仅靠出现故障后被动应急的方式很难满足证券行业对客户体验的高要求，广发证券信息技术部运维团队的工作不局限于传统运维部门快速处理故障和发布变更等被动性工作，而是积极推动转变角色定位，要求运维人员需要参与到软件设计开发阶段，包括从架构设计到关键功能逻辑实现的环节，协同构建具有架构韧性的系统；在推进变更自动化同时，构建变更管控解决方案，实现变更风险可感知、变更因子可观测及变更风险可防御，防范因变更引入的故障风险；同时结合 chatops 和人工智能技术，构建在故障发生时应急处置总线，提升应急恢复能力，形成一个闭环。

通过将故障管理贯穿于整个系统生命周期，我们才能真正实现从"救火"到"防火"的转变，系统地提升系统的可靠性与韧性。

二、全生命周期管理—稳定性保障全景

为保障复杂系统稳定性，广发证券将稳定性保障前移至系统设计与测试阶段，构建了围绕六大核心能力

的稳定性保障全景体系，强化软件生命周期各环节的紧密协作。在系统上线前将非功能性需求左移至研发设计阶段，上线后持续挖掘并防范线上风险，推进优化解决，形成稳定性保障闭环。

故障可恢复:

关注"系统关键业务功能"的容错能力设计，使其具备限流、熔断、降级、隔离等容错能力，确保系统面对故障时能快速恢复正常运行。

性能可扩展：

在业务增长、用户流量突增或面临生产性能故障时，系统能够通过灵活调整资源和服务配置来满足不同的性能需求，展现出架构的弹性。除在架构层面确保性能的可扩展性，还应制定性能扩展预案，并定期进行压力测试和演练，以确保系统在实际业务增长或高流量事件中能够平稳过渡。

变更可管控：

实现"变更计划、变更申请、变更评审、变更发布、变更验证、首日保障、故障应急、复盘优化"全变更周期管理闭环，推动应用软件在制品、应用配置、数据库脚本的持续交付(CD)自动化发布，构建变更影响评估、变更风险防御、变更状态感知、异常变更定位能力解决方案，降低变更风险。

业务可监控:

构建应从基础设施层向上延伸至客户体验层，涵盖基础设施监控、硬件服务器监控、平台软件监控、应用可用性监控、应用功能监控以及性能和客户体验监控。。

问题可观测：

制定可观测数据埋点的技术标准，在多团队达成共识，围绕问题剖析，设计面向运维、研发、测试、供应商等多种视角的数据分析场景，通过指标、日志、链路追踪等手段快速定位问题。

部署可感知：

将构成系统所有软硬件及其依赖资源数字化，实现对资源对象状态及变化的实时感知。部署可感知能力使得从整体系统资产到微观的环境配置或业务参数，都能被随时观测和掌握。。

上图中间蓝色的区域是 SRE 稳定性保障的关键过程和主要工作内容，涵盖了从容量管理、变更管理、监控管理，到架构管理、应急管理、风险治理和运行评估等多个方面。

上图下面红色部分是我们梳理的推进稳定性保障进行左移的切入点。以容量管理为例，去年底的天量行情大家一定记忆犹新，不少同业都出了稳定性事故，面对天量行情不是说加满 CPU/内存/带宽，多部署几个实例副本就能提升系统容量，我们需要评估哪些组件能 scale up，哪些能 scale out，是否有组件是瓶颈，在系统设计阶段就应该定义系统的设计容量，考虑垂直和水平扩容方案，消除容量瓶颈，在测试阶段就应该进行压力测试，绘制系统容量曲线，这样才能在稳定性保障过程中做到心中有数。

三、全生命周期故障管理—故障可恢复

故障的可恢复性指，当系统因内部或外部因素导致业务中断或面临故障风险时，系统应具备及时发现故障的能力，并能够迅速采取降级、限流、切换、重置、隔离等韧性措施以恢复业务连续性

在去年的天量行情应对过程中，我们也再次感受到良好的架构韧性在稳定性保障中发挥的重要作用，除了系统扩容，我们还需要依赖降级/限流/熔断/隔离等手段，比如行情降低刷新频率，就能在不太影响体验的情况下，降低系统和带宽压力，对不重要的交易功能做限流，保障关键交易，或者通过调整业务特性开关的方式，改变用户与系统交互逻辑或者短路一些有问题的业务逻辑，大大降低系统压力，提升系统稳定性等，这些能力都依赖研发阶段在架构韧性上的考量，使系统内建架构韧性。

为了实现故障快速恢复，我们大概在 22 年开始投入资源做故障发现与处置，在运维平台上打造了应急管理场景工具，实现一站式应急指挥和从发现、定界、处置、定位、复盘的应急过程管理，持续完善业务监控、智能日志监控，系统全景感知提升故障发现能力，开发了故障自愈、数字化应急预案等场景工具，提升处置效率。但是出故障后被动应急的方式很难满足证券行业对客户体验的高要求。因此，我司运维团队逐渐将工作重心左移，

以更积极的姿态参与到软件设计开发阶段，从架构设计到关键功能逻辑，构建具备良好可运维性的软件交付机制。

四、全生命周期故障管理

在全生命周期故障管理实践中，我们构建了一套涵盖系统内建稳定性、主动变更防御与管控分析，以及数智化重塑故障管理全过程的体系，以期在故障发生前、发生时及发生后都能具备更强的应对能力。

运维左移，内建系统稳定性：

目标是使系统从设计之初就具备更高的稳定性。通过架构评审、上线评审、非功能规范的制定与执行，以及定期的架构韧性评审站会，我们将稳定性要求前置到研发和设计阶段，避免运维在应急时陷入只会采取有限手段、过度依赖研发和厂商的被动局面。

构建主动变更防御和管控分析：

鉴于变更常常是引发故障的主要因素，我们对变更实施了全过程的严格管控。这包括变更评审、时序管控、同步对齐、技术验证等环节，确保变更的规范性和安全性。通过变更事件感知、自动化变更、变更复核以及功能验证、压力测试等手段，我们致力于提前发现和解决变更引入的潜在故障风险。标准化的操作流程、首日保障、首笔验证、关键巡检等措施构成了变更生效后的防御机制。强大的变更防御能力结合实时的变化感知，使我们能够快速识别变更带来的异常，从而将风险扼杀在萌芽阶段。

持续提升应急管理能力应急发现与定界：

利用智能化手段提升预警和告警的及时性和准确性，快速确定故障的影响范围。建立高效的应急指挥体系，基于 chatops 打造应急总线，高效建立应急连接，联动人、工具、数据、系统，持续打造应急预案和应急工具，提升处置效率。

数智化重塑故障管理全过程：

有效的故障管理离不开工程化、数字化和智能化的支撑。我们持续在混沌工程、可观测性、变更管控和应急管理等领域进行建设。通过线上化、数字化和智能化的手段，我们不断地重塑故障管理的各个环节。建立高效的指挥体系，联动应急预案和应急工具，组织人员快速应急止损，控制故障扩散。

（二）体系设计详情

一、事前防范：以运维左移提前做好风险防范

近年来 Devops 理念已然深入人心，它极力倡导产研、测试、运维各职能紧密协作，通过自动化手段，显著提升业务价值交付的效率与质量，核心聚焦于业务价值的快速流转。然而，DevOps 的推动常常以研发团队为主导，其核心关注点在于业务价值的交付速度，这使得它在满足 SRE 对系统极端稳定性、高可靠性等方面深层要求时，有点力不从心，难以完全契合 SRE 的核心使命。

从软件完整的生命周期视角审视，问题的发现和解决越在早期阶段，其所需的修复成本便越低廉。这无疑对我们 SRE 团队提出了更高要求，需要我们主动出击，**积极拥抱并推进"运维左移"（Left Shift）策略**。这意味要将诸如可靠性、可维护性、可观测性等一系列非功能性需求，前置到系统设计与研发的早期阶段，从源头嵌入稳定性基因。

然而，"运维左移"的实际落地挑战巨大，不像 Devops，它拥有成熟的文化理念、丰富的工具链以及大量的成功实践可供借鉴，"运维左移"在落地过程中，往往需要从零开始构建共识、设计新的协作机制，**并在团队内部逐步培育相应的文化氛围**，更离不开扎实的工程实践作为支撑。

接下来，我将尝试从如何构建共识、建立有效机制、推动文化变革以及夯实工程性支撑等多个层面，为大家详细剖析落地"运维左移"的具体策略和实践路径。

1、围绕系统稳定性制定“架构韧性模型”

运维左移的关键路径 达成共识：

首先需要在各团队间就非功能性需求及标准达成明确共识。

运维左移的核心工作：

架构韧性建设

广发证券运维团队基于实践经验，将系统架构韧性细致划分为 L1 到 L5 五个等级，以便更好地指导和评估系统的鲁棒性。

架构韧性 L1～L5 模型，根据这个架构韧性的模型我们定期会组织一些技术评审

- L1 基础保障级
- L2 具备逃生通道（可恢复）
- L3 可热操作止损（无单点风险）
- L4 可止损隔离（自愈 / 半自愈）
- L5 跨系统间止损免疫

通过系统梳理和明确要求，团队从以下四个关键维度，支撑和优化系统的架构韧性，并作为我们工作的牵引：

- 故障可恢复性：涵盖了冗余设计、故障隔离策略、自动化故障转移、熔断与降级机制等，确保系统能在发生故障后迅速恢复服务。
- 性能可扩展性：关注通过自动化扩容、数据库性能调优、多级缓存应用、负载均衡配置以及异步任务处理等技术手段，保障系统在面对流量增长时的稳定与高效。

- 运行可观测性：注重构建全面的监控指标体系、统一日志规范、Tracing 全链路追踪能力、以及拨测支持等，从而实时掌握系统运行状态，提升问题定位效率。
- 系统可运维性：强调实现自动化部署、建立可靠的回滚机制、完善配置管理、提供清晰的维护文档及细致的访问控制，以提高系统的日常管理与维护效率。

2、通过技术架构评审检验系统准入

运维左移的落地，需要建立工作机制，形成闭环，在系统立项和上线以及架构重大改造等环节，我们设立了技术委员会评审机制，技术委员会的委员包括了各运维条线的专家，将非功能性需求和稳定性保障要求纳入评审范围，同时这些委员也有推动技术规范的落地和培训。另外在故障应急复盘、应急演练、三线风险挖掘等运维过程中发现的风险，我们也建立了风险揭示和问题单等流程，闭环反馈给研发/测试，从而形成一个持续改进、不断强化的闭环。

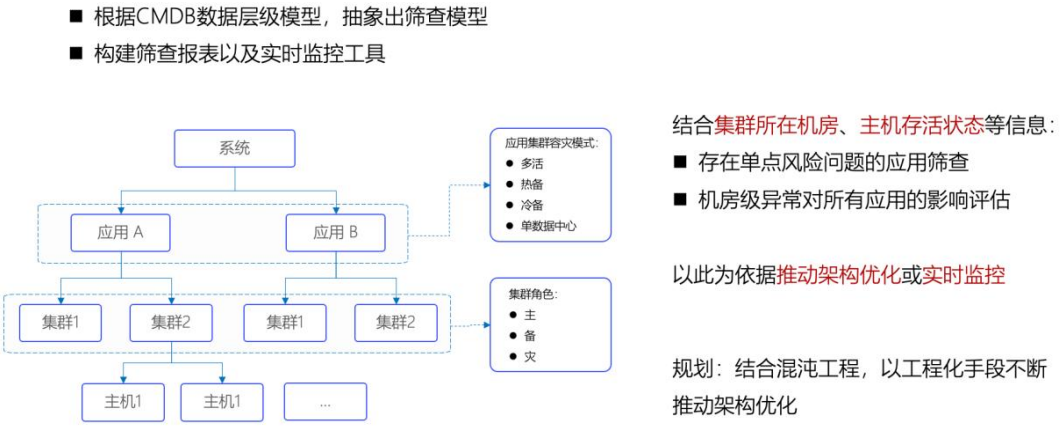
3、由一线专家牵头挖掘线上系统架构韧性

运维团队要实现从被动响应到主动出击、推动"运维左移"的核心转变，其前提是一个深刻的观念革新。为了系统性地在整个组织内促进和强化这种"运维左移"的文化，我们定期组织和举办系统架构韧性评审站会，在一个开放包容的讨论氛围下，专注架构韧性探讨，定期开展持续提升运维能力。

评审站会现场照片

通过这样的评审机制，我们不仅能对现有系统架构的鲁棒性、弹性进行评估，更重要的是，它建立了一个跨团队的学习交流平台。在评审会上，来自不同团队的运维专家可以互相分享经验、最佳实践以及遇到的挑战及其解决方案，从而在开放、协作的环境中相互启发、共同成长，最终目标是整体性地提升各个团队的运维能力水平。

4、推动应用消除单点风险达成架构韧性 L3 工程实践



最后，运维左移落地，离不开工程性平台/工具的支撑，比如可观测行落地，需要建立可观测平台和最大程度方便研发埋点，以最小成本实现系统可观测，变更发布也需要交付平台的支撑，落地运维对自动化发布标准和要求。

上图是我们在做业务连续性治理时，通过工程性手段推动系统及应用架构单点风险优化的例子，结合集群所在机房以及主机存活状态等信息，筛查单点风险应用，建立机房级异常对应用影响的感知。

未来，我们也会通过混沌工程等工程手段，持续挖掘架构风险，建立对系统故障处置的信心。

二、变更管控：持续强化准入交付门槛

1、建立一下数字化的变更管控闭环

在 SRE 的实践当中啊，这个变更管控是属于非常重要的一个环节， 我们将会从 变更生命周期管控，变更操作执行痛点，主动变更防御，变更感知与风险分析四个方面进行阐述。

"变更"无疑是系统稳定性的头号杀手。尤其对于我们金融证券行业而言，更要如履薄冰，因为任何一次变更完成后，都可能直接面临业务流量的上限井喷高峰。在我们的金融交易系统中，变更通常按照周进行迭代发布。每次变更上线后，严格的测试验证必不可少，而上线首日更是需要我们倾注全部精力进行重点保障。

备注：

监管层如证监会、央行、发改委 等部门常在节假日前后发布重磅政策如降准、IPO 调整、行业扶持等，是为了利用假期期间消化影响，节后开盘市场剧烈波动，节后首日常被称为"政策窗口期"或"情绪释放日"，股民集中交易，券商 APP、交易系统、行情数据平台等压力剧增，假期前后往往也是 IT 部门推进系统升级、数据库调优、发布新版本的时间窗口（因业务量相对低），**但节后立刻遭遇市场交易井喷，就会放大变更引入的隐患。**

为了最大限度地降低变更风险，我们构建了一套严密的变更管控流程，涵盖了从最初的评审到最终的保障各个环节：

- **评审阶段：**

这是变更的第一道关口。我们会在这里进行深入的沟通变更内容，从全局视角把控所需资源，进行严格的系统内评审，并通过必要的 CAB 评审（即变更咨询委员会评审），从源头确保变更的合理性和可行性。

- **同步阶段：**

在评审通过后，至关重要的是确保相关方高度同步。这包括运维团队间的对齐，明确操作时间对齐，进行细致的重要保障评估，并对整个变更的操作时序进行评审，避免因信息差或误解导致的执行偏差。

- **实施阶段：**

进入实际操作层面，我们严格遵循见单操作、操作复核规范，确保每一步变更实施都可追溯、可审计。同时，引入双岗复核机制，通过二人相互校验，进一步降低人为失误的概率。

- **验证阶段：**

这是检验变更效果的关键环节。我们会进行全面的测试验证并记录问题，基于验证结果最终确认是否上线。这其中包含了必不可少的功能测试，以及在高流量场景下的压力测试，确保系统在真实负载下表现稳定。

- **重点保障阶段：**

正如前面所说，变更后首日业务重点保障至关重要。为了应对潜在的突发情况，我们会进行关键时刻巡检，甚至启动保障集结机制，快速调动相关资源。更会进行首笔验证，确认核心交易流程畅通无阻。在一天交易结束后，还会进行盘后复盘，总结经验教训，为后续变更提供指导。

2、以痛点驱动设计变更管控

即使我们已经采取了多种变更管控措施，但在实践中仍面临一些棘手的痛点。

首先，**事前变更影响难以评估**

是一个普遍存在的挑战。这主要体现在审批过程过度依赖个人的经验和判断，往往难以进行全面客观的影响分析；同时，审批所依据的信息可能与实际的执行操作存在脱节，导致评估结果不够准确。尤其是在跨团队和跨系统复杂交互的场景下，准确预判变更的连锁反应变得异常困难。

其次，**操作不标准引发操作风险**

也是常见的症结所在。许多常规操作缺乏统一的执行范式，难以形成可沉淀、可复用的标准流程。这不仅容易导致关键操作步骤的遗漏，还可能使得技术检查和验证工作不到位，无形中增加了非预期故障的概率。

再次，对于**高危变更缺乏事中管控**

也是一个亟待解决的问题。在系统处于高危时段，或者变更本身就具有高风险特性时，往往缺乏有效的实时管控手段。一旦出现异常，无法快速启动阻断机制来停止或回滚问题变更，这可能导致故障范围的快速扩大，造成更严重的损失。

此外，我们还发现**变更对象关键变化无感知**

是导致不稳定性的一大诱因。许多故障的根源在于变更触发了变更对象（例如系统配置、关键参数等）的稳定性因子发生了异常变化，而我们通常对此缺乏有效的感知能力，也未能建立相应的预警和事后分析机制。

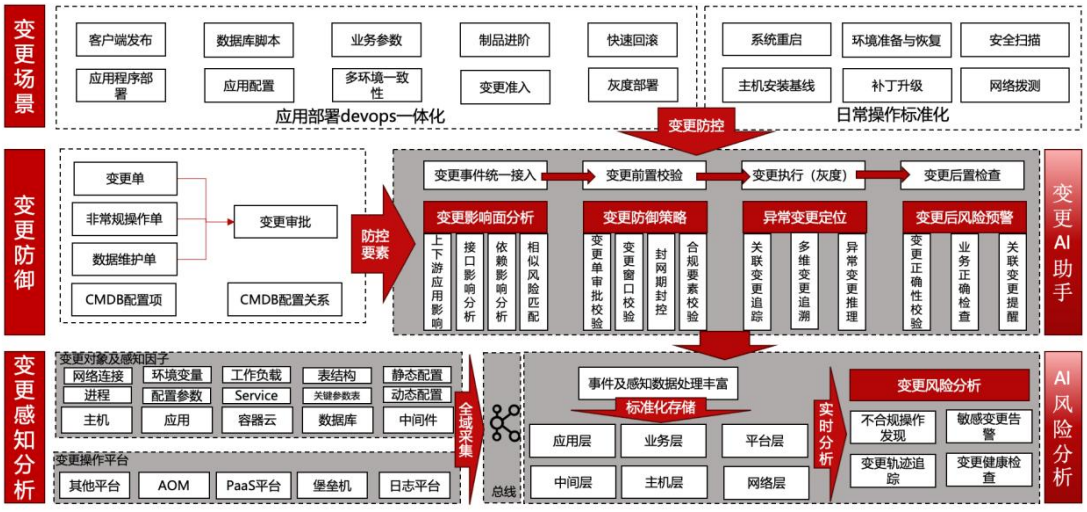
同时，**变更操作行为无感知**

也带来了不少困扰。我们常常缺乏对全局变更操作事件的整体视图，难以获取跨团队之间关联的变更操作信息。这使得一些非预期的操作行为难以被及时发现并预警，同时也给事后的审计分析带来了困难。

最后，当异常或故障发生时，**快速定位引发故障的变更**

往往变得非常困难。这主要是由于缺乏全局的变更信息视野，难以迅速锁定是哪一个具体的变更操作导致了问题。现有的变更流程未能有效地将变更审批、变更对象状态变化与实际操作事件关联起来，导致故障溯源效率低下。

3、将变更管控场景工程化



为了应对这一挑战，我们设计了一套全面的变更管控解决方案，其核心理念在于将日常标准化的变更操作都纳入统一的变更管控平台进行管理。

首先，通过构建强大的变更防御能力，我们对这些纳入平台的标准变更操作进行管控，减少那些容易出错的"黑屏"操作和繁杂的手工执行，确保每一步变更都清晰可追溯。

然而，并非所有变更都能完全纳入平台进行自动化管控，例如在堡垒机或某些 PaaS 平台上的操作。对于这些游离于标准化流程之外的变更，我们建立了一套强有力的兜底机制，着重于构建变更感知和分析能力。这套机制包括两个关键组成部分：

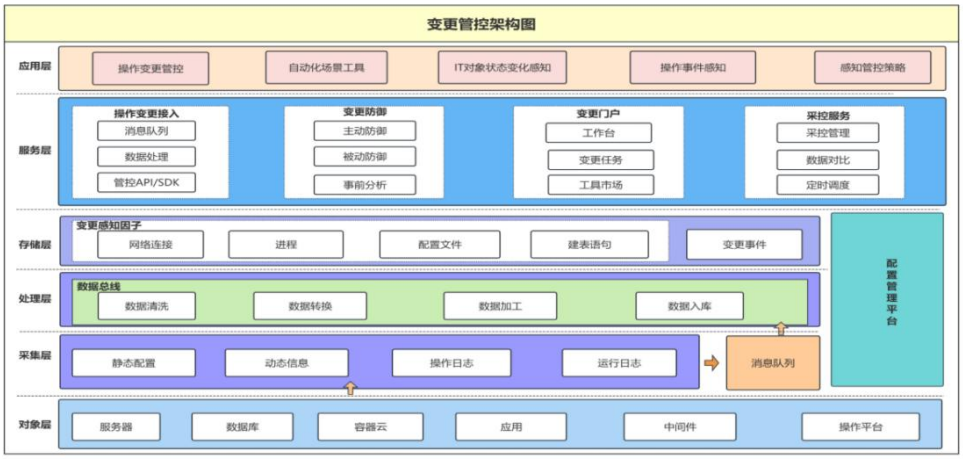
- **操作事件的全局采集：**

我们会把那些难以直接管控的操作平台，例如堡垒机以及各类 PaaS 平台产生的操作事件，全面地采集回来，汇聚到统一的数据库中。

- **变更感知因子的深度采集：**

针对主机、应用、容器云、数据库、中间件等各种核心变更对象，我们对其关键的变更感知因子进行细致的采集，例如配置变化、网络连接状态、进程启动停止等。

通过对这两类数据的全面采集，我们便为后续深入的变更感知分析奠定了坚实的基础，从而能够更好地识别、理解并应对系统中发生的各类非标准化变更。



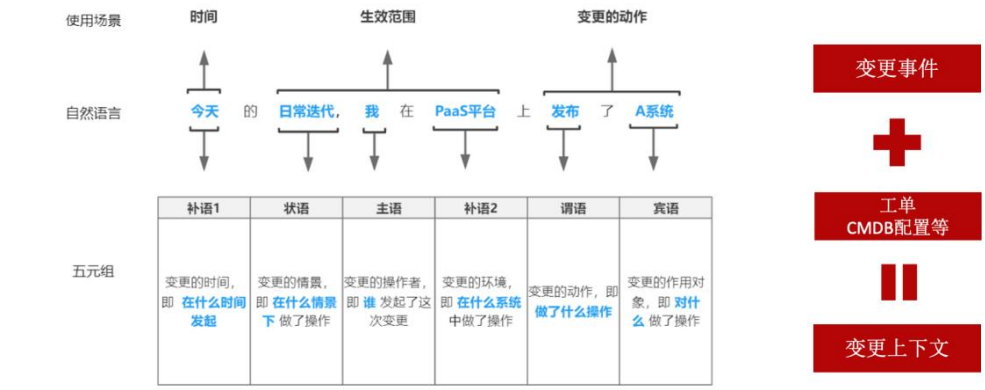
变更管控的系统架构如上图

- 对象层（底部）：最底层是我们的变更对象，包括服务器、数据库、容器云、应用、中间件以及各种操作平台。这些是整个系统要管控的核心资产。
- 采集层与处理层（中部）：再往上是数据的采集和处理。采集层负责获取各种数据，例如静态配置、动态信息、操作日志和运行日志等。这些采集到的数据会通过数据总线（处理层）进行清洗、转换、加工，最终入库。

- 服务层（中部）：在数据处理之上，我们根据上层应用的需求，对服务层进行了细致的微服务划分。这包括：
  - 操作变更接入：提供统一的变更入口，确保所有变更操作都经过规范的流程。
  - 变更防御：这是非常核心的部分，提供了主动防御、被动防御和事前分析等多种机制，在变更执行前、执行中、执行后进行风险识别和拦截。
  - 变更门户：为用户提供操作工作台、变更任务管理以及工具市场等功能，是用户与系统交互的主要界面。
  - 采控服务：负责采控管理的执行、数据的对比以及定时调度任务的管理，实现对变更对象的精细化控制。
- 应用层（顶部）：最上层是应用层，包含了多种应用能力，例如操作变更管控、自动化场景工具、以及对 IT 对象状态变化、操作事件的感知能力，并基于感知到的信息进行感知管控策略的执行。

整个架构通过清晰的分层和模块化的设计，实现了变更管控的全流程覆盖，从底层的对象到上层的应用，形成一个完整的、可观测和可防御的体系结构。

4、数字化变更管控的核心模型：变更事件



注：参考蚂蚁变更管控开源项目AlterShield（https://altershield.io）

作为变更管控的基础，我们对变更事件进行了标准化建模，这借鉴了蚂蚁开源的变更管控项目 AlterShield 的理念。一个变更事件通常包含以下核心要素：

- 变更时间：何时发起变更。
- 变更操作者：由谁执行变更。
- 变更情景：在何种场景或情景下进行操作。
- 变更环境：在哪个平台或系统中执行操作。
- 变更动作：具体的修改或操作是什么。

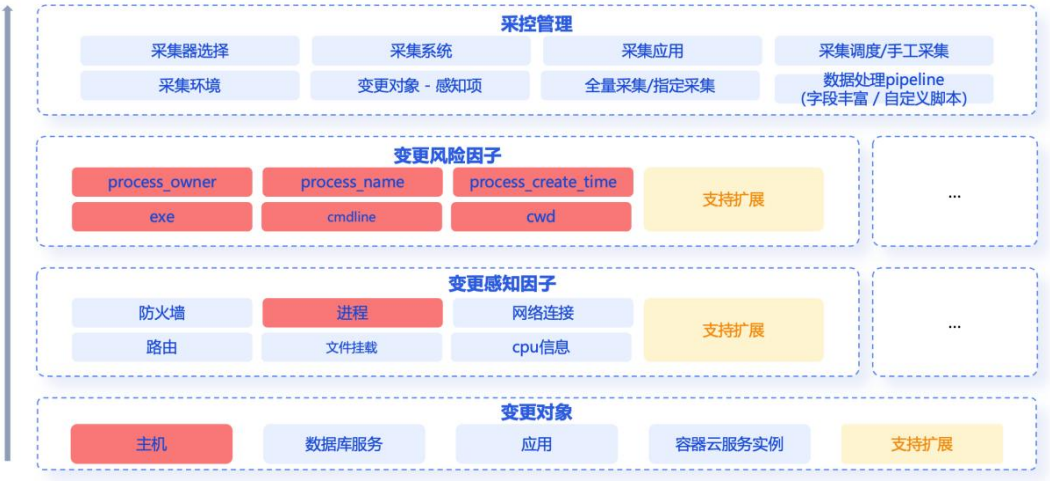
- 变更作用对象：变更影响的是什么实体。

我们将这些要素结构化，形成一个五元组来描述变更事件，例如，"今天的日常迭代，我在 PaaS 平台上发布了 A 系统"这句话，就可以拆解为具体的五元组信息：

- 补语 1 (变更的时间)：今天（在什么时间发起）
- 状语 (变更的情景)：日常迭代（在什么情景下做了操作）
- 主语 (变更的操作者)：我（谁发起了这次变更）
- 补语 2 (变更的环境)：PaaS 平台（在什么系统中做了操作）
- 谓语 (变更的动作)：发布（做了什么操作）
- 宾语 (变更的作用对象)：A 系统（对什么做了操作）

然而，单纯的变更事件信息不足以支撑全面的管控。我们将结构化的变更事件信息与工单、CMDB 配置等其他相关数据相结合，构建出一个完整的"变更上下文"。后续的变更管控、主动防御措施的应用，以及对变更事件的感知和分析，都将依赖于这个丰富的变更上下文。

5、风险来自于变更因子的变化



变更管控体系中的一个核心概念，是我们构建的"变更感知因子"模型。这里所谓"变更感知因子"，是指在执行变更操作后，能够反映系统关联 IT 对象部署和运行状态变化的那些关键因素。请注意，它不等同于我们日常监控的可观测指标，而是更侧重于捕捉系统特定状态的快照信息。

我们的目标是，将这些能够描述系统状态的"变更感知因子"与最终的可观测指标紧密关联起来。

以主机为例，我们会针对其上的多种感知因子进行精细化采集，例如：

- 防火墙规则
- 网络路由配置
- 进程运行状态信息
- 文件挂载点状态（NFS/NAS 盘挂载等）
- 网络连接状态
- 内核参数
- CPU 等硬件信息快照

当主机发生重启或经历了其他类型的变更后，这些因素的状态变化（比如动态路由的意外丢失或关键防火墙规则的缺失）都可能直接导致系统出现异常行为或潜在风险。

6、标准化变更操作工具

以下重点阐述如何通过标准化的工具实现变更流程的有效管理。

核心在于构建"能力原子化，标准化编排"的变更操作工具集，以应对日益复杂的系统变更。整个标准化变更流程被细致地划分为五个紧密衔接的关键阶段：

- **主动防御：**

这是变更流程的第一道防线，旨在前置识别和规避潜在风险。具体措施包括严格执行变更窗口检查、深度校验变更工单信息的准确性和完整性，并针对潜在的高危操作进行全面检查，确保变更在安全、可控的范围内进行。

- **前置准备：**

在正式执行变更前，进行充分的准备工作。这涵盖了向相关方进行变更通知、合理设置监控维护期以避免误报，以及采集变更对象的关键感知因子。同时，对变更前置条件进行细致确认，确保变更具备执行的基础和环境。

- **变更执行：**

此阶段是变更实施的核心，依赖于一系列标准化的变更工具和自动化流程，例如"变更工具 1"、"变更工具 2"或"自动化流程 3"等，通过这些工具高效、准确地完成既定的变更操作。

- **后置检查：**

变更执行完毕后，必须进行全面的验证。这包括及时关闭监控维护期，发送变更完成通知，并对变更后的感知因子进行采集和与变更前的状态进行比对，以确认变更是否达到了预期效果，并进行必要的技术检查。

- **变更复核：**

最后一步是进行最终的复核和确认。系统会发送复核待办通知，并由指定人员执行复核操作，确保变更的合规性和有效性得到最终确认。

通过上述标准化流程和工具的应用，我们旨在实现以下核心目标：

- 能力原子化：将复杂的变更操作分解为更小、更易于管理和组合的基本操作单元。
- 标准化编排：基于原子化的能力，通过标准化的流程引擎进行灵活、可靠的编排和组合。
- 主动防御：在变更发生前就构建起强大的风险识别和规避机制。
- 落实合规要求：严格遵循"双人复核"、"线上留痕"等合规性要求，确保变更操作的可追溯性和安全性。
- 工具共建共享：促进和鼓励各团队之间在变更工具上的协同构建和共享使用，提升整体效率和经验沉淀。

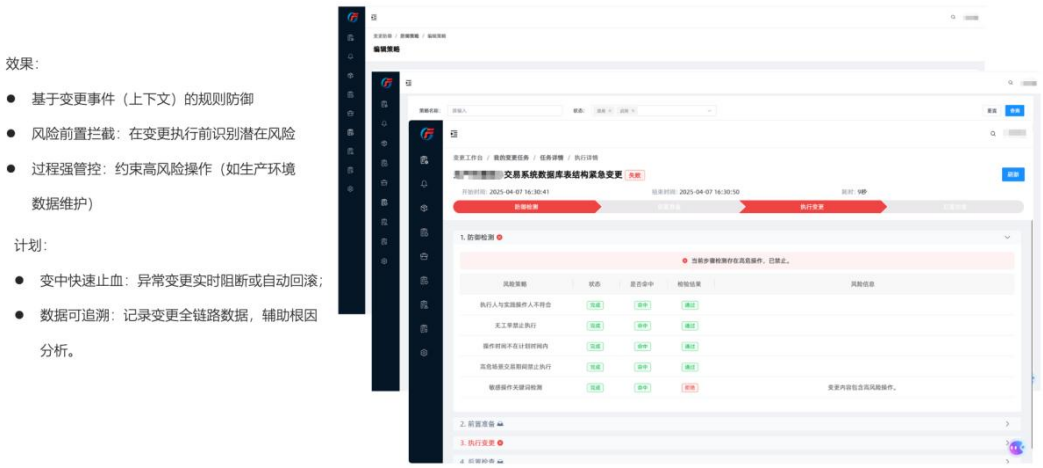
上图是主机标准化重启工具，其设计核心在于通过引入精细化的"变更感知因子"采集与比对机制，显著提升运维操作的安全性与可控性。

具体流程如下：

- 执行主机重启任务时，平台会在操作 前后 对目标主机的运行状态进行全面的"变更感知因子"自动化采集。
- 采集完成后，系统会智能生成一份详细的 比对报告。
- 这份报告能够清晰地展现主机在重启操作前后所发生的所有 细微变化，使得操作人员能够一目了然地掌握系统的变更详情。
- 通过这种方式，我们成功地构建了一道坚实的防线，能够有效地 防范 那些潜在的、由于主机重启本身或相关操作可能引入的各种 风险。

本质上，这个工具是将复杂且容易出错的手动比对过程自动化、标准化，赋予操作人员更强的洞察力，从而规避因信息盲区或遗漏导致的故障隐患，是我们在数智化故障管理体系中构建主动防御能力的一个具体实践。

7、构建主动防御的防控能力



作为 SRE 专家，我们深知变更操作是引发系统故障的最常见原因之一，因此，如何有效地管理和防御变更风险，是保障系统稳定性的重中之重。这张幻灯片的核心思想就是构建一套基于变更事件上下文的主动防御体系，将风险拦截前置，并对高风险操作进行强力管控。

核心目标：

- **基于变更事件（上下文）的规则防御：**

这意味着我们的防御策略不再是僵化的规则，而是能够根据具体的变更任务、执行环境以及系统上下文，动态地评估风险并触发相应的防御措施。

- **风险前置拦截：**

在变更执行前识别潜在风险,我们不再寄希望于问题出现后再去补救，而是在变更执行之前，通过自动化的风险检测机制，提前识别出可能导致故障的潜在因素，例如不符合规范的操作、与当前系统状态冲突的变更等，从而在源头阻止风险的扩散。

- **过程强管控：**

约束高风险操作（如生产环境数据维护）：特别是针对那些对生产环境有潜在巨大影响的高危操作，例如数据库的修改、关键配置的调整等，我们会施加更强的管控，确保只有经过充分评估和授权的变更才能执行，并且在执行过程中全程监控，杜绝人为失误。

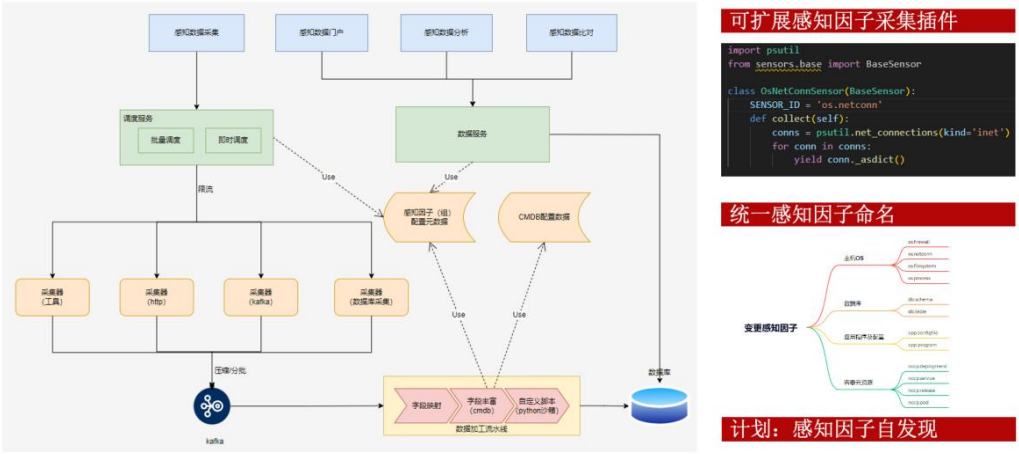
- 除此之外，这套主动防御体系还能带来以下实际效果：
- **变中快速止血：**

异常变更实时阻断或自动回滚：在变更执行过程中，如果系统检测到任何异常情况，如服务性能急剧下降、错误率飙升等，防御体系能够立刻触发停止变更进程（止血），或者自动将系统回滚到变更前的稳定状态，最大限度地减少故障对业务的影响范围和时间。

- **数据可追溯：**

记录变更全链路数据，辅助根因分析：系统会详尽地记录每一次变更从发起、审批、执行到最终结果的全过程数据，包括执行人、执行时间、具体操作、系统状态等。这些全面的数据链对于后续进行故障的根因分析（RCA）和复盘改进至关重要，帮助我们不断优化变更流程和防御策略。

8、实现变更因子的采控

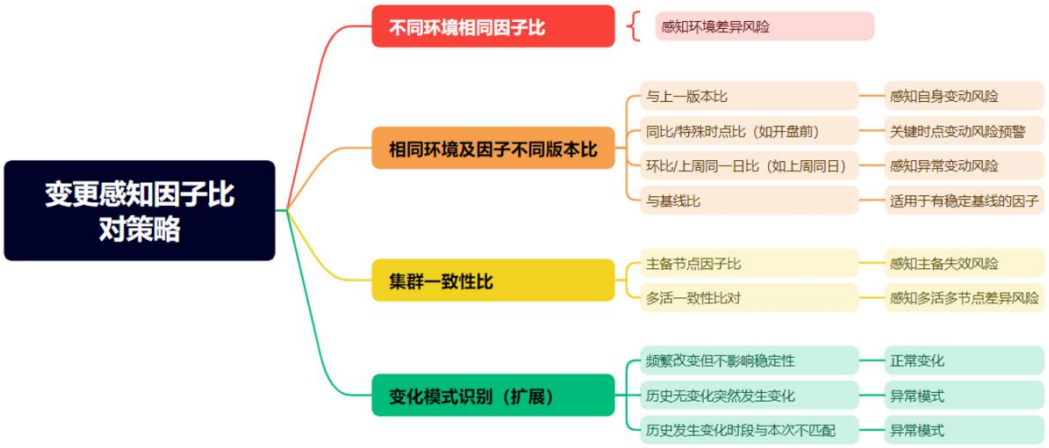


好的，我们这里聚焦的是"变更管控：变更感知采集逻辑"，核心在于构建一套能够高效、灵活采集变更数据的体系，以便我们能及时洞察系统中的细微变化，为后续的分析和决策提供坚实基础。

为了实现这一目标，我们专门开发了一套强大的采集框架：

- 通过插件的方式扩展采集能力，插件只需要关注核心采集逻辑，通过继承采集插件基类，并覆写采集方法，即可实现感知因子采集能力。
- 采集所涉及的调度、限流、压缩、分批上报等通用且复杂的逻辑，由采集框架统一处理。
- 为更好的支持感知因子的扩展和标准化，我们采取了统一命名方式，借鉴 opentelemetry 的命名规范，进行层次化的命名
- 未来，我们还将增加自动发现能力，发现并采集主机上常见的应用、中间件、数据库感知因子。

9、比不同：变更感知对比策略



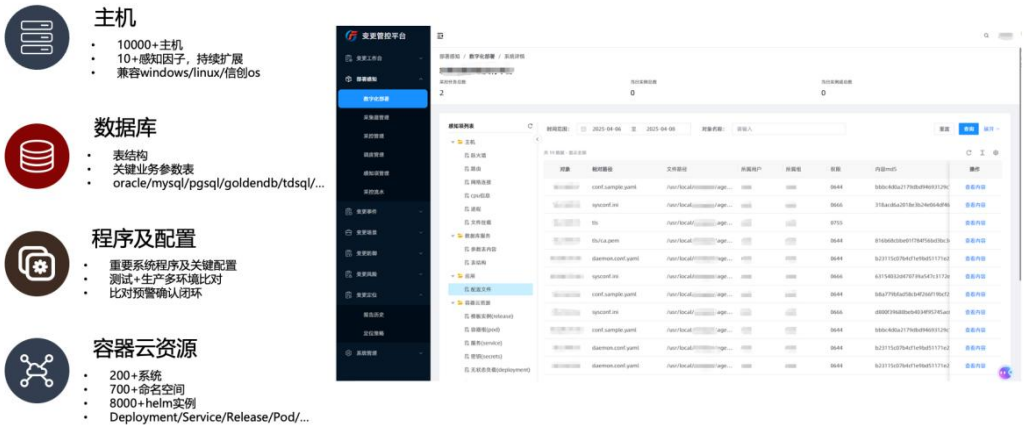
变更感知的核心在于感知因子的比对策略通过构建灵活、多维度、精细化的比对方式，主动感知并应对变更可能引入的风险。

这项策略主要包含以下几个关键的比对维度：

- 不同环境相同因子比对：这是一个基础但非常重要的维度，通过比较相同配置或状态因子在不同环境（例如测试环境与生产环境）中的差异，能够有效感知环境之间可能存在的配置差异带来的风险，及时发现那些部署或配置不一致的地方。
- 相同环境及因子不同版本比对：这是在同一个环境中进行更加细致的比较，用于感知因子自身的变动风险。我们采取了几种不同的比对方式：
- 与上一版本比对：这是最直接的方式，通过与最新的稳定版本进行对比，能够快速识别当前变更引入的变动风险。

- 同比/特殊时点比对：针对业务高峰或关键时点（例如交易系统的开盘前），通过与历史同期数据或特定时点数据进行比对，可以及时预警潜在的关键时点变动风险，确保系统在重要时刻的稳定性。
- 环比/上周同一日比对：这种周期性比对策略，例如与上周同一天的数据进行对比，能够帮助我们感知和识别异常的周期性变动模式，及时发现不符合常规规律的变化。
- 与基线比对：对于那些具有相对稳定基线的因子，通过与预设的稳定基线进行比对，能够有效识别偏离正常状态的变动。
- 集群一致性比对：针对我们广泛应用的集群化部署架构，确保集群内部的一致性至关重要，因此我们引入了集群一致性的比对：
- 主备节点因子比对：通过比较主备节点之间的关键因子，能够及时感知主备节点可能存在的配置不一致或状态差异，从而预警主备失效的风险。
- 多活一致性比对：在多活部署的复杂场景下，确保不同多活节点之间的数据和服务状态一致性非常关键，这项比对能够帮助我们感知多活节点之间的差异风险，保障多活架构的可靠性。
- 变化模式识别（扩展）：除了静态的比对，我们还引入了更高级的变化模式识别，通过分析历史数据，对当前的变化进行智能化判断：
- 如果某个因子历史表现为频繁变动但从未对系统稳定性产生影响，当前的变动可能被视为"正常变化"。
- 反之，如果一个因子历史从未发生变化，却突然发生了变动，这很可能是一个"异常模式"，需要重点关注。
- 同样，如果当前的变化时段与历史上的变化时段不匹配，也可能预示着"异常模式"的出现。

完成这些多维度的比对后，系统会自动生成详细的比对报告，并可以根据用户的订阅或配置，主动发送给相关的技术人员或团队。



基于变更因子采集框架，我们构建了数字化变更感知，针对主机、数据库、程序及配置以及容器云资源等变更对象，实现了感知因子多环境采集比对：

- **主机感知因子采集：**

覆盖超过 1 万台 主机，持续扩展感知因子种类。全面兼容 Windows、Linux 以及 信创操作系统 等多样化环境，为跨平台统一监控奠定基础。

- **数据库关键信息采集：**

深入到数据库层面，采集包括 表结构 和一些 关键业务参数表 在内的重要信息，以便及时发现与数据库相关的潜在问题。

- **程序与配置的采集及比对：**

对重要系统的 程序 和 关键配置 进行采集与比对。

不仅支持 测试 和 生产 等多环境间的差异比对，而且比对结果会进行 闭环确认，确保配置变更的合规性与安全性。

-

容器云资源采集与比对：**针对容器云环境，**

实现了对 200 多个系统、700 多个命名空间**的各类资源进行全面采集与比对，确保容器化应用的资源配置健康、一致且可控。**

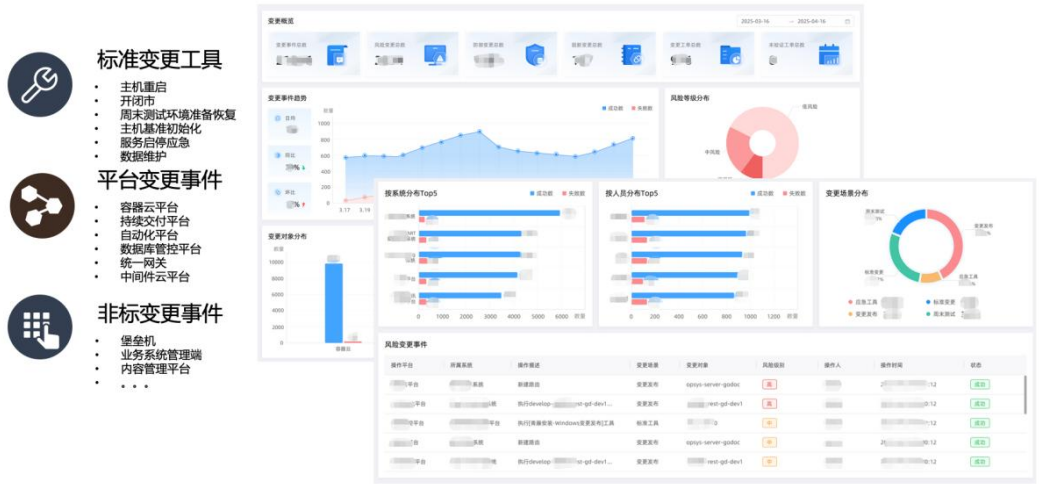
上图是数字化感知比对的案例截图：

关键配置文件比对：当发生变更操作，修改关键配置文件时，会将变动推送给负责人，由负责人二次确认，防止预期外变动。

主机感知因子比对：当主机层面发生路由、防火墙规则、进程状态等变化时，推送给负责人二次确认，防止因未持久化的变动失效导致的风险。

IP 访问关系构建：通过采集主机网络连接，构建主机间访问关系，为上层各应用场景（系统上下游关系构建、告警关联性分析等）提供数据支撑。

10、变更实践态势感知



作为数智化重塑证券公司故障管理体系的关键一环，我们针对变更事件构建了全面的态势感知能力，这对于主动预防和快速响应故障至关重要。这项能力主要通过统一采集和汇聚不同来源的变更事件来实现。

具体而言，我们对以下三类主要的变更事件进行了整合：

- 标准变更工具产生的事件：指的是变更管控平台上统一管理的标准化的变更流程和工具产生的事件，例如自动化的部署、配置更新等。这些事件通常结构化程度高，易于解析和处理。
- 变更操作平台的事件：这是指我司各类变更操作平台中记录的各种操作行为和状态变化。这类操作通常通过平台间对接即可实现采集，包括容器云平台、持续交付平台、数据库管理平台等。
- 非标准变更事件：指的通过堡垒机或者直接登录主机或者业务系统自带管理后台进行操作的变更事件，通常这类事件采集和接入的难度、成本比较高，我们将这些事件的采集优先级放在最后，完成前面 2 类操作事件采集覆盖后逐步采集接入。通过推广标准变更工具，逐步减少这类变更方式的占比。

通过将这些多样化的、来源于不同系统的变更事件进行统一的采集和汇聚，我们得以建立起对整个变更生命周期的全面视图。这种整合能力是我们能够更深入地理解变更与潜在故障之间的关联，从而实现对变更事件态势的实时感知和分析，为构建主动防御体系奠定坚实基础。

11、场景驱动的变更管控

变更验证待办

集结通知

关键时刻重要巡检推送

为了确保变更能够安全、高效地进行并顺利落地，我们依托运维平台打造了一系列场景化的工具，它们如同一个个前沿阵地，在变更过程的不同阶段提供坚实支撑。

- **变更验证表：**

变更实施完毕后，系统将自动编制一份详尽的变更验证任务清单，并向所有相关利益相关者进行推送。此机制确保了每项验证工作均有人专门负责，并且有明确的记录可查，从而形成一个严密的验证循环，显著减少了因验证不足可能引发的风险。

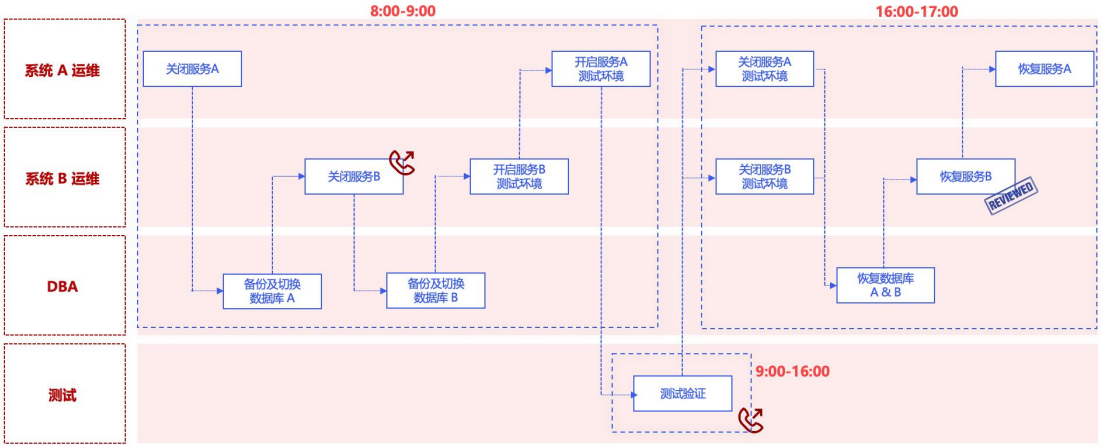
- **集结通知：**

变更部署完成后，平台会自动触发拉群动作，并将所有变更相关的干系人拉入一个群聊，发送集结通知。这确保了在变更后的关键时期，所有相关人员都能迅速集结，确保在第一时间对可能出现的问题进行响应和处置。

- **巡检推送：**

在变更完成后的关键业务时点或周期性检查点，系统会自动执行预设的巡检任务，并将巡检结果以消息推送的形式发送给运维人员。这些推送内容高度聚焦于系统和数据的关键状态，让运维团队能够实时、清晰地掌握变更后的系统运行状况。

12、大规模团队变更时序管控



券商行业的全网测试，通常是一个涉及几百人的大型变更场景，包括了运维/测试/研发/业务等多个团队的协作，我们针对大规模变更场景，开发了时序工具，确保任务之间的时序依赖，高效推进整个变更过程，任务节点支持对接自动化平台，自动执行一些像环境准备和恢复的例行性的操作，减少操作风险。

时序配置汇总

个人任务列表

逾期升级督办

往下我们谈一下应急指挥体系

三、最后一道防线：应急指挥

1、应急指挥总体思路

即使我们在研发流程前端左移运维责任，并辅以严格的变更管理机制，系统异常的发生仍然是一个无法完全避免的现实。

因此，应急指挥体系成为我们保障系统稳定性的最后一道也是至关重要的防线。

应急指挥目前已经在公司各个条线形成了一个应急的协同文化，出现故障时不仅系统相关的运维、研发、测试能够快速参与到应急，上下游系统也能够按需订阅故障实时处置的信息。

为了最大限度地降低异常事件的影响，我们持续投入资源，不断完善应急保障体系，其构建核心围绕以下关键要素展开：

- **应急保障体系：**

构建坚实可靠的框架，确保在突发状况下，能够迅速、有序地调动资源、明确职责。

- **一站式应急总线：**

打造统一的平台，整合各类告警信号、监控数据和处理工具，实现信息的高效汇聚与流转，避免信息孤岛和人工传达的延迟与错误。

- **事件处理流程：**

建立清晰、可执行的标准化流程，从事件发现、定级、处理、恢复到复盘，每一步都有章可循，减少混乱和不确定性。

- **机器人应急协同：**

应急指挥是由一系列机器人（监控响应升级、故障应急、值班经理、智能影响分析等机器人）辅助建立一个跨团队的应急调度，确保所有的故障都能高效的运作。

- **数字化应急预案：**

将预案电子化、结构化，方便快速检索、更新和执行。这些预案是团队在面对常见或复杂故障时的行动指南，能够显著提升应急响应的速度和有效性。

- **智能化应急：**

将应急指挥与大模型、智能体、知识库关联，实现智能化的应急。

- **信息共享：**

让不同的人能够在一个平台上共享数据，支持按需获取应急信息。

2、ECC 运营指挥中心

IT运营指挥中心（ECC）

- 主要包括故障监测、应急指挥、数据运营、重大演练等核心职能
- OnCall 人员的值守、应急作战指挥室的线下场所

IT 运营指挥中心（ECC）作为应急指挥体系的关键构成要素，同时承担着线下场所与应急核心职能，在应急指挥体系中扮演着至关重要的角色。

如左侧星型图所示，IT 运营指挥中心（ECC）是整个应急保障体系的枢纽，围绕着它，辐射出多个关键职能和协作方：

- **OnCall 值班经理 & 一线：**

作为值班体系的前沿，负责初步响应与处理。

- **故障发现：**

负责通过各类监控手段及时感知系统异常。

- **应急响应 作战室：**

应急事件发生时，快速集结相关人员进行协同处置。

- **故障定界 & 定位：**

通过分析与排查，明确故障影响范围及根本原因。

- **应急处置 决策：**

在获取故障信息后，迅速制定并执行恢复策略。

- **重大保障 集结：**

在关键时期或发生重大事件时，集中力量进行重点保障。

ECC 其自身具备多项核心职能，是我们在故障、数据运营、重大演练以及应急指挥中的关键抓手：

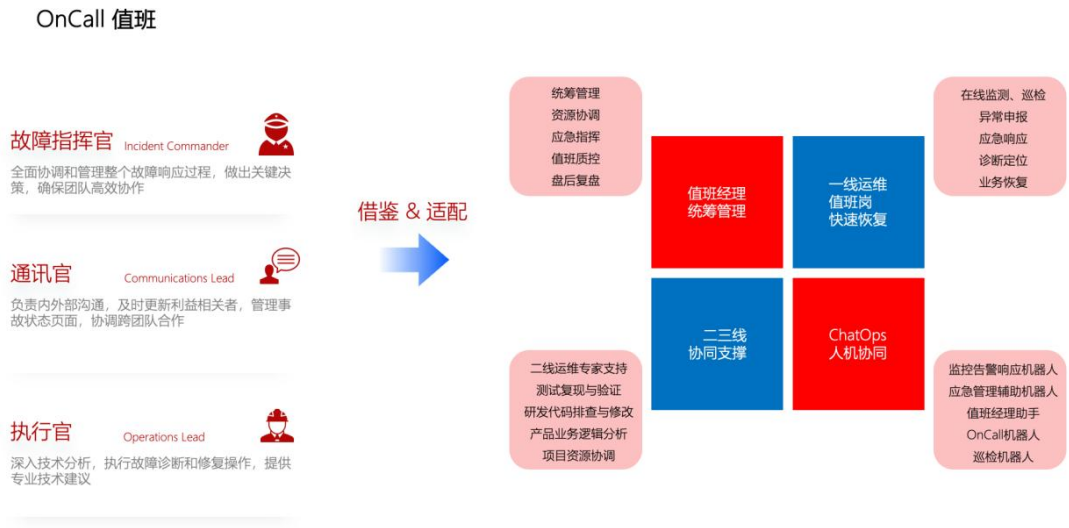
- 值班保障：每日组织一线人员值班保障。
- 故障监测： 负责对各类系统、服务进行不间断的监控。

- 应急指挥：作为事件处置的统一指挥中枢，统筹协调。
- 数据运营：负责收集、分析与展示各类运营数据，为决策提供支持。
- 重大演练：组织和执行各类应急演练，提升团队的应急响应能力。

ECC 不仅仅是一个虚拟的概念，更是一个具有线下场所的实体，它是我们 OnCall 人员值守以及应急作战指挥的基地，正如右侧图片直观呈现的监控指挥中心场景。

3、一二三线协同的 onCall

我们参考 Google SRE oncall 机制，在运维保障体系中设定了分工明确的保障角色，例如值班经理，其职责涵盖统筹管理、资源协调和应急指挥，是保障高效响应的关键角色之一。



我们将整个应急保障体系划分为清晰的职责和协作层级：

- **一线运维值班岗：**

这是我们快速响应的第一道防线，肩负着异常申报、通过在线监测和巡检及时发现问题，并进行应急响应、诊断定位和快速恢复业务的关键使命。

- **二三线协同支撑：**

当问题需要更深入的技术分析和根治时，会激活二三线的协同体系。这包括二线运维专家的技术支援，还有测试团队对问题的复现和验证，以及研发团队进行代码排查和必要的 Bug 修复与代码修改。此外，还可能涉及到产品团队进行业务逻辑分析以及跨项目的资源协调。

为了大幅提升应急响应的效率和自动化能力，我们在平台层面实现了 **ChatOps 人机协同 的能力**

。这意味着在故障应急过程的各个关键环节，都会有智能化机器人提供强大的辅助支撑。比如：

- **监控告警响应机器人：**

自动化处理告警信息并触发初步响应动作。

- **应急管理辅助机器人：**

协助管理事故状态、流程推进和信息同步。

- **值班经理助手：**

为值班经理提供数据统计、状态汇总和任务提醒等功能。

- **OnCall 机器人：**

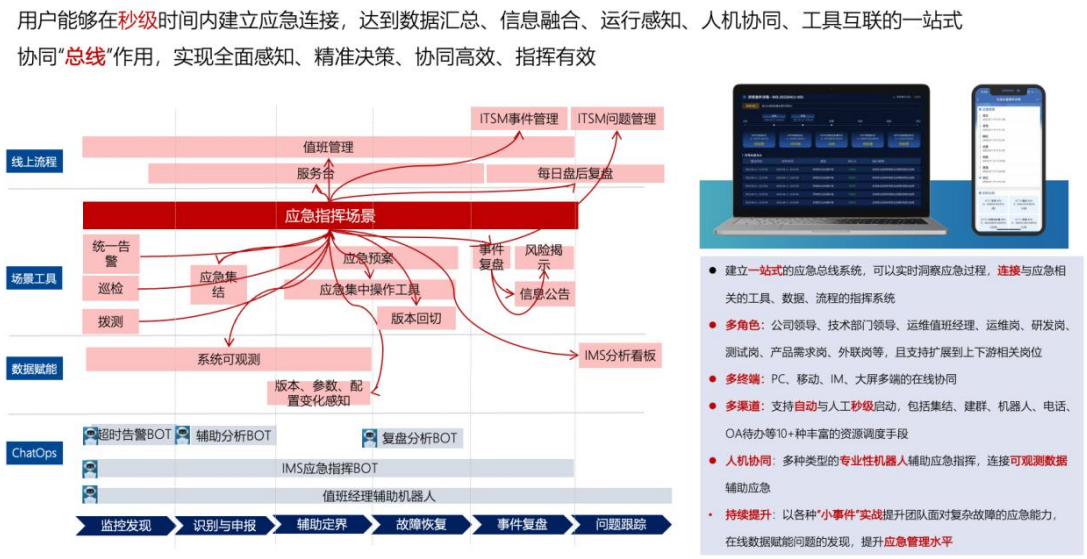
协助 OnCall 值班人员进行故障诊断和操作执行。

- **巡检机器人：**

自动化执行例行检查任务，提前发现潜在隐患。

整个体系的设计，既有清晰的人员职责划分，又有强大自动化工具的支撑，旨在确保在面临生产环境问题时，能够实现快速准确的发现、高效 的协同以及迅速有效的恢复，最大化地保障业务的连续性和稳定性。

4、人机协同的应急的总线



应急总线的核心目标在于：打破信息孤岛，将与应急相关的各类工具、数据和流程无缝连接起来，形成一个能够实时洞察应急全过程的统一平台，确保在故障或突发事件发生时，能够以秒级的速度建立应急连接。

为了达成这一目标，我们构建的应急总线具备以下关键特性：

- **一站式接入与实时感知：**

系统汇集了分散在各处的应急工具、数据和业务流程，构建了一个统一的入口，从而能够实时、全面地感知应急事件的进展，消除信息不对称带来的沟通和协作障碍。

- **多角色广泛协同：**

系统天然支持公司领导、技术领导、运维值班经理、各级运维、研发、测试、产品需求及外联等广泛角色的在线协同，并具备向下游相关岗位扩展的能力，确保应急事件处理能够囊括所有相关方。

- **多终端无缝接入：**

通过提供 PC、移动、IM、大屏等多种终端支持，系统保证了无论值班人员或决策者身在何处，都能第一时间通过熟悉的设备接入应急流程，确保信息的秒级同步。

- **多样化秒级启动：**

无论是通过自动化规则触发，还是由人工发起，系统都能在秒级内启动应急流程，并提供十余种资源调度方式，包括自动集结人员、创建 IM 群组、调用专用机器人、进行电话通知、触发 OA 待办等，确保应急响应的快速启动和人员到位。

- **人机协同智能赋能：**

系统集成了多种专业化机器人，这些机器人能够与可观测数据深度集成，在应急指挥过程中提供辅助分析、智能判断和自动化执行能力，显著减轻人工负担，提升决策的精准性和效率。

同时，我们强调通过日常的"小事件"实战演练来持续提升团队处理复杂故障的能力，并利用系统沉淀的在线数据赋能问题发现与根因分析，从而实现应急管理水平的不断优化与提升。

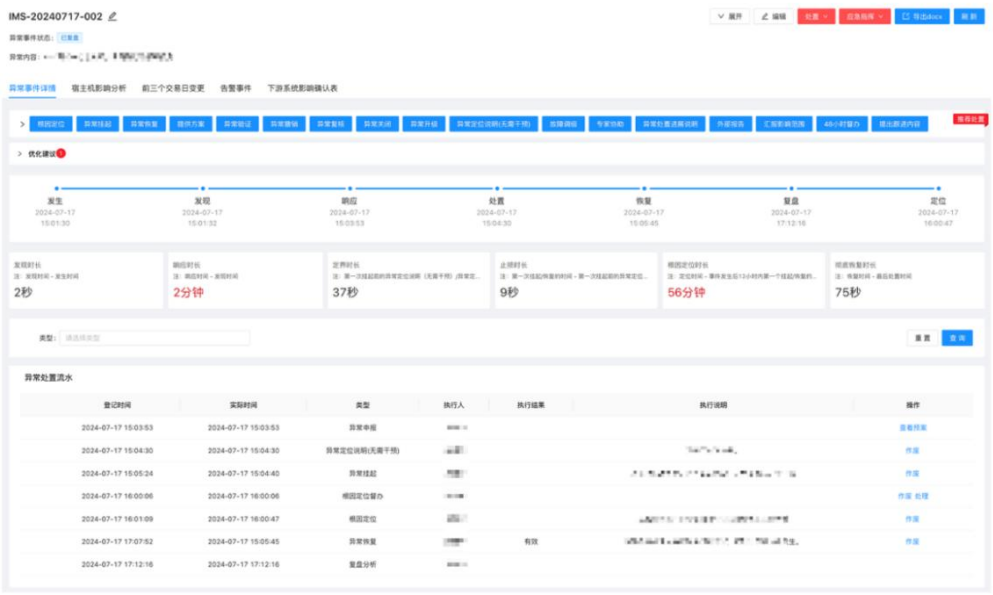
结合幻灯片左侧的流程图，我们可以更清晰地看到这个应急总线是如何将各个环节串联起来的：

- 以应急指挥场景为中心枢纽。
- 线上流程中的值班管理和服务台与 ITSM 流程紧密集成，保证事件处理的规范性。
- 统一告警、巡检、拨测等场景工具直接为应急指挥提供信息和执行能力。
- 系统可观测、版本、参数、配置变化感知等数据源为决策和根因分析提供关键数据支持。

- ChatOps 能力通过各类机器人辅助应急流程的各个阶段，从告警发现到复盘分析，实现基于 IM 的高效协同和自动化操作。

正是通过这样的体系构建，我们旨在打造一个能够快速响应、智能辅助、协同高效的一站式应急总线平台，为业务的稳定运行提供坚实保障。

5、实时在线感知应急过程



最终，所有的故障处理实践与方法都将汇集并落地于我们标准化的事件处理流程之中。这一流程体系的设计，旨在确保在面对任何突发状况时，我们都能遵循预设的、高效的步骤，从而最大程度地减少故障影响，保障业务的连续性与系统的稳定性。

6、数字化应急预案

为了有效地提升故障应急处置的效率，我们打造了数字化应急预案工具。

该工具围绕两个核心理念构建：

- **原子化：**

将复杂的应急流程拆解为独立、可复用的"原子预案"。这种模块化的设计极大简化了预案的编排、更新以及与自动化的关联，提升了整体的灵活性和效率。

- **数字化：**

通过平台化实现预案的可编排、可演练、可统计、可遍历、可自愈能力，为预案的持续优化和扩展奠定基础。数字化平台不仅能够轻松组合原子预案形成完整的应急流程，还能支持模拟演练以验证其有效性，对历史事件和执行情况进行统计分析以指导改进，方便追溯整体流程，并最终目标是实现系统异常时的自动触发和恢复。

如图所示，一个典型的数字化应急处置流程被清晰地呈现。流程按照阶段划分，从左向右对不同服务/组件的应急处置进行细化描述。

具体而言，流程自上而下推进：

- **判断阶段：**

在此阶段，针对服务 A、组件 B、服务 C，首先进行快速判断是否能执行处置动作，并辅以链接和确认表来获取信息和进行决策。

- **处置阶段：**

识别出可行的处置方向后，便进入具体的处置步骤。该阶段包含处置动作编排，然后执行针对不同服务/组件的具体预案：

- 服务 A 执行应急卡片 A01 和应急卡片 A02。
- 组件 B 执行自动化原子预案。
- 服务 C 执行主机异常策略。
- 验证阶段: 处置完成后，流程进入整体的验证阶段，以确认修复效果。

通过将应急预案进行数字化和原子化，我们旨在构建一个更高效、更具操作性且可持续改进的应急响应体系。

针对突发事件的应急响应，我们的核心策略是构建可高效执行的数字化应急预案，并强调其"原子化"与"数字化"特性。

原子化

的理念在于将复杂的应急处置流程拆解成具备复用能力的最小单元。这样一来，无论是针对特定故障场景进行预案编排、后续的更新迭代，还是与自动化系统进行无缝对接，都能变得异常灵活和便捷。

数字化

则赋予了预案一系列关键能力：

- **可编排：**

能够根据具体事件、受影响的服务或组件，像是服务 A、组件 B、服务 C 等，灵活地组合和调整“处置动作编排”，比如调用“应急卡片 A01”、“自动化原子预案”或“主机异常策略”。*

- **可演练：**

允许我们在非生产环境中模拟执行预案流程，验证其有效性，从而在真正面临故障时游刃有余。

- **可统计：**

能够记录应急处理的全过程数据，包括每个步骤的耗时、执行结果等，为后续的复盘分析和持续优化提供数据支撑。

- **可遍历：**

整个应急处置流程，从“判断”、“处置”到“验证”三个阶段，以及在每个阶段中针对不同服务的详细步骤和验证项（如“黄金指标”、“日志输出”、“业务验证”、“下游确认”），都可以清晰地被追踪和理解，毫无盲区。

- **可自愈：**

通过与自动化能力的结合，理想状态下，数字化预案能够触发一系列自动化操作，帮助系统在一定程度上实现自我修复。

- **可持续扩展：**

随着业务和系统的演进，应急预案可以轻松地进行调整和改进，以适应新的需求和场景。

如幻灯片所示，数字化应急预案的流程清晰地展示了从判断故障是否符合处置条件（通过"链接"、"确认表"等原子预案或工具快速辅助）到执行处置动作，最终再对处置效果进行全面"验证"的过程。其核心价值在于，在故障发生时，我们可以基于预先编排好的、由各类原子化能力（如"链接"、"确认表"、"应用异常策略"、"主机异常策略"等）组成的数字化流程，一键启动应急响应，快速完成故障定界、处置，并最终验证处置的有效性，大幅提升应急效率和准确性。

四、智能化与场景融合

智能化如何与运维场景深度融合，我们将围绕以下三个核心点展开：

- **广发证券对智能运维的理解：**

我们将深入探讨广发证券在智能运维领域的认知框架，包括企业对智能运维的定义、期望达成的目标、目前所处的阶段以及对智能运维价值的深刻洞察。

- **运维平台智能化升级思路：**

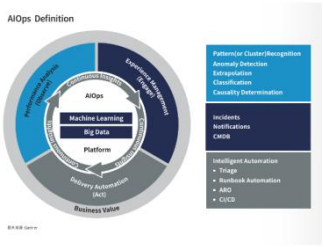
详细剖析我们将如何对现有运维平台进行智能化改造与升级，这涉及架构层面的优化、引入智能化组件的设计思路以及技术选型的考量，旨在构建一个更智能、更高效的运维支撑体系。

- **案例分享：**

通过具体的实践案例，生动展示智能运维在实际运维场景中的应用效果，用真实世界的经验来阐释前述理念和升级思路。

1、大模型给智能运维带来新的突破

Gartner提出了AIOps的理念，定义为：AIOps平台的人工智能将大数据，AI 机器学习和其他技术相结合，通过主动，个性化和动态的洞察力支持所有主要IT运营功能。AIOps平台支持同时使用多个数据源，数据收集方法，分析技术和可视化等技术。
算法式的智能运维主要应用在指标与日志数据分析，仅适用于部分应用场景。随着大模型的兴起，新增了**生成式智能运维发展路线**，形成“**传统智能算法+专家经验+平台智能体+生成式算法**”相结合的人机协同的智能运维发展。



Gartner 早在多年前就提出了 AIOps 的概念，即人工智能驱动的 IT 运营，其核心是通过大数据、机器学习和 AI 技术，为 IT 运营提供持续、个性化和动态的洞察，从而支持性能分析、体验管理以及自动化操作。AIOps 平台通常具备多数据源整合、数据采集、分析和可视化等能力，旨在实现 IT 运营的智能化和自动化。

然而，传统的基于算法的智能运维并非没有局限。它主要依赖于预设算法处理指标和日志数据，应用场景相对有限。

但随着大模型的异军突起，我们惊喜地看到，智能运维领域出现了一条全新的发展路径——生成式智能运维。

未来的智能运维发展模式，将是**"传统智能算法 + 专家经验 + 平台智能体 + 生成式算法"深度融合的人机协同模式**。这种模式将集各家所长，极大地提升运维效率与准确性。

左侧的图表清晰地展示了 AIOps 的核心构成及其主要功能领域。它是一个多层结构的环，中心是 AIOps 本身，内层是支撑技术——机器学习、大数据和平台。中间层强调持续洞察（Continuous Insights）是贯穿始终的流程。而最外层则勾勒出了 AIOps 所覆盖的主要 IT 运营职能：

- 性能分析 (Observe): 利用模式识别、异常检测、分类、因果确定等技术进行深入洞察。
- 体验管理 (Engage): 涉及事件处理、通知、CMDB 等，关注用户和业务体验。
- 交付自动化 (Act): 通过智能自动化实现故障分类分级、运行手册自动化、自动修复、CI/CD 等，提升运营效率。

右侧的表格则对传统的基于半监督算法式智能运维与新兴的生成式 AI 智能运维进行了直观对比，突出了生成式 AI 带来的范式转变：

对比项目	传统基于半监督的算法式智能运维	生成式 AI 智能运维
工作原理	用少量标注和大量未标注数据建模型	大模型先学再调，找数据隐藏规律
学习数据	主要用指标数据，少量人工标注	融合各种数据，啥都能学
学习能力	用未标注数据，适应新场景弱	大数据学习，适应新场景强
优势	标注少也能训练，算法不复杂	诊断疑难故障，给出周全方案
局限	依赖人工标注，新问题难处理	训练费资源，不知咋做的决策
主要应用场景	数据稳定、故障常见的情况	复杂系统和要求高的运维场景
对于人机协同模式的帮助	帮着核对处理，完善标注数据	给分析建议，和人配合超

总结来看，生成式 AI 展现出显著的优势，但传统模式依然有用武之地。未来的智能运维将是两者的协同发展，并通过将专家经验和平台能力融于一体，最终构建起图中间模拟对话界面所展示的，一个能够与人流畅交互、提供智能协助的新型运维体系。正因如此，随着大模型的兴起，智能运维正朝着一个以传统智能算法、专家经验、平台智能体以及生成式算法深度结合的人机协同方向加速演进。

2、我们对智能运维的理解

那我们对智能运维的理解是打造智能化的人机协同运维体系，其核心在于：

- 通过智能技术与专家经验的沉淀，构建具有专业技能的机器人。
- 辅助运维专家进行系统的感知、决策与执行。

这也就意味着，智能运维需要持续提升以下六大核心能力：

- **实时获得"发生了什么？"：**

快速感知系统当前状态。

- **关联分析"为什么会发生？"：**

深入探究问题根本原因。

- **智能预测"将会发生什么？"：**

预判未来潜在风险。

- **决策判断"采取什么措施？"：**

根据分析结果制定应对策略。

- **自动执行"如何快速执行？"：**

实现自动化操作与响应。

- **实时感知"工作执行的效果？"：**

对执行结果进行反馈与评估。

而实现"人机协同"的关键，在于有效结合运维专家与机器人的特点，并通过"数据+平台"进行管理。这包括：

- 沉淀专家经验，赋能机器人：将运维专家的宝贵知识和经验转化为可复用的、自动化的处理逻辑。
- 以"数据+算法"赋能运维专家：利用数据和算法能力增强运维专家的实时感知与辅助决策能力。
- 增加运维机器人岗位：通过机器人承担或重塑操作性、规律性、大计算、7x24 小时运行以及提升人机体验等工作类型。
- 建立平台化管理：通过统一平台实现对整个智能运维流程的协同与自动化执行，确保决策的有效落地。

总之，智能运维旨在通过人机协同，利用智能技术提升运维体系的感知、分析、预测、决策和执行能力，从而更高效、更可靠地保障系统的稳定运行。

3、智能运维能力建设方案

智能运维建设的整体思路，概括来讲，我们构建了"四个中心"的建设体系，旨在整合现有资源，实现运维能力和服务智能化升级。

首先，我们强调"以场景驱动"，将现有运维平台的能力转化为"运维 Agent 智能体能力中心"。这就像是把运维平台的各种"服务"打包成一个个智能化的工具箱，能够快速响应不同的运维场景需求，并最终落地到公司统一的智能体编排平台。

其次，我们将现有的智能算法能力，例如异常检测、根因分析、风险预测等，进行统一封装和整合，形成"智能算法中心"。这个中心将作为智能运维的大脑，为各种智能应用场景提供技术支撑。

再者，我们对现有庞大的机器人和聊天群生态进行智能化升级，希望为这些工具"插上翅膀"。通过关联知识库，实现智能化的问答（比如使用 RAG 问答技术），并为它们绑定智能体，让用户可以直接通过机器人与智能体进行交互。这不仅提升了现有工具的智能化水平，也鼓励并赋能了员工自主构建更多智能机器人。

最后，也是至关重要的，我们致力于激发全体运维专家沉淀宝贵的运维经验，以此构建一个"智能专家知识中心"。这个知识中心是智能化的基石，它不仅仅是文档和数据的集合，更是经过训练和提炼的、可供智能化应用调用的运维智慧库。例如，我们将各类运维领域的原始数据，包括厂商文档、API 文档、技术文档以及系统运行数据，都纳入训练范畴，确保知识的全面性和时效性。

智能运维建设

为了实现运维的智能化转型，我们在既有的运维平台基础上进行了核心能力的增强与重塑。

首先，我们构建了一个全局智能助手，它的强大之处在于：

- 能够动态加载和启用各类智能体。
- 具备强大的上下文感知能力，能理解当前页面的状态与用户的操作意图。
- 能够集成我们的运维知识库，提供智能问答服务。
- 支持将多个智能体的工作流程进行编排，实现复杂任务的自动化执行。

其次，我们落地了"无代码 AI + ChatOps" 的新范式，让运维人员能够更便捷地利用智能化能力，便捷地维护与更新运维知识库，为一线团队提供最新信息赋能。

- 在日常的企业微信群聊中，可以直接基于知识库进行智能问答，将知识转化为即时支持。
- 通过采集分析企业微信的聊天记录，系统能够感知到聊天的上下文，提供更加精准和智能的帮助。

最后，我们将所有智能体进行了统一集成和管控，这不仅能最大化工具的复用效率，也使得上层敏捷场景能够灵活地复用这些智能体能力，加速业务创新与问题的解决。

4、大模型应用案例：AI 辅助故障定位案例

AI 辅助故障定位系统，旨在利用大型模型（AI）的能力，在系统故障发生时，为 SRE 和运维人员提供智能化的支持，从而加快故障的定位和解决速度。

图示界面清晰地划分了两大区域：左侧是智能定位辅助信息模块，右侧则是故障详情与智能分析的核心交互区。

在左侧的智能定位辅助信息区域，系统会根据发生的具体事件（如 IMS-2025*-006. 和受影响的系统，推送一系列有价值的辅助信息，包括：

- 前三个交易日变更情况：列出近期可能与故障相关的系统变更（如 CAB-20250307-001, CAB-20250313-017. ，帮助快速锁定导致故障的潜在变更源。
- 应急中心可观测感知：展示应急中心层面汇集的系统异常感知，例如系统健康度（当前为 97%），并提供链接供进一步查看详细的可观测数据。
- 统一告警事件：汇总与当前故障关联的统一告警信息，如近 2 小时的告警数量（当前显示 6 条），便于了解故障范围和影响。
- 历史相似事件：智能推荐与当前故障现象或类型相似的历史事件（如 IMS-20220418-004, IMS-20220916-006. ，为排查提供宝贵的历史经验参考。

- 推荐应急策略：基于大模型的分析，系统会提供针对性的应急处理策略建议（例如针对"对账单发送失败"、"中登报盘服务器故障"等具体场景的建议）。特别需要指出的是，这些建议由大模型筛选生成，使用时需要人工核对确认其有效性与适用性。

- 右侧区域是故障详情和核心的 AI 排障智能体交互区。这里详细展示了故障的原始描述（如"上周发版的服务连接出现'Unknown prepared statement handler (5) given to mysql_stmt_precheck'"），并允许用户或系统向 AI 发起排查请求（例如询问"这个故障怎么排查？"）。

- 排障智能体收到请求后，会基于对故障描述和相关信息的综合分析，提供初步的故障原因分析和定位建议。以"Unknown prepared statement handler (5) given to mysql_stmt_precheck"错误为例，AI 给出了以下四种可能的深层原因分析和解释：

- SQL 语句预处理问题：详细阐述该报错通常指向预处理语句的状态异常，可能因其被提前关闭或初始化不当所致。

- 数据库连接问题：分析数据库连接层面的问题也可能导致预处理语句执行失败，诸如连接池配置错误、连接超时或非正常连接关闭等都可能是原因。

- 代码逻辑问题：指出应用层代码中可能存在的逻辑缺陷，例如预处理语句的创建和执行流程不符合规范或顺序错误，会影响其正常处理。

- 数据库版本兼容性问题：考虑到近期可能的数据库版本变动，AI 提出了版本不兼容的可能性，认为新版本与现有预处理语句机制可能存在冲突，导致工作异常。

- 通过这样一个 AI 辅助系统，我们将故障发生后的被动响应，提升至了 AI 驱动的智能分析与主动建议模式，极大地增强了故障定位的效率和准确性，让运维团队能够更快速、更精准地解决问题，最大程度地降低故障对业务的影响。

5、大模型应用案例：异常变更定位助手

变更查询
异常系统及上下游系统
近期变更情况

变更操作事件查询
异常系统及上下游系统发
生过哪些（风险）操作

部署状态感知
异常系统及上下游系统各变
更感知因子是否发生变化

异常变更定位
哪些变更、操作事件、变更
感知因子变化导致异常

借助之前变更管控系统沉淀的变更信息、变更对象变动数据以及操作事件记录，我们构建了一个异常变更定位助手。

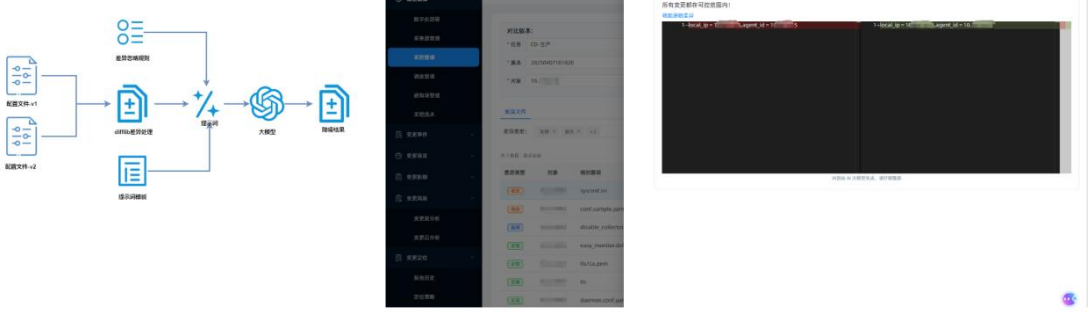
通过这个助手，我们可以便捷地查询异常系统及其上下游系统近期的变更情况。

更进一步，我们能够结合这些数据，利用大模型的推理能力，协助快速定位是哪些具体的变更或变化导致了系统的异常。

在实践中，我们采用的是千问的 QwQ 推理模型，它的推理能力令人印象深刻，能够有效地识别不同变更之间的时间关联性，以及它们与上下游系统之间的复杂关联关系，从而提升异常变更定位的效率和准确性。

6、大模型应用案例：感知采集对比一键降噪

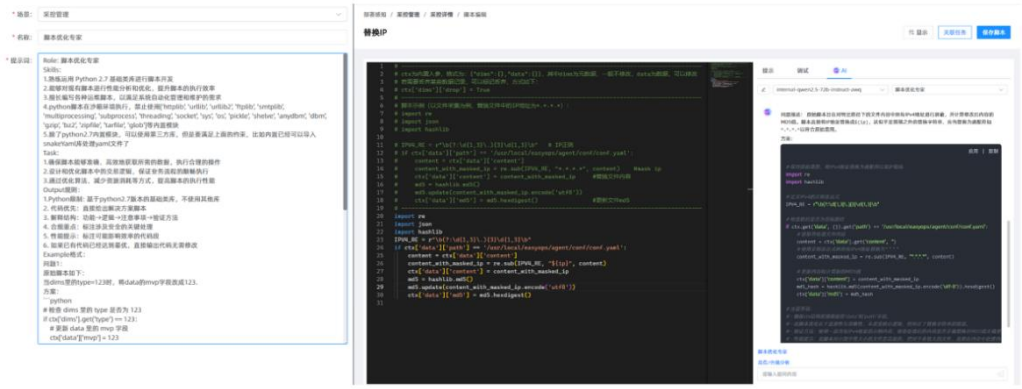
大模型应用案例：感知采集比对一键降噪



在构建感知采集体系时，我们面临着一个关键挑战：如何在不同环境（例如生产环境与测试环境）之间进行比对时，有效处理那些确定已知、但因环境差异而必然不同的信息，比如 IP 地址或密码。这些差异信息会产生"噪音"，干扰我们对真正问题的判断和定位。

为了解决这个问题，我们专门利用大模型的强大能力，开发了一个降噪功能。其核心思想在于识别并忽略掉这些已知差异，从而使我们在比对不同环境的数据时，能够专注于那些更能反映潜在问题的关键信息。这极大地提升了我们数据分析的效率和准确性，确保我们能更快、更精准地发现系统异常。

7、大模型应用案例：脚本优化与合规检查



大模型目前展现出的一个非常突出的能力，在于它强大的代码生成与优化能力，基于此，我们在当前运维平台中深度融入了多项智能化功能，旨在通过集成大模型技术，实现：

- **代码优化能力：**

大模型能够对存量代码进行深入分析，识别潜在的性能瓶颈及冗余逻辑，并提出更高效、更简洁的实现方案，显著提升代码质量与运行效率。

- **合规检查功能：**

自动化地对照预设的技术规范、安全标准以及内部最佳实践，对代码进行全方位的合规性检查，确保所有提交的代码都能满足严苛的要求，从源头减少潜在的技术风险。

- 通过将这些智能化能力嵌入到日常运维流程中，我们正积极探索大模型技术在提升运维效率、保障系统稳定性方面的巨大潜力，这不仅是运维体系从数字化迈向智能化的关键一步，更是应对复杂系统挑战的必然选择。

8、大模型应用案例：多模态场景探索

在对智能化的持续探索中，我们也尝试了多模态场景的可能性，旨在探索大模型在更广阔领域的应用边界。

我们首先尝试将 Zabbix 的监控看板输入给多模态大模型进行识别分析，初步结果显示其具备一定的识别能力。

紧随其后，我们考虑到运维场景下监控看板作为最普遍的数据呈现形式，决定进一步利用无头浏览器对 Grafana 看板进行截图，并将这些截图输入给大模型进行分析。实验结果令人振奋，模型确实能够识别出图表中存在的异常模式，例如**尖峰**

或**突变**

等情况。然而，需要清醒地认识到，目前这项功能尚处于起步阶段，距离在生产环境中真正可用仍有 不小的差距。

尽管当前成果有限，但这方面的想象空间是巨大的，预示着大模型在辅助运维智能化决策方面拥有广阔的应用前景。

9、大模型探索与展望

好的，基于大模型在运维领域的探索与展望，我们目前聚焦放在三个核心方向。

"当下"我们正在进行的努力：

- 专家知识库的建设：这块我们主要通过 RAG (Retrieval-Augmented Generation) 技术来构建运维专家知识库，以期实现更精准的知识检索和应用。
- 沉淀专家经验的智能体编排：我们致力于将优秀的运维专家经验转化为可复用的智能体，并通过智能体编排的方式进行应用，让专家的智慧能够规模化赋能。
- 智能体与运维场景的深度融合：将这些沉淀下来的智能体与具体的运维场景紧密结合，让智能体在实际生产环境中发挥作用。
- **展望"下一步"的发展：**
- 进一步探索更高级的智能体策略，充分利用大模型的推理能力去探索更复杂的运维的问题，尤其是在以下几个方向：
- 智能体策略探索：这包括对 FunctionCall (调用外部工具与函数)、ReAct (Reason + Act，推理与行动结合)、Plan&Execute (规划与执行) 以及可能更 High-level 的 MCP ReAct (结合 MCP 的 ReAct 框架) 等多种智能体策略进行深入的研究和实践。
- MCP 与多智能体协作：我们也希望通过探索 MCP 和多智能体协作的能力，让不同的智能体能够协同工作，解决更复杂的运维挑战。
- 运维领域的 Deep Research：我们计划在运维领域开展更深入的研究，重点探索智能排障和深度健康检查等核心痛点。

（三）总结及展望

至于"未来"，随着大模型技术的飞速发展，我们预见将涌现出能力更强大的模型，拥有更卓越的推理能力和多模态处理能力，并且将具备更强的运维通用知识。同时，行业会逐步构建 更加专业的知识库，不仅包括通用的运维知识，更会细化到银行、证券等金融行业的 专业运维知识库。我们对此的态度是积极观望，一旦这些 基础设施成熟，将其应用到我们的实践中。

5.4 混沌工程与全链路压测

5.4.1 混沌工程与全链路压测概述

（1）混沌工程与全链路压测的定义

混沌工程（Chaos Engineering）是一种通过主动向系统注入故障来验证系统韧性的工程实践方法。其核心理念是"在可控的环境中主动引发故障，以发现系统中潜在的脆弱点"。正如流感年年都会来袭，我们无法预知它何时爆发，但可以肯定的是，故障迟早会发生。与其被动等待故障的打击，不如主动出击，有计划地模拟故障。通过这种方式，我们能够提前检验系统的"免疫系统"——包括应急响应机制、告警系统，以及业务自身的恢复能力——是否真正有效。

全链路压测是一种在生产环境或高仿真环境下，模拟真实用户行为和流量模式，对系统进行端到端压力测试的方法。与传统压测不同，全链路压测基于真实的业务场景，从最贴近用户行为的入口侧进行施压，压测报文力求高度模拟真实用户报文的离散度、内容和广度。这种方式能够覆盖跨服务、跨业务部门的复杂链路，真正验证系统在真实业务场景下的性能表现。

两者相辅相成，形成"主动预防"与"主动防御"的组合拳：压测侧重于验证系统在预期内高负载下的性能表现，精准识别容量瓶颈；混沌工程侧重于暴露系统在预期外故障下的脆弱性，验证其容错能力。将两者结合使用，可以实现"1+1>2"的协同效应——压测发现的性能瓶颈将作为混沌工程的输入项，而混沌工程的对抗与模拟结果又将反哺压测能力，形成一个螺旋式上升的优化闭环，持续推动业务稳定性的提升。

（2）传统压测的局限性

传统压测存在三大"谎言"，限制了其在复杂系统中的有效性：

环境等价的谎言：传统压测声称测试环境可以等同于生产环境，但实际工作中，无论是机器配置、服务器实例规格、数据量规模，还是网络拓扑结构乃至基础设施的部署模式，测试环境与生产环境之间都存在显著差异。这种差异导致在测试环境获得的压测结果，难以真实推断生产系统的性能表现。

用户真实的谎言：传统压测试图通过模拟流量来完全还原真实用户行为。然而，如果仍然采用传统压测方式，通过人工构建流量模型，则很难全面覆盖业务链路中每一个细微的分支流程。以游戏场景为例，玩家可能在开局前 5 分钟在大厅，并未进行射击操作，随后进入匹配对战，在房间内才出现大量的移动、射击、换弹等行为。这种动态性导致不同协议在不同时间段内的占比呈现出明显的波动，而传统压测模型往往采用固定比例，无法反映这种真实的行为模式。

链路完整的谎言：传统压测声称能够覆盖全链路的所有环节，但无论是在测试环境中进行单集群或单服务的压测，其最大的弊端是无法实现链路的完整覆盖。例如，它往往无法触及第三方依赖服务、负载均衡设备或核心网络基础设施等关键组件。然而，在流量高峰或极端大流量涌入时，这些传统压测未能覆盖、看似边缘的基础服务，恰恰可能成为系统性能的关键瓶颈点。

因此，传统压力测试本质上不过是一种功能性验证手段，它难以胜任真实的容量规划任务，更无法有效地承担系统稳定性验证的重任。

（3）混沌工程的核心价值

- **暴露系统脆弱性**：通过主动注入故障，发现系统在异常情况下的薄弱环节，尤其是那些介于"正常运行"与"明显故障"之间的，非零一判定的性能衰退或体验有损问题。
- **验证容错机制**：检验熔断、降级、限流、重试等容错策略是否按预期生效，即使部分组件发生故障，核心业务依然能够保持可用性，并且系统能够实现快速自愈。
- **提升应急能力**：通过常态化演练，锻炼团队的故障响应和处置能力，提升多业务协同确认影响面和快速止损的效率。
- **增强系统韧性**：持续发现和修复问题，使系统在面对真实故障时更加健壮，将核心场景的 RTO 控制在可接受范围内（如 10 分钟以内）。

（4）全链路压测的核心价值

- **验证系统容量**：在真实生产环境中验证系统的实际承载能力，探究潜在的容量瓶颈，为容量规划提供数据支撑。
- **发现性能瓶颈**：识别系统在高负载下的性能短板，包括内存泄漏、死锁、线程池耗尽等问题，指导性能优化方向。
- **验证架构设计**：检验系统架构在极端流量下的稳定性和可扩展性，尤其是跨服务、跨业务部门的长链路场景。
- **保障业务大促**：为电商大促、节假日抢票、演唱会门票秒杀等关键业务场景提供容量保障，确保系统能够应对业务增长。

（5）压测与容灾演练的通用痛点

在实际落地过程中，压测与容灾演练面临四大通用挑战：

- **准不准**：压测结果能否准确反映系统真实性能与容量。如果压测模型与真实玩家行为差异较大，无论结果是"通过"还是"失败"，其数据都将失去指导意义。
- **快不快**：压测与演练的执行效率是否满足快速迭代的需求。尤其在代理业务中，压测工作经常被延迟到临近上线才启动，导致准备仓促。
- **全不全**：压测和演练对场景和系统范围的覆盖是否全面。不同项目的容灾演习科目不统一，粒度差异大，难以穷尽所有可能的故障情况。
- **贵不贵**：压测与演练所投入的成本是否经济合理。尤其是采用"云真机"或"云手机"进行大规模压测时，运营成本非常高昂。

5.4.2 混沌工程与全链路压测体系设计

（1）混沌工程体系设计

混沌工程体系的建设需要遵循 PDCA（Plan-Do-Check-Act）循环框架，在不断发现问题、解决问题中持续改进。一个完整的混沌工程体系通常包含以下核心要素：

故障注入能力：覆盖多个层次的故障注入能力是混沌工程的基础。

- 基础设施层：CPU 负载、内存压力、磁盘故障、网络延迟/丢包/分区
- 容器层：Pod 故障、资源限制、容器重启
- 应用层：进程异常、进程假死、依赖故障

- 业务特性化品类注入：针对特定业务场景（如游戏、金融）的定制化故障注入能力

爆炸半径控制：通过精确的目标选择、时间窗口控制和自动熔断机制，确保故障注入的影响范围可控。这是在生产环境进行混沌实验的前提条件——必须具备完善的安全边界。

实验编排与调度：支持单次执行、定时执行和周期性执行的灵活调度能力，以及复杂故障场景的编排能力。包括演练管理、故障库、演练历史记录等功能模块，实现数据驱动的持续改进。

可观测性集成：与监控、日志、链路追踪系统深度集成，实现故障注入前后的指标对比和影响评估。包括基础设施监控、业务监控大盘、智能熔断机制，作为实验过程中的"瞭望塔"。

知识库与智能化：沉淀故障案例、最佳实践和行业经验，结合 AI 能力实现故障场景智能推荐和实验效果智能分析。

（2）全链路压测体系设计

全链路压测体系是一个分层架构，需要从流量发起到监控告警形成完整闭环。一个完整的全链路压测体系通常包含五个逻辑分层：

流量发起层：由压测平台和施压机集群组成，负责依照既定场景，不断地生成并向目标系统输送压测流量。核心能力包括：

- 用户模拟：预先申请大量压测专用账号（如 200 万个 UID），确保身份与真实用户无异
- 场景搭建：根据预设目标构造压测场景，模拟出各种可能的业务流和压力情况
- 报文构造：精细化准备压测所需的请求报文，确保请求数据的真实性和多样性

流量治理层：包括流量染色、网关、智能路由隔离策略等核心功能，确保带有"专属标签"的压测流量能在复杂的系统拓扑中精准穿梭，与正常生产流量有效区分，实现"精准投送，避免误伤"。核心机制包括：

- 埋入标识：在发出的每个请求中植入压测特定的标识
- 隔离配置：配置压测环境与生产环境的流量隔离策略

业务逻辑层：承载并处理压测流量的业务逻辑服务，策略性地将正常生产集群与专用的压测集群加以区分，以彻底避免压测对生产环境的任何潜在影响。

数据隔离层：通过引入"影子"副本或专用实例机制，确保压测过程中产生的所有数据，都被严格隔离在独立的沙盒环境中。三层隔离策略包括：

- 数据隔离：影子 DB、影子表
- 流量隔离：独立压测集群
- 消息隔离：影子 MQ、影子 Redis

监控告警层：作为压测过程中的"瞭望塔"，实时、细致地监测各项核心指标，一旦侦测到任何潜在的问题或异常，立即触发预设的告警机制，同时通过熔断机制，在极端情况下自我保护，防止"雪崩效应"。

（3）容灾架构设计

容灾架构是混沌工程和全链路压测的重要验证对象，也是系统韧性的根本保障。分层设计是容灾架构的核心理念：

流量调度层：这是用户请求进入系统的第一关，也是架构韧性的重要支撑。通过 DNS 解析引导用户流量，借助 GSLB（全局负载均衡）实现跨地域、跨数据中心的流量智能分发，结合 CDN 实现资源加速与就近访问。

业务逻辑层：通过多活部署策略，尤其是异地多活甚至三活的部署架构，确保即使部分地域或可用区出现故障，业务也能持续对外提供服务。辅以强弱依赖治理，梳理业务自身的依赖关系以及业务模块间的依赖逻辑。

基础架构层：核心在于跨可用区（AZ）的数据复制与一致性保障机制，确保不同可用区之间的数据实时同步与校验，为上层业务提供可靠的数据基础。

基础设施层：作为整个容灾架构的根基，提供计算、存储、网络资源，包括自建的数据中心、服务器集群与专线网络，以及基于公共云的服务。

5.4.3 混沌工程与全链路压测关键流程

（1）混沌工程实施流程

混沌工程实施需要遵循严谨的流程，确保实验的安全性和有效性：

1. **定义稳态假设**：明确系统正常运行时的关键指标和预期行为，作为实验成功与否的判断基准。这些指标应覆盖技术指标（如 CPU 利用率、内存使用率、响应时间）和业务指标（如订单成功率、核心接口成功率）。

2. **设计实验方案**：选择故障类型、确定注入目标、设置爆炸半径和持续时间，制定回滚预案。需要考虑故障注入对业务的潜在影响，平衡容灾目标与业务效果。

3. **环境检查**：验证目标环境状态正常，确认监控告警就绪，通知相关干系人。在演练前进行 checklist 检查，确保各项前置条件满足。

4. **执行故障注入**：按计划执行故障注入，实时观察系统行为和指标变化。监控面板需要实时展示演练状态（如"环境检查执行中"、"成功"、"演练流程异常"等），便于快速掌握演练进展。

5. **结果分析**：对比实验前后的指标变化，评估系统韧性表现，记录发现的问题。对演练结论进行定性评估，以颜色标签直观展示："不符合预期"标红警示，"成功"则以绿色彰显。

6. **改进闭环**：根据实验结果制定改进措施，推动 S0 级（最高优先级）故障场景的业务架构进行多活改造，跟踪改进落地，验证改进效果。

（2）全链路压测实施流程

全链路压测实施需要经历多个阶段，确保压测的完整性和安全性：

1. **业务场景梳理**：确定压测入口、梳理核心链路、识别需要隔离或 Mock 的依赖。典型的全链路压测场景包括：节假日前的火车票抢票、大型演唱会门票秒杀等涉及跨部门、跨服务的复杂链路。

2. **历史容量分析**：分析历史流量峰值和系统表现，为压测目标设定提供数据依据。结合 FinOps 思路，利用压测数据预测业务增长所需的资源规模，支撑更精准的容量规划与成本控制。

3. **确定压测目标**：根据业务增长预期和容量规划，确定目标 QPS 和响应时间指标。压测目标应从纯技术指标（如 QPS、RT、CPU）提升到业务指标层面（如订单成功率、关键链路转化、核心接口成功率）。

4. **元数据准备**：准备压测账号、请求报文、影子资源等基础数据。需要预先申请大量压测专用账号，并对其进行登录授权认证，确保下游服务能够将压测账号识别为真实用户。

5. **隔离策略配置**：配置数据隔离、流量隔离、消息隔离策略，确保生产安全。这是全链路压测能够在生产环境安全执行的关键保障。

6. **压测计划审批**：提交压测计划，经过 SRE 和业务负责人审批后方可执行。大规模架构改造需 SRE 团队深度参与评审，确保改造方案符合容灾建设目标。

7. **压测执行与监控**：在业务低峰期执行压测，实时监控系统指标和业务指标。压测过程中需进行实时、全面的监控，捕捉系统性能的每一次波动。

8. **结果分析与报告**：生成压测报告，评估系统容量，识别性能瓶颈，输出优化建议。压测报告应直接映射到业务风险与收益，从"测得快不快"变成"业务能不能稳稳跑"。

（3）容灾演练与持续运营

容灾建设是一个需要持续投入、不断优化和迭代的系统工程。基于 PDCA 循环框架，容灾运营包含以下关键环节：

风险治理与巡检：对日常巡检发现的风险进行有效治理，确保系统健康无虞。结合公司平台能力、历史内部外部故障案例进行深入分析，识别出高风险场景并将其纳入管理范畴。

公司级容灾演练：定期组织并执行 S0 级、公司级的容灾演练，验证预案的有效性，提升团队在真实故障场景下的响应能力和协作效率。演练不能流于形式，需要通过红蓝演练等形式，主动挖掘和识别潜在的业务风险点。

预案建设与评审：针对每一个 S0 场景，建立和完善详细的故障预案。当业务线进行大规模架构改造以满足性能或容灾需求时，SRE 团队需深度参与评审，确保改造方案符合容灾建设目标。

智能化未来：随着运行经验的不断积累和丰富，未来大量需要人工介入的调试和分析工作，将逐步由 AI 自主进行验证与优化，通过 AI 分析历史流量数据自动生成更贴近真实业务的压测场景，利用 AI 自动识别性能问题根因并提供优化建议，从而大幅减少人工干预。

5.4.4 腾讯游戏 SRE 混沌工程及压测实践

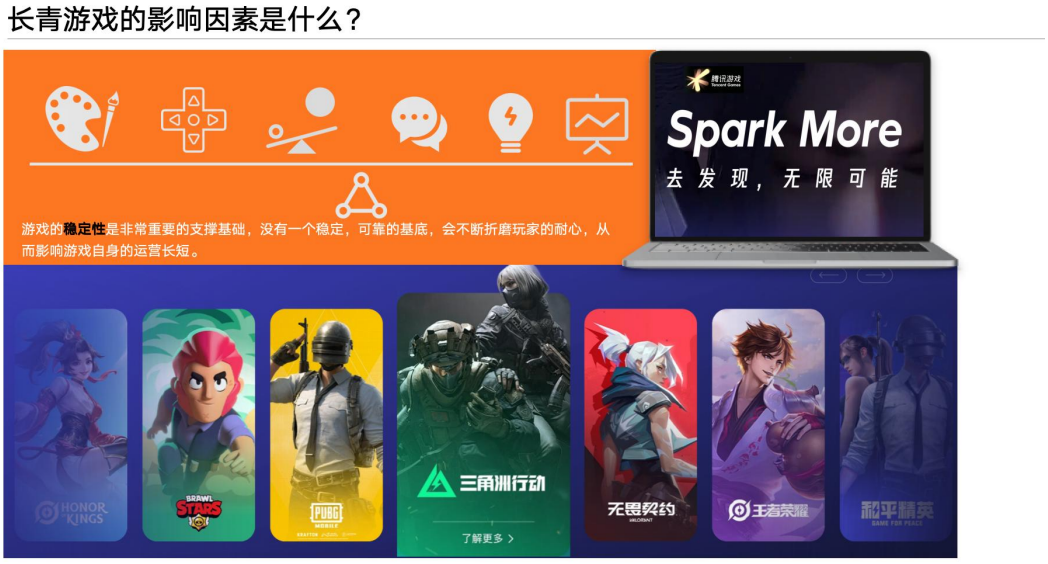
SRE Elite 精选原因

腾讯游戏旗下运营着《王者荣耀》《和平精英》《三角洲行动》等 300 余款游戏产品，日活用户数以亿计。游戏行业在构建高可用系统时面临独特挑战：业务高度异构，300 多款游戏各有不同的技术栈和架构需求；实时性要求极高，竞技类游戏延迟需控制在 20ms 以内；单机承载能力低，CPU 利用率超过 60%即可能引发卡顿；流量洪峰冲击频繁，活动期间连接数可能在一分钟内暴增数倍，极易触发雪崩效应。

面对这些挑战，腾讯游戏 SRE 团队将混沌工程和全链路压测作为核心手段，构建了游戏稳定性保障体系。本案例基于真实线上场景和亿级用户规模，分享了从上线前的容量验

证、风险发现，到上线后的常态化演练、故障快速恢复的完整实践经验，对其他高并发、强实时性业务同样具有借鉴意义。

（一）背景及设计原则



到底是哪些核心要素决定了一款游戏能走向"长青"，不断延长其生命周期？

我们经常讨论，驱动游戏长线发展的因素多种多样：

- **视觉与创意表现：** 美术风格、画面质量、独特的艺术设计。
- **核心玩法体验：** 操控手感、游戏机制的深度与乐趣，以及持续的玩法创新。

- **平衡性与公平性：**数值设计、角色强度、付费与非付费玩家之间的生态平衡。
- **社区生态构建：**玩家互动、用户生成内容（UGC）的引导、社区氛围的维护。
- **长线运营策略：**版本更新、活动设计、与玩家持续的沟通反馈机制。
- **前沿技术驱动：**创新技术应用、性能优化、实现低延迟与快速响应。
- **平台兼容与用户体验：**跨平台支持、直观的 UI/UX 设计、全球化的部署策略。
- **系统架构韧性：**稳定可靠的互联结构、高效的数据流动与强大的技术生态支撑。

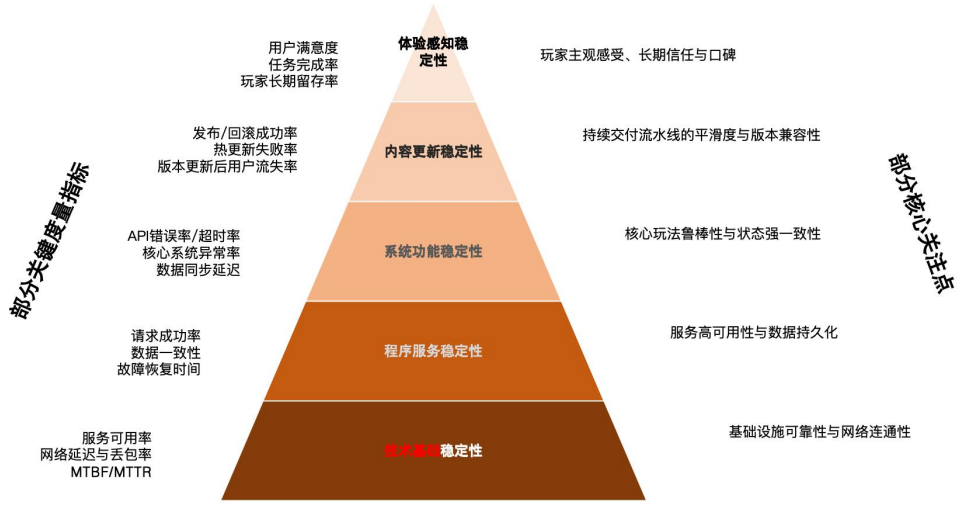
然而，在这一切之上，**游戏的稳定性是至关重要的支撑基础。**这好比金字塔的基石，没有稳固可靠的底层支撑，任何再精彩、再丰富的内容也可能因不稳定而不断消磨玩家的耐心，严重影响游戏的运营寿命。想象一下，如果每一次精彩的游戏体验都被突如其来的中断或性能问题所破坏，玩家的留存和忠诚度将无从谈起。

因此，我们始终将稳定性视为金字塔结构的最底层和最核心部分——它是确保用户拥有流畅、愉悦体验的根本保证。在稳定性这一基石之上，我们才能持续构建和优化上层的所有体验要素，确保游戏能够真正实现长青。

这张幻灯片展示了多款成功的长青游戏，如《王者荣耀》、《荒野乱斗》、《和平精英》、《三角洲行动》和《无畏契约》，它们都深谙此道。正是对技术稳定性的持续追求，结合创意与运营的不断投入，才让这些游戏得以在激烈的市场竞争中脱颖而出，持续"Spark More"，为玩家带来无限可能。

稳定性的金字塔结构：从基础设施到用户体验的乘数效应

稳定性的金字塔结构



稳定性在产品或服务中的分层级体现及其衡量方式，通过一个直观的金字塔模型，自下而上地揭示稳定性的不同层面、关键度量指标和核心关注点。

越是金字塔顶端，越是贴近终端用户的真实体验，即用户主观感受；而越是金字塔底部，则越偏向物理层面，聚焦于我们技术基础的稳定性。稳定性并非孤立存在的，它具有显著的乘数效应。我们可以将其形象地比喻为，当你向平静的水池中投入一块小石子，它所泛起的涟漪波及的范围，往往远超石子本身，影响到整个水体。这正是要强调的，金字塔的每一层都并非独立，而是紧密相连。任何一个层面的波动，都可能沿着金字塔向上层层传递，最终影响到终端用户，导致用户体验受损，甚至对业务收入造成巨大冲击。

- **金字塔底部：技术基础设施稳定性**
    - **概述**：这是整个稳定性的基石，关注最底层的硬件、网络和基础服务的可靠性。设想一下，如果服务器不可用、网络延迟高企、丢包率居高不下，那么在此之上构建的所有系统与服务都将摇摇欲坠。
    - **部分关键度量指标**：
        - 服务可用率 (Service Availability)
        - 网络延迟与丢包率 (Network Latency & Packet Loss Rate)
        - MTBF (Mean Time Between Failures，平均故障间隔时间)
        - MTTR (Mean Time To Repair/Recover，平均恢复时间)

    - **部分核心关注点**：基础设施可靠性与网络连通性。我们不能忘记，这一切稳定性的根基正在于此。
- **金字塔第二层：程序服务稳定性**
    - **概述**：在坚实的基础之上，这一层关注单个程序或服务的稳定运行。这包括确保我们的服务能够持续响应，并且在面对高并发、突发流量时，仍能保持其应有的性能与功能。
    - **部分关键度量指标**：
        - 请求成功率 (Request Success Rate)
        - 数据一致性 (Data Consistency)
        - 故障恢复时间 (Fault Recovery Time)
    - **部分核心关注点**：服务高可用性与数据持久化。任何的数据不一致或服务中断都可能在用户心中留下难以磨灭的负面印象。
- **金字塔第三层：系统功能稳定性**
    - **概述**：继续向上，我们关注的是整个系统各功能模块之间的协调稳定性与正确性。尤其是对于像游戏这样的产品，核心玩法的鲁棒性、系统状态的强一致性至关重要。这直接关系到用户能否顺畅地体验产品的核心价值。
    - **部分关键度量指标**：
        - API 错误率/超时率 (API Error Rate / Timeout Rate)
        - 核心系统异常率 (Core System Anomaly Rate)
        - 数据同步延迟 (Data Synchronization Delay)
    - **部分核心关注点**：核心玩法鲁棒性与状态强一致性。
- **金字塔第四层：内容更新稳定性**
    - **概述**：这一层聚焦于软件版本、内容发布和更新过程的稳定性，以及更新对用户的影响。我们致力于实现平滑的版本更新体验，确保玩家能够第一时间体验到最新的玩法与内容，而不会因为更新故障或延迟而感到沮丧。
    - **部分关键度量指标**：
        - 发布/回滚成功率 (Release / Rollback Success Rate)
        - 热更新失败率 (Hot Update Failure Rate)

- 版本更新后用户流失率 (User Churn Rate After Version Update)
- **部分核心关注点**：持续交付流水线的平滑度与版本兼容性。

- **金字塔顶部：体验感知稳定性**
  - **概述**：这是稳定性最终体现在用户端的感受，是所有底层稳定性的最终目标。它代表了用户对产品的主观认知，是用户满意度、长期信任与口碑的直接体现。我们所有的努力，最终都将汇聚于此，影响用户对产品的整体感知。
  - **部分关键度量指标**：
    - 用户满意

稳定性的乘数效应：小问题引发大灾难

系统稳定性问题并非孤立存在，一个小小的隐患，往往像投入池塘的一颗石子，会激起层层涟漪，不断放大，最终可能引发严重的级联故障。这种乘数效应，要求我们 SRE 必须未雨绸缪，主动出击，提前发现并规避风险。

核心问题：程序服务稳定性不佳，引发连锁反应

我们的经验表明，程序服务的不稳定性常常是引发一系列后续问题的核心根源。频繁的程序更新，有时是为了解决旧问题，却意外地导致了新的稳定性挑战，形成恶性循环。

典型案例剖析：

- **初始问题**：某次内部测试期间，服务器进程频繁崩溃（Crash）。

- **初步诊断：**经过层层排查，问题最终锁定在一个数据库存储过程（SP）的执行时长异常。
- **首次"修复"：**开发团队为了解决当前的Crash问题，将原有的串行执行逻辑修改为并行处理。
- **新问题浮现：**然而，在正式发布后，随着玩家数量的急剧增加，系统又出现了严重的数据库锁表问题，进一步加剧了系统的不稳定性。

这个案例清晰地揭示了，仅仅为了解决眼前的故障而采取的草率或不完善的方案，不仅无法彻底根治，反而可能引入更深层次、更难以预料的复杂问题。

我们的挑战与思考：如何提前发现和预防问题，提前规避风险？

为了更直观地理解各类稳定性问题的分布，我们统计了过去一年对外测试过程中发现的稳定性问题。下方的饼图数据，清晰地展示了各项稳定性问题的占比，为我们的SRE策略制定提供了重要依据：

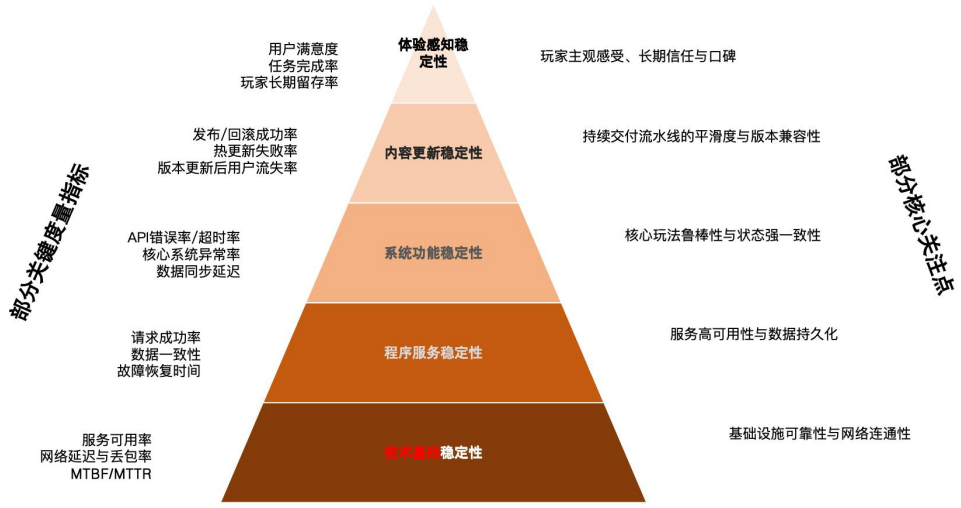
- **系统功能稳定性：**49.06%
- **内容更新稳定性：**28.30%
- **程序服务稳定性：**13.21%
- **体验感知稳定性：**5.66%
- **基础稳定性：**3.77%

从饼图可以看到，"系统功能稳定性"占据了近一半的比例，是我们需要重点关注和投入的主要领域。其次，"内容更新稳定性"和"程序服务稳定性"也占据了相当大的比重。这与我们前面提到的"程序服务稳定性问题较多，导致内容更新频繁，进一步导致其他稳定性受损"的观点是相互印证的。

02 SRE的实战工具箱：压测与混沌工程

“主动预防”和“主动防御”。协同效应：1+1>2

稳定性的金字塔结构



"稳定性的金字塔结构"中的第四层，也就是内容更新稳定性。

众所周知，内容更新的稳定性直接关乎产品迭代的平滑度，它处于从底层技术到用户体验感知这一递进关系中的中上层。当底层的"程序服务稳定性"受损时，我们常常需要进行紧急修复或版本迭代，这直接导致了新内容的发布。因此，内容更新的稳定性，实际上是"技术基础设施稳定性"和"程序服务稳定性"的最终呈现与检验。

具体而言，内容更新稳定性层有以下核心关注点与关键度量指标：

- **核心关注点：**

- **持续交付流水线的平滑度：** 核心在于发布流程是否顺畅、自动化程度高，以及是否有完备的回滚机制，确保新版本能够安全、高效地推向用户。
- **版本兼容性：** 新旧版本之间的接口、数据和功能兼容性至关重要，避免因版本更新导致的用户体验中断或数据异常。

- **关键度量指标：**
  - **发布/回滚成功率：** 衡量每次版本发布和回滚操作的成功比例，确保新旧版本切换的可靠性。
  - **热更新失败率：** 对于支持热更新的系统，监控热更新的成功率，识别并解决导致更新失败的问题。
  - **版本更新后用户流失率：** 这是衡量更新稳定性对用户影响的最直观指标，过高的流失率往往意味着更新过程中存在严重的用户体验问题。

在整个稳定性金字塔中，内容更新稳定性尤为关键，因为它连接了技术实现与用户感知。任何在底层服务中暴露出的问题，往往会通过内容更新的滞后、失败或兼容性问题反馈给用户，进而影响到最顶层的"体验感知稳定性"。因此，保障"内容更新稳定性"不仅是技术层面的挑战，更是提升用户满意度和维护品牌形象的重要一环。

（二）体系设计及关键流程

SRE 的实战工具箱：压测与混沌工程

在 SRE 的实战中，我们手握两大核心利器：压测与混沌工程。它们并非独立作战，而是通过"主动预防"与"主动防御"形成一套组合拳，从而实现 1+1>2 的协同效应。

首先，我们聚焦于压测，它承载着主动预防的关键功能。我们可以将其理解为一场高强度、全方位的系统体检。

- **模拟真实用户行为**：压测旨在通过模拟海量的用户访问行为，对系统施加巨大的负载。
- **提前发现瓶颈与故障**：
  - 在系统上线前或更新后，通过压测来探测其性能极限和容量瓶颈，及时发现潜在的性能问题、资源不足、配置不当等隐患。
  - 挖掘系统在极端负载下的稳定性问题，例如内存泄漏、死锁、线程池耗尽等。

正是通过这种高强度的体检，我们才能在用户真正接触系统之前，预判并解决可能出现的各种问题，为业务的稳定运行打下坚实的基础。

压测与混沌工程的关系及价值

主动预防：压测的价值
通过“高强度体检”模拟真实用户的访问行为，甚至施加远超日常的峰值流量，来问系统这个问题“这你受得了吗？”

主动防御：混沌工程的价值
通过“疫苗接种”提前应对流感（故障）的到来，虽然不知道它什么时候会来，但迟早会来。
与其被动等待，不如主动、有计划地模拟故障，来检验系统的“免疫系统”是否有效。

对比维度	压测	混沌工程
核心目标	验证系统在预期内高负载下的性能表现，探知容量瓶颈	暴露系统在预期外故障下的脆弱性，验证容错能力
主要手段	模拟正常/峰值用户访问流量，如高并发请求	主动注入各类故障，如服务器宕机、网络延迟
关注焦点	性能与容量（吞吐量、响应时间、资源利用率）	韧性与可靠性（故障隔离、自动恢复、冗余机制）
理想结果	系统能平稳处理目标流量，性能指标达标	系统在部分组件故障时，核心业务仍可用，能快速自愈

接下来，我们将深入探讨压力测试（Load Testing）与混沌工程（Chaos Engineering）之间的关系、各自的独特价值，以及如何通过两者的协同作用提升系统可靠性。

首先，我们来谈谈压测的价值，我将其类比为"高强度体检"。我们通过模拟用户访问行为，包括超日常的峰值流量，对系统进行严苛的负载考验。其核心目标是验证系统在高负载下的承受能力，探究潜在的容量瓶颈。就好比我们想问系统一个灵魂问题："你到底能承受多大的压力？"

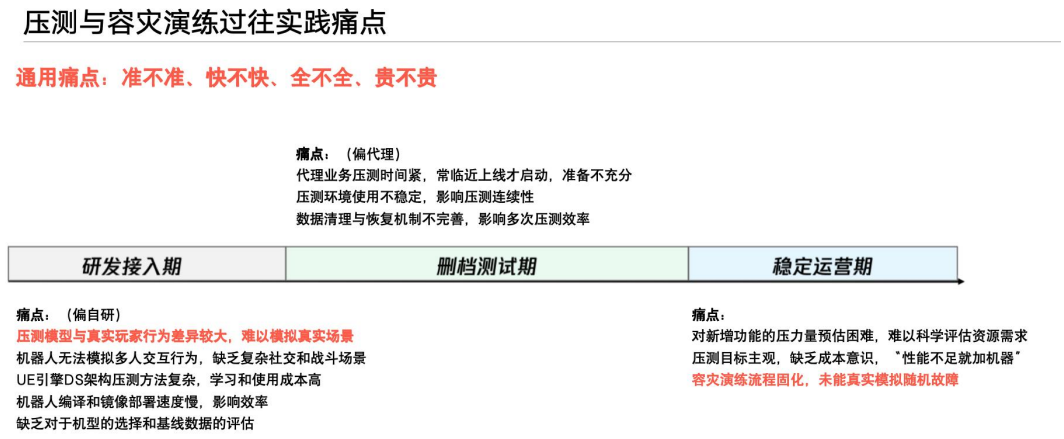
而混沌工程的价值，我们则可以将其理解为"主动防御"，它如同为系统接种疫苗。正如流感年年都会来袭，我们无法预知它何时爆发，但可以肯定的是，故障迟早会发生。与其被动等待故障的打击，不如主动出击，有计划地模拟故障。通过这种方式，我们能够提前检验系统的"免疫系统"——包括你的应急响应机制、告警系统，以及业务自身的恢复能力——是否真正有效，是否在预期范围内。

我们通过以下表格，详细对比了压测与混沌工程在不同维度的异同：

- **核心目标：**
  - **压测：**验证系统在**预期内**高负载下的性能表现，精准识别容量瓶颈。
  - **混沌工程：**暴露系统在**预期外**故障下的脆弱性，从而验证其容错能力。
- **主要手段：**
  - **压测：**模拟正常或峰值用户访问流量，例如高并发请求。
  - **混沌工程：**主动注入各类故障，比如服务器宕机、网络延迟，甚至是更复杂的系统性失效。
- **关注焦点：**
  - **压测：**侧重于性能与容量，包括吞吐量、响应时间以及资源利用率等关键指标。
  - **混沌工程：**强调韧性与可靠性，例如故障隔离能力、自动化恢复机制以及系统的冗余设计。
- **理想结果：**
  - **压测：**系统能够在目标流量下保持平稳运行，各项性能指标均达到预设标准。
  - **混沌工程：**即使部分组件发生故障，核心业务依然能够保持可用性，并且系统能够实现快速自愈。

因此，我们必须将这两种能力有机地结合起来，而非割裂地看待它们。它们相辅相成，共同构筑起系统高可靠性的坚实屏障。正如同我之前在分享中多次提到的，尤其是在细致聆听了银行和携程等行业的经验分享后，我深切体会到在复杂系统环境中，这两种实践对于保障业务连续性是何等关键。

压测与容灾演练的痛点分析



通用痛点

首先，我们面临四大通用挑战，它们贯穿于整个测试与演练过程：

- **准不准：** 压测结果能否准确反映系统真实性能与容量。
- **快不快：** 压测与演练的执行效率是否满足快速迭代的需求。
- **全不全：** 压测和演练对场景和系统范围的覆盖是否全面。
- **贵不贵：** 压测与演练所投入的成本是否经济合理。

分阶段痛点

针对产品生命周期的不同阶段，我们遇到了更具体的痛点。如下方示意图所示，产品生命周期通常经历研发接入期、删档测试期和稳定运营期。

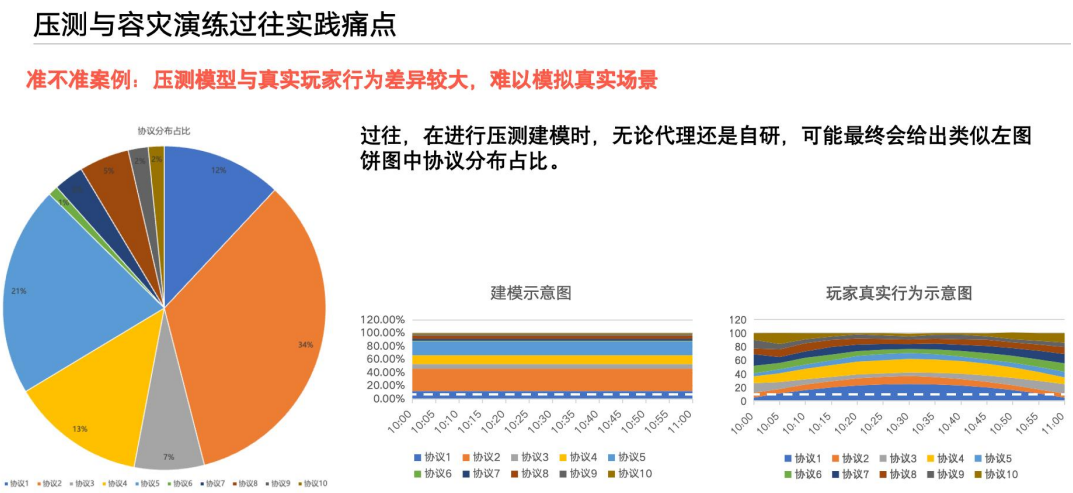
- **研发接入期（偏自研项目）**
  - **压测模型与真实玩家行为差异大：** 这是核心问题，自研压测模型未能有效模拟真实用户行为，导致测试结果无法准确映射实际场景。
  - **机器人无法模拟复杂交互：** 在需要多人交互的复杂场景（如社交、战斗），机器人模拟能力不足。
  - **UE 引擎 DS 架构压测方法复杂：** 特定技术栈（如虚幻引擎的 DS 架构）的压测方法复杂，学习与使用成本高昂。

- **机器人编译与镜像部署速度慢：**压测环境准备效率低下，耗时过长，影响测试进度。
- **缺乏机型选择与基线数据评估：**压测方案未充分考虑不同硬件配置对性能的影响，也未能建立有效的性能基线。

- **删档测试期（偏代理项目）**
  - **压测时间紧迫，准备不充分：**代理业务常临近上线才启动压测，导致准备仓促，无法充分验证。尤其对于腾讯代理的海外游戏，尽管它们可能已在海外市场经过长期验证，但在国内需要进行大量本地化改造或组件接入，以及满足合规性要求。如果缺少足够的时间与海外开发商沟通并进行改造验证，会大大影响压测效果。
  - **压测环境不稳定：**压测环境经常出现故障或中断，影响测试的连续性。
  - **数据清理与恢复机制不完善：**多轮压测之间的数据清理与环境恢复操作效率低下，拖慢了整体测试进度。
- **稳定运营期**
  - **新增功能负载预估困难：**对于新发布功能，难以准确预测其负载压力，导致资源规划可能不合理，存在资源闲置或不足的风险。
  - **压测目标主观，缺乏成本意识：**压测过程缺乏量化的目标，在遇到性能瓶颈时，团队往往倾向于通过"加机器"来解决问题，而非深入优化，缺乏成本控制的理念。
  - **容灾演练流程固化，未能真实模拟随机故障：**容灾演练流程过于固定，未能有效模拟实际生产环境中复杂的随机故障，从而降低了演练的实战意义。当演练的 check list 固定后，所有干系人都已知晓操作步骤，这导致演练难以覆盖真实生产环境可能会出现的复杂突发情况。毕竟，生产环境的故障往往是随机且不可预测的。

与银行或携程这类业务长期处于稳定运营期的模式不同，网络游戏具有鲜明的生命周期，从立项、删档测试、稳定运营，直至最终退市。这使得游戏自研业务面临独特的挑战，因为新业务通常缺乏可参考的历史数据。即使是相同类型的游戏，后台技术栈的差异也会导致显著的性能差异，使得简单的参考变得困难，特别是在压测模型与真实玩家行为差异巨大的情况下。因此，准确模拟真实场景、评估性能成为一大难题。

准不准案例：压测模型与真实玩家行为差异较大



问题描述： 如果压测数据无法反映实际情况，它就失去了指导意义。过往在进行压测建模时，无论是通过第三方工具代理还是团队自研，最终呈现的协议分布占比往往与真实玩家行为存在显著差异。

SRE 专家解读：

我们不直接基于用户行为日志进行建模，而是从程序层面的客户端与服务端协议来统计占比，原因在于：

- 用户行为日志本质上属于经营分析的流水数据，而程序侧可能并未记录所有行为日志，导致其不完整或不准确。
- 基于协议层面的统计，能够更准确地反映系统实际承载的请求类型和数量。

问题呈现：协议分布"准不准"？

- **协议分布占比（饼图）：** 这张饼图展示了压测建模初期设定的协议分布比例，例如协议 1 到协议 10 的具体百分比。这通常代表了在压测启动前，我们预设的请求类型分布。
- **建模示意图：** 这张堆叠图进一步阐述了饼图所揭示的建模模式。它描绘了在建模场景下，不同协议在特定时间段内（例如 10:00 至 11:00）的相对占比。从图中可以看到，各协议的份额在整个压测期间保持相对恒定不变。这意味着我们的质量保障团队在进行压测时，可能会依据这个固定的比例来编写自动化脚本和机器人，然后进行施压。

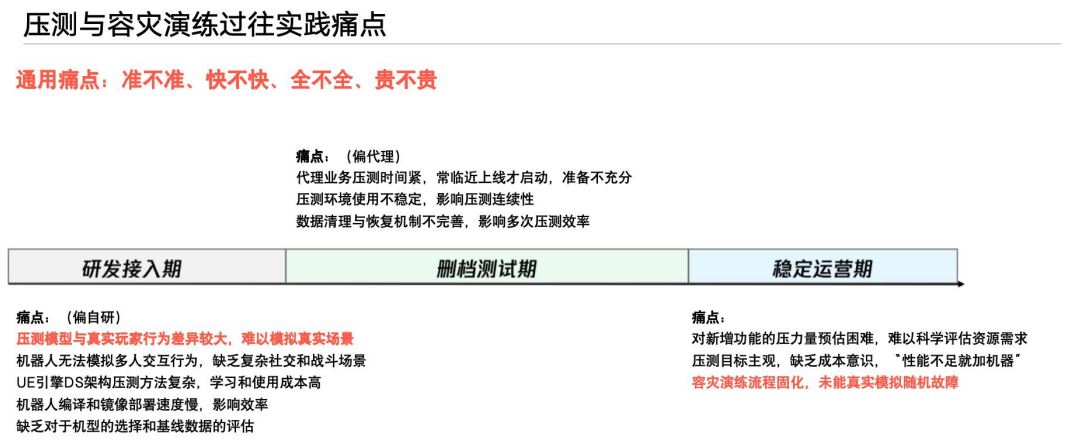
- **玩家真实行为示意图**：然而，真实用户的行为并非如此一成不变。这张图对比展示了玩家的真实行为模式，其中各协议的分布比例在时间轴上动态变化。以一款第一人称射击（FPS）游戏为例，玩家可能在开局前 5 分钟在大厅，并未进行射击操作，随后进入匹配对战，在房间内才出现大量的移动、射击、换弹等行为。这种动态性导致了不同协议在不同时间段内的占比呈现出明显的波动。

核心痛点与风险：

如果将模型中设定的峰值压力（例如图中白色虚线所示的协议 1 峰值）与真实玩家行为进行对比，我们会发现：

- **未被覆盖的压力场景**：在玩家真实行为中，存在部分场景（如右图中白色虚线上方协议 1 的蓝色区域）的压力强度远超建模设定。这意味着在压测中，这些真实场景压根没有被充分测试到。
- **压测结果失真**：这种模型与真实行为之间的差异，直接导致压测结果与实际生产环境的表现发生偏差。
- **线上故障隐患**：许多业务可能经历过"压测通过，上线即崩溃"的窘境，这往往是由于压测未能有效模拟真实业务峰值导致。
- **资源浪费与效率低下**：反之，有时压测会持续在某个场景发现性能瓶颈，难以通过。但这很可能是因为压测模型设定了过高的、实际永远不会触及的峰值压力，导致团队在解决一个"伪问题"上耗费了大量时间和精力。

因此，如果压测建模不准确，无论结果是预期的"通过"还是"失败"，其数据都将失去指导意义，为系统的韧性埋下了巨大的隐患。



我们的痛点可以用四句话高度概括，即："准不准、快不快、全不全、贵不贵。"这反映了压测结果的准确性与真实性、执行效率、覆盖范围以及投入成本等方面的普遍困境。

我们将这些痛点与产品生命周期相结合，详细阐述了在不同阶段所遇到的具体问题。

A. 研发接入期痛点 (偏代理业务):

- **压测时间紧张，准备仓促：**代理业务的压测工作经常被延迟到临近上线才启动，导致预留时间不足，准备工作往往不够充分。
- **压测环境不稳定：**压测环境自身存在不稳定性，频繁中断或失效，从而影响压测的连续性和有效性。
- **数据清理与恢复机制不完善：**在多次压测循环中，压测后的环境数据清理和恢复流程存在缺陷，显著降低了再次进行压测的效率。

B. 删档测试期痛点 (偏自研业务):

- **压测模型与真实玩家行为差异大：**当前的压测模型难以准确模拟真实玩家的行为模式和复杂交互，导致压测结果与实际生产环境存在较大偏差，无法有效揭示系统在真实负载下的表现。
- **机器人无法模拟多人交互场景：**特别是在游戏行业，涉及复杂社交、战斗互动场景时，现有压测机器人无法有效模拟多用户间的协作与对抗行为。
- **UE 引擎 DS 架构压测方法复杂，成本高昂：**
    - **方法复杂性：**针对基于 Unreal 引擎和 DS（Dedicated Server）架构的游戏，其压测方法极其复杂。与传统的纯协议或网络通信模式不同，这类架构要求压测必须通过真实的客户端与服务器进行通信，因为 Unreal 引擎需要管理完整的用户状态上下文。如果采用传统压测模式，是无法满足其对状态维护的需求的。
    - **成本高昂：**若采用模拟真实用户的"云真机"或"云手机"进行大规模压测，运营成本将非常高昂，这对于预算有限的开发团队而言是巨大的负担。
    - **无头客户端的应对与挑战：**部分预算有限的团队会采用"无头客户端"模式来降低成本。这相当于一个精简版的游戏客户端，阉割了音频、视频渲染等非核心功能，以最小实体形式存在，从而解决压测的技术难题。
    - **编译与镜像部署效率低下：**然而，"无头客户端"的编译构建过程本身就是一个重大挑战。将压测机器人（即无头客户端）编译完成需要耗费

大量时间。考虑到游戏行业的快速迭代周期（可能每天都有新版本在内部进行测试），用于构建的资源通常是恒定的，这使得 QA 团队难以在需要时快速获取到所需的压测资源和机器人，严重影响了测试效率。

- **缺乏机型选择与基线评估：** 压测计划中未能全面考虑不同硬件机型（尤其是移动端设备）的性能差异，也缺乏对性能基线数据的科学评估。

C. 稳定运营期痛点：

- **新增功能压力预估困难：** 在系统稳定运行后，对于新增功能的压测，难以准确预估所需的压力负载，从而无法科学评估所需资源，可能导致资源浪费或不足。
- **压测目标主观，缺乏成本意识：** 压测目标设置往往过于主观，普遍缺乏成本意识，导致"性能不足就简单通过增加机器解决"的粗放式做法。
- **容灾演练流程固化，无法模拟随机故障：** 容灾演练的流程过于僵化，未能真实模拟生产环境中可能出现的随机、突发性故障场景，导致演练效果不佳，未能有效提升系统的韧性。

D. 全部全的案例（特定类型游戏容灾演练挑战）：

- 针对不同类型的游戏，其系统架构、业务逻辑各异，导致容灾演练的科目也存在显著差异。单一、固定的演练方案无法覆盖所有场景。例如，某些游戏可能对网络延迟更为敏感，而另一些则更关注数据库的读写

全不全案例：容灾演练覆盖面不足

压测与容灾演练过往实践痛点

全不全案例：不同项目容灾演习科目不一，粒度不同，难以覆盖所有故障情况

业务A

类别	检查项
客户端	服务器维护感知
客户端	不能频繁调用授权页面
客户端	单个后台服务的异常不能导致客户端不相关逻辑不可用（覆盖所有进程）
客户端	非必需的第三方组件异常时不能影响业务核心路径
服务器	不能有单点故障或单点故障时间少于15分钟影响小于5%的用户
服务器	服务上云，有状态服务使用GameStateKubSet工作负载
服务器	模块隔离-单个服务支持独立重启
服务器	模块隔离-单个服务进程异常结束不能引发其他服务进程异常
服务器	模块隔离-游戏业务服务进程与接入层进程剥离
服务器	模块隔离-外部接口异常不能影响用户关键路径
服务器	平滑扩缩容能力
服务器	过载保护-接入层需要有阈值控制
服务器	过载保护-战斗服需要有阈值控制
服务器	过载保护-服务器
服务器	过载保护-服务器对一段时间内不活跃连接要强制断开
服务器	服务启动耗时小且不影响用户体验
服务器	服务支持进程消失后自动拉起
服务器	关键进程和性能计数器需要接入监控告警
服务器	服务器可用性
服务器	第三方组件容灾能力确认
服务器	单个硬件永久损坏不能导致用户数据丢失
服务器	单个服务器重启不能导致用户关键数据丢失
服务器	单个服务进程异常结束不能导致用户关键数据丢失
服务器	非业务逻辑或操作资源限制
服务器	机器故障与重启
组件	
DB	
通信异常	模拟通信层异常

各业务间演练科目不同，覆盖面难保障
反复调试耗时耗力

业务B

检查项	描述	用例名称
客户端：启动时先去公告服务拉取停服公告	拉取间隔不能小于1分钟，如果拉取到停服公告，需要禁止用户登陆，公告服务不可用时，玩家可以登录登录。	客户端：停服公告拉取 客户端：分端公告（Android、iOS、PC） 客户端—维修公告 客户端—白服/审核公告 客户端—单点登录/单一强制登录限制一
客户端：不能频繁调用授权页面	授权授权过期，且用不允许重新拉取授权页面，防止服务器主动关闭客户连接时，需要退出登录后重新拉起授权页面并连接，客户端需要根据玩家进行主动登录，玩家点击登录拉起授权页面，客户端需要有明文提示的功能和重连机制。	客户端：拉取授权（服务器已满） 客户端：拉取授权（服务器未开放） 客户端：拉取授权（客户端版本过低） 客户端：拉取授权（停服维修时） 客户端：拉取授权（授权重连 通过） 客户端：拉取授权（多次重连失败）
客户端过期保护：连续三次登陆失败禁止一段时间登陆	登陆失败登陆失败时，不允许程序自动重新登陆，服务产品过去多重试，连续三次登陆失败后，进行用户等待1分钟，才能登陆，登陆与服务打不可用，再次登入次失败后需要等待5分钟以上，客户端重启也要记住间隔时间。	连续登录失败后禁止一段时间登录 服务器异常状态
客户端模块隔离：单个后台服务的异常不能导致客户端不相关逻辑不可用	单个后台服务全部停掉停止，无法连接，连接超时，连续错误响应，不能影响用户使用不相关的其他功能，例如：支付接口异常，不应该导致客户端完成不能登陆	单个后台服务器的异常不能导致客户端不相关逻辑不可用（参考用例编号10）
服务器：实现可靠的停服公告机制	需要有独立且高可用的公告服务（单个公告服务器的要求是>=4个9/3个台服集群），且服务发现机制正常，1、玩家通过初始和客户端拉取服务器公告；停服后公告服务持续发布公告，玩家不允许登陆	服务器：可靠停服机制
服务器：不能有单点故障或单点故障时间少于15分钟影响小于5%的用户	每个服务都有多个节点进行应用容错和健康检查节点支持至少三个节点集来自，单个服务进程异常结束，单个服务器异常关机等不能让用户感知，如果有单点故障必须修复	
服务器：单个服务进程异常结束不能导致用户关键数据丢失	游戏应用异常结束不能对客户造成大量损失，内存异常的情况下，数据能即时保存，不能数据损坏或丢失，如果内存异常的情况下，服务器重启后用户数据损坏或丢失，则不允许玩家登陆，如果数据损坏或丢失，则只能接受单点丢失玩家，如果重要数据的损坏，一般发生不允许丢失，如果重要数据的损坏，则只能接受单点丢失玩家。	服务器：数据落盘

在"全不全"这一维度上，我们面临的核心痛点是：现有性能测试（压测）与容灾演练方法论存在明显局限性。由于缺乏统一标准，不同项目的容灾演习科目不一、粒度不同，导致难以全面覆盖所有潜在故障场景。

普遍存在的痛点：

- **演练覆盖不全**: 过去的容灾演练实践存在遗漏，未能实现故障场景的全面覆盖。
- **标准化缺失**: 不同项目间的容灾演练科目不统一，粒度差异大，难以穷尽所有可能的故障情况。
- **重复与低效**: 各业务线在容灾演练上的差异性导致覆盖面难以保障，同时耗费大量人力进行重复且零散的调试工作。这种固化且僵硬的演练流程往往流于形式，无法有效提升系统的实际韧性。

为了展现这些痛点，幻灯片通过"业务 A"和"业务 B"两个案例，列举了在客户端、服务器、组件、DB 和通信异常等多个维度下，需要关注的容灾检查项。这些检查项体现了在不同层面可能出现的故障以及对应的容灾考量。

幻灯片主体内容：

由"业务 A"和"业务 B"两个表格构成，展示具体的容灾检查项。

业务 A 表格：

- **结构**: 两列，左侧为"类别"，右侧为"检查项"。
- **内容**:
  - **类别**: 涵盖系统层面的"客户端"、"服务器"、"组件"、"DB"及"通信异常"。
  - **检查项**: 关注容灾与高可用性需求，例如：
    - **客户端**: 服务器维护感知，避免频繁授权调用，单后台服务异常不影响不相关逻辑。
    - **服务器**: 规避单点或影响较小故障，支持服务上云，实现模块隔离（独立重启、进程异常不影响其他进程、与接入层剥离），具备平滑扩缩容能力，提供过载保护（接入层、战斗服、服务器、强制断开不活跃连接），优化服务启动耗时，实现进程消失自动拉起，关键进程与性能计数接入监控告警，确保服务器可用性，确认第三方组件容灾能力，保证数据不丢失（硬件永久损坏、服务器重

启、进程异常结束），限制非业务逻辑或操作资源，应对机器故障与重启。

- **组件/DB**: 预留检查项，待细化。
- **通信异常**: 模拟通信层异常。

- **特点**: 业务 A 的检查项侧重于系统架构及基础服务的韧性、容错能力和资源管理。

业务 B 表格:

- **结构**: 三列，分别为"检查项"、"描述"及"用例名称"。
- **内容**:
  - **检查项**: 具体的容灾演练场景，例如：客户端启动拉取停服公告、客户端授权调用频率控制、客户端过载保护、客户端模块隔离、服务器可靠停服公告机制、服务器单点故障、服务器关键数据丢失。
  - **描述**: 详细说明检查项的具体要求或情景，如"拉取间隔不小于 10 秒；若拉取到停服公告，禁止用户登录"。
  - **用例名称**: 提供具体的用例或故障模拟场景，如"客户端：停服公告"。

案例分享：护航游戏稳定上线与运营

本次分享接下来将通过具体的案例，分析我们在 SRE"左移"和"右移"实践中的经验和成果。

SRE左移研发服务，为“主动预防”提效降本

一体化压测提效点：免去人工视图编辑、人工数据统计、人工总结的时长，提升压测机器人构建效率

SRE 左移研发服务：一体化压测实践

SRE 通过"左移研发服务"策略，将稳定性保障前置到研发阶段，实现"主动预防"，从而提升效率、降低成本。在一体化压测实践中，我们的核心目标是解放工程师，通过自动化手段免去人工视图编辑、数据统计和总结等重复性工作，提高压测机器人的构建效率。

在游戏行业中，压测机器人的构建速度和科学评估方法的缺失是普遍痛点。过往，新业务立项时，QA 团队往往需要依赖游戏后台工程师的经验，根据业务架构来大致拍板机型选择。例如，参照腾讯的标准推荐机型后，QA 团队会根据运营目标（如 40-50 万 PCU）进行压测。若无法达标，则可能采取粗放式增加资源的方式。这种方式存在隐患：如果初期机型选型并非最优，资源可能过剩，导致业务上线后成本大幅攀升。

此外，我们与 QA 团队沟通时发现，鉴于业务多运行在云上，数据收集和整理异常困难。数据可能来自服务本身、云控制台以及其他周边系统，QA 需要投入大量时间进行手工整理。SRE 团队也面临类似挑战，例如海量可观测性视图的编写与展示工作，随着版本快速迭代，视图也需频繁更新，这无疑增加了工程师的负担。

为此，我们构建了一体化压测解决方案，实现了从 **构建即刻交付**、**交付即刻观测**、**观测即刻分析**、**分析即刻实施**到**实施即刻总结** 的闭环流程：

- **构建即刻交付**：我们通过 CI 流水线实现代码的自动构建和即时交付。一旦游戏版本发布，所有可观测内容会根据预埋点自动构建。同时，我们通过异步方式解决客户端构建过程中的资源抢占问题，使得 QA 团队在前端测试功能时，后台能并行进行机器人构建，提升效率。

- **交付即刻观测**：作为 SRE 团队，我们基于游戏品类实践，沉淀了通用的可观测性解决方案和数据埋点要求。这些成为版本交付的标准，确保研发团队在版本发布前已将必要的数据埋点集成到产品中。结合可观测平台，这些 JSON 文件能够直接展示。我们特别关注版本维度属性的展示，通过不同颜色标识，使团队能直观识别不同版本间的指标差异和可观测内容的变化。
- **观测即刻分析**：在交付的可观测内容基础上，我们会立即进行数据分析。尤其在业务首次对外测试或多轮删档测试期间，我们会结合所有相关进程日志，分析协议分布，并展示协议在不同时间段的变化趋势。这使得 QA 团队能基于最真实的用户行为，去修改和构建其压测模型。
- **分析即刻实施**：通过一体化压测能力，QA 团队可以将压测用例沉淀到平台。他们可以根据不同的游戏品类和业务特性，选择或定义压测用例，并启动自动化压测。这意味着压测过程无需人工干预，系统会自行完成。
- **实施即刻总结**：压测完成后，我们利用 AI 能力，汇聚来自不同平台和系统（包括内部和外部）的数据，并通过大模型进行总结。这简化了原本分散、耗时的数据整理工作，提高了总结效率。

通过这套一体化压测实践，我们减少了大量人工操作。例如，生产环境中可能多达几十甚至上百个视图，过去需要频繁手动调整。现在，借由"构建即交付，交付即观测，观测即分析"的模式，视图会随版本更新而自动调整，确保我们看到的始终是最新的信息，即使版本迭代带来内容新增或废弃，视图也能自动适配。

SRE左移研发服务，为“主动预防”提效降本

智能机型压测降本点：并行多机型压测，兼顾性能和成本

智能机型压测：FinOps 理念在 CI 阶段的落地

SRE 左移研发服务，旨在将运维视角前置到研发阶段，以期实现"主动预防"并达到提效降本的目的。此次分享将重点阐述智能机型压测在实现性能与成本兼顾方面的实践。

压测流程：在 CI 阶段早期融入 FinOps 理念

传统压测往往侧重于性能瓶颈与容量短板的识别，而较少关注成本因素。然而，资源成本在规模化应用中会迅速放大，因此，在持续集成（CI）阶段，将 FinOps 理念引入早期研发服务至关重要。

- **痛点认知：** 多数业务团队在压测中鲜少考虑成本，而作为资源管理者，SRE 团队深知资源成本倍增后的巨大影响。
- **能力赋能：** SRE 团队对现有资源池具备掌控能力，能够识别可用的各种资源类型。
- **降本思路：**
    - **匹配阶段**
        - 根据压测模块对 CPU 和内存的机型要求，从节点池获取匹配的机型列表。
    - **准备阶段**
        - 调整模块的 YAML 文件，设置节点亲和性标签。
        - 节点池按照匹配机型申请机器，将其加入服务器集群并设置相应标签。
    - **部署阶段**
        - 根据压测版本需求部署服务器环境。

数据驱动决策：通过 AI 智能分析优化机型选择

SRE 团队通过平台化能力，在业务方提供压测用例并明确 CPU 核心数和内存预期后，能够匹配互娱资源池中满足需求的多种设备类型（如 Intel 或 AMD 架构）。

- **并行压测：** 改变单一机型压测的传统模式，例如不再固守 S8 这种"锚点机型"，而是采用并行多机型压测。
- **AI 分析：** 结合测试用例，分析不同机型的性能变化，并提供以下关键洞察：
    - **机型性能对比（SA5.4XLARGE64 vs S9.4XLARGE64）：**
        - **综合结论：** 两种机型各有优劣，需根据具体业务需求权衡。
        - **性能方面：**

- **CPU 使用率：** S9.4XLARGE64 表现更优，性能提升约 75%。
- **内存使用：** SA5.4XLARGE64 表现更优，性能提升约 1.5%。
- **ds fleet 进程数：** S9.4XLARGE64 表现更优，平均进程数高出约 60%。
- **ds fleet CPU 核数：** S9.4XLARGE64 表现更优，平均值高出约 50%。
- **DSA 耗时统计：** S9.4XLARGE64 在所有指标上均表现更优，尤其在 PlayGameSession（提升约 98%）和 TerminateProcess（提升约 94.6%）方面提升明显。
- **内存稳定性：** SA5.4XLARGE64 表现更优，内存使用更为平稳，波动较小。

- **成本方面：** SA5.4XLARGE64 的成本比 S9.4XLARGE64 高，具体数值待定。

- **选择建议：**
  - 若重视内存稳定性与成本控制，推荐 SA5.4XLARGE64。
  - 若重视 CPU 性能、进程数、CPU 核数和 DSA 耗时，推荐 S9.4XLARGE64。
  - 最终决策需基于具体业务需求和优先级。

- **单用户成本评估：** 报告还会结合整个互娱大盘的品类数据，分析业务的单用户成本，并评估其在同品类中的位置（中游、上游或下游）。这有助于研发团队客观判断是否应通过逻辑优化来降低单用户成本，或选择新的机型以达到更优效果。

通过这些客观数据，研发团队能够在项目初期即对机型选择做出更明智的判断，从而在提升性能的同时有效控制成本，实现 SRE "左移研发服务"的价值。

SRE右移持续运营，为“主动防御”对症下药

混沌工程+AI：充分结合品类故障案例和业务知识库，通过信息流随时完成混沌实验和分析

SRE 右移持续运营：混沌工程与 AI 结合

我们正在从传统的"左移防范"——即在研发环节前端介入，进行主动预防——向"持续运营"阶段的"右移"演进，旨在通过主动防御机制，实现对症下药。过往的容灾演练多依赖固化的检查清单，这种方式覆盖面有限，且由于演练时间预设，参与者可能因提前准备而缺乏真实感，难以模拟真实故障场景。我们希望在业务负责人不知情的情况下，随时随地进行异常注入，以提升演练的拟真度。为此，我们引入了**混沌工程与 AI 相结合**的模式，旨在通过信息流服务，整合品类故障案例和业务知识库，以自动化方式落地和分析混沌实验。

一、混沌工程与 AI 结合的工作流程

这一工作流程旨在利用我们长期积累的行业经验和数据资产，特别关注那些开源工具或平台因缺乏具体业务情境而无法覆盖的故障类型。我们致力于将业务自有的故障场景也纳入混沌工程的范畴，从而实现更完整的韧性测试。

- **Prompt 输入与智能体响应：**
    - 用户（或系统）输入 Prompt，明确实验目标，如"游戏品类"、"游戏名称"、"实验场景"、"实验集群"和"命名空间"等。
    - "混沌工程智能体（MCP 协议）"——我们的 AI 智能体（bkchat）——接收 Prompt 后，依据**"业务知识库"**（包含服务链路、故障因子、故障原子等信息）和**"混沌工程知识库"**（涵盖故障案例、同品类故障共性等）进行智能处理。

- **实验生成与资源确认：**
    - 智能体生成初步的混沌实验方案，并调用"tool"进行方案的二次确认，确保其符合业务实际情况。
    - 智能体进一步查询 Kubernetes（K8S）资源，以确定混沌实验的"爆炸半径"，并同样利用"tool"完成确认。
    - 基于 K8S 资源信息，智能体生成具体、随机的实验内容。
- **执行与分析反馈：**
    - 实验方案获得审批后被自动执行，全程由"tool"辅助完成。
    - 实验结束后，智能体对收集到的实验数据进行分析，并利用"tool"进行结果解读。
    - 最终，智能体输出详细的实验报告，明确包含"实验结论"、"性能数据"及"场景指标"。

二、AI 智能体交互与实验案例

通过这种机制，团队成员无需频繁点击页面，只需通过企业内部聊天工具，如企业微信机器人，即可便捷地发起一场容灾演练。

- **交互示例：**
    - 用户直接向@bkchat 智能体提问："业务，体育竞技类游戏，需要针对大厅场景的服务请求链路设计混沌实验工作流"。
    - AI 智能体迅速响应："正在执行工具混沌工程方案生成"，并显示相应参数。
    - 在后续交互中，智能体还会执行"k8s 资源获取"等操作，并显示 cluster、namespace 参数。
- **混沌实验配置（chaos_yaml）：**
    - 智能体生成的 chaos_yaml 配置清晰定义了混沌实验的细节，例如：

```
workflow:
  name: lobby-club-chaos
  entrypoint: lobb-club-chaos
  templates:
```

```
- name: workflow-pod-chaos
  template:
    kind: PodChaos
    spec:
      action: pod-kill
      mode: one
      selector:
        labels:
          app: lobby
- name: workflow-network-chaos
  template:
    kind: NetworkChaos
    spec:
      action: delay
      mode: all
      selector:
        labels:
          app: club
      duration: 180s
      delay: 60ms
```

- 此配置包含了在一个名为 lobby-club-chaos 的工作流中，先执行 workflow-pod-chaos（模拟 Pod 故障）再执行 workflow-network-

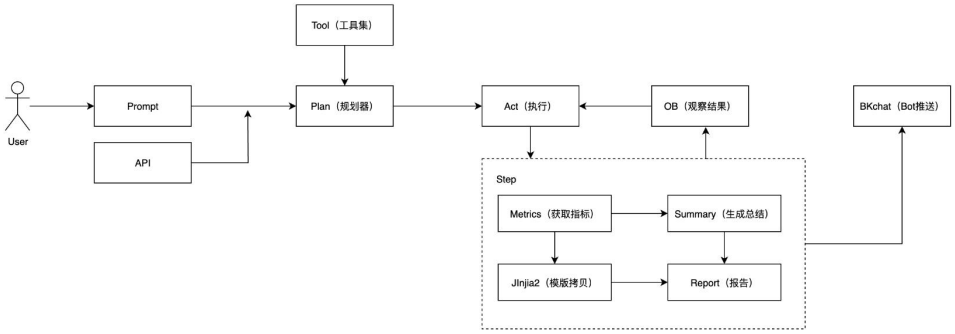
chaos（模拟网络延迟）两种故障注入方式，并设定了故障类型、持续时间、选择器等关键参数。

三、混沌测试报告与 AI 智能分析

每次实验结束后，系统会自动

数据驱动决策，智能赋能研运

智能报告实现流程



智能报告实现流程

各位 SRE 同仁，接下来我们将分析智能报告的实现流程，这也是我们多 Agent 智能体协同工作的基础框架。

核心理念是：**数据驱动决策，智能赋能研运。**

下图展示了一个标准化的流程，旨在通过自动化和智能化的方式，提升研发运营的效率与决策质量。

一、流程总览：智能报告生成与推送

整个流程从用户输入到报告最终交付，围绕"智能报告实现"的核心目标，划分出以下关键环节：

- **用户与输入：**
    - 流程启动于用户的需求，可以是直接的 **Prompt (提示)**，或是通过 **API** 接口触发。
- **智能规划 (Plan)：**
    - 用户输入将被 **Plan (规划器)** 接收。

- **规划器**会与 **Tool (工具集)** 进行交互，以选择和编排必要的工具来执行后续任务。这意味着规划过程能够根据业务场景动态调度所需能力。

- **执行与观察 (Act & OB)：**
  - **Plan (规划器)**决策后，将指令传递给 **Act (执行模块)**。
  - **Act (执行模块)**在实际操作中会产生 **OB (观察结果)**。
  - **Act (执行模块)**与 **OB (观察结果)** 之间形成一个反馈闭环，确保执行过程能够基于实时观测进行动态调整和优化。

- **报告生成核心步骤 (Step)：**
  - 这一阶段是智能报告的核心生成引擎。
  - **Metrics (获取指标)：** 率先进行数据的收集与关键指标的提取。
  - **Summary (生成总结)：**基于获取到的指标，系统会自动生成一份摘要总结。此总结也将作为 **OB (观察结果)** 的一部分，反馈至执行与规划环节。
  - **Jinja2 (模版拷贝)：**智能报告的生成依赖预设的 **Jinja2 模板**。系统会将获取到的数据填充至模板中。
  - **Report (报告)：** 最终输出标准的专业报告。

- **报告输出与推送：**
  - 生成的 **Report (报告)**将通过 **BKchat (Bot 推送)** 机制，发送给最终用户或相关自动化系统。

二、能力注册与平台化

我们的平台不仅集成了混沌工程中自带的原生故障注入能力，更能够将内部平台化的业务原子功能，例如基于业务场景的特定操作，注册为 **MCP (原子能力)**。这些能力的注册中心允许 **Plan (规划器)** 在需要时进行调用，极大地丰富了可编排的能力集合：

- 创建演练活动原子：负责接收演练对象、注入指标等信息自动生成演练活动，实现演练活动的快速编排与执行；
- 获取指定业务演练活动信息原子：负责接收演练业务信息，对相关的演练活动进行检索返回，以便对目标演练活动进行操作；
- 获取演练活动执行任务状态和日志信息原子：按照接收演练活动信息，精准返回演练当前的执行状态、执行日志等，以便快速定位执行层面的问题；

- 填入演练任务执行结论原子：提供演练活动执行结论回填接口，实现对于演练结果的精准记录；
- 获取演练历史稳态指标和监控告警原子：基于演练任务自动获取关联的稳态指标和告警信息等以实现对于故障注入效果的精准量化；
- 获取演练历史信息原子：支持按照时间、业务等多维度获取，演练历史信息，实现对于演练历史数据的智能分析。

三、灵活分工与协同

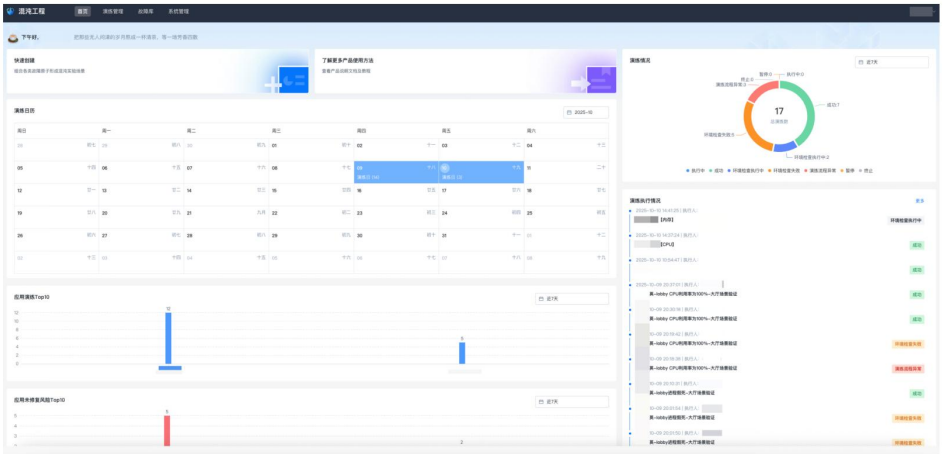
在我们的多 Agent 体系中，每个智能体都具备清晰的职能划分。例如：

- 在混沌工程场景中，特定的智能体被专门设计用于执行混沌工程任务。
- 此智能体在完成任务后，会调用智能报告智能体，生成并输出详尽的实验报告。

这种细致的分工确保了各 Agent 智能体能够高效协同，共同完成复杂的研运任务，从而促进整个系统的智能化决策和运营。

数据驱动决策，智能赋能研运

混沌工程系统截图



混沌工程系统演示

这张幻灯片以"数据驱动决策，智能赋能研运"为核心，通过展示混沌工程系统的用户界面截图，具体阐述了如何利用混沌工程提升系统韧性并支持运维工作。

混沌工程系统概览

- **平台入口与导航：**

- 主界面顶部展示"混沌工程"的 Logo，并提供"首页"、"演练管理"、"故障库"和"系统管理"等导航选项，标志着这是一个混沌工程平台。
- 主页面的问候语后，设有"快速创建"功能，使用户能够便捷地组合各种故障原子以构建混沌实验场景；同时，还辅以"了解更多产品使用方法"的链接，引导用户深入学习产品功能与使用指南。

- **演练调度与统计：**
  - **演练日历：** 界面清晰呈现了以日历形式组织的演练活动，例如，标注了"初七 29"、"初八 30"等日期，并突出显示了存在演练安排的日期。具体的演练数据显示，2025 年 10 月 09 日（周四）共有 14 场演练，而 2025 年 10 月 10 日（周五）则有 3 场演练，使用户能够一目了然地掌握混沌工程演练的日程安排。
  - **近 7 天演练概览：** 日历右侧的"甜甜圈图"直观展示了近 7 天内的演练统计，总计 17 次演练。图例详尽标示了各项演练状态及其对应的数量，其中包括：7 次"成功"，3 次"演练流程异常"，5 次"环境检查失败"，以及 2 次"环境检查执行中"；同时，系统中报告"暂停"、"执行中"和"终止"的演练均为 0 次。
- **演练执行详情：**
  - 界面下方以时间倒序排列的列表记录了详细的演练执行日志，涵盖了时间戳、执行者及具体演练内容和执行结果。
  - **示例如下：**
    - 2025-10-10 14:41:25 | 执行人: [内容] - 环境检查执行中
    - 2025-10-10 14:37:24 | 执行人: [CPU] - 成功
    - 2025-10-10 10:54:47 | 执行人: - 成功
    - 2025-10-09 20:37:01 | 执行人: 英-lobby CPU 利用率为 100%-大厅场景验证 - 成功
    - 2025-10-09 20:30:18 | 执行人: 英-lobby CPU 利用率为 100%-大厅场景验证 - 环境检查失败
    - 2025-10-09 20:19:42 | 执行人: 英-lobby CPU 利用率为 100%-大厅场景验证 - 演练流程异常

- 2025-10-09 20:18:38 | 执行人: 英-lobby CPU 利用率为 100%-大厅场景验证 - 环境检查失败
- 2025-10-09 20:10:31 | 执行人: 英-lobby 进程假死-大厅场景验证 - 成功
- 2025-10-09 20:01:54 | 执行人: 英-lobby 进程假死-大厅场景验证 - 环境检查失败
- 2025-10-09 20:01:50 | 执行人: 英-lobby 进程假死-大厅场景

数据驱动决策，智能赋能研运

混沌工程系统截图

故障库：丰富的故障注入场景

这张幻灯片展示了混沌工程平台的故障库界面，汇集了多种可用于混沌实验的故障注入场景，旨在通过数据驱动的决策和智能化的研运，提升系统的韧性与稳定性。

经验库：快速引用演练专家经验

系统提供标准化演练经验库，支持将零散的专家操作固化为平台统一管理的数字资产。通过封装故障原子与流程编排的成为开箱即用模板，用户无需配置复杂参数即可一键引用，大幅降低演练门槛，通过经验中预置熔断策略和稳态假设，确保基于该经验发起的每一次演练都是在安全可控的范围内的，防止乱演练导致生产事故。

核心功能概览

平台的核心是一个故障库界面，汇集了多种可用于混沌实验的故障注入场景，帮助团队主动发现系统中的潜在弱点。

- **系统导航栏：**幻灯片顶部展示了"混沌工程"系统的导航栏，涵盖"首页"、"演练管理"、"**故障库**"（当前选中项）和"系统管理"等模块，清晰呈现了平台的整体结构和核心功能。
- **故障分类侧边栏：**页面的左侧提供了故障分类导航，便于用户快速筛选和定位故障类型，包括：
    - **全部：**默认展开，包含"主机"和"Kubernetes"两大类。
    - **品类故障：**针对特定业务场景设计的故障类型，支持高度定制化。
- **故障列表：**页面的主体区域以直观的卡片形式展示了丰富的故障列表，每张卡片都详细描述了一种可注入的故障类型，具体信息包括：
    - **故障名称：**统一以"混沌工程-"开头，清晰标识故障内容，例如"CPU 故障注入"、"网络包乱序"、"进程假死"等。
    - **对象：**指明故障作用的实体，例如"主机"。
    - **类型：**说明故障的实现机制或所使用的工具，例如"cy-chaos"、"sub_process"、"gse_agent"。

丰富多样的故障注入场景

平台集成了广泛的故障注入能力，不仅覆盖了基础架构层面，更深入到业务特定场景，以确保全面性。

- **硬件/资源相关故障：**涵盖了底层资源层面的各种失效模式，例如：
    - 混沌工程-CPU 故障注入 (对象:主机, 类型:cy-chaos)
    - 混沌工程-内存故障注入 (对象:主机, 类型:cy-chaos)
    - 混沌工程-磁盘故障注入 (对象:主机, 类型:cy-chaos)
    - 混沌工程-CPU 利用率高 (对象:主机, 类型:cy-chaos)
    - 混沌工程-内存利用率高 (对象:主机, 类型:cy-chaos)
    - 混沌工程-磁盘使用率高 (对象:主机, 类型:cy-chaos)
    - 混沌工程-磁盘 IO 高 (对象:主机, 类型:cy-chaos)
- **网络相关故障：**模拟网络层面的不稳定性和异常情况，例如：
    - 混沌工程-网络故障注入 (对象:主机, 类型:cy-chaos)
    - 混沌工程-网络包乱序 (对象:主机, 类型:cy-chaos)
    - 混沌工程-网络延迟 (对象:主机, 类型:cy-chaos)
    - 混沌工程-网络丢包 (对象:主机, 类型:cy-chaos)

- 混沌工程-网络 DNS 解析异常 (对象:主机, 类型:cy-chaos)
- 混沌工程-网络 DNS 不可访问 (对象:主机, 类型:cy-chaos)

- **进程/应用程序相关故障：** 针对应用和进程层面的异常行为进行模拟，例如：
  - 混沌工程-进程假死 (对象:主机, 类型:cy-chaos)
  - 混沌工程-进程退出 (对象:主机, 类型:cy-chaos)
- **平台/工具集成操作：** 融合了现有运维工具的能力，拓展了故障注入的广度，例如：
  - 标准运维(SOPS)-子流程 (对象:主机, 类型:sub_process)
  - 作业平台(JOB)-快速执行脚本 (对象:主机, 类型:gse_agent)

可视化演练流程编排

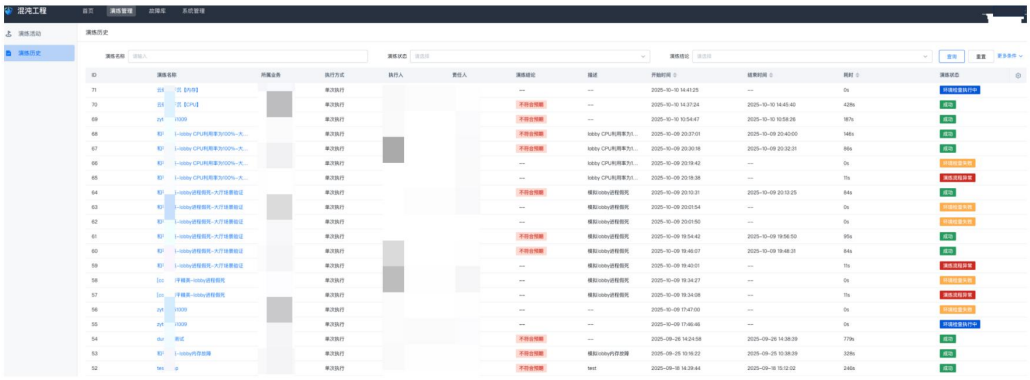
系统支持基于 GUI 的可视化流程编排能力，通过"低代码"交互方式降低混沌工程的使用门槛。用户无需编写复杂的底层脚本，只需通过简单的拖拽操作，即可将标准化的"故障原子"组装成完整的演练流程。

业务特性化品类注入

相较于业界通用的开源解决方案，该混沌工程平台加入了**业务特性化的品类故障注入能力**。例如，针对网络游戏等具有定制化场景的业务，平台能够提供用户自定义的注入能力。这确保了在不同业务场景下，团队都能进行有针对性的混沌实验，从而识别和解决业务风险，提升系统的韧性。

数据驱动决策，智能赋能研运

混沌工程系统截图



演练历史记录：数据驱动的持续改进

这张幻灯片展示了混沌工程系统的核心功能界面——演练历史记录。这不仅是一个简单的日志列表，更是我们通过数据驱动方法提升研发与运维智能化水平的基石。

混沌工程作为保障系统韧性的重要手段，其核心模块在此一览无余：

- **主体导航**：清晰呈现"混沌工程"为主模块，下辖"首页"、"演练管理"、"故障库"和"系统管理"等子模块，当前页面正是"演练管理"下的"演练历史"。

这张幻灯片最核心的部分，就是那份详尽的混沌工程演练历史列表。它犹如实验报告的索引，将每一次混沌实验的关键信息完整地呈现出来，为后续的数据分析与决策提供了坚实的基础，具体包括以下丰富信息：

- **演练基本标识**：
  - **ID**：每场演练的唯一编号，便于精准追溯。
  - **演练名称**：直观描述演练内容，例如"云沉【内存】"、"E-lobby CPU 利用率为 100%-大…"、"E-lobby 进程假死-大厅场景验证"等，深刻反映了各种故障注入与压力测试场景。
  - **所属业务**：明确标识演练所针对的业务系统或组件。
  - **执行方式**：如普遍采用的"单次执行"模式。
  - **执行人与责任人**：明确演练的发起者与结果负责人，确保权责清晰。
- **演练结果与状态概览**：
  - **演练结论**：对演练结果进行定性评估，以颜色标签直观展示："不符合预期"标红警示，"成功"则以绿色彰显，帮助我们快速识别问题与成就。
  - **演练状态**：标签化展示演练的实时或终结状态，如"环境检查执行中"（蓝色）、"成功"（绿色）、"环境检查失败"（橙色），以及"演练流程异常"（红色），便于我们迅速掌握演练进展与潜在问题。
  - **描述**：对演练内容进行补充说明，如"lobby CPU 利用率为 1%…"、"模拟 lobby 进程假死"、"模拟 lobby 内存故障"，进一步精细化演练场景。
- **时间维度与耗时分析**：
  - **开始时间与结束时间**：精确到年-月-日 时:分:秒的时间戳数据，为后续的时间序列分析提供了依据。
  - **耗时**：直观展示演练持续时长，如"0s"、"428s"、"187s"，帮助我们评估演练效率。

此外，为了实现高效的数据查询与管理，系统还提供了：

- **灵活的筛选与查询功能**：列表上方预设了"演练名称"、"演练状态"、"演练结论"等筛选条件输入框和下拉菜单，配合"查询"、"重置"及"更多条件"按钮，极大地提升了用户查找特定演练记录的便捷性。

这张幻灯片的核心内容正是以**数据表格**的形式，承载了混沌工程"演练历史"的各项数据，它的具体列结构如下：

- ID
- 演练名称
- 所属业务
- 执行方式
- 执行人
- 责任人
- 演练结论（通过颜色标签直观展示，如"不符合预期"为红色标签，空值不显示）
- 描述
- 开始时间（精确至年-月-日 时:分:秒，例如"2025-10-10 14:41:25"）
- 结束时间（精确至年-月-日 时:分:秒）
- 耗时（以秒为单位显示，如"0s"，"428s"，"187s"）
- 演练状态（同样通过颜色标签直观展示，例如"环境检查执行中"为蓝色标签，"成功"为绿色标签，"环境检查失败"为橙色标签，"演练流程异常"为红色标签）

这份详细的演练历史不仅记录了我们为提升系统鲁棒性所做的努力，更是我们持续改进、优化系统韧性的数据基石。

演练操作日志：全链路操作审计

操作日志提供对所有操作的全生命周期记录与追踪能力。系统自动捕获并持久化存储用户的关键行为数据，确保每一次故障注入、策略变更或演练执行都有迹可循。

Linux&Windows:支持对多种操作系统进行故障注入

支持 Linux 与 Windows 两大主流操作系统的故障注入，通过统一的管控平面，用户可灵活编排针对不同操作系统 CPU 满载、进程异常、网络丢包等多种故障场景，助力精准识别系统隐患，全方位验证系统保障业务连续性与高可用的能力。

GPU 故障注入：支持对 GPU 进行故障注入

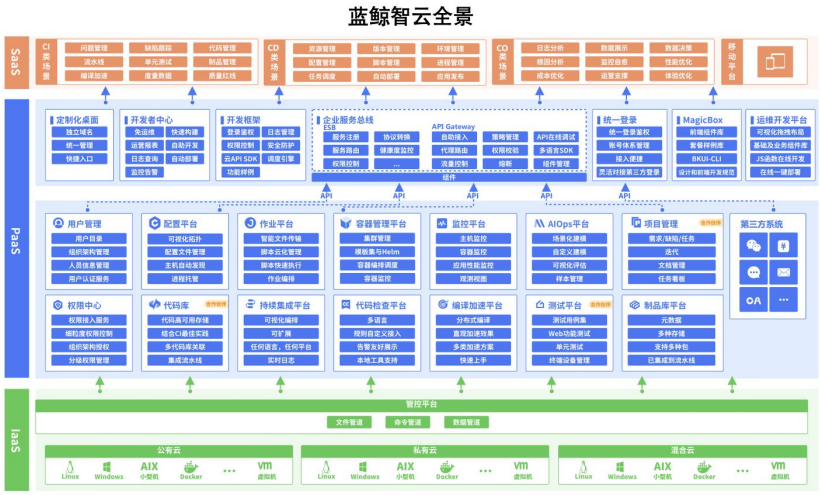
支持对底层 GPU 硬件进行精细化的异常模拟，涵盖 GPU 温度过高、显存耗尽、使用率高等关键故障模式，帮助 SRE 团队提前发现图像渲染、视频转码等高负载业务在硬件异常下的系统瓶颈，确保在算力异常时仍具备优雅降级与快速自愈的能力；

云服务故障注入：支持对主流云服务商的托管服务进行故障模拟故障注入 涵盖数据库主备切换、对象存储异常、API 限流及可用区隔离等关键故障场景，帮助 SRE 团队提前验证混合云架构的容错能力与应急策略的有效性，确保在云服务异常时业务仍能保障连续性并实现快速故障转移。

（三）总结及展望

平台工程底座：蓝鲸智云全景

坚实的平台工程底座



本次分享聚焦于如何在 SRE 实践中实现"左移"与"右移"的无缝衔接，其核心在于构建一个平台工程底座。我们团队历经十余年，持续深耕于此，并基于"蓝鲸智云全景"这一平台工程架构，实现了平台能力的封装与应用场景的适配。

这张幻灯片展示了蓝鲸智云的平台工程生态，它涵盖了 SaaS、PaaS 和 IaaS 三个核心层次，致力于提升 SRE 团队的效率和稳定性保障能力。

- **SaaS 层：直观呈现了平台面向开发运维全生命周期的服务能力。**

- **CI 类场景（持续集成）：** 涵盖了从问题管理、流水线、编译加速、缺陷跟踪到单元测试、度量数据、代码管理、制品管理以及质量红线的全链路集成优化。
- **CD 类场景（持续部署）：** 专注于部署环节的效率与安全，包括资源、配置、任务调度、版本、脚本、自动部署、环境管理、进程管理和应用发布等功能。
- **CO 类场景（持续运营）：** 侧重于系统运行后的持续优化与洞察，提供日志分析、根因分析、成本优化、数据展示、监控自愈、运营支撑、数据决策、性能优化和体验优化等能力。
- **移动平台：** 独立于 CI/CD/CO 场景，旨在为移动应用的开发与管理提供专属能力支持，图标形象地描绘了移动应用开发和文件管理的场景。

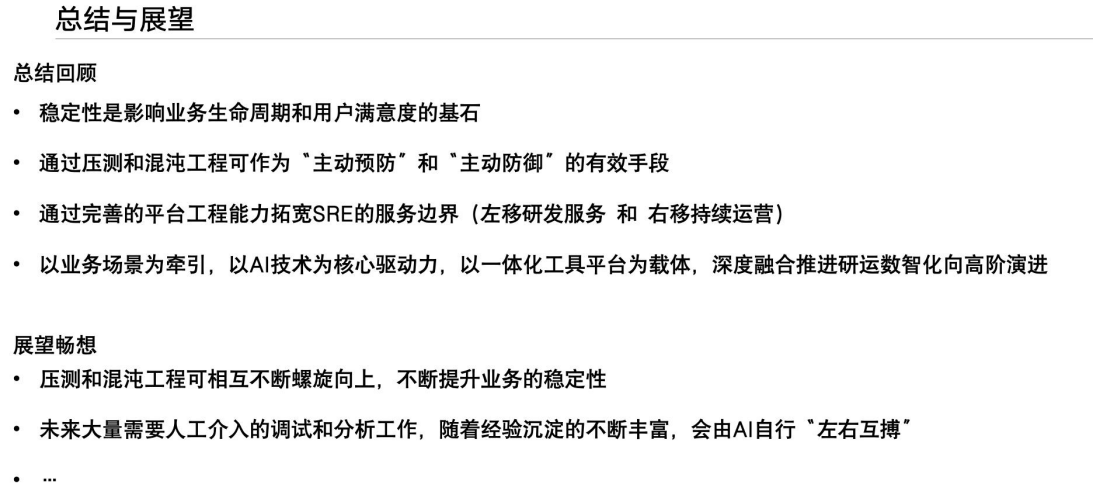
- **PaaS 层：作为平台的核心能力集合，通过开放的 API 接口对外提供服务，确保其灵活性与可扩展性。**
  - **门户能力：** 定制化桌面提供独立域名、统一管理和快捷入口，为用户提供个性化工作区。
  - **开发者效率工具（开发者中心）：** 赋能开发者，提供免运维、运营报表、日志查询、监控告警、快速构建、自助开发及自动部署等功能，加速研发进程。
  - **通用开发组件（开发框架）：** 包括登录鉴权、权限控制、云 API SDK、功能样例、日志管理、安全防护及调度引擎，为应用开发提供坚实基础。
  - **集成层（企业服务总线 ESB）：** 作为核心集成枢纽，实现服务注册、路由、权限控制、协议转换及健康度监控等关键能力。
  - **API 管理（API Gateway）：** 提供自助接入、代理路由、流量控制、熔断、策略管理、权限校验、API 在线调试及多语言 SDK，构筑健壮的 API 生态。
  - **统一认证（统一登录）：** 统一登录鉴权，实现账号体系管理，便捷接入第三方登录。
  - **前端开发支持（MagicBox）：** 提供前端组件库、套餐样例库、BKUI-CLI 以及设计和前端开发规范，提升前端开发效率与一致性。

- **低代码/无代码开发（运维开发平台）：** 通过可视化拖拽布局，结合基础与业务组件以及在线 JS 函数开发，实现应用的一键部署。

- PaaS 层还包含一系列核心子平台，共同构建了蓝鲸智云的全面能力：
  - **用户管理：** 提供用户目录、组织架构管理、人员信息管理与用户认证服务。
  - **配置平台：** 实现了可视化拓扑、配置文件管理、主机自动发现与进程托管。
  - **作业平台：** 支持智能文件传输、脚本云化管理、脚本快速执行与作业编排。
  - **权限中心：** 提供权限接入服务、细粒度权限控制、组织架构授权与分级权限管理。
  - **代码库：** 确保代码高可用存储，结合 CI 最佳实践，支持多代码库关联与集成流水线。
  - **持续集成平台：** 具备可视化编排、高度可扩展性，支持任何语言/平台，并提供实时日志。
  - **容器管理平台：** 提供集群管理、模板集与 Helm、容器编排调度与容器监控。
  - **代码检查平台：** 支持多语言、规则自定义接入、友好的告警展示与本地工具支持。
  - **监控平台：** 覆盖主机监控、容器监控、应用性能监控与观测视图，提供全面洞察。
  - **编译加速平台：** 实现了分布式编译，提供直观的加速效果、多

04 总结与展望

总结与展望：SRE 的高阶演进与未来图景



SRE 工作的核心与未来发展，可以概括为以下几个关键点：

一、协同增效：压测与混沌工程的组合拳

- **主动预防与防御**：压力测试（压测）和混沌工程两者结合，能实现"1+1>2"的协同效果，显著提升系统稳定性。
- **循环优化机制**：压测发现的性能瓶颈将作为混沌工程的输入项，而混沌工程的对抗与模拟结果又将反哺压测能力，形成一个螺旋式上升的优化闭环，持续推动业务稳定性的提升。

二、拓展边界：平台工程赋能下的 SRE

- **服务边界左移与右移**：通过完善的平台工程能力，SRE 的服务边界得以拓宽，向研发前置（左移）和持续运营（右移）延伸。
- **研运数智化高阶演进**：以业务场景为牵引，以人工智能（AI）为核心驱动力，依托一体化工具平台，SRE 将深度融合并推进研发运营的数字化、智能化向更高阶演进。

三、智能化未来：AI 自主"左右互搏"

- **经验沉淀与自动化**：随着运行经验的不断积累和丰富，未来大量需要人工介入的调试和分析工作，将逐步由 AI 自主进行"左右互搏"式的验证与优化，从而大幅减少人工干预。
- **业务层面的韧性思考**：
  - **用户体验指标**：在多人同屏等高并发场景下，通过压测或混沌工程，可以发现客户端帧数直线下滑，甚至卡顿至幻灯片模式等严重影响用户体验的问题。
  - **业务与技术协同**：一旦发现这类指标已低于用户体验的最低门槛，SRE 需与业务方紧密沟通，共同制定解决方案。例如，针对多人同屏游戏场景，可通过以下方式进行优化：
    - **用户自主选择**：提供选项，允许用户在极端卡顿情况下优先保障连接不中断，或选择性屏蔽部分玩家显示。
    - **客户端被动优化**：实施视野裁剪或多边形渲染分层等技术，在用户无感知的情况下提升性能。
    - **逻辑分线**：通过后端逻辑优化，在不影响用户感官体验的前提下，减少单屏幕实际承载的玩家数量，以降低网络负载和渲染压力。
  - **对缺陷的区分**：SRE 的混沌工程更多聚焦于系统在复杂、有损场景下的韧性表现，而非明显的程序错误（Bug）。明显的缺陷应在测试环节被发现并修复。混沌工程旨在寻找那些介于"正常运行"与"明显故障"之间的，非零一判定的性能衰退或体验有损问题。

四、混沌、应急攻防一体化联动

此外，团队还规划构建"攻防一体"的自动化"红蓝对抗"体系，打破平台间的壁垒，实现混沌工程（攻）与应急平台（防）的互通和流程编排，在发起针对性的故障注入后，监控系统捕捉到异常后自动触发早已配置好的应急处置预案。

- 发起攻击（红军）：混沌工程根据"演练计划"自动触发故障注入；
- 感知响应（蓝军）：监控捕获异常后，在应急中心创建应急事件并启动计时；
- 自动防御（蓝军）：应急平台根据预设策略，自动匹配应急预案；
- 闭环验收（裁判）：混沌平台持续监听业务指标，校验系统是否在目标恢复时间内自动恢复；
- 演练报告：自动生成演练复盘报告，全景展示演练时序与防御效果，输出量化的系统韧性报告。

5.4.5 携程全链路压测平台的建设与运营

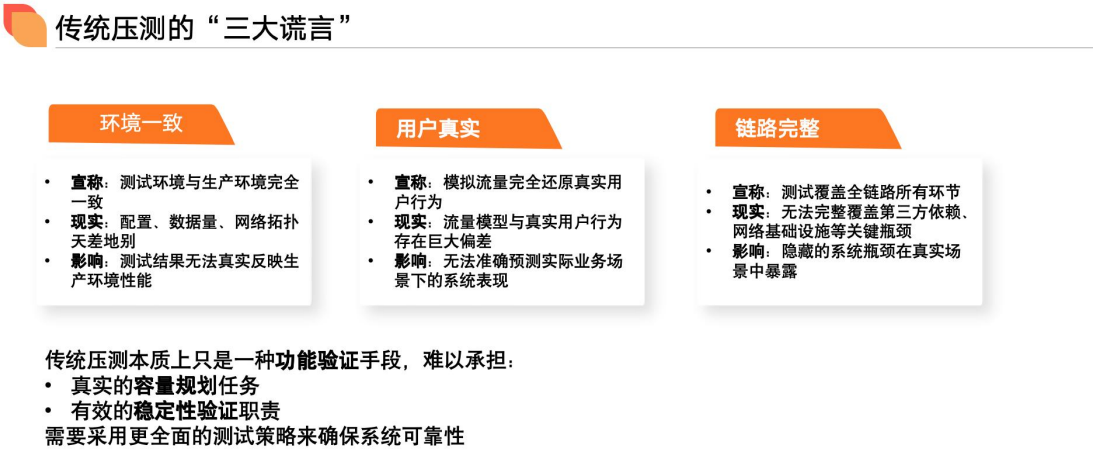
SRE Elite 精选原因

携程作为国内领先的在线旅游服务平台，业务覆盖酒店、机票、火车票、门票等多条核心产品线，服务数亿用户。在线旅游行业在构建高可用系统时面临独特挑战：业务场景高度复杂，涉及跨部门、跨服务的超长调用链路；流量特征极度不均衡，节假日抢票、热门演唱会门票秒杀等场景下流量可达日常数十倍；对交易准确性要求极高，任何支付、库存环节的问题都可能直接影响用户体验和业务收入。

面对这些挑战，携程将全链路压测作为保障系统稳定性的核心手段，构建了一套完整的生产环境压测体系。本案例基于真实生产环境和亿级用户规模，系统分享了从压测账号体系设计、流量染色与透传、数据/流量/消息三层隔离机制，到压测计划管理与文化建设的全链路实践经验。其"在生产环境安全压测"的方法论，以及"让业务方自助完成压测"的平台化思路，对其他复杂业务链路、高并发交易场景的系统同样具有重要借鉴价值。

背景与挑战

传统压测的三大谎言



谎言一：环境等价

- **宣称：** 传统压测声称测试环境可以等同于生产环境。
- **现实：** 实际工作中，我们都能深刻体会到，无论是机器配置、服务器实例规格、数据量规模，还是网络拓扑结构乃至基础设施的部署模式，测试环境与生产环境之间都存在显著差异。

- **影响：**这种差异导致在测试环境获得的压测结果，难以真实推断生产系统的性能表现，更不用说精准识别潜在的性能瓶颈。

谎言二：用户真实

- **宣称：**传统压测试图通过模拟流量来完全还原真实用户行为。
- **现实：**如果我们仍然采用传统压测方式，通过人工构建流量模型，则很难全面覆盖业务链路中每一个细微的分支流程。
- **影响：**这种基于报文的压测，所获得的测试结果难以有效验证系统在真实业务场景下是否能满足性能需求。

谎言三：链路完整

- **宣称：**传统压测声称能够覆盖全链路的所有环节。
- **现实：**无论是在测试环境中进行单集群或单服务的压测，其最大的弊端是无法实现链路的完整覆盖。例如，它往往无法触及第三方依赖服务、负载均衡设备或核心网络基础设施等关键组件。
- **影响：**然而，在流量高峰或极端大流量涌入时，这些传统压测未能覆盖、看似边缘的基础服务，恰恰可能成为系统性能的关键瓶颈点。

传统压测的本质与局限

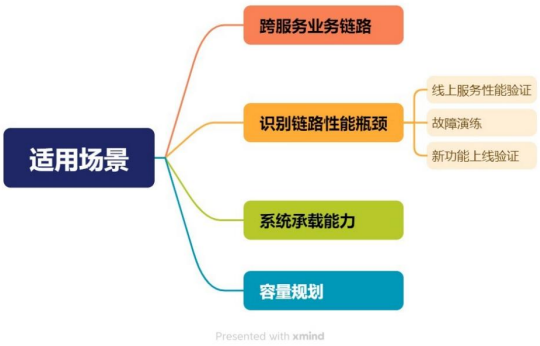
- **本质：**传统压力测试本不过是一种功能性验证手段。
- **难以承担：**它难以胜任真实的容量规划任务，更无法有效地承担系统稳定性验证的重任。
- **改进方向：**因此，我们深刻认识到，为了确保系统的真正可靠性，唯有采取更为全面、更具前瞻性的测试策略才能有效应对。

正是基于对传统压测这些深层次的分析和理解，我们才能清晰地认识到，为何转向全链路压测乃至更全面的混沌工程实践，对于构建弹性可靠的生产系统而言，是如此必要和关键。

全链路压测的定义与价值

全链路压测适用场景及解决的问题

基于真实的业务场景，在生产环境最贴近用户行为的入口侧，模拟海量用户请求对整个系统进行压力测试，从而识别业务链路中的性能瓶颈。



全链路压测是一种在生产环境下进行的全面性能测试方法，具有以下核心特征：

- **压测环境：**
    - 在真实的生产环境下执行，确保测试结果的准确性和有效性。
- **压测方式：**
    - 基于真实的业务场景，模拟用户行为，从最能贴近用户行为的入口侧进行施压。
    - 压测报文的构造力求高度模拟真实用户报文，覆盖其离散度、内容和广度。
    - 通过这种方式对系统进行直接压测。

这种压测方式的适用场景及其解决的问题如下：

- **跨服务夸业务部门链路**
    - 覆盖复杂、跨多个微服务或业务系统的端到端业务链路，确保整体流程的稳定性与性能。在携程，我们有两个非常典型的全链路压测场景：一是节假日前的火车票抢票，二是门票业务，尤其是大型演唱会门票秒杀。这类场景链路长、涉及部门和服务众多，包括支付、风控及其他公共基础服务。针对这些复杂业务场景，在生产环境下进行全链路压测，可以有效发现如支付环节等潜在的性能瓶颈，及时识别并解决可能影响扣款和业务稳定的关键问题。

- **识别链路性能瓶颈**
    - **线上服务性能验证**：在生产环境中验证现有服务的性能表现，确保满足 SLA 要求。
    - **故障演练**：在压力下模拟故障场景，验证系统的容错能力、降级策略和恢复机制。
    - **新功能上线验证**：在新功能发布前，验证其在真实流量和压力下的性能影响，避免上线后出现性能问题。
- **系统承载能力**
    - 评估系统在高并发或大数据量下的最大负载能力，确定系统能够支撑的用户量或交易量上限。
- **容量规划**
    - 基于系统承载能力的评估结果，为未来的硬件资源采购、服务扩容、架构优化等提供数据支撑，确保系统能够应对业务增长。

携程全链路压测体系架构

在第二部分，我们将系统地介绍携程整个压测体系的技术架构，深入了解携程如何通过其压测体系来保障系统稳定性、应对高并发挑战并在亿级用户和设备规模下实现全链路的高可用目标。

压测架构全景图

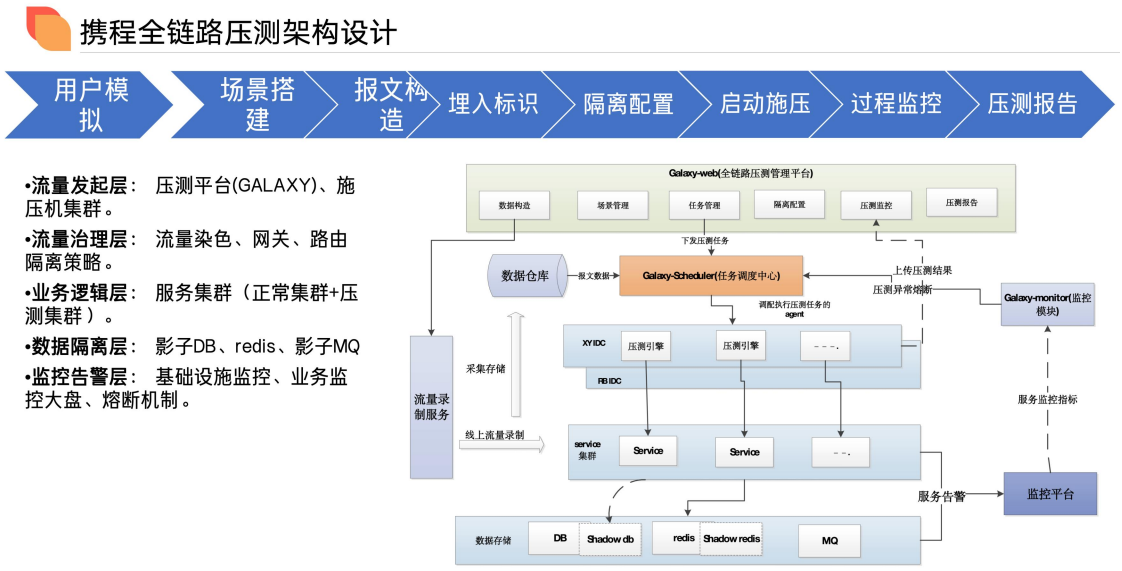


image-20251224145446274

一、用户侧压测流程：七个核心阶段

- **用户模拟：**这是压测的起点，还原真实用户的行为模式，确保压测的有效性。
- **场景搭建：**根据预设目标，构造压测场景，模拟出各种可能的业务流和压力情况，这是策略制定与执行的关键环节。
- **报文构造：**精细化准备压测所需的请求报文，确保请求数据的真实性和多样性。
- **埋入标识：**在发出的每个请求中植入压测特定的标识，为压测流量打上“专属标签”，是后续隔离和追踪的基础。
- **隔离配置：**基于这些标识，配置压测环境与生产环境的流量隔离策略，确保压测活动不对线上业务造成任何潜在干扰。
- **启动施压：**一切准备就绪后，正式触发压测，让系统接受预设的流量洪峰冲击。
- **过程监控：**压测进行中，需进行实时、全面的监控，捕捉系统性能的每一次波动。
- **压测报告：**压测结束后，生成详尽数据报告并进行深度分析，这是总结经验、发现瓶颈、指导优化的关键产出物。

二、架构分层与核心组件：支撑全链路压测的五大基石

全链路压测架构的五个逻辑分层：

- **流量发起层：**
    - **核心构成：**压测平台 (GALAXY)、施压机集群。
    - **职责所在：**负责依照既定场景，不断地生成并向目标系统输送压测流量。
- **流量治理层：**
    - **核心功能及组件：**流量染色、网关、智能路由隔离策略。
    - **职责所在：**确保带有“专属标签”的压测流量能在复杂的系统拓扑中精准穿梭，与正常生产流量有效区分，不偏不倚地流向指定压测环境，实现“精准投送，避免误伤”。
- **业务逻辑层：**
    - **核心构成：**服务集群 (分为正常生产集群与专属压测集群)。

- **职责所在：**承载并处理压测流量的业务逻辑服务，我们通常会策略性地将正常生产集群与专用的压测集群加以区分，以彻底避免压测对生产环境的任何潜在影响。

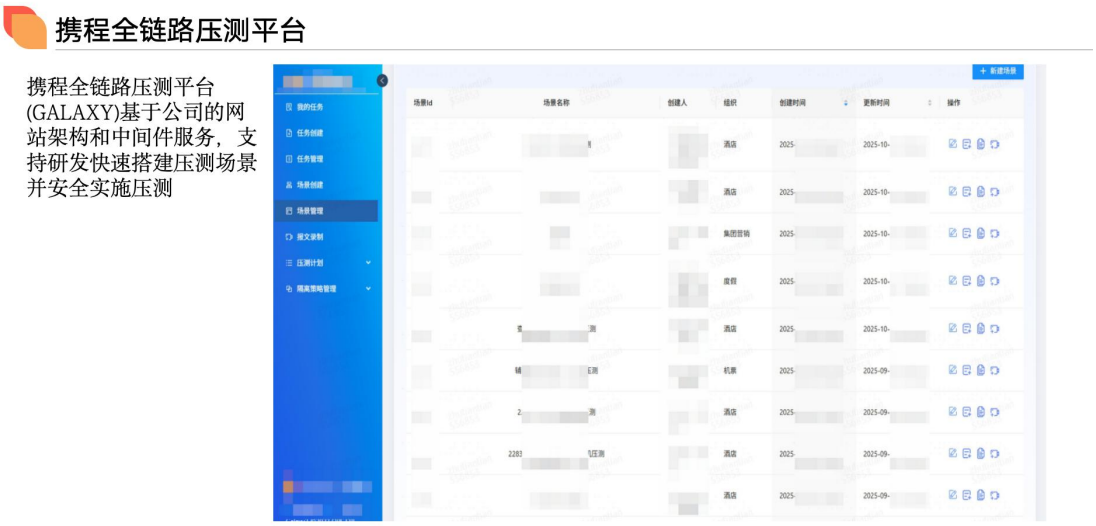
- **数据隔离层：**
  - **核心构成：**影子 DB、影子 Redis、影子 MQ 等。
  - **职责所在：**通过引入“影子”副本或专用实例机制，确保压测过程中产生的所有数据，都被严格隔离在独立的沙盒环境中，绝不污染生产环境的真实数据。
- **监控告警层：**
  - **核心功能及组件：**基础设施监控、业务监控大盘、智能熔断机制。
  - **职责所在：**作为压测过程中的“瞭望塔”，实时、细致地监测各项核心指标，一旦侦测到任何潜在的问题或异常，立即触发预设的告警机制，同时通过熔断机制，在极端情况下自我保护，防止“雪崩效应”。
- **右侧主体：GALAXY 压测平台架构** 中间是具体的系统组成和数据流向，大致是这样：
  - 顶部是 **Galaxy-web 全链路压测管理平台**：包括数据构造、场景管理、任务管理、隔离配置、压测监控、压测报告等功能模块。
  - Galaxy-web 将压测任务下发到 **Galaxy-Scheduler（任务调度中心）**。
  - 调度中心再调度下游的 **压测引擎 / agent**，在不同 IDC（例如 XYIDC、FB IDC）发起压测流量。
  - **流量录制服务**：负责线上流量的录制与回放，采集后的数据进入**数据仓库** / 存储，用于构造压测数据。
  - 下方是 **service 集群**：正常服务 + 压测服务，后端接入影子数据层：
    - DB + Shadow DB
    - Redis + Shadow redis
    - MQ+
  - 右侧是 **Galaxy-monitor（监控模块）**与**监控平台**：接收服务告警、服务监控指标以及压测异常，形成压测过程监控与结果上报。

整体来说，这张图展示的是：从“用户侧压测流程”出发，结合“流量、业务、数据、监控”的分层隔离策略，在底下用 Galaxy 系列组件（web 管理平台、调度中心、压测引擎、流量录制、影子库、监控平台等）实现一套完整的**全链路压测平台架构**。

压测平台管理界面

03 压测实践与踩坑指南

压测场景管理



压测场景列表界面

如界面所示，平台提供了全面的压测场景管理功能：

- **直观的用户界面：** 左侧导航菜单包含“我的压测报表”、“压测场景管理”等关键模块，当前视图高亮显示的是“压测场景管理”，即为一系列已创建压测任务的列表。

- **详细的场景列表：** 主内容区以表格形式展示了压测场景列表，包含“场景名称”、“创建人”、“创建时间”、“更新时间”以及“操作”列，方便用户概览和管理。
- **便捷的操作入口：** 表格中的“操作”列提供了“管理”等功能按钮，允许用户对特定压测场景进行进一步配置和管理。
- **快速检索与创建：** 表格上方设有搜索框和“新建场景”按钮，支持用户快速查找现有场景或创建新的压测任务。

我们希望搭建一个**低技术门槛、易操作、强实用性**的平台，让研发、测试，甚至产品经理，都能基于各自业务**自主配置压测场景**，并在**生产环境中安全地进行施压和实验**。

用户模拟

压测账号体系设计

image-20251224150110705

在正式介绍核心模块之前，我们首先关注模拟真实用户行为这一关键环节。为了尽可能贴近真实世界，携程在压测中采取了如下策略：

- **压测账号体系与安全授权**
    - **真实用户身份模拟：** 我们预先从用户中心系统申请了多达 200 万个压测专用的用户 ID（UID），这些账号在用户中心进行了真实注册，确保其身份与真实用户无异，从而为压测提供一套真实且受控的用户池。
    - **用户登录认证：** 针对涉及创建订单等需要用户登录认证的业务场景，为了确保下游服务（尤其是支付扣款功能）能够将压测账号识别为真实用户，系统会对其进行登录授权认证。

- **业务方管理与申请：** 压测账号的管理由业务方负责。当业务方需要进行涉及订单创建、扣款或支付等场景的压测时，他们会提前向用户中心提交申请，说明需使用的压测账号。
- **授权流程：** 用户中心在审核通过后，会为这些指定的压测账号授权一个登录态 Token。凭借此 Token，压测账号便能以真实用户的身份登录并参与业务流程。
- **安全考量：** 在压测账号的使用管理上，最核心的考量是安全性。

在确保模拟真实性的同时，账号安全与网络安全是用户模拟场景下不可或缺的防护层：

账号安全

我们将账号安全视为重中之重，实施一系列严密的防护措施：

- **登录态时效性控制：** 对模拟用户的登录会话进行严格的时效性管理。这意味着只有在压测活动期间，所生成的登录态才被允许有效。一旦压测结束，这些登录态将立即失效并被回收，彻底杜绝未经授权的持续访问。
- **账密安全管理：** 模拟账户不具备传统的、可显式存储的用户名和密码。从设计之初，这些账户的凭证就处于不透明状态，甚至在某种程度上可以说是不存在的，极大地降低了泄露风险，有效防范了撞库攻击及弱密码漏洞的威胁。
- **风控管理：** 整合独立的风险控制策略与过滤规则，对模拟行为进行实时监控。这使我们能够识别并过滤掉任何潜在的异常或高风险操作，从而在模拟过程中保障账户资产及信息的安全。

网络安全

同时我们在网络层面部署了多重拦截机制：

- **请求源检测：** 在网络接入层实施严格的请求源检测。当前，所有模拟请求都必须源自内部网络环境。任何被识别为来自外部网络的请求，都会在初期被 KIll 掉，确保外部无法利用压测账号发起恶意交互。
- **数据脱敏：** 对所有涉及敏感信息的传输和处理过程进行数据脱敏。这确保了即使在测试、分析或非生产环境中，敏感数据也以匿名或假名形式存在，从而在不影响测试有效性的前提下，保障了用户隐私和数据合规性。

场景搭建

压测场景配置流程

场景搭建流程

核心目标：搭建高效的压测场景

- **明确用户行为与入口**
  - **用户行为界定：** 首先，需要清晰定义压测所针对的用户行为，这是整个压测场景的基石。例如，在酒店预订场景中，用户行为可能从搜索、筛选到下单支付等一系列动作。

- **施压入口识别：**一旦用户行为明确，便能精准定位与其对应的压测入口，通常表现为一个 API 接口或其他服务端点。所有的压力将通过此入口施加，以模拟真实的用户请求。

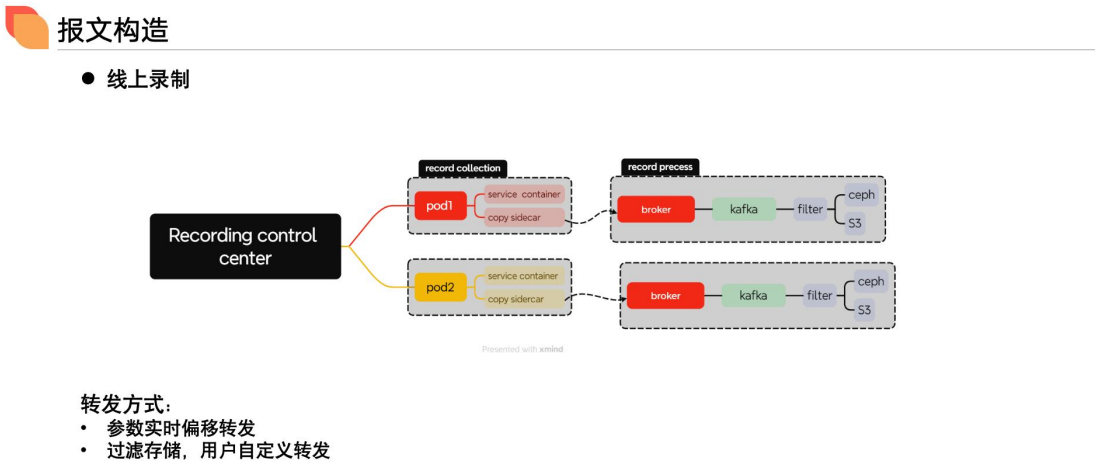
- **链路梳理**
  - **上下文关联核心流程：**链路梳理并非要求业务方描绘完整的业务流程，而是应聚焦于核心交互流程及其上下文传递，识别出其中关键的业务节点。以酒店创单为例，其核心链路可能涉及：
    - 酒店可订查询：确认酒店及房型库存。
    - 下单：提交预订请求。
    - 支付页面跳转与支付方式选择：引导用户完成支付。
    - 扣款：实际资金交易。
    - 回调：支付结果的异步通知。
    - 扣库存：完成库存更新。
  - **子场景构建：**通过将上述核心节点进行上下文串联，便形成了独立的“子场景”。这些子场景是构成复杂业务场景的基本单元。后续的压测报文构造、数据隔离等都将以这些子场景为基础。
- **报文构造与选择**
  - **精细化报文生成：**在确定每个子场景后，下一步是针对这些子场景构造压测报文。我们致力于提供多种方式，使业务方能够构建最接近真实用户行为的报文，确保压测的有效性和准确性。
  - **贴近真实性：**全流程压测的核心在于模拟真实用户行为，因此报文的构造需高度还原用户在实际操作中可能发出的请求格式与内容。
- **隔离策略准备**
  - **场景化隔离配置：**隔离作为全流程压测的重要一环，被融入到场景的搭建过程中。业务方在完成场景搭建后，可根据其特点配置所需的隔离策略。这包括但不限于对数据库（DB）、缓存（Redis）、消息队列、ES 集群或服务集群的隔离设置。
  - **数据隔离独立管理与复用：**尽管隔离策略与具体场景关联，但数据隔离的配置是独立管理的。一个关键优势是，一次隔离配置可以被多个场景复用。例如，某个支付服务所依赖的数据库，由于其作为中台服务会

被所有业务场景访问，因此只需配置一次该数据库的隔离策略，所有相关压测场景均可关联该配置，从而实现高效管理和复用。

- **场景特定修改：**若特定场景对通用隔离策略有特殊需求，业务方可以在该场景内进行局部修改。需强调的是，这类修改仅在该特定场景生效，且仅限于该场景中的某个链路节点的应用访问数据库时的策略调整。

报文构造

报文录制



报文录制流程

一、数据采集

- 在 K8s 环境下，我们通过向服务实例所在的 Pod 注入一个专用 Sidecar 容器实现数据采集。
- 该 Sidecar 的主要职责是监听指定端口的 HTTP 报文，并对其进行录制。
- **录制内容与采样：**可根据需求灵活选择录制特定时间段的报文，例如业务高峰期、低峰期，甚至是特定活动期间的用户请求报文，将其整合形成完整的报文集，以供后续压测使用。

二、数据处理与存储

录制下来的报文会进入预处理流程，以确保数据安全性和可用性：

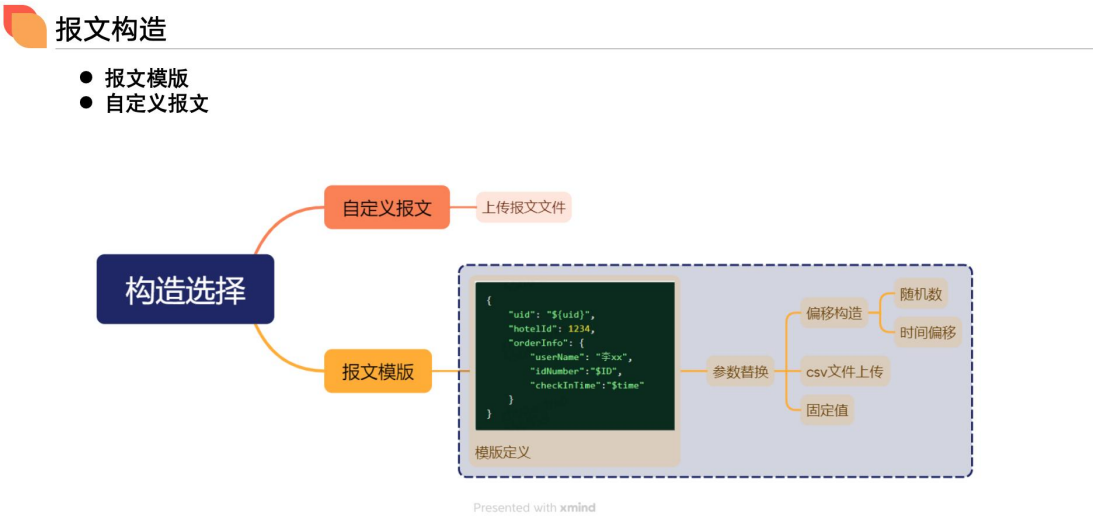
- **消息队列（Kafka/Broker）**：采集到的报文通过 broker（如 Kafka 代理）进入消息队列 Kafka。Kafka 作为缓冲和解耦层，确保了数据生产者和消费者之间的稳定传输。
- **报文过滤（Filter）**：在数据存储之前，报文会经过 Filter 组件进行处理，其中最关键的是对敏感信息进行**脱敏处理**。
- **离线存储**：过滤和脱敏后的报文将被存储到外部存储系统，目前支持 Ceph（分布式存储）或 S3（对象存储服务）。

三、录制报文的再利用与转发

存储后的录制报文可被压测系统高效复用，并支持灵活的转发策略：

- **参数实时偏移转发**：在压测时，系统会从录制报文中提炼出与特定压测入口 API 相关的报文。由于原始录制报文可能包含真实用户信息且已脱敏，系统支持对这些报文进行参数偏移，例如替换脱敏后的 UID 或手机号码为不存在或虚拟的值，从而确保压测报文的真实性与安全性。
- **过滤存储与用户自定义转发**：系统支持对数据进行过滤后存储，并允许用户根据自身业务需求定义个性化的转发规则，这极大地提升了数据的灵活性和定制化能力。

报文模板



报文模板配置

除了录制真实流量，我们还支持通过自定义报文和报文模板的方式来构造压测数据。其中报文模板提供了灵活的参数化能力：

参数替换机制：

- **偏移构造：**支持**随机数**生成任意值，以及**时间偏移**以根据时间戳动态调整数值，这在压测场景中尤为实用。
- **CSV 文件上传：**用户可以将预先准备好的参数值集合通过 CSV 文件导入，系统会依照顺序依次替换模板中的占位符，适用于批量生成测试数据。
- **固定值：**对于不需变化的参数，可以直接指定一个不变的固定值，确保特定字段的稳定性。
- **自定义报文（Direct Customization）：**这种方式提供了一种更为直接和刚性的报文构造方法。用户可以直接上传一个完整的报文文件。系统会直接使用上传的报文进行后续操作。

出入参配置

出入参配置界面

在复杂的业务链路中，服务之间往往需要进行参数传递和数据提取。为此，我们提供了全面的出入参配置能力。

一、入参配置

入参配置用于定义如何为每个请求生成或获取输入参数，我们支持多种灵活的参数来源：

- **固定值**：直接使用预设的静态数值作为参数。
    - **上下文传递**：在服务调用链中，我们将实现 A 服务响应报文中特定字段的值，作为 B 服务的输入参数，从而实现跨服务的参数传递。例如，A 调用 B，B 调用 C，A 的输出可以作为 B 的输入。
    - **随机函数**：为了模拟更真实的环境或进行数据扰动，支持生成随机值作为参数，细分为：
        - **时间偏移**：基于当前时间生成带偏移的数据。
        - **随机数**：生成指定范围内的随机数值。
    - **CSV 上传**：当用户需要使用预定义的大量参数集时，可将包含参数值的 CSV 文件上传。在压测或其他操作中，系统将实时读取这些文件，替换报文中的占位符，实现批量参数化处理。此外，用户还可以自定义脚本来生成参数，以满足更为复杂的逻辑需求。
- **出参配置** 出参配置主要解决如何从服务交互中提取关键信息的问题，以便后续处理或传递给其他服务。我们提供了两种主要的提取方式：
    - **响应报文出参**：从服务请求的响应报文中提取所需的参数。
    - **请求报文出参**：从已发出的请求报文中提取参数。

二、参数表达式输入类型

为了适配不同的报文结构和参数化需求，我们的参数表达式支持以下两种主要类型，并提供丰富的解析能力：

- **模板类型**：当请求报文以模板形式存在时，参数表达式直接填写模板中的占位符名称。
    - **示例报文模板**：
- {

  "id": "${id}",

  "orderId": "${orderId}",

```
  "name": "${name}"
}
```

- **参数表达式**：在这种情况下，参数表达式可以直接填写如 id、orderId 或 name 等，系统会识别并进行替换。

- **非模板类型（Non-template Type）**：当请求报文不是模板形式时，参数表达式则根据 JSON 结构层级，采用 a.b.c 的格式进行填写。

  - **支持 JSONPath**：为了处理复杂或动态的 JSON 结构，我们广泛支持 JSONPath 表达式。当表达式以$开头时，系统会将其视为 JSONPath 进行解析。

  - **示例实际报文**：

- ```
  {
    "orderInfo": {
      "orderName": "TestOrder",
      "orderId": "12345"
    },
    "userInfo": {
      "name": "JohnDoe"
    }
  }
  ```

  - **替换示例**：若要替换 orderName 字段的值，可以使用的表达式是 orderInfo.orderName。当使用 JSONPath 时，例如 $.head[?(@.name=="uid")]，可以精确地定位到报文中符合特定条件的字段。
  ```

通过上述灵活的出入参配置方式以及强大的参数表达式解析能力，我们旨在提供一套全面、易用的解决方案，赋能 SRE 团队高效地管理和自动化服务调用中的数据流。

请求识别及透传

压测标识设计

压测标识与透传机制

为了确保压测流量能够在整个调用链路中被正确识别和处理，我们设计了一套完善的标识机制。

一、标识定义

我们定义了两类核心标识：

- **通用标识 (key=x-ctx-sre-xxxxxx1)**
  - **用途：** 仅用于明确这是一个压测请求。
  - **特点：** 默认必须携带，业务部门可以基于此标识对压测请求进行定制化的特殊处理，例如记录压测路径或定制压测数据。
  - **处理方式：** 系统中所有中间件或服务仅识别此标识，不做任何特定处理，以便我们可以完整地追踪压测请求所经过的链路。
- **隔离标识 (key=x-ctx-xxxxxxx2)**
  - **用途：** 告知中间件采取相应的隔离策略。

- **特点：**用户可根据需求决定是否携带此标识，例如当压测需要使用隔离资源时。
- **处理方式：**
  - 中间件一旦识别到此隔离标识，会检查当前服务节点是否配置了特定的隔离策略。
  - 如果没有配置，则会执行默认的隔离策略。
  - 以数据库（DB）为例，如果我们的数据访问层（DAL）中间件识别到带有隔离标识的压测请求，但未明确告知如何处理，它将默认拒绝该请求的所有读写操作。

二、标识透传

为了确保压测标识能在整个分布式调用链路中传递，我们依赖于分布式追踪系统（例如基于 CAT 的埋点能力）。

- **链路入口点：**无论是 Service Mesh、HTTP Web 服务器、Node.js Web 服务器，DUBBO 或其他 RPC 服务，作为请求进入系统的第一个节点，都负责处理初始的标识。
- **透传机制：**后续链路中的所有服务节点通过中间件自动进行标识的透传。这是通过追踪上下文（TraceContext）实现的。
  - **获取标识：**通过 traceContext = Cat.getTraceContext() 获取当前请求的追踪上下文，并从中提取压测标识，例如 String pressureLane = traceContext.get("pressure.lane")。
  - **修改/添加标识：**将压测标识嵌入到追踪上下文，例如 traceContext.add("pressure.lane", pressureLane)，确保它能够继续传播给下游服务。

三、标识识别

标识识别的核心在于所有中间件的统一改造，以确保它们能够一致地获取和判断请求的压测属性。

- **统一识别机制：**通过 CAT 埋入的压测标识，携程所有中间件都进行了改造，能够统一地获取并识别这些标识。

- **标准化判断：** 这意味着所有尚未支持的中间件，在将来需要支持时，都将遵循这套统一的标准。
- **判断方法：** 使用 Boolean pressureLane = Cat.isPressureLane() 等封装方法来判断当前请求是否为压测请求。
- **后续处理：** 一旦中间件识别到请求是压测流量，其后续处理逻辑（例如是否进行隔离、流量整形或数据记录）将由该中间件根据其自身特性和配置来确定。

通过以上三层机制，我们实现了对压测请求在整个系统中的精准识别、高效透传，并提供了灵活的隔离与处理能力，这对于在高并发环境下进行 SRE 实践并确保系统稳定性至关重要。

压测隔离配置

隔离策略总览

隔离策略架构

全链路压测的核心挑战之一是如何在生产环境中进行压测，同时确保压测流量不会对真实业务造成任何影响。为此，我们设计了三个维度的隔离策略：

一、数据隔离：确保数据安全的基础防线

- **采用"影子资源"机制：** 我们通过引入"影子库"、"影子表"或其他"影子资源"来解决此问题。即，压测请求产生的所有数据操作都将作用于这些专为压测而创建的影子资源，而非真实的生产数据存储。

- **简化数据清理：**这种隔离方式带来的附加效益是数据清理的便捷性。一旦压测结束，我们只需回收这些影子资源，即可实现数据的快速、彻底清理，大大降低了运维负担。

二、流量隔离：保障真实用户体验的核心策略

- **识别与管理压测请求：**服务层需要具备识别压测请求的能力。当压测请求流经业务链路时，我们需要对其进行有效管理，以防止其与真实用户请求混淆或相互影响。
- **应对无状态服务的弹性扩缩容：**绝大多数无状态服务都具备弹性扩缩容能力，例如通过 HPA（Horizontal Pod Autoscaler）在流量高峰时自动扩展节点。在资源充足的前提下，即使有大量压测请求涌入，服务实例也能快速扩容来应对，从而避免单个服务集群被压垮。这种情况下，业务层可能无需为流量进行物理隔离，但仍需确保弹性扩缩容机制能够及时响应。
- **部署独立压测集群：**尽管弹性扩缩容机制强大，但对于某些核心业务，如果其扩容响应时间不足以满足避免影响真实用户的需求，则建议部署独立的压测集群。压测请求将通过路由转发机制，精确导向这些专为压测而设的独立集群，从而彻底将压测流量与生产流量隔离。

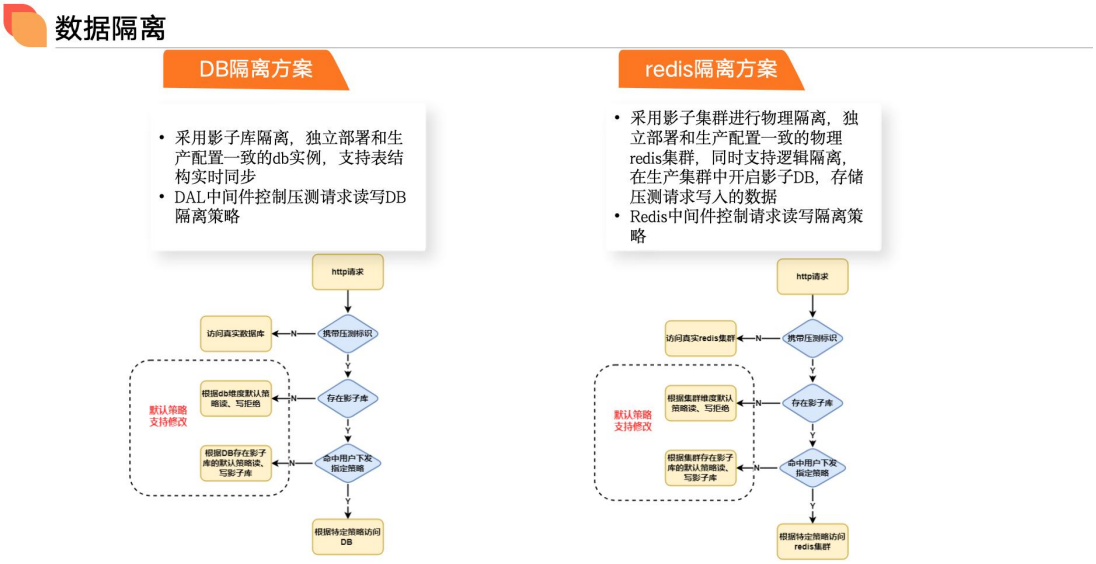
三、消息隔离：防止消息积压，保障异步通信的顺畅

在分布式系统中，消息中间件承担着关键的异步通信角色。压测产生的消息如果处理不当，可能严重影响真实用户的消息处理。

- **消息抑制机制：**
  - **生产端抑制：**在消息生产时，通过中间件配置或业务逻辑判断，直接抑制压测消息的发送，阻止其进入任何消息队列。
  - **消费端识别与忽略：**消费端服务能够识别出哪些是压测消息，并选择不对其进行处理，从而避免无效消费和资源浪费。
- **“影子消息队列”：**
  - **避免队列积压：**这是目前我们推崇并广泛使用的方式。通过创建与生产队列并行的“影子队列”，压测产生的消息将发送到这些专用队列中，与真实用户消息完全隔离，有效避免了队列积压对真实业务的影响。
  - **专用消费集群：**为确保影子队列中的消息也能被有效处理（例如用于验证消息流程的完整性），我们会支持用户创建专用的消费集群。这

些集群通过配置环境变量等方式，被消息中间件识别为专门处理影子队列消息的消费者，它们会订阅并消费相同 Topic 下的影子队列消息，保证压测中的异步流程也能得到完整验证，而不会干扰到生产环境的消费者。

数据隔离实施方案



数据隔离方案

一、DB 与 Redis 隔离方案

DB 隔离方案

- **核心机制：** 采用物理部署的“影子库”进行彻底隔离，确保压测流量与真实业务流量完全分离。
    - **部署与配置：** 影子库独立部署，并与生产环境数据库保持相同的配置和表结构，支持表结构的实时同步，以最大限度模拟真实环境。
    - **控制逻辑：** 通过 DAL (数据访问层) 中间件智能识别 HTTP 请求是否带有压测标识：
        - **无压测标识：** 请求直接访问真实数据库，保证业务正常运行。
        - **有压测标识：** 请求根据预设策略（包括用户自定义策略或默认策略）路由至影子库进行读写操作。若无影子库或特定策略，系统将根据默认配置拒绝读写，防止意外数据污染。
- **Redis 隔离方案**

- **核心机制：**采用物理部署的“影子集群”实现严格的物理隔离，同时在生产集群内通过“影子 DB”实现逻辑隔离，以优化资源成本。
- **部署与配置：**影子集群独立部署，并与生产环境 Redis 集群保持相同的配置，确保隔离的有效性与压测的真实性。
- **压测数据管理：**压测写入的数据存储在生产集群的“影子 DB”（逻辑隔离区），而非直接污染生产数据。
- **控制逻辑：**通过 Redis 中间件识别 HTTP 请求的压测标识，并根据预设策略（用户自定义或默认）路由请求：
    - **无压测标识：**请求直接访问真实 Redis 集群。
    - **有压测标识：**请求根据策略访问影子集群或在生产集群的“影子 DB”进行操作。若无可用影子资源或特定策略，系统将拒绝读写，避免数据混淆。

二、资源效率与成本优化：精准控制影子资源的生命周期

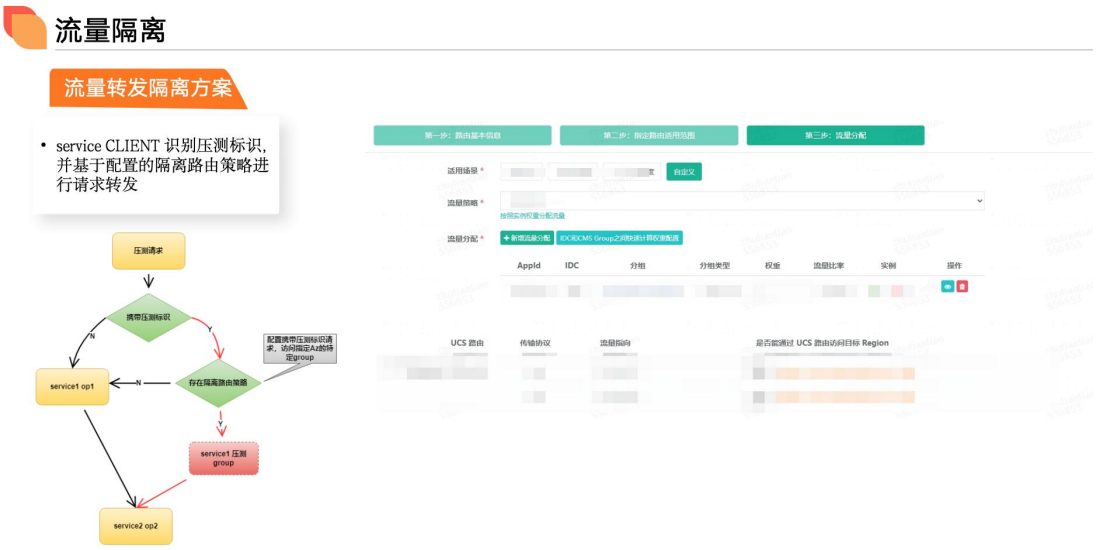
为了平衡资源隔离的彻底性与资源投入的经济性，设计了以下资源管理策略：

- **按需初始化：**影子资源并非预先创建，而是在压测任务真正开始时，通过“初始化影子库/资源”环节动态创建，避免长期占用资源。
- **及时回收：**压测任务一旦完成，对应的影子资源会立即被回收，确保资源的高效周转。
- **资源池化：**建立统一的影子资源池，供所有业务部门共享使用，提升资源复用率。
- **压测计划管理：**通过压测计划管理机制，协调各业务部门的压测时间，避免在同一时间段进行大规模压测导致的影子资源抢占与不足，例如，若某一时间段已被抢占，其他部门需重新协调压测时间。

三、中间件控制与策略下发：实现动态与精细化隔离

- **策略识别：**隔离的识别核心依赖于中间件，通过将隔离策略下发到应用服务节点，使得应用能够感知并执行相应的隔离逻辑。
- **动态路由：**应用服务节点上引用的框架中间件会根据当前请求以及对应的策略，动态决定访问真实资源还是影子资源，确保隔离策略的准确执行。

流量隔离实施方案



流量隔离与路由策略

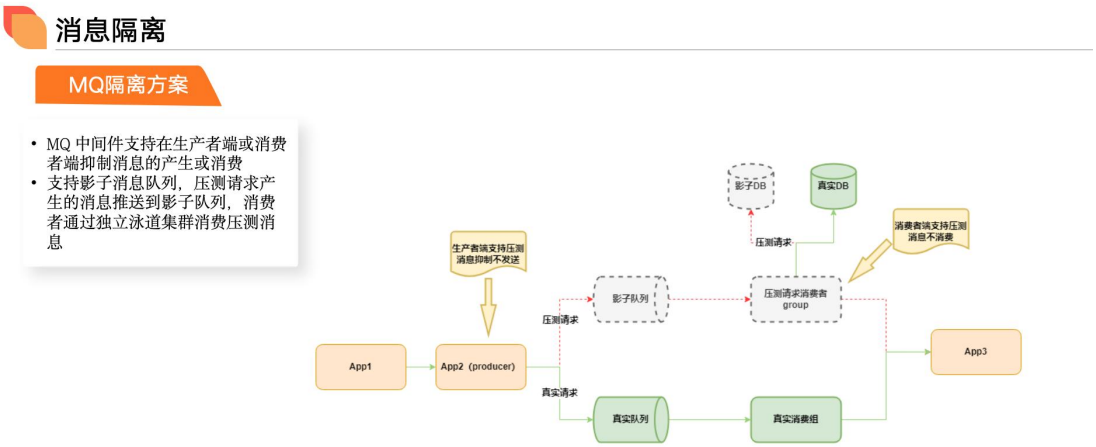
一、智能路由策略

流量隔离的核心是基于压测标识进行智能路由，确保压测流量被导向专用的隔离集群：

- **路由决策流程：**
  - 当系统接收到请求时，首先判断其是否“携带压测标识”。
  - 若**不携带压测标识**，请求将进入正常的生产服务链路，例如流向“service1 op1”。
  - 若**携带压测标识**，系统将进一步核查是否存在“隔离路由策略”。
    - **存在隔离路由策略：**针对携带压测标识且配置了指定可用区（Az）特定组的路由策略，请求将被精确转发到**“service1 压测 group”**等隔离环境进行处理。
    - **不存在隔离路由策略：**压测请求将按正常业务逻辑或其他默认规则处理，可能流入“service1 op1”或“service2 op2”。
- **细粒度路由配置：**平台提供直观的用户界面（UI）进行路由策略的配置，核心配置步骤包括：
  - **路由基本信息：**定义路由的适用场景（如自定义模式）及流量策略（如按实例权重分配）。
  - **指定路由适用范围：**明确路由规则生效的具体范围。

- **流量分配：** 这是配置的核心，支持：
    - “新增流量分配”功能，允许针对流量进行灵活的分配配置。
    - “IDC 和 CMS Group 之间快速计算权重配置”，便于在不同数据中心和集群管理系统分组间进行流量均衡。
    - 通过配置“AppId”、“IDC”、“分组”、“分组类型”、“权重”、“流量比率”、“实例”等参数，实现对流量路由目标和比例的精细化控制。例如，将特定 AppId 和 IDC 的流量，以指定权重或比率路由至某个隔离分组。
- **跨区域流量管理：** 平台支持通过统一控制系统（UCS）进行路由管理及传输协议配置，实现跨区域（Region）的流量转发与隔离。特别地，可以通过 UCS 路由，将流量导向目标区域，这为多区域部署下的压测和流量管理提供了强大能力。

消息隔离实施方案



MQ 隔离架构

MQ 隔离方案详解

- **影子消息队列机制：** 引入独立的"影子消息队列"来承载压测请求产生的消息。这些压测消息不与真实业务消息混合，而是由独立的"泳道集群"进行消费和处理。

架构与消息流向：

- **消息生产与分流（App2 - producer）：**

- 上游应用程序 App1 发送请求给作为消息生产者的 App2。
- App2 具备智能识别能力，能够区分“真实请求”和“压测请求”。
- **真实请求流向（绿色）：** 按照正常业务流程，发送至“真实队列”。
- **压测请求流向（红色虚线）：** 在“生产者端支持压测消息抑制不发送”的策略下，压测请求的消息会被路由并推送到专门的“影子队列”。

- **消息队列层：**
  - **真实队列（实线）：** 存放真实的业务消息。
  - **影子队列（虚线）：** 专用于存放压测请求产生的消息，与真实队列物理或逻辑隔离。
- **消息消费与下游处理：**
  - **真实消费组（实线）：** 从“真实队列”拉取消息，进行真实业务处理，并与“真实 DB”进行交互，最终可能将结果传递给 App3。
  - **压测请求消费者 group（虚线）：** 作为独立的“泳道集群”，专门从“影子队列”拉取压测消息。
    - 在“消费者端支持压测消息不消费”的机制下，这个消费者组确保压测消息得到独立处理。
    - 这些压测消息将访问专用的“影子 DB”，以避免对生产环境数据造成污染。
    - 压测处理结果或后续请求同样会传递给 App3。
- **数据存储层：**
  - **真实 DB（实线）：** 用于存放和处理真实业务数据。
  - **影子 DB（虚线）：** 专用于存放和处理压测数据，保障与真实业务数据的完全隔离。
- **App3：** 作为下游应用，同时接收来自真实消费者组和压测请求消费者组的处理结果或进一步请求。

整体而言，这套方案通过在生产者端识别并分流压测消息，配合独立的影子队列、隔离的消费者集群以及专用的影子数据库，构建了一个完整的消息隔离“泳道”，确保压测活动能在独立环境中进行，不会对生产服务的 SLA 造成任何影响，从而极大地提升了系统在常态化压测下的安全性和稳定性。

配置任务

压测计划管理

压测计划管理

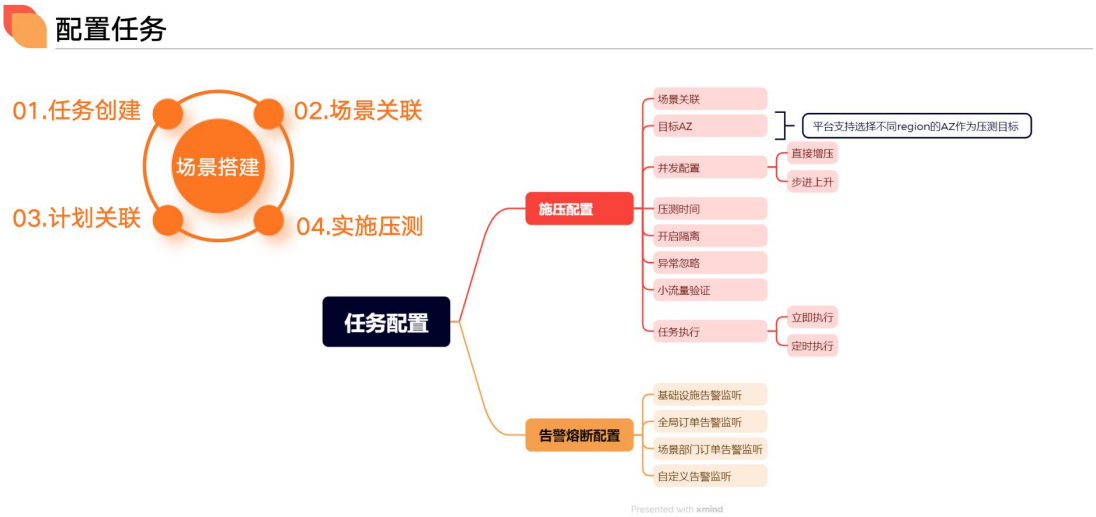
一、压测计划的必要性

- **压测窗口稀缺性与管理必要性：** SRE 正在面对资源竞争的现实，以今年十月份的压测窗口占用情况为例，表明资源已基本饱和，亟需精细化管理以避免冲突和无序。此举旨在将分散的压测需求纳入统筹规划，保障有限资源的合理利用。
- **压测计划的核心目的与推行策略：**
  - **赋能控制：** 压测计划管理旨在赋予 SRE 部门对业务研发乃至测试团队压测执行的控制权。任何团队欲对特定场景进行压测，均须先行创建压测计划。
  - **计划性与审批流程：** 此机制强制性地要求所有压测活动具备明确的计划性。计划创建后，将触发多级审批流程，包括但不限于相应的 SRE 负责人及更高层级领导的审查。审批重心在于：
    - **场景适用性评估：** 考量当前压测场景是否适合进行大规模压力测试。
    - **时间窗口合理性判断：** 评估所选压测时间段是否恰当，避免对线上系统造成不必要的冲击。

- **QPS（每秒查询率）合理性审核：** 对业务研发或测试团队提出的预期压测 QPS 进行严格审查。SRE 站在全局视角，审视其对基础设施的潜在影响，若发现不合理或存在风险，将坚决驳回。

- **实现通盘掌控与安全保障：**
  - **宏观态势一览无余：** 通过压测计划管理，SRE 团队能够鸟瞰全局，清晰掌握如十月份这般，整个月份内各业务部门或团队在不同时间窗口预约的压测安排，从而合理调配资源，并及时识别潜在冲突。
  - **强制平台控制与安全区间运行：** 一旦压测计划通过审核，平台将严格锁定其执行时间窗口。若有团队试图在未经授权的时间段执行压测，平台将直接拦截任务，防止未经批准的操作，以最大程度保障系统安全。
  - **避免业务高峰期冲击：** 此机制重点在于限制在业务高峰期进行压测的可能性，除非有高级别主管经过权衡并审批通过的特殊场景。例如，某团队可能在评估自身风险并获得总监批准后，选择在高峰期对特定业务流进行压测，这凸显了计划管理的灵活性与安全性兼顾的特性。

任务配置流程



任务配置界面

一、任务配置流程

压测任务的配置遵循清晰的四步流程：

- **01. 任务创建：** 这是压测工作流的起始点，定义了新的压测需求。

- **02. 场景关联**：将创建的任务与预设的测试场景进行精准绑定，确保测试内容的准确性。
- **03. 计划关联**：确保任务与具体的执行计划有效关联，如果没有关联的计划，任务将无法执行。
- **04. 实施压测**：在完成所有前置配置后，正式启动压力测试。

二、任务配置详情

任务配置是压力测试能否成功执行的关键，尤其注重安全性与灵活性，其主要分为施压配置与任务执行两大部分：

1. **施压配置**：
    - **场景关联**：此选项再次强调了任务与场景的紧密结合，是施压的基础。
    - **目标 AZ**：鉴于平台支持选择不同区域（region）的可用区（AZ）作为压测目标，这允许精细化地进行跨 AZ 压力测试，从而验证异地容灾和分区故障隔离能力，确保系统在复杂部署下的韧性。
    - **并发配置**：
        - **直接增压**：直接设定并达到目标并发量进行测试。
        - **步进上升**：通过逐步提升并发量来模拟真实流量增长，监测系统性能瓶颈。
    - **压测时间**：精确设定压力测试的持续时间。
    - **开启隔离**：此项是任务独有的重要配置，它允许用户在任务层面决定是否触发隔离措施，即使场景本身配置了隔离,该任务也可以选择性地不开启，以便在小流量验证或特定情况下观察系统真实行为，避免压测对生产环境造成影响。
    - **异常忽略**：允许在压测过程中忽略特定范围的异常，以评估系统在非理想情况下的表现。
    - **小流量验证**：在大规模压测前，先用小部分流量进行验证，确保测试场景与配置的有效性。
2. **任务执行**：
    - **立即执行**：支持任务配置完成后即刻启动。

- **定时执行**：提供弹性，允许用户预设任务启动时间，实现自动化执行。

三、告警熔断配置

为确保压力测试的安全性和有效性，平台提供了多维度的告警监听与熔断机制：

- **基础设施告警监听**：实时监控 CPU、内存、网络等核心基础设施指标。
- **全局订单告警监听**：对整个系统的订单相关告警进行监控，聚焦业务核心指标。
- **场景部门订单告警监听**：针对特定场景或部门的订单告警，提供更细粒度的业务监控。
- **自定义告警监听**：赋予用户根据业务需求灵活定义告警规则的能力，进一步增强了监控的普适性和专业性。

启动压测

压测执行流程

压测执行流程

一、执行流程概览

- **01. 创建计划：** 这是任何任务启动的首要环节，旨在明确任务目标与预期。
- **02. 关联场景：** 在计划创建之后，需要将计划与已定义的具体业务场景进行绑定，这个环节至关重要，它确保了任务能够针对性地执行。此处，我们特别强

调场景的独立管理——在任务配置时，必须明确指定将要进行压测的特定场景，并确保该场景已包含相应的计划。

- **03. 提交审批：** 一旦计划与场景关联完成，任务将进入审批流程，这是确保变更受控与合规性的重要保障。
- **04. 实施压测：** 审批通过后，方可启动压测任务，此步骤的顺利进行，离不开前期的严谨规划与审批。

二、任务调度日历：可视化排期与资源管理

右侧区域则以"2025 年 10 月"的日历视图，展示了任务的排期与调度情况。

- **时间排期概览：** 日历清晰地标记了从 1 日到 26 日的日期。
- **任务持续时间：** 每个日期格内嵌的蓝色矩形条块代表了已预定的任务时段。例如，在 10 月 1 日和 10 月 14 日，均有标记为“01:00 - 02:00”的任务块，这揭示了该时间段很可能被用作系统压测、维护或其他 SRE 操作的共享资源或固定时间窗口。10 月 15 日还展示了一个更长的蓝色任务块，可能意味着一个持续时间较长的任务或一系列连续任务。
- **导航与筛选：** 日历上方提供“2025 年 10 月”的月份显示及左右箭头导航，便于用户切换月份查看。

压测监控与分析

压测监控大盘

一、监控体系

- **过程监控：**这是压测运行中的第一道防线。在压测请求持续冲击系统时，我们必须通过实时监控平台，密切关注各项系统指标，如 CPU 利用率、内存消耗、I/O 吞吐、网络延迟等。这不仅有助于发现潜在瓶颈，还能及时调整压测策略，确保压测的有效性。
- **告警监听：**与监控紧密相连的是告警机制。当系统指标超出预设阈值，或者出现错误率飙升、响应时间剧增等异常情况时，告警系统应立即触发通知，将问题准确地传达给相关负责人。及时响应告警是避免问题扩大的关键，它能让我们在压测过程中迅速定位并解决关键故障。
- **报告输出：**压测并非一劳永逸。在压测结束后，一份详尽且专业的报告至关重要。这份报告不仅要汇总压测期间收集到的所有数据，更要基于这些数据进行深入分析，评估系统在不同负载下的表现，识别性能瓶颈，并为后续的性能优化和容量规划提供决策依据。报告的质量直接决定了压测的价值。

结果分析：链路全覆盖与动态容量评估

压测结果的对比分析是我们进行容量规划和系统优化的核心。

查询/筛选功能，允许我们根据"调用方"、"被调用方"以及"压测流量类型"等维度，灵活地筛选和组合压测数据，以便对特定业务链路进行细致分析。

数据卡片聚焦于压测结果与日常数据之间的关键对比，其核心目标是评估压测的仿真度以及系统容量：

- **压测仿真度评估：**
    - **链路分析报告：**压测的核心在于模拟真实用户行为。我们通过“链路分析报告”，将压测请求所经过的业务链路与真实用户请求的链路进行细致对比。
    - **Trace 覆盖率：**结合“压测高码”与“日常高码”下的“trace 覆盖率”指标，我们能够量化压测流量对实际业务链路的覆盖程度。例如，如果压测的某个中间件节点，“压测高码”的 Trace 覆盖率显著低于“日常高码”，这可能意味着压测报文的构造未能充分覆盖该节点的真实业务场景。
    - **调用总数比值与覆盖率比值：**这些比值进一步细化了压测流量与日常流量在调用量和链路覆盖上的差异。如果发现压测流量未触达真实用户访问的某个关键路径，或者触达了真实用户不常访问的路径，这都将提示我们需要优化压测报文，以提高其仿真度。

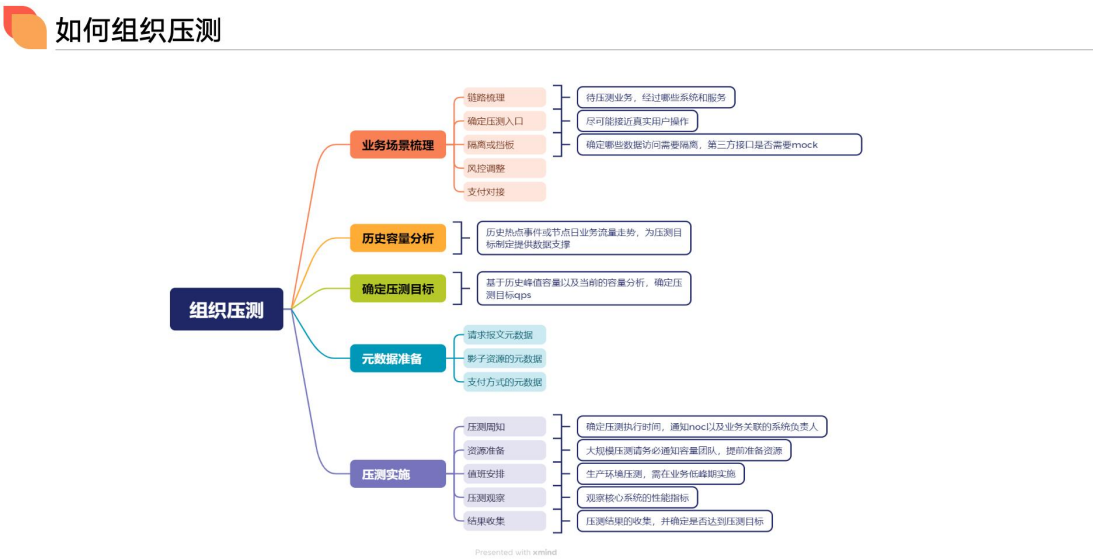
- **异常检测：**“异常数”或“异常比例”，如“异常数: 1”，是评估压测健康度的重要指标。这些异常可能来源于压测过程中出现的错误、超时或其他非正常行为，它们是系统性能瓶颈或潜在缺陷的直接反映。

- **动态容量评估：**
  - **容量分析：**结合本次压测所消耗的资源状况，我们会与容量团队协作，进行精准的容量分析。这项工作旨在回答两个核心问题：当前系统是否需要扩容？如果需要，具体的扩容量是多少？
  - **资源利用率：**容量分析不仅关注服务实例的数量，还会细化到CPU、内存、网络带宽、数据库连接等各项关键资源的利用率，以确保资源的合理配置，避免过度投入或资源瓶颈。

最终，通过这一整套严谨的“压测流程 + 数据驱动的对比分析”，我们可以精确评估系统在压力下的抗风险能力，优化压测报文的仿真度，并为容量团队提供

压测流程组织

标准化压测流程



标准化压测流程

一个完整的全链路压测项目，需要遵循标准化的流程，确保每个环节都得到充分考虑和执行。

一、业务场景梳理：还原用户路径，确定压测边界

- **确定压测入口：**找出最能模拟真实用户行为的流量入口，这是压测流量注入的关键点，例如用户登录、商品下单等核心操作。
- **隔离或挡板：**明确在压测过程中哪些数据读写需要进行隔离，哪些外部服务（如第三方支付接口、短 X 服务等）需要使用挡板（Mock）或沙箱环境进行模拟，以避免对生产环境造成不必要的影响。
- **风控调整、支付对接：**特别关注风控策略和支付环节的适配，可能需要临时调整风控规则以允许大流量通过，并确保支付链路在压测期间能够正确模拟或真实对接（如果允许）。

二、历史容量分析：以史为鉴，预判未来

历史数据是我们制定压测目标的宝贵财富。通过细致地分析，我们可以为压测目标设定提供坚实的数据支撑：

- **历史热点事件：**回顾过往的大促活动、秒杀场景等热点事件的流量峰值和系统表现。
- **日常业务流量走势：**分析日常业务的流量波动规律，识别高峰期和低谷期，了解系统的常态化承载能力。

三、确定压测目标：量化预期，指明方向

在充分理解业务场景和分析历史容量的基础上，我们需要明确本次压测的具体目标，这通常以 QPS（Queries Per Second，每秒查询数）作为衡量的核心指标：

- 基于历史峰值容量数据，并结合当前业务增长预期和容量规划，综合确定预期的目标 QPS。这个目标 QPS 应该是我们系统在未来一段时间内需要能够稳定承载的流量上限。

四、元数据准备：精雕细琢，确保真实性

元数据是压测能够真实模拟业务行为的基础。它的准备工作尤为重要，直接影响压测结果的有效性：

- **请求报文元数据：**准备用于压测的请求数据格式和内容，确保其多样性和真实性，尽可能覆盖实际业务中的各种请求场景。
- **影子资源的元数据：**如果采用了影子模式或隔离环境进行压测，需要准备对应的影子环境所需的基础数据、配置数据等元数据。

- **支付方式的元数据：**精心准备与支付交易相关的元数据信息，如不同的支付渠道、支付金额、用户 ID 等，以真实模拟支付流程。

五、压测实施：有条不紊，高效执行

压测的实施是一个系统性的工程，需要周密的计划和严谨的执行：

- **压测周知：**压测开始前，务必提前确定压测执行时间，并广泛知会 NOC（网络运营中心）以及所有业务关联的系统负责人，确保大家知情并做好配合准备。
- **资源准备：**针对大规模压测，务必提前通知容量团队，协同准备所需的硬件资源、网络带宽等，确保压测环境的充足。
- **值班安排：**若进行生产环境压测，需选择业务低峰期执行，并安排专门的值班人员全程监控，随时准备应对突发状况。
- **压测观察：**压测过程中，持续密切观察核心系统的各项性能指标，包括 CPU 利用率、内存使用、网络 IO、数据库连接数、响应时间、错误率等，及时发现瓶颈。
- **结果收集：**压测结束后，系统性收集所有压测数据和日志，并对结果进行深入分析，评估是否达到了预设的压测目标。这包括分析系统的各项性能指标是否在预期范围内，是否存在稳定性问题或性能瓶颈。

压测文化的建设与未来展望

压测文化建设

04 压测文化的建设与未来展望

压测文化体系

压测文化要素

技术体系的建设固然重要，但更关键的是如何在组织内部建立起一种压测文化，让全链路压测成为团队的日常实践。

一、价值共识：建立压测的文化认同

全链路压测并非形式主义，其核心在于为企业带来实实在在的价值。

- **不为压测而压测：** 我们需要根据公司具体的业务场景，精准定位压测的适用范围和期望成效，确保每一轮压测都聚焦于解决实际问题。
- **统一认知：** 从高层管理者到业务研发、测试人员，再到 SRE 团队，所有参与者都必须对全链路压测的意义和价值形成高度共识。这种自上而下的认同是推动压测工作顺利开展的关键驱动力。

二、责任共担：跨部门的协同作战

全链路压测的复杂性决定了它无法由单一团队独立完成，而是一项需要多部门、多角色紧密协作的任务。

- **协同机制：** SRE、研发、测试等团队需要共同参与压测的全过程。
- **稳定共维：** 我们持续向所有团队强调，网站的整体稳定性是大家共同的责任。任何环节出现问题，都可能对整个公司造成影响，因此我们提倡共同维护系统稳定性的团队意识。

三、标准化流程：规范高效的执行保障

一套清晰、标准化的压测流程，是确保全链路压测可重复、高效和有效执行的基础。

- **流程标准化：** 一旦决定推行全链路压测，其从准备、执行到分析的每一个环节、每一个节点都必须标准化。
- **统一规范：** 应尽可能为所有业务部门制定统一的压测流程规范，以提升整体执行效率和结果可比性。

四、工具支持：便捷易用的自助服务平台

强大的工具支持是降低压测门槛、提升效率的关键。

- **平台化工具：** 我们构建了一套高效的全链路压测管理平台。
- **用户友好：** 该平台设计宗旨是让业务部门能够快速上手，甚至实现“傻瓜式”操作。如果配置过程过于繁琐，耗费大量人力成本，用户在使用时就会产生抵触心理。
- **自助服务：** 我们的目标是实现压测过程的完全自助化。目前，大部分压测活动已由业务方自行完成。通过提供简洁快速的工具，我们不仅为用户提供了便利，也有效降低了公共团队的技术支持成本和人力投入，让平台团队无需直接参与，用户就能通过平台获取所有所需的信息和支持。

未来演进方向

未来演进路线图

技术永不止步，携程的全链路压测体系也在不断演进。我们的演进路线遵循以下思路：

核心演进思路： 先把性能压测能力尽可能"产品化"和"自动化"，覆盖研发交付链路与线上运行环境；再把压测目标从技术指标升级到业务指标，并进一步用压测数据反哺容量与成本；最后探索用 AI 提升场景生成与根因分析效率。

已在推进的能力建设

1. 压测左移：把性能验证嵌入 CI/CD 为尽早发现性能回归问题，团队将性能压测能力左移到交付链路，在 CI/CD 中嵌入自动化性能验证。每当新功能准备投产时，系统会在发布前自动执行相应的压测或性能基线校验，用于提前识别新版本的性能风险，避免问题进入生产环境后才暴露。

落地要点 - 性能压测嵌入 CI/CD - 投产前提前发现新版本性能问题

2. 无人值守压测：线上版本发布后的自动触发与周期性验证 在业务部门的需求驱动下，团队建设了"无人值守压测"模式：当服务在生产环境发布新版本后，系统会在业务低峰期自动触发一次压测，用于验证线上真实环境下的新版本承载能力与性能表现。

除"发布后自动触发"外，还支持"周期性压测"：业务方可配置定时任务（job），在低峰期周期性运行某个固定场景的压测并自动沉淀报告。这样，业务同学次日只需查看报告即可完成例行验证，实现持续性能验证与风险前置发现。

落地要点 - 上线版本自动化触发性能压测，实现持续性能验证，提前发现性能隐患

3. 混沌工程嵌入：压测与故障注入结合验证韧性 在已有压测能力基础上，团队同步推进与混沌工程的结合：对日常、常见的故障注入场景（如随机故障模拟），在注入后继续执行压测，用于验证系统在异常情况下的容错能力与性能退化边界，从而提升系统韧性与可恢复性。

落地要点 - 引入随机故障模拟，验证系统在异常情况下的容错能力，提升韧性

下一阶段演进方向（未来规划）

1. 业务价值驱动与成本优化：从"技术压测"升级为"业务压测" 团队计划将压测目标从纯技术指标（如 QPS、RT、CPU）进一步提升到业务指标层面（例如订单成功率、关键链路转化、核心接口成功率等）。业务方可以更直观地看到压测对业务结果的影响，压测报告也能直接映射到业务风险与收益，从"测得快不快"变成"业务能不能稳稳跑"。

在此基础上，压测数据将与 FinOps 思路结合：利用压测结果预测业务增长所需的资源规模，支撑更精准的容量规划与成本控制，让压测数据从“验收工具”升级为“经营与治理数据”。

目标要点 - 以业务指标为压测目标，提供最直观的业务影响分析
- 与 FinOps 结合，压测数据预测业务增长所需的资源规模，进行精准的容量规划和成本控制

2. AI 驱动智能化压测：降低配置成本、提升定位效率（探索中） 团队也在探索将 AI 引入压测全流程，重点聚焦两类价值： 1) 场景生成智能化：基于历史流量与调用特征，自动生成更贴近真实业务的压测场景，减少人工配置成本与经验依赖。 2) 分析诊断智能化：在压测结果出现异常时，自动识别潜在性能问题根因，并给出优化建议，提升问题排查效率。

该方向目前仍处于摸索与方案探索阶段，目标是逐步沉淀可复用的方法与工程化能力，最终形成“更准的场景、更快的定位、更低的门槛”的智能压测体系。

目标要点 - 通过 AI 分析历史流量数据，自动生成更贴近真实业务的压测场景，减少人工配置成本
- 利用 AI 自动识别性能问题根因，并提供优化建议，提升问题排查效率

5.4.6 小红书混合云容灾治理实践

SRE Elite 精选原因

小红书作为国内领先的生活方式社区平台，月活用户超过 3 亿，日均笔记曝光量达数百亿次。随着业务规模从千万 DAU 跃升至亿级，其云架构经历了从单一公有云到异地多活、再到混合云的持续演进。这一过程中，容灾能力建设面临独特挑战：架构复杂度陡增，微服务规模从百千量级增长至上万量级，依赖链路错综复杂；混合云环境下专线故障风险频发，流量治理难度倍增；S0 级核心场景从最初 4 个发展到近百个，快速逃逸和止损的协同效率要求极高。

面对这些挑战，小红书 SRE 团队构建了一套端到端的混合云容灾治理体系，以 PDCA 循环为方法论指导，从容灾洞察、容灾治理到容灾运营形成闭环。本案例分享了从 GSLB 流量调度优化、专线故障应对策略、容灾中控平台建设，到分层金字塔演练体系的完整实践经验，将核心场景 RTO 控制在 10 分钟以内。对于正在进行多云部署、异地多活架构建设的企业，本案例具有较高的参考价值。

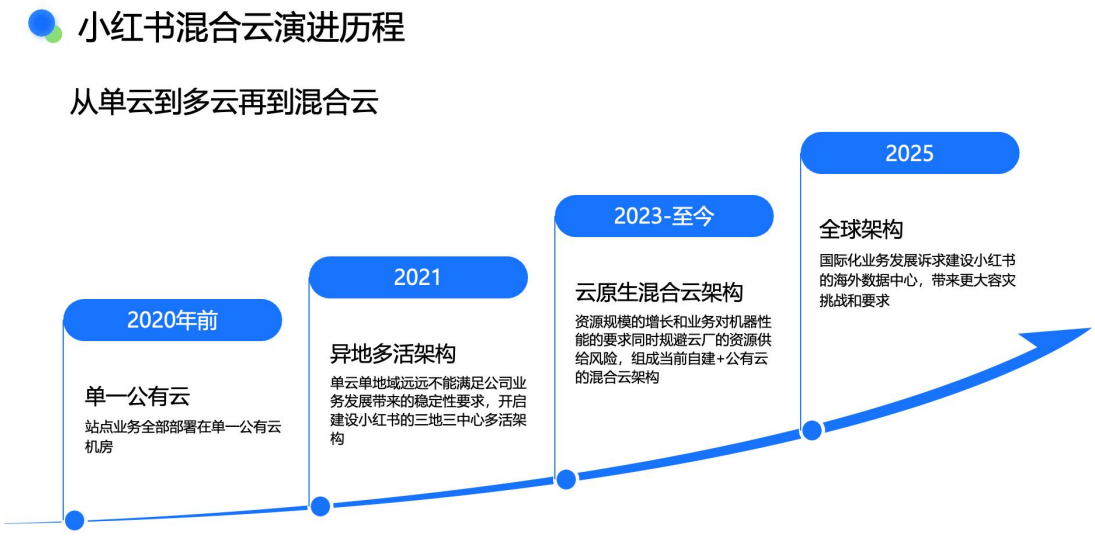
（一）背景及设计原则

混合云演进与容灾挑战概览

01

小红书混合云演进以及面临的容灾风险与挑战

小红书云架构演进历程



小红书的云架构演进历程，清晰地展现了其从单一公有云发展到多云异地多活，最终实现混合云的部署路线，并展望了未来全球化架构的布局。

1. 2020 年前：单一公有云阶段

- **业务背景：**小红书自 2013 年起，经历了从跨境电商到社交属性的业务转型，特别是在 2018 年后结合疫情，社交业务实现了爆发式增长。
- **架构特点：**在这一阶段，小红书的业务全面依赖单一公有云。所有计算、存储、网络资源以及容器技术，均构建于公有云的基础能力之上。这种模式适用于业务初期，但随着业务规模的增长，其稳定性面临挑战。

2. 2021 年：异地多活架构的实践

- **驱动因素：**随着用户规模突破千万 DAU，单一地域的云部署已无法满足公司业务对稳定性和容灾能力的需求。频繁的故障造成了公众影响，促使小红书开始寻求更完善的容灾策略。
- **架构特点：**
    - **启动建设：**小红书着手构建三地三中心异地多活架构。
    - **技术探索：**针对推荐业务等核心场景进行异地多活的 POC（概念验证），打磨了流量调度和资源调度等关键技术。
    - **核心业务改造：**历时两年多，至 2023 年，小红书 S0 级别的核心业务全面完成了异地多活架构的改造，显著提升了系统的韧性。

3. 2023 年至今：云原生混合云架构的建立

- **驱动因素：**
    - **资源规模：**业务规模持续扩大，资源需求达到百万级，单一云厂商的资源供给能力逐渐显现瓶颈。
    - **风险规避：**为了摆脱对单一云厂商的过度依赖，规避潜在的资源供给风险，小红书开始引入自建数据中心。
- **架构特点：**
    - **混合部署：**形成了由自建机房与公有云共同组成的混合云架构，将部分核心业务或对性能、成本、合规性有特殊要求的业务部署在自建环境中。
    - **技术挑战：**这种混合架构也带来了新的容灾挑战，需要更精细化的资源协同与故障管理策略。

- **云原生赋能：** 深入集成云原生技术，进一步增强了架构的弹性、可伸缩性和跨环境可移植性。

4. 2025 年：迈向全球化架构

- **驱动因素：**
  - **国际化业务增长：** 随着国际化业务推进，小红书在海外的用户规模持续增长，对“跨区域高可用”和“就近访问”的要求越来越高。
  - **外部事件刺激：** 年初某云厂商在海外区域发生的大规模机房故障事件，也直接提醒我们：如果全球业务过度依赖单一区域或单一云厂商，一旦出现区域性故障，损失会非常难以接受。这成为我们系统性推进全球化多区域容灾布局的重要触发点。
- **架构特点：**
  - **海外数据中心：** 计划建设小红书的海外数据中心，以支撑全球业务的稳定运行。
  - **全球容灾：** 面临更复杂的全球容灾挑战与要求，需要考虑跨地域的服务与数据调度、全球负载均衡以及“One-Key / One-Community”的理念，即实现业务流量和数据的全球共享与协同，这一目标带来的技术挑战极高。

小红书的云架构演进，是一段持续应对业务挑战、主动提升技术韧性的过程，通过混合云和全球化部署，致力于为用户提供更稳定、更高效的服务。

容灾风险识别和面临的挑战（外部）

容灾风险识别和面临的挑战（外部）

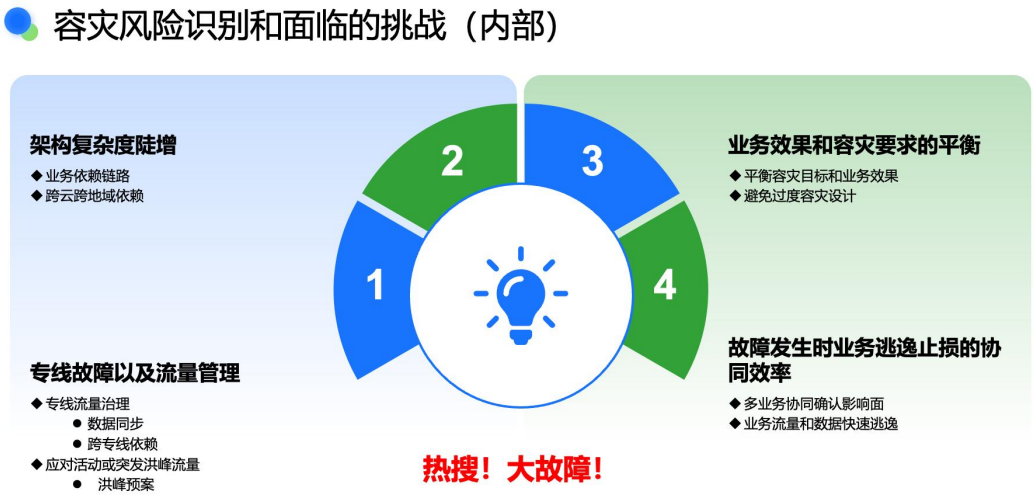
基础设施故障

面对基础设施的外部挑战，容灾能力十分重要。从这些数据可以看到，基础设施相关的风险暴露是高频的：不同产品和业务在不同日期都可能出现告警甚至故障。这意味着，在做架构设计时，不能把基础设施故障当作“小概率意外”，而要按“它会反复发生”的前提来建设容灾能力，把应对机制做成日常能力，而不是临时救火。

- **突发硬件故障：**
  - **机房火灾**：例如，去年某云服务提供商就曾遭遇火灾，导致业务持续数日中断，迫使客户进行被动且复杂的跨机房业务迁移。
  - **意外物理破坏**：如挖掘机不慎挖断光缆的场景，导致专线被切断，进而造成云上客户业务大面积不可用。正如俗语所说，"功夫再好，也怕挖机"，面对这类物理损伤，技术架构难以规避直接影响。
- **软件层面广泛影响：**
  - **DNS 级别故障**：即使是看似微小的软件配置问题，也可能引发全球性的连锁反应。例如，近期 AWS 就曾因一个 DNS 问题，导致全球范围内的服务出现数小时的大面积中断。

这些案例说明，外部因素对基础设施的潜在破坏较大，需要在容灾策略中考虑并应对这些不可预见的挑战。

容灾风险识别和面临的挑战（内部）



1. 架构复杂度陡增

- **业务依赖链路错综复杂：**随着小红书业务微服务化架构的持续迭代，微服务规模已从早期的百千量级增长至如今的上万量级，形成了一个高度复杂且相互依赖的系统图谱。这种高耦合性意味着任一强依赖服务的微小抖动或不可用，都可能引发“雪崩效应”，严重影响整个业务链路的可用性。
- **跨云跨地域混合架构挑战：**在核心业务实现多活架构后，系统不仅涉及业务模块间的依赖，更扩展到不同机房间的、跨专线的依赖。这种混合云架构的复杂性，以及随之而来的地域间数据一致性和专线稳定性管理挑战，对系统的韧性提出了极高要求。

2. 专线故障与流量管理难题

- **专线流量治理的复杂性：**在多活部署，尤其是异地三活的场景下，各单元间需实现互联互通及冗余，导致存在大量专线链路。这些专线承载着离线、在线业务流量和数据同步，其任何抖动或中断都可能导致大规模的业务异常或非预期事件。
- 数据同步：实时数据同步及一致性保证是专线流量治理的核心挑战。
- 跨专线依赖：系统在不同专线间存在的多重依赖，使得单一专线故障可能波及依赖其传输的多个业务。
- **应对活动和突发洪峰流量：**
- 洪峰预案：针对大促活动（如 CNY）、直播抢购或“热搜”效应等突发性流量洪峰，需要制定并实施前瞻性的预案，以确保系统在高负载下仍能保持稳定运行，避免资源过载。

3. 业务效果与容灾要求的平衡

- **平衡容灾目标与业务效果：**在实践中，容灾和业务效果之间一定存在 trade-off。我们的目标不是“为了高可用可以无限牺牲业务效果”，而是在约束条件下找到最优解：***在极端故障场景下，我们允许在一个可控时间窗口内，适度牺牲推荐/搜索效果的精细度，换取核心功能“可用”；但在常态场景下，我们不接受为了追求理论上的极端容灾能力，而长期牺牲核心业务指标。***
- **避免过度容灾设计：**例如，异地多活方案在提高容灾能力的同时，一定程度上会引入跨机房延迟，带来部分业务效果折损。如果不加选择地“全量多活”，很容易导致成本和复杂度失控。因此，我们会和业务方在设计阶段充分对齐：哪些是必须“容灾优先”的 S0 场景，哪些可以“效果优先 + 兜底预案”，在容灾能力、业务体验和资源成本之间做精细的权衡。

4. 故障发生时业务逃逸止损的协同效率

- **多业务协同确认影响面：**当发生专线中断、机房断电等容灾级别的故障时，快速、准确地识别故障影响范围，并推动多个核心业务团队高效协同止损，是实现业务快速恢复的关键。
- **业务流量和数据快速逃逸：**团队需要具备在短时间内将故障业务流量迅速切换至健康的系统或区域的能力，并能够及时止损，将 RTO（恢复时间目标）控制在极短时间（例如，将核心场景的 RTO 降至 10 分钟以内）。随着 S0 级核心场景数量从最初的 4 个发展到近 100 个，快速逃逸和止损的协同效率面临巨大挑战。

（二）体系设计及关键流程体系设计

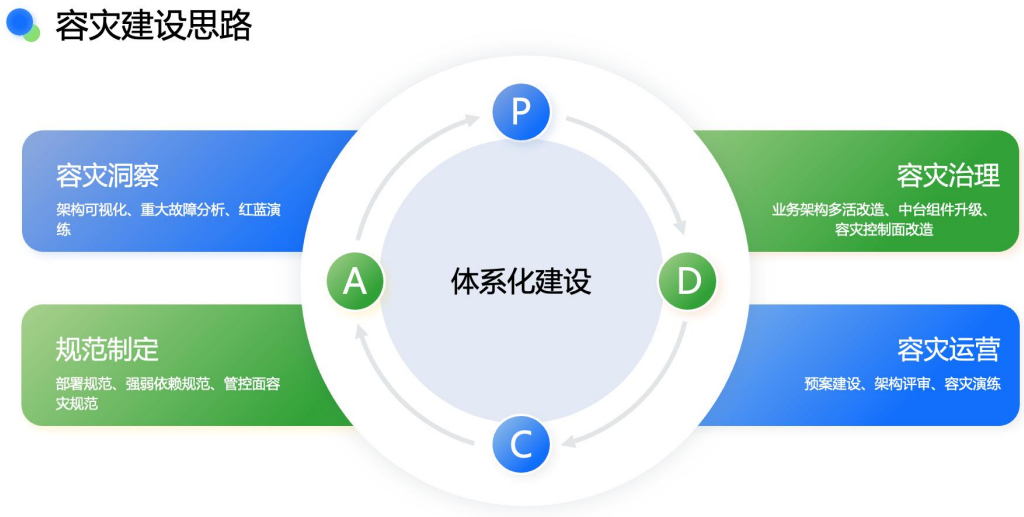
小红书混合云容灾体系建设

02

小红书混合云容灾体系建设

在分析完所面临的各项挑战之后，接下来将阐述小红书容灾项目组的建设思路。面对复杂且多样的问题，需要构建一个端到端、体系化的解决方案，以应对混合云环境下的各种不确定性。

容灾建设思路：PDCA 循环体系



容灾建设并非一蹴而就，它是一个需要持续投入、不断优化和迭代的系统工程。体系化建设是核心思想，以 PDCA（Plan-Do-Check-Act）循环框架为指导，在不断发现问题、解决问题中持续改进。

- **P (计划/Plan) - 容灾洞察**
    - **规范制定与推动：** 通过标准化定义符合高可用要求的架构，明确部署规范、强弱依赖规范以及管控面容灾规范，并与业务和 SRE 团队达成共识。基于这些规范，我们可以识别出不符合要求的业务，推动其标准化迭代，并使其跟上公司基础设施发展的步伐。
    - **风险识别与分析：** 结合公司平台能力、历史内部外部故障案例进行深入分析，识别出高风险场景并将其纳入管理范畴。此外，通过红蓝演练等形式，主动挖掘和识别潜在的业务风险点，以支撑后续的针对性治理。
- **D (执行/Do) A (Act / 改进) - 容灾治理**
    - **S0 场景改造：** 持续识别 S0 级（最高优先级）故障场景，并推动其业务架构进行多活改造。这包括对现有系统进行工程化升级，以提升其在极端情况下的韧性。
    - **基础设施升级：** 定期对核心中台组件进行升级，以确保其性能和可靠性满足日益增长的业务需求。

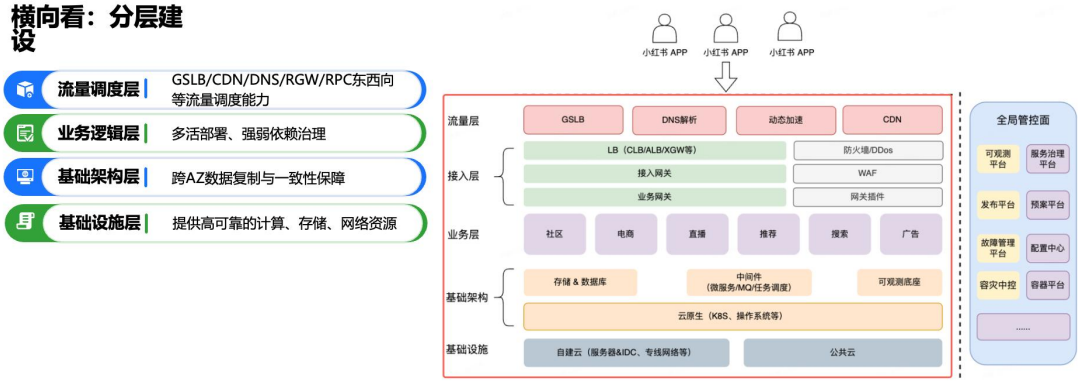
- **控制面容灾建设：** 改造并优化容灾控制面，使其能够与定义的规范和要求保持一致，确保在故障发生时能够有效地编排资源切换和恢复。

- **C (检查/Check) - 容灾运营**
  - **风险治理与巡检：** 对日常巡检发现的风险进行有效治理，确保系统健康无虞。
  - **公司级容灾演练：** 定期组织并执行 S0 级、公司级的容灾演练，验证预案的有效性，提升团队在真实故障场景下的响应能力和协作效率。
  - **预案建设与评审：** 针对每一个 S0 场景，建立和完善详细的故障预案。同时，当业务线进行大规模架构改造以满足性能或容灾需求时，SRE 团队需深度参与评审，确保改造方案符合容灾建设目标。

通过上述四个阶段的持续循环和反馈，容灾建设能力得以不断提升。将规范、治理、运营和演练融入到日常工作中，从而构建一个具有韧性、能有效应对各种故障挑战的系统。

小红书混合云容灾架构设计

小红书容灾架构



小红书的容灾架构，与业界主流互联网架构思路一致，秉持分层建设的理念，从用户请求直至底层基础设施，层层递进，通过技术设计与实践，达成全链路的高可用与业务连续性。

横向来看，小红书的容灾架构可划分为以下几个关键层面：

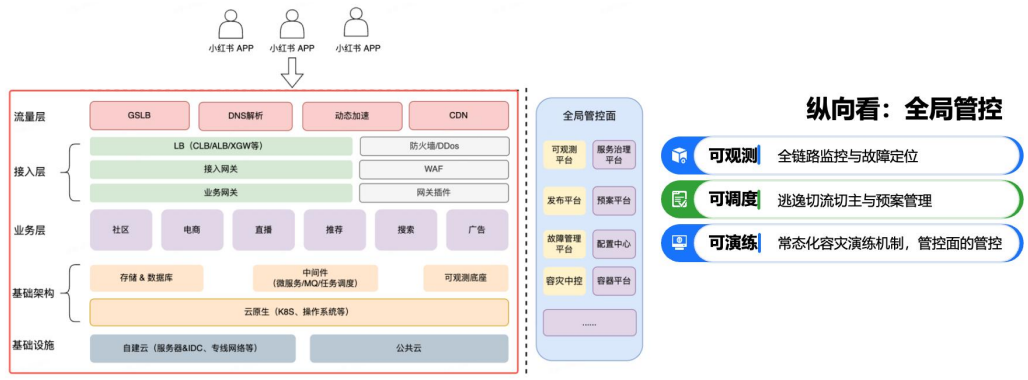
- **流量调度层：** 这是用户请求进入系统的第一关，也是架构韧性的重要支撑。
  - 通过 **DNS 解析** 引导用户流量；

    - 借助 **GSLB (全局负载均衡)** 实现跨地域、跨数据中心的流量智能分发；
    - 结合 **CDN (内容分发网络)** 实现资源加速与就近访问，提升用户体验并分散流量压力；
    - 利用 **网关** 提供包括南北向与东西向在内的流量调度能力，确保请求能够有效地抵达目标服务。
- **业务逻辑层：** 承载着小红书核心功能的运转。
    - 通过 **多活部署** 策略，尤其是异地多活甚至三活的部署架构，确保即使部分地域或可用区出现故障，业务也能持续对外提供服务；
    - 辅以 **强弱依赖治理**，梳理业务自身的依赖关系以及业务模块间的依赖逻辑，从而在故障发生时，减少级联效应，保障核心业务的健壮性与弹性。
- **基础架构层：** 位于业务逻辑之下，提供稳定的技术支撑。
    - 核心在于 **跨可用区 (AZ) 的数据复制与一致性保障** 机制，确保不同可用区之间的数据实时同步与校验，为上层业务提供可靠的数据基础。
- **基础设施层：** 作为整个容灾架构的根基。
    - 提供 **计算、存储、网络资源**，包括自建的数据中心、服务器集群与专线网络，以及基于公共云的服务，构建可靠的物理与虚拟资源环境。

这些层次共同构成了小红书业务侧的整体容灾框架，从根源上保障系统的稳定运行。

容灾架构的分层设计与全局管控

小红书容灾架构



小红书的容灾架构设计，不仅涵盖了从应用层到基础设施层的全方位技术考量，更通过一套全局管控机制，确保了业务连续性与稳定性。

架构总览：

- **分层设计：** 整个架构自上而下分为流量层、接入层、业务层、基础架构和基础设施五大核心层级，同时辅以强大的全局管控面。
- **全局管控面：** 容灾工作不仅仅是技术架构，更需一套体系化的机制，实现对整个系统的监控、调度与演练，从而达成高效的灾难恢复能力。

从纵向看全局管控：

- **平台支撑：** 依靠一系列平台和系统，包括可观测平台、服务治理平台、发布平台、预案平台、故障管理平台、配置中心、容灾中控、容器平台等，实现对全链路的精细化管理。
- **三大核心能力：**
  - **可观测：** 建立全链路监控体系，能够快速发现业务层中的问题，判断故障是局限于特定业务场景、多个业务线，还是某个单机房的异常，并迅速定位根因。
  - **可调度：** 在故障发生时，同步进行问题定位与止损。这包括对流量进行精细化调度，如流量切流、逃逸以及数据库的主备切换能力，以最大限度降低影响。

- **可演练：**具备常态化的容灾演练机制，通过演练平台对上述能力进行定期验证，包括演练生命周期管理和容灾中控能力的建设，确保整个容灾体系的有效性和可靠性。

混合云容灾规范与标准要求

混合云容灾规范是小红书在复杂混合云环境下，为保障业务高可用和连续性，所建立的一套重要标准。该规范旨在明确混合云环境下的容灾建设要求，特别是针对单机房故障及专线故障等常见场景提供标准化的解决方案，最终目标是赋予系统强大的“逃逸能力”，确保在面对灾难时能够快速响应、恢复或转移业务。这份规范从流量层、业务层、基础架构层和管控面等多个维度，构建了一个全面的容灾体系：

- **流量单元：域名 (流量层)** 在流量层面，我们聚焦于构建弹性且可调度的流量体系。
    - **GSLB (全局负载均衡)与 DNS：** 通过全局负载均衡（GSLB）和域名系统（DNS）的精妙协同，我们实现了跨地域的流量智能调度与分发，这不仅能有效引导用户请求至最优数据中心，更是确保流量层在面对局部故障时能快速切换，保障业务永续的基石。
    - **预案建设：** 针对流量层可能出现的各种故障，我们都制定了详细的应急预案并定期演练，确保在突发情况下，能够迅速启动预案，将故障影响降至最低。为了避免“木桶效应”中的短板问题，我们可能还会从每个

业务线的视角进行爆炸半径解耦，甚至为每个业务线独立配置域名，以此来分散风险，增强整体的韧性。

- **SO 场景 (业务层)** 在业务层面，我们关注核心业务的连续性，并与研发、产品团队共同定义和 S0、S1 等级的关键业务场景。
    - **核心接口与多活部署机房容量冗余：** 我们对业务的核心接口设定了严格的容灾要求，并通过跨数据中心的多活部署以及充足的容量冗余来确保，即便某个机房发生故障，业务也能无缝切换，持续对外提供服务。
    - **跨单元强弱依赖管理：** 深入分析业务单元间的强弱依赖关系至关重要。
        - **弱依赖熔断：** 对于定位为弱依赖的服务，我们实施了熔断机制。一旦被依赖的服务出现故障，系统能够迅速切断关联，防止故障蔓延，确保核心业务不受影响。
        - **强依赖单元闭环：** 而对于强依赖服务，我们着重确保其在容灾场景下能够形成业务闭环，维护整体业务的完整性和一致性。
- **中间件&数据库 (基础架构&设施)** 在基础架构与设施层面，我们严格管理核心组件，确保其稳定性和容灾能力。
    - **标准中间件版本与关键参数底限要求：** 统一并强制采用标准化的中间件版本，是保障系统稳定性和可维护性的基础。同时，对中间件和数据库的关键参数，如超时设置和连接数管理，也设定了明确的底线要求，以避免资源耗尽或长时间阻塞。例如，在数据库选型上，我们可能强制要求生产环境优先使用 MySQL，而非其他不易运维或不符合公司标准的数据库服务，实行强管控。
    - **数据库、MQ 等多活方案选型：** 对于数据库、消息队列（MQ）等核心基础设施，我们不仅选择并且推行了多活架构方案，以确保它们在关键时刻能够提供持续的服务能力。
    - **专线 QoS：** 同时，对底层专线的服务质量（QoS）也提出了严格要求，保障跨地域数据传输的可靠性和效率。
- **管控面分级 (管控面)** 在管控面，其核心在于保障容灾逃逸能力和指令的精准高效下发。

- **容灾逃逸能力，指令100%下发：** 管控面必须具备强大的容灾逃逸能力，能够在灾难发生时，确保所有指令（如配置下发、自愈操作等）能够100%精准且及时地下发至所有受影响的组件和设备，快速引导系统恢复。
- **管控面强弱依赖和降级策略：** 管控面本身也是一个服务集合，其内部组件间的强弱依赖关系同样需要清晰梳理。我们针对管控面自身的依赖关系制定了详细的降级策略，确保在部分组件故障时，管控面依然能够保持核心功能运行，避免因管控面自身问题而加剧灾情。

管控面容灾建设思路

管控面容灾建设的整体思路，是基于管控面在故障场景下的作用所构建的。管控面平台作为止损专家应对故障的关键工具，在保障业务高可用性方面起到重要作用。因此，需要围绕其核心特点，构建一套全面的容灾体系。

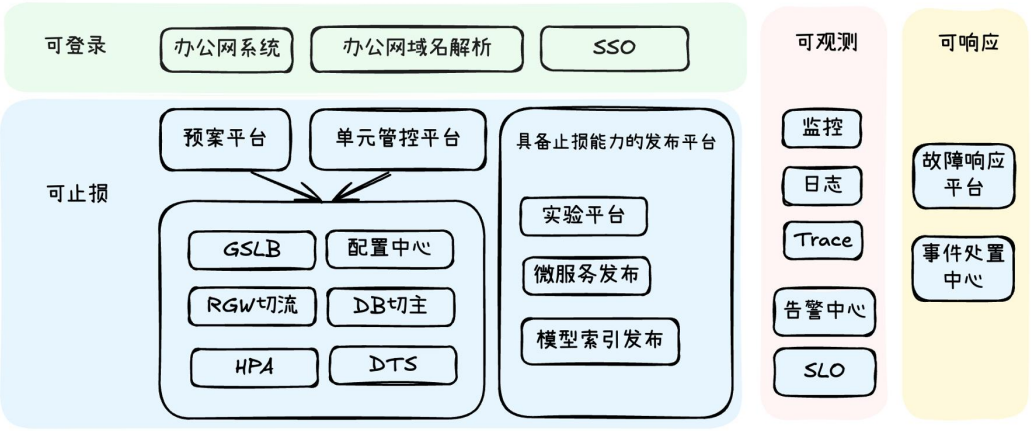
以下是管控面容灾建设的四大核心能力：

- **可登录：确保止损专家能够访问平台。**
  - 在故障发生时，第一要务是确保止损专家能够顺利登录止损平台，以便执行应急操作。
  - 尽管通常我们会依赖企业内部的单点登录（SSO）系统，但这本身也需要有足够高的服务等级目标（SLO）保障。
  - 对于这种强依赖SSO的登录模块，需要设计降级方案，以保证专家在紧急情况下，能够绕过潜在的SSO故障，成功登录系统并执行预案。

- **可止损：确保止损命令能够有效执行，并覆盖所有相关平台。**
    - 止损能力是管控面最核心的功能，它要求止损命令能够顺利下发并生效。
    - 这不仅包括预案管理平台，更需要将电商业务涉及的单元管控平台纳入止损范围。
    - 此外，所有具备发布能力的平台，都应将止损能力作为核心要求，并整合进整体容灾体系中，确保在发布过程中一旦出现问题，能够立即进行止损操作。
- **可观测：实时掌握止损效果和业务恢复趋势。**
    - 在故障期间及止损操作之后，我们需要借助可观测系统，实时监控各业务域的核心指标。
    - 通过北极星指标等关键数据，能够清晰地观测到业务的表现，判断止损措施是否有效，以及业务恢复的趋势和进度。
    - 这种端到端的实时监控能力，对于快速评估故障影响和决策很重要。
- **可响应：构建应急响应机制。**
    - 应急响应体系是缩短平均恢复时间（MTTR）的关键。
    - 需要依托故障响应平台，明确定义包括指挥员、定位专家、止损专家在内的核心角色及其职责。
    - 通过明确的角色分工和协同流程，保证在真实故障发生时，组织能够有序地进行响应和处置，减少故障对业务的影响。

管控面四大核心能力详解

管控面容灾建设



管控面容灾建设旨在确保核心控制功能在灾难面前保持可用性，并能快速恢复。我们将管控面能力归纳为四大关键维度：可登录、可止损、可观测和可响应。通过这些能力的构建，我们能够为管控面提供最基本的容灾保障。

1. 可登录 (Login Availability)

- **办公网系统与办公网域名解析**：确保在任何容灾场景下，内部办公系统及其域名解析服务能够持续可用，为员工提供稳定的访问入口。
- **SSO (Single Sign-On, 单点登录)**：提供统一且安全的身份认证服务，保障用户在灾备期间能够顺畅登录到各类管控系统。

2. 可止损 (Loss Minimization/Damage Control)

- **预案平台**：集中存储和管理应急预案与操作流程，在灾难发生时迅速指导恢复，以最小化损失。
- **单元管控平台**：依照业务单元或区域进行流量与资源精细化管控，通过局部故障隔离机制，有效防止故障跨区域扩散。
    - **GSLB (Global Server Load Balancing)**：实现跨数据中心或区域的流量智能调度，是异地多活和快速流量切换的基础。
    - **配置中心**：集中管理应用配置，支持动态刷新与版本回滚，确保配置变更的可靠性与安全性。

- **RGW 切流 (RGW Traffic Switching)**：通过特定网关服务实现快速流量切换，应对区域性故障。
- **DB 切主 (Database Master Switch)**：数据库主备自动/手动切换机制，保障数据库服务的连续性。
- **HPA (Horizontal Pod Autoscaler)**：根据负载动态调整 Pod 数量，提供弹性扩缩容能力，增强系统韧性。
- **DTS (Data Transmission Service)**：保障数据在不同系统间的同步、迁移与备份，维护数据完整性与高可用。

- **具备止损能力的发布平台**：发布流程融入风险控制与回滚机制，能够及时发现并阻止潜在问题，甚至自动回滚至稳定版本，规避发布风险。
  - **实验平台**：在生产环境进行小流量或灰度实验，验证新功能或配置的稳定性，有效降低发布风险。
  - **微服务发布**：针对微服务架构，提供精细化的流量控制和部署策略。
  - **模型索引发布**：针对 AI 模型或搜索索引等特定业务数据，具备可靠的发布与回滚能力。

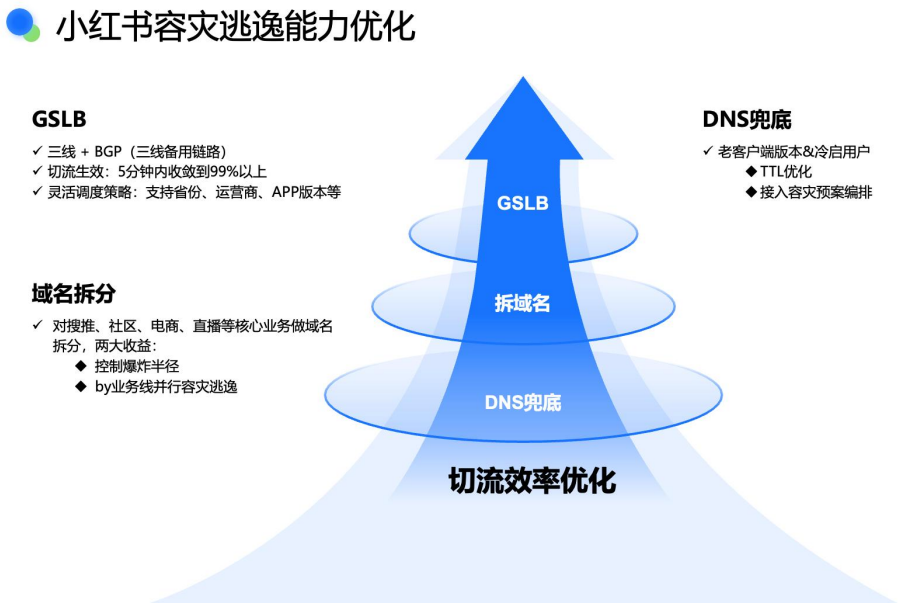
3. 可观测 (Observability)

- **监控**：实时采集系统各项关键指标（如 CPU、内存、QPS、延迟等），用于快速发现异常状态。
- **日志**：收集系统与应用运行日志，为故障诊断提供详尽的历史数据。
- **Trace**：分布式调用链追踪，清晰展现请求在复杂微服务架构中的流转路径和时间消耗，协助定位性能瓶颈与故障点。
- **告警中心**：基于监控数据与日志配置告警规则，及时通知 SRE 团队处理异常。
- **SLO (Service Level Objective)**：定义服务的可用性、性能等预期目标，作为衡量服务健康状况与容灾能力的重要基准。

4. 可响应 (Responsiveness)

- **故障响应**：构建高效的故障响应机制，确保在问题发生时能快速、有效地处理，从而将影响时间最小化。

容灾逃逸能力优化



针对容灾逃逸能力进行技术优化，通过提升切流效率及相关机制，将故障逃逸时间控制在 10 分钟以内，确保业务连续性。

在实践中不断优化容灾能力，主要围绕以下几个核心策略展开：

- **接入层优化：GSLB 调度与切流**
    - **GSLB 流量调度**：小红书通过自研的 GSLB（Global Server Load Balancing）管控层，实现了对 DNS 依赖的解耦，能够根据实时流量和机房负载，灵活调整各机房的流量权重，实现流量调度。
    - **客户端 SDK 联动**：联动客户端 SDK，将 GSLB 能力嵌入其中，使用户流量能够接入到最优或指定的接入点。
    - **多链路高可用**：GSLB 接入了包括电信、联通、移动在内的“三线”网络链路，并辅以 BGP（Border Gateway Protocol）备用链路，确保网络传输的稳定性和冗余性。
    - **切流效率提升**：通过 GSLB SDK 的 TTL 优化，将流量收敛效率提升至 5 分钟内生效 99%以上，缩短了故障恢复时间。
    - **灵活调度策略**：GSLB 支持基于省份、运营商以及 APP 版本等多维度进行流量调度，实现故障隔离和流量引导。
- **域名拆分：控制故障爆炸半径，实现业务线并行逃逸**

- **单一域名问题**：历史上，小红书使用一个大型的单一域名承载所有业务，这导致在故障发生时，切流操作会依赖于单一域名的生效，影响面广泛。
- **核心业务域名独立**：为了解决这一痛点，我们对搜推、社区、电商、直播等核心业务进行了域名拆分，使每个核心业务都拥有独立的域名。
- **两大收益**：
  - **控制爆炸半径**：当某个业务发生故障时，影响范围仅限于该业务对应的独立域名，有效降低了故障对整体业务的影响。
  - **按业务线并行容灾逃逸**：不同业务线可以根据自身情况独立进行容灾切换，提升了容灾效率和灵活性。

- **DNS 兜底：保障老版本与冷启用户的韧性**
  - **兜底场景**：针对部分老客户端版本用户和冷启动用户，他们可能无法及时获取 GSLB 下发的最新调度指令，从而需要依赖传统的 DNS 解析。
  - **TTL 优化**：对此，我们对 DNS 记录的 TTL（Time To Live）进行了优化，以缩短 DNS 缓存更新时间，加速 DNS 层面的故障切换。
  - **容灾预案编排**：我们将域名切换能力纳入接入层容灾预案编排流程，确保在 DNS 层面出现故障时，也能有预设的应对方案，并能快速启动。

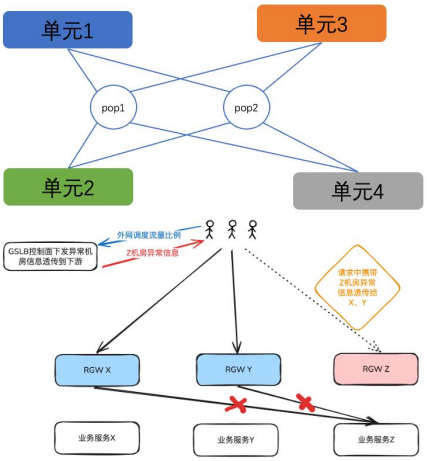
所有这些优化措施都指向一个核心目标：**提升切流效率**。它们构成了小红书在亿级设备与千万用户规模下，从预防到响应、从发布到复盘的全链路高可用保障。

专线故障应对策略

小红书容灾逃逸能力优化

应对专线故障我们做好准备了吗?

- ✓ 切流指令下发强依赖管控面 -> 多活机房的专线链路
  - ✓ 机房专线物理对等，点对点且冗余
  - ✓ 切流架构优化：GSLB联动内网调度



针对专线故障，设计了相应的应对策略：

- **应对专线故障核心问题** 核心问题在于，日常中许多管控面的切流指令下发，都依赖于机房间的专线链路。一旦专线出现故障，这些关键操作可能受阻。
- **解决方案与准备措施** 为了解决这一潜在风险，从两个层面进行了优化：
  - **基础设施层面：专线物理冗余与自动回切**
    - 对跨机房专线链路进行了物理层面的冗余设计。每个业务单元与另一个单元的连接，不是单点链路。
    - 当其中任意一条专线链路因故中断（例如物理受损），流量将自动回切（failback）到另一条备用链路，确保链路的持续可用性。这种"点对点且冗余"的部署，增强了网络的弹性。
  - **流量调度层面：外网与内网联动调度优化** 在极端场景下，若多条专线链路同时失效，通过结合上游流量调度层与接入层网关，构建了一套内外网联动的机制。具体实践如下：
    - **GSLB 联动调度：** 利用 GSLB（全局负载均衡）的外网切流能力，结合内网接入网关的协作，实现流量调度。
    - **故障信息透传：** 如下图所示，当某个机房（如 Z 机房）出现专线故障时，GSLB 控制面会感知并向下游组件透传异常机房信

息。GSLB 会调整外网流量的调度比例，将流量从故障机房切走，并向客户端或上游发送 Z 机房的异常信息。

- **客户端感知与流量纠偏：** 用户请求首先流向 RGW（Request/Router Gateway）。当客户端接收到故障机房异常的 Header 信息后，它将此信息携带到具体的业务服务接入网关层。接入网关识别到此信息后，进行流量纠偏，不再将请求转发至故障机房，从而避免对不可用服务的调用。

容灾中控平台建设与能力拆解

容灾中控：管控面的管控面

"容灾中控"这一核心平台，将其定位为容灾管理体系中"管控面的管控面"，不同于传统管控面侧重于指令和配置的下发，容灾中控更扮演着"监考官"的角色，确保指令的端到端执行成功，并提供状态感知和决策支持。

容灾中控的核心能力包括：

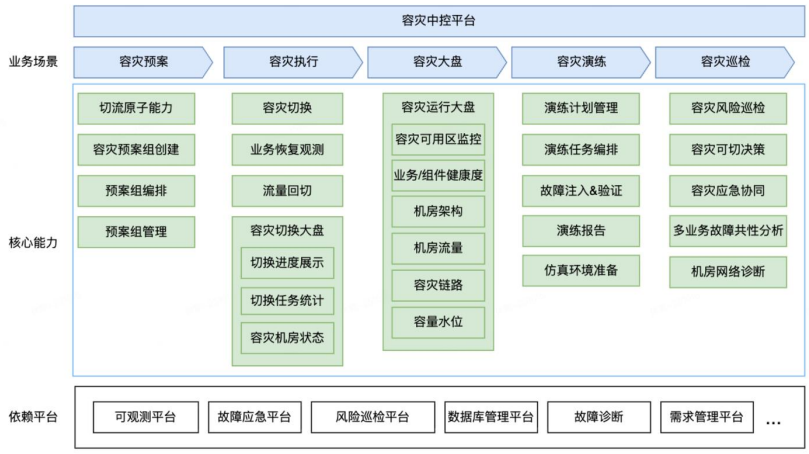
- **容灾监控大盘** 这一模块提供在容灾故障场景下，对业务可用性、基础组件健康度以及基础设施健康状态的实时观测能力。它将分散的监控数据聚合成一个统一视图，使 SRE 团队能够掌握整个系统在灾难情况下的运行态势，及时发现异常并评估影响范围。这能确保在故障应对时刻，决策者能够基于信息做出判断。
- **全局容灾预案** 全局容灾预案旨在实现对容灾逃逸预案执行过程的状态追踪与可视化。它能够显示每个指令或预案是否成功执行、执行进度如何，以及在此过程中是否出现了任何问题。通过提供端到端追溯能力，容灾中控能够辅助容灾指挥

员对逃逸进展进行判断，从而在故障场景中，做出决策，例如调整应对策略、切换备用方案等，确保业务能够快速恢复并稳定运行。

- **容灾演练生命周期管理** 考虑到大型故障的发生频率相对较低，容灾中控通过定期的容灾演练来提升团队的应对能力和平台的成熟度。此模块为容灾演练提供全生命周期的管理能力，从演练的准备阶段（包括风险确认、演练方案设计、预案制定、剧本编写等），到演练的实际执行，再到后续的复盘总结和经验存档。通过演练管理，团队能够不断测试和优化容灾预案、提升操作熟练度，从而在面临灾难时能够有效应对。

容灾中控平台的原子能力拆解

容灾中控：管控面的管控面



基于整体建设思路，将容灾中控平台拆解为一系列原子能力，并逐步优化，目标是在容灾演练或真实故障场景中，确保所有动作有序、有效执行，并在 10 分钟内达成 RTO（恢复时间目标）。这主要涵盖预案执行、业务表现及响应效率等多个核心维度。

业务场景 (Business Scenarios): 容灾中控平台全面覆盖容灾的生命周期，从预防到执行再到评估，包括：

- **容灾预案 (Disaster Recovery Plan):**
  - 切流原子能力：构建流量切换的基础单元。
  - 容灾预案组创建：将相关预案整合，形成可管理集合。
  - 预案组编排：对预案组进行有序调度与组合。
  - 预案组管理：对预案进行全生命周期维护，确保其时效性和准确性。

- **容灾执行 (Disaster Recovery Execution):**
    - 容灾切换：在灾难发生时，快速引导业务流量至可用区域。
    - 业务恢复观测：实时监控业务在切换后的恢复状态与健康状况。
    - 流量回切：在原区域恢复正常后，平稳地将业务流量迁移回原区域。
- **容灾大盘 (Disaster Recovery Dashboard):**
    - 容灾运行大盘：提供容灾系统宏观运行状态的概览。
    - 容灾可用区监控：实时监测各可用区健康度。
    - 业务/组件健康度：细致监控业务及其依赖组件的健康状况。
- **容灾演练 (Disaster Recovery Drill):**
    - 演练计划管理：规划和协调容灾演练的所有环节。
    - 演练任务编排：定义演练的具体步骤和执行顺序。
    - 故障注入&验证：通过模拟故障，检验容灾机制的实际效果。
    - 演练报告：生成详细报告，评估演练成效及发现的问题。
    - 仿真环境准备：为演练构建高仿真度的模拟环境。
- **容灾巡检 (Disaster Recovery Inspection):**
    - 容灾风险巡检：定期审查潜在的容灾风险点。
    - 容灾可切决策：基于巡检结果，评估系统当前是否具备切换能力，并支持决策。

核心能力 (Core Capabilities): 为支撑上述业务场景，容灾中控平台沉淀了以下关键能力：

- **预案管理维度：**
    - 切流原子能力
    - 容灾预案组创建
    - 预案组编排
    - 预案组管理
- **切换执行维度：**
    - 容灾切换
    - 业务恢复观测
    - 流量回切
    - 容灾切换大盘

    - 切换进度展示
    - 切换任务统计
    - 容灾机房状态
- **实时监控与诊断维度：**
    - 容灾运行大盘
    - 容灾可用区监控
    - 业务/组件健康度
    - 机房架构
    - 机房流量
    - 容灾链路
    - 容量水位
- **演练维度：**
    - 演练计划管理
    - 演练任务编排
    - 故障注入&验证
    - 演练报告
    - 仿真环境准备
- **巡检与决策维度：**
    - 容灾风险巡检
    - 容灾可切决策
    - 容灾应急协同
    - 多业务故障共性分析
    - 机房网络诊断

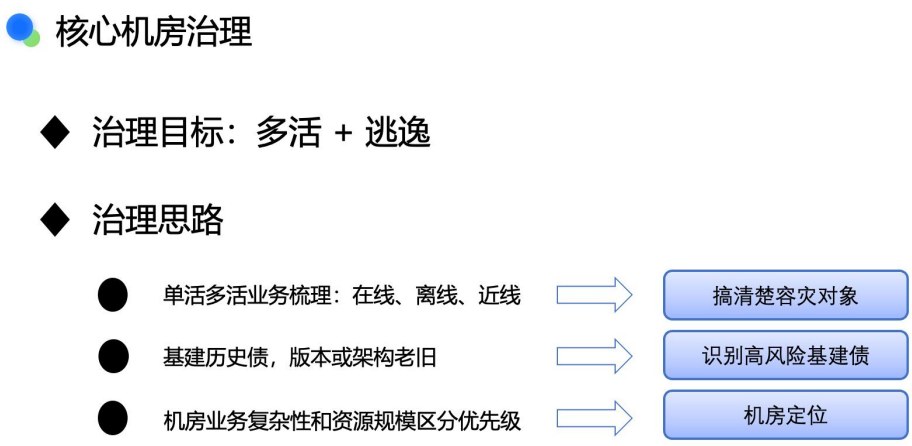
依赖平台 (Dependent Platforms):

容灾中控平台的有效运作，需要依赖一系列底层平台和服务的支撑，通过与其他平台的集成，实现从数据集成到策略执行的闭环，支撑业务连续性保障。具体包括：

容灾中控平台作为业务连续性保障的核心枢纽，其高效运作依赖于多个底层平台和服务的协同支撑。通过与这些平台的深度集成，实现从数据采集、风险识别、策略制定到自动化执行的完整闭环。主要依赖平台包括：

- 可观测平台：提供实时监控指标、日志聚合、链路追踪等能力，为容灾决策提供数据基础和异常告警。
- 故障应急平台：集成故障发现、应急响应、故障自愈等能力，在容灾场景下快速响应和处理突发事件。
- 风险巡检平台：定期执行容灾配置巡检、资源健康度检查，提前发现潜在风险点。
- 数据库管理平台：管理数据库实例的主从关系、备份恢复策略，支持数据层面的容灾切换。
- 故障诊断平台：提供根因分析、影响面评估等能力，辅助容灾决策和事后复盘。
- 需求管理平台：记录和追踪容灾演练计划、容灾切换审批等流程，确保操作合规可追溯 等等

核心机房治理策略与思路



核心机房治理的目标聚焦于“多活”与“逃逸”能力的建设，旨在保障业务连续性和高可用性。在这一过程中，我们面临着许多挑战，尤其是在处理小红书早期部署的、承载复杂业务的“老旧”异地机房时。例如，AWS 曾经的故障就揭示了在一个默认大型机房中，当多种组件混杂部署时，发生问题后难以快速定位和处理的困境。因此，对核心机房，尤其是主用机房进行治理，是容灾建设中至关重要的一环。

治理目标：多活＋逃逸

构建一套能够支持业务多活运行，并在灾难发生时能够平滑、有效地实现业务“逃逸”的机制，是核心机房治理的终极目标。这要求我们确保重要业务在任何单一数据中心故障时，都能够正常运行或快速恢复。

治理思路

1. 单活多活业务梳理：在线、离线、近线

在机房治理的第一步，我们需要对所有业务进行精准分类和深入梳理，明确其对容灾的需求和优先级。

- **业务属性识别：**分析业务是属于高实时的“在线业务”，支持异步或延迟处理的“离线业务”，还是介于两者之间的“近线业务”。这直接影响其容灾策略的制定。
- **多活可行性评估：**识别当前单活业务中，哪些可以通过架构调整，升级为具备多活能力的业务，提升整体可用性。
- **搞清楚容灾对象：**通过对业务的全面梳理，明确哪些是需要纳入容灾保护范围的“容灾对象”，确保重要业务无遗漏。

2. 基建历史债，版本或架构老旧

历史遗留的技术债务，如老旧的中间件版本、过时的基础设施组件，以及陈旧的系统架构，是核心机房治理中的一大挑战。

- **识别风险点：**持续地识别这些由历史原因形成的、可能导致系统不稳定的高风险“基建债”。
- **跨团队协作：**与基础设施团队紧密合作，共同评估并制定风险消减计划，推动相关治理工作的落地。
- **识别高风险基建债：**建立技术债务清单，划分优先级，逐步偿还，以提高系统的稳定性和可维护性。

3. 机房业务复杂性和资源规模区分优先级

不同的机房其内部承载的业务复杂性、资源投入会有很大差异，因此需要结合整体规划，对不同机房赋予明确的定位和治理策略。

- **机房定位评估：**结合小红书整体的机房布局和演进规划，重新审视各个机房的角色和战略定位。例如，某个机房过去可能是主用，现在是否需要调整为备用或逐步淘汰。

- **业务资源分析：** 综合评估机房中业务的复杂程度及其资源占用规模，以此作为治理工作优先级和投入的决策依据。
- **机房定位：** 明确每个机房在多活和逃逸体系中的具体职责，例如主中心、容灾中心等，并根据定位优化其治理策略和资源配置。

核心机房风险治理实践

1. 业务梳理与风险确认

准确识别核心业务是风险治理的起点。采取以下措施：

- **稳定性元数据定期刷新：** 构建并定期更新系统的稳定性元数据，包括业务线、场景、接口、底层服务、存储，以及相关负责人等关键信息。这些元数据作为信息源，能有效补充传统文档的不足。此外，持续识别新的 S0（最高级别关键）场景，并按需进行治理。
- **历史故障分析：** 分析过往故障，从中吸取经验教训，找出系统薄弱环节，避免同类问题再次发生。
- **日常客诉补充：** 用户投诉是发现潜在风险和系统问题的重要线索。通过分析客诉，一些原本可能被标记为较低重要级别的业务功能，在用户眼中却很重要。基于这些实际反馈，刷新业务的重要性评级，并进行相应的治理。

2. 老 K8S 集群风险治理

随着云原生技术的普及，K8S 集群已成为基础架构的核心。但历史遗留问题不容忽视：

- **高风险版本集群汇总：**小红书自 2017 年起引入 K8S，至今已积累了大量老版本集群。我们将与容器团队紧密协作，明确高风险 K8S 集群的定义及其具体风险点。
- **风险解除方案：**针对识别出的高风险集群，我们将推动：
  - **集群原地升级：**在满足条件的情况下，优先尝试进行原地升级，提升集群的稳定性和安全性。
  - **业务迁移：**若原地升级不可行，我们将协同业务团队，将应用服务从高风险集群迁移至更安全、更稳定的新集群，从而彻底解除风险。

3. 数据引擎风险治理

数据是业务的生命线，其链路的稳定至关重要。针对数据引擎的风险，我们与数据链路团队紧密合作，制定了分层级的灾备策略：

- **双活场景（实时数据 L0）：**针对最高优先级（L0 级）的实时数据场景，建设**全链路双活能力**。这意味着通过冗余设计，在任何一条数据链路出现异常时，都能实现无缝切换，确保**数据流 0 中断**。
- **单活逃逸场景（实时数据 L1）：**对于次优先级（L1 级）的实时数据场景，我们建设**单链路同云逃逸能力**。其目标是在单链路异常时，能在**30 分钟内完成止血**，即快速恢复服务，最大程度减少业务影响。这可能通过有状态服务的额外容量部署（如 150%容量），或无状态任务在故障时进行资源抢占来实现。

4. 存储风险治理

存储作为所有业务数据的最终归宿，其可靠性是核心机房治理的重点：

- **管控层多机房部署：**存储的控制平面和管理平台实现**多机房部署**，以提升自身的可用性和韧性，确保在单一机房故障时，存储的管理能力不受影响。
- **数据层多机房部署与零丢失保障：**推动核心存储，特别是 **S0 集群和 MySQL 数据库，实现多机房部署**。这一举措旨在实现 **RPO（Recovery Point Objective，恢复点目标）为 0**，确保在故障情况下**数据零丢失**，从而为各业务线的核心场景提供数据完整性保障。

以上是围绕核心机房治理所规划的四大方向和具体举措。通过这些系统性的工作，能够持续巩固核心机房的稳定性、可用性，并支撑业务的持续发展。

容灾演练实践总览

03

小红书容灾演练实践

容灾演练三阶段发展路径

小红书容灾演练实践

小红书自 2024 年起，便开启了体系化的容灾演练实践，并规划了清晰的发展路径。

第一阶段：容灾演练支撑 (Y24)

去年，我们工作的核心聚焦于组织建设与基础能力保障，旨在确保容灾演练的有效执行。

- **容灾组织及机制：**

- 建立了“容灾全局双周例会”的常态化沟通机制。
- 明确了“周期性演练（双月，半年）”的定期演练制度。

- **容灾演练能力：**
  - 着重建设了“预案能力”，确保应对策略清晰完备。
  - 提升了“攻击能力”，以模拟真实故障场景。
  - 强化了“可观测能力”，为演练过程提供透明视图和数据支持。

在这一阶段，我们通过文档和手动指令执行，成功地将演练跑起来，并识别出机房及业务维度的容灾风险。

第二阶段：演练深度 (平台化) (当前)

目前，我们正致力于通过平台化手段，提升演练的广度和深度，以实现更高效的组织协同和更显著的演练效果。

- **演练平台升级：**
  - 目标是实现“容灾风险&架构可视化”，清晰展现架构脆弱点。
  - 将“演练生命周期在线化”，通过平台工具管理整个演练流程。
  - 实现“模拟攻击能力平台化”，将攻击模拟能力整合到统一平台，以发现更多潜在的容灾风险。
- **演练场景升级：**
  - 拓展至“重大历史故障演练支撑”，复现并解决历史遗留问题。
  - 覆盖“基础设施（机房，网络）演练支撑”，确保底层设施的韧性。
  - 强化“基础组件演练支撑（基础设施自演练）”，使基础设施能够自主进行演练，提升其可靠性。

第三阶段：演练体系化 (常态化) (Y26)

展望未来，我们将进一步推动容灾演练的常态化、指标化和自动化，迈向自主闭环的韧性架构。

- **演练指标体系：**
  - 构建完善的评估体系，通过“基于架构巡检识别架构风险”。
  - “基于架构风险指引演练指标”，量化风险并指导演练方向。
  - 关注“风险治理率/演练覆盖率，成功率”等关键指标，确保演练效果可衡量、可优化。
- **架构自演练支撑：**

- 实现“业务自演练”，使业务自身具备故障模拟与恢复能力。
- 实现“基础设施自演练”，确保基础设施能够自主进行演练，提升其可靠性。
- 最终目标是实现“常态化全局演练——常态化，自动化”，通过混沌工程等日常机制，对业务强依赖和过去业务架构发现的问题进行小范围的验证，形成风险发现与闭环的自主机制。

容灾演练的四大核心维度

小红书容灾演练实践

小红书的容灾演练体系主要围绕以下四大核心维度展开：

- **组织**
  - **制定演练计划：** 容灾演练的成功首先依赖于周密的计划。我们会自上而下制定详细的演练计划，并与公司高层充分对齐，以确保演练目标与整体业务目标高度一致。
  - **明确演练目标：** 演练需要承载具体目的，无论是验证新系统的可用性、提高团队响应速度，还是发现潜在的架构缺陷。
  - **明确组织角色和分工：** 划分参演人员的职责和权限是演练顺畅执行的关键，确保每位成员都能各司其职，高效协同。
- **混沌能力**
  - **模拟机房孤岛：** 通过模拟数据中心间的网络隔离，测试系统在部分资源失效时的独立运行能力。

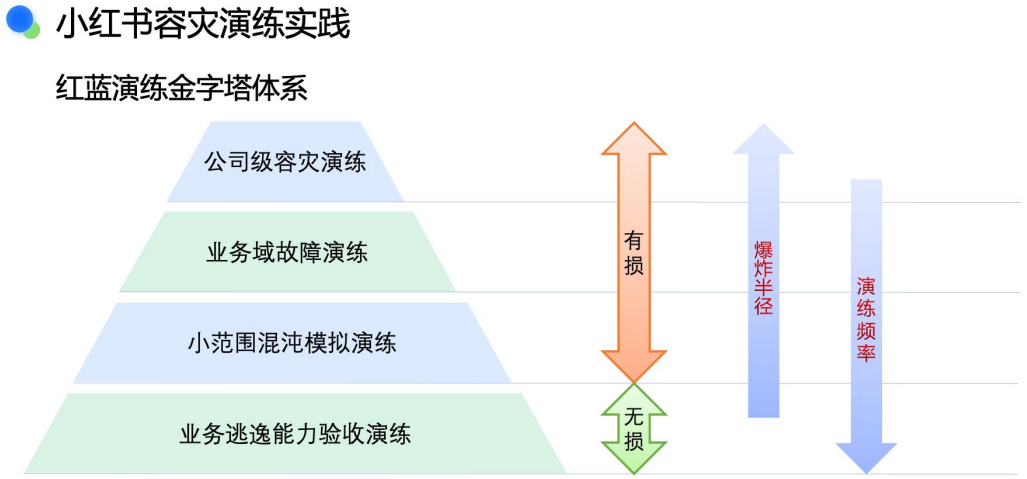
- **模拟专线断网：**模拟核心网络链路中断情境，验证服务通信的韧性。
- **模拟弱网丢包环境：**复现网络质量不佳导致的丢包和延迟，评估系统在非理想网络条件下的表现。
- **模拟调用延迟：**注入服务间调用响应时间增长的故障，测试系统在慢速请求下的稳定性和降级处理机制。 容灾演练的效果在很大程度上依赖于这些混沌能力，它们能够模拟真实且多变的故障场景，从而检验系统的鲁棒性。

- **剧本**
- 演练剧本是容灾演练成功的关键，它能够引导我们发现系统深层次的问题，还能在演练过程中暴露团队协作的短板，这些问题可能涉及人员操作、架构设计或流程规范。具体环节包括：
  - **前置风险评估：**演练启动前，我们会对潜在风险进行全面评估，确保演练过程可控。
  - **演练通告：**及时向所有相关方发布演练通知，以保证信息透明并为各项准备工作预留充足时间。
  - **蓝军攻击：**由专门的“蓝军”团队模拟攻击或系统故障注入，正式启动演练流程。
  - **组织执行预案逃逸：**演练团队依据预置的应急预案，迅速执行止损、隔离等操作。
  - **故障恢复：**在故障发生后，迅速执行恢复操作，使系统恢复正常运行。
  - **预案回切和风险复盘：**故障恢复后，系统将安全回切至正常状态，并对整个演练过程进行详细总结与反思，以便持续改进和优化。
- **预案设计**
  - **预案编排：**设计并编写各种故障场景下的应急处理预案，确保每一种可能情况都有应对之策。
  - **切流效率：**评估并持续优化在故障发生时，流量快速、平滑切换到备用系统的效率。

- **存储切主：**关注提升数据库或存储系统在主备切换时的效率和成功率。
- **容灾预案覆盖：**持续评估并扩大现有容灾预案所能覆盖的故障场景范围。

- 预案设计是容灾能力的基础，是应对故障挑战的依据。没有一套完善有效的预案，任何容灾演练都难以发挥作用。核心目标是构建体系化的容灾能力，确保在面对故障时，能够快速、有效地响应和恢复。

红蓝演练金字塔体系



在小红书的容灾演练体系中，我们遵循业界普遍采用的金字塔模型，其核心思想是分层、逐步递进地提升演练的深度和广度。

红蓝演练金字塔体系

该体系由下至上共分为四个层次，旨在通过不同规模和频率的演练，全面提升系统的容灾能力：

- **业务逃逸能力验收演练：**
  - 作为金字塔的基础，这类演练频率最高。
  - 主要聚焦于单一业务模块或组件在特定故障下的快速脱离故障影响，确保核心功能不受损。
  - 通常以“无损”方式进行，爆炸半径最小，对生产环境影响极低，但演练频率最高。

- **小范围混沌模拟演练：**
    - 在逃逸能力验收的基础上，进一步引入小规模的混沌工程实践。
    - 通过注入随机故障，模拟局部网络抖动、服务异常等场景，验证系统的弹性与自愈能力。
    - 这类演练同样倾向于“无损”操作，爆炸半径有限，且具备较高的演练频率。
- **业务域故障演练：**
    - 此层演练提升到业务域层面，涉及多个相互关联的业务模块。
    - 模拟业务逻辑层面的复杂故障，例如关键服务中断、数据链路异常等，测试跨服务间的协同容灾能力。
    - 演练频率适中，可能伴随一定的“有损”风险，爆炸半径相对扩大，挑战性也随之增加。
- **公司级容灾演练：**
    - 位于金字塔的顶端，是最高级别的综合性演练。
    - 模拟系统级或数据中心级的灾难性故障，覆盖端到端的业务流程和异地多活等复杂场景。
    - 这类演练频率最低，通常是“有损”的，具有最大的爆炸半径，涉及的业务线众多，对业务影响和挑战也最大。

演练属性趋势

该金字塔体系还通过趋势箭头形象地展示了不同类型演练的属性：

- **有损/无损：**
    - 越是底层的演练，如业务逃逸能力验收和小范围混沌模拟，越趋向于“无损”进行，对生产环境几乎不产生影响。
    - 而随着演练层级的升高，尤其是业务域故障演练和公司级容灾演练，其演练性质更可能趋向于“有损”，这意味着在演练过程中可能需要中断部分服务或造成业务影响，以真实模拟灾难场景，从而更全面地评估容灾方案的有效性。
- **爆炸半径：**

- 从金字塔底部到顶部，演练的“爆炸半径”逐渐增大。这意味着底层演练影响范围小，而顶层演练则会涉及更广泛的业务系统和用户群体，因此其潜在影响和风险也更大。

- **演练频率：**
  - 演练频率与金字塔层级呈反向关系，即越是底层的演练，其频率越高，可在日常开发运维中频繁进行。
  - 而越是顶层的演练，例如公司级容灾演练，由于其复杂度、资源投入及潜在风险，演练频率会相对较低。

通过这样的分层金字塔体系，小红书能够在日常中高频次地开展底层的"无损"小范围演练，基于这些日常实践和业务线风险验收情况，我们也会组织更高层级的、更真实的容灾演练，以验证容灾机制的有效性，确保业务在面对各类故障时都能具备韧性。

演练组织分工与职责划分

小红书容灾演练实践

演练组织分工

组织阵型

	人员（技术风险驻场；其他按需，可以Online）	职责
总指挥		整体把控
组织&播报		• 协助建设演练方案，拉齐各方同学评审 • 组织演练，确认参与人及角色安排 • 把控演练现场节奏 • 收集各方向关键信息并播报
专家组	蓝军专家组： 红军专家组： 红军观察组：	• 盯全局可用性大盘包括舆情等信息，及时感知异动情况 • 异常情况下及时评估业务影响做出判断 • 观察演练过程识别异常点 • 把握整体红蓝流程，充当横向红/蓝军
PMO		• 信息同步、周知以及必要的横向协作支持
HRBP		• 驻场人员的保障、提供居家办公咨询
舆情		• 知晓并关注舆情问题
红军业务组	各业务域POC同学	• 充当红军各领域指挥角色 • 负责各自业务内的事故应急和其他突发场景决策 • 有客诉或内部反馈，及时在演练群同步

本章旨在阐释在面临复杂的容灾挑战时，如何通过精细化的组织架构与明确的职责划分，确保演练的高效进行与真实效果。

容灾演练并非单一技术团队的使命，它需要公司多部门、多维度的协同参与。我们将参与演练的各个角色及其职责进行了界定，不仅包括技术人员，还涵盖了项目管理、人力资源以及业务部门等职能，共同构建起应对突发事件的协同机制。

在具体的演练组织分工中，我们**细致考量了公司内部的各类角色**，不局限于技术同学，更将项目管理、HRBP、职能部门乃至舆情组纳入其中。

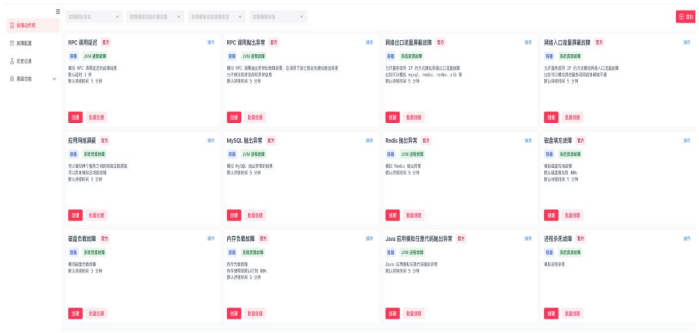
我们明确了每个角色的具体职责，确保所有参与者都清楚自己在整个容灾演练中的定位与贡献，并通过充分的对齐与沟通，使各方能够高效协同，共同完成目标。无论是整体的方案建设、演练组织执行，还是关键信息的收集播报，以及对全局可用性、业务影响的评估判断，以及人员保障、舆情监控等，每个环节都有对应角色承担，从而共同支撑起容灾演练的实践。

混沌动作库与故障模拟

小红书容灾演练实践

混沌动作库->真实故障模拟

故障类型：
- ✓ 网络类故障
  - ➢ 延迟、不可达、抖动、丢包
- ✓ 模拟服务全挂
- ✓ 连接异常



在小红书，仅依靠常规演练难以覆盖所有风险，因此在应对大规模或特定场景演练时，会依赖一套设计的"混沌动作库"。这套系统旨在模拟真实的故障场景，从而验证系统的容灾能力。

核心理念：混沌动作库 → 真实故障模拟

通过主动注入各类故障，来检验和提升系统对异常情况的韧性。幻灯片左侧列举了在容灾场景中重点关注的故障类型：

- **网络类故障：** 这类故障在容灾场景中极为常见，且影响广泛。
  - **延迟：** 模拟网络通信的响应迟缓，测试系统在弱网络条件下的表现。
  - **不可达：** 模拟服务间通信中断，验证路由和自动故障转移机制。
  - **抖动：** 模拟网络不稳定，测试系统的健壮性。
  - **丢包：** 模拟网络数据丢失，检验数据传输的可靠性。
- **模拟服务全挂：** 模拟核心下游服务完全不可用的极端情况，以评估系统对服务依赖的容错能力。

- **连接异常：**模拟数据库连接、缓存连接或其他关键服务链接出现异常，例如链接数被打满，导致整个服务不可用。

这些故障场景的模拟，为业务团队提供了辅助工具，以验证其容灾策略的有效性。

幻灯片右侧展示了我们实际使用的混沌工程平台界面，其中的每项“混沌动作”都以卡片形式呈现，清晰地描绘了可注入的故障类型，覆盖了日常应用中的多种场景：

- **RPC 类故障：**
  - RPC 调用延迟：模拟微服务间通信耗时增加。
  - RPC 调用异常：模拟微服务调用失败。
- **数据库故障：**
  - MySQL 跑出异常：模拟数据库服务异常，如连接中断或查询失败。
- **缓存故障：**
  - Redis 异常：模拟分布式缓存系统出现故障。
- **网络故障：**
  - 网络出口流量异常：模拟服务对外通信受到限制。
  - 网络入口流量异常：模拟服务接收外部请求受到影响。
- **资源类故障：**
  - 磁盘饱和故障：模拟服务器存储资源耗尽。
  - 内存饱和故障：模拟服务器内存资源不足。
- **应用层及其他故障：**
  - Java 应用模拟任意代码跑出异常：模拟应用代码层面的运行时错误。
  - 进程异常：模拟关键进程崩溃或非预期行为。

每个卡片不仅包含故障的名称和简要描述，还提供"创建"与"恢复故障"的操作按钮，使得故障注入与恢复过程直观且易于管理。通过这种理论与实践相结合的方式，能够系统地提升小红书的整体服务韧性。

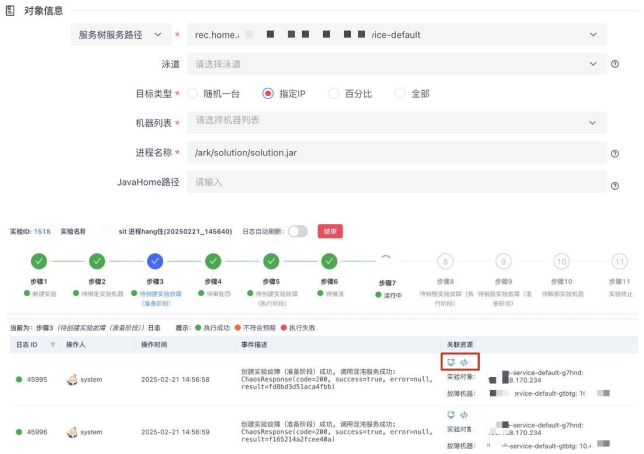
混沌动作库的应用场景与实践

小红书容灾演练实践

混沌动作库->真实故障模拟

应用场景:
- ✓ 强弱依赖治理
  - ➢ 预期依赖 -> 链路加固 -> 混沌验证
- ✓ 机房断网
  - ➢ 数据链路双活演练
  - ➢ 在线多活业务演练

施加范围:
- ✓ 单实例（随机、指定）
- ✓ 百分比
- ✓ 全部下游



小红书在容灾演练方面通过"混沌动作库"来模拟真实故障，以验证系统韧性，并保障业务连续性。

应用场景与验证目标：

- **强弱依赖治理：**
  - 通过混沌工程验证服务调用拓扑中的强弱依赖关系对接口成功率的影响。
  - 对强弱依赖进行链路加固（如超时配置、熔断配置）后，使用混沌演练验证加固效果。
- **机房断网演练：**
  - **数据链路双活演练：** 验证跨机房数据链路的双活机制在高压下的稳定性和一致性。
  - **在线多活业务演练：** 模拟机房级故障，确保业务在多个活跃数据中心间能够平稳自动切换。

故障注入施加范围（爆炸半径控制）： 为了精细化控制实验影响，混沌演练支持多种粒度的故障注入目标选择：

- **单实例：** 可以随机选择或精确指定单个服务实例进行故障注入。
- **百分比：** 以一定的服务实例比例作为目标进行故障注入。
- **全部下游：** 将故障扩散至所有相关的下游服务，以模拟更广范围的影响。

混沌演练工具平台：

- **故障注入配置：**
  - **对象信息：** 明确故障注入的服务目标，如通过“服务树服务路径”rec.home...../ice-default 锁定特定服务。
  - **泳道选择：** 支持在特定的环境或流量分组中执行实验。
  - **目标类型：** 提供“随机一台”、“指定 IP”、“百分比”、“全部”等灵活的目标选择方式。
  - **机器列表与进程指定：** 可手动指定目标机器，并通过“进程名称”/ark/solution/solution.jar 和“JavaHome 路径”精准定位故障注入点。
- **实验执行与监控：**
  - **实验 ID 与名称：** 每个实验拥有独特的 ID（如 1518）和描述性名称（如 sit 进程 hang 住(20250221_145640)），便于追溯与管理。
  - **流程图：** 展示实验从“新建实验”到“实验终止”的 11 个步骤，实时反馈各步骤的执行状态（成功、不符合预期、失败）。
  - **当前步骤提示与日志：** 提供当前实验所处步骤的提示信息，并详尽记录操作日志（日志 ID、操作人、操作时间、事件描述、关联资源），特别是在故障创建准备阶段成功时，会显示 ChaosResponse，表明混沌服务响应成功。其中，“关联资源”会具体列出实验对象和服务 IP，以及受影响的故障机器实例。

混沌演练的核心目标在于模拟真实故障，在受控环境中识别并解决潜在的系统脆弱性，从而构建更具韧性的业务系统。

灰度机房环境建设

小红书容灾演练实践

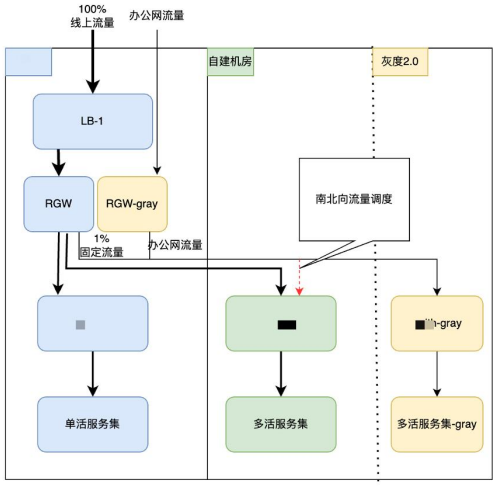
灰度机房：提升演练效率的利器

目标：建设支撑真实、常态化、自动化红蓝演练环境

隔离性

真实性（架构对等）

低成本



本次分享旨在深入探讨小红书的容灾演练实践，特别是如何利用“灰度机房”这一核心工具，显著提升演练效率。

核心目标：建设支撑真实、常态化、自动化红蓝演练环境

为了有效控制演练过程中的潜在风险，避免因演练操作不当而引发大规模生产故障，我们构建了一个专供业务团队、SRE 工程师及 QA 测试人员使用的灰度机房环境。目标是提升整体演练效率，实现从被动响应到主动预防的转变。

为此环境设定了以下构建原则：

- **隔离性：**确保演练环境与线上生产环境之间完全隔离，避免交叉影响，从而有效控制"爆炸半径"。
- **真实性 (架构对等)：**演练环境的架构、依赖链路及其上部署的服务都需要与生产环境保持一致，实现端到端部署，以保证演练结果的准确性和有效性。
- **低成本：**在满足隔离性和真实性的基础上，注重成本效益。例如，对于存储类服务，倾向于与线上环境共享集群，以优化资源利用率。

灰度机房流量调度与服务部署架构

灰度机房的设计核心在于其灵活的流量调度能力，能够支持多种演练模式。

- **生产流量分流机制**

1. **全量线上流量与办公网流量入口：** 100% 的线上用户流量以及办公网内部流量首先汇聚于第一层负载均衡器 (LB-1)。

2. **RGW 层面的流量路由：** 经过 LB-1 后，流量进入小红书的边缘网关 (RGW)。在此层级，大约有 1%的固定生产流量会被精准地路由到“RGW-gray”模块，标志着灰度流量的正式启动。

3. **办公网流量直入灰度：** 此外，办公网用户流量也可直接接入“RGW-gray”，这为内部员工在灰度环境进行深度测试提供了便捷通道。

- **服务部署与灾备能力**
    - **自建机房：** 包含核心的 RGW 组件以及承载主业务流量的“单活服务集”和“多活服务集”。其中，“南北向流量调度”组件负责主流量在不同服务集间的分配。
    - **灰度 2.0（灰度机房）：** 作为容灾演练的核心区域，它承接来自“RGW-gray”的灰度流量。
        - “多活服务集-gray”是灰度机房中与生产环境多活服务对应的副本，专门用于处理灰度流量及执行各类演练。
- **南北向流量调度：** 这一关键组件实现自建机房与灰度 2.0 之间的互联互通。它能将来自自建机房的流量按需调度至本机的“多活服务集”，或导向至“灰度 2.0”机房的“多活服务集-gray”，实现精细化的流量控制和演练隔离。

灰度机房的演练模式

灰度机房提供了多种灵活的演练场景：

- **生产流量切割演练：** 可以将线上生产环境 1%的真实流量切换至灰度机房，并在受限的"爆炸半径"内（例如针对单实例）进行故障注入和验证，有效控制风险。
- **办公网流量模拟演练：** 在需要更彻底隔离的场景下，可以完全切断生产流量，转而引入办公网 WiFi 流量。通过员工的日常使用行为，模拟用户场景进行故障注入，并观察其功能表现是否符合预期。这种方式能够验证在真实用户交互下的系统韧性。

X 机房断网演练实战案例

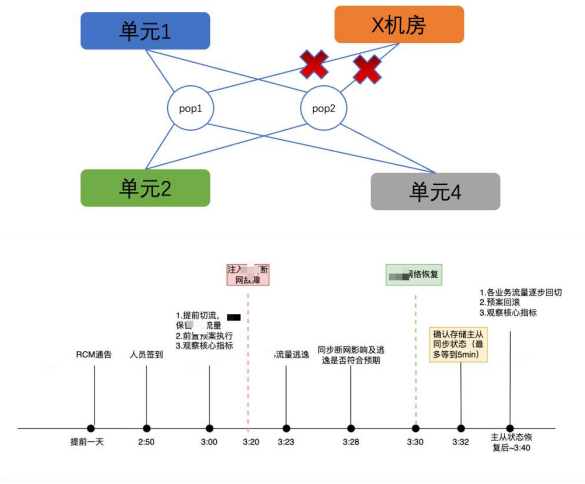
小红书容灾演练实践

X机房断网演练实战

断网方案是关键

剧本是灵魂

逃逸预案是武器



接下来，将分享小红书近期一次真实的容灾演练案例，主题聚焦在"X 机房断网实战"。

本次演练情景模拟了机房专线全断、形成"孤岛"的极端情况，旨在验证在这一特殊故障模式下，核心业务能否持续稳定运行，以及团队的应急响应能力。

要成功组织此类容灾演练，以下三点至关重要：

- **断网方案是关键**：需要与专线团队紧密合作，共同设计并明确具体的断网模拟方案。方案需要足够真实，例如，通过专业的指令下发或直接操作端口等方式模拟专线中断，避免与真实故障场景出现偏差，确保模拟的有效性。这项工作通常由"蓝军"团队负责落实。
- **剧本是关键**：对于像公司级容灾演练这样涉及众多部门和角色的活动而言，一份详尽且可执行的剧本是确保演练顺利进行的基础。它需要清晰定义所有参与方（包括职能部门、业务团队等）的角色职责、具体操作步骤以及每个环节的时间节点（精确到几点几分），从而将所有参与者高效地串联起来。
- **逃逸预案是基础**：在极端故障下，除了常规的恢复策略，需要准备一套能够"断臂求生"的备用方案。它确保在常规手段无法奏效时，系统依然能够以降级或最小影响的方式维持服务可用性。

现在，结合这张幻灯片上的示意图和时间轴，回顾这次"X 机房断网演练"的实战过程。

示意图解读：网络拓扑与故障影响

这张示意图展示了我们简化的网络架构。图中，“单元 1”、“单元 2”和“单元 4”通过两个 POP 节点（pop1 和 pop2）与“X 机房”实现互联互通。当“X 机房”发生断网故障时（如图中红色“X”所示），所有与 X 机房相关的网络连接都将中断，直接影响依赖 X 机房的服务与通信。

容灾可观测大盘

小红书容灾演练实践

X机房断网演练-容灾可观测

全局监控大盘

通过一套可观测的系统，感知业务和系统的健康状况。核心在于将复杂的信息分层可视化，从而实现对容灾全过程的掌控。

- **业务线与场景核心指标洞察：**
    - 在容灾演练中，首先要聚焦于小红书各个业务线及其细分场景的核心指标。通过监控大盘，能够看到"直播"、"推荐"、"搜索"、"社区"、"广告"、"电商"以及"音视频架构"等核心业务模块的关键数据表现，例如 CTR、人均 Feed 总时长、用户登录成功率等。这些指标是业务健康状况的晴雨表，其正常与否直接反映了用户体验和业务连续性。
- **北极星 SLO 实时监控与预警：**
    - 为不同场景设定了"北极星 SLO（Service Level Objective）"，即衡量服务可用性和性能的关键指标。监控大盘能够实时展示这些 SLO 是否临近或已经突破预设阈值。一旦 SLO 出现破线趋势或实际破线，系统会立即给出警示，确保能第一时间发现潜在或已发生的可用性问题。
- **故障感知与推送机制：**

- 无论是处于演练阶段的模拟故障，还是真实场景中的偶发故障，一旦某个场景发生异常，故障信息都会被实时推送到这个统一的监控大盘上。这使得可以集中化地观测所有故障事件，快速定位问题发生的具体场景，缩短故障发现时间。

- **多维度观测与决策支撑：**
  - 通过这个大盘，能够以三层递进的视角来评估系统状态：
    - **核心指标是否异常？** 这是最直接的业务表征。
    - **SLO 是否破线？** 这反映了服务质量是否受到影响。
    - **是否发生实际故障？** 这是最严重的事件。
  - 这三层信息通过一张大盘集中呈现，使得 SRE 团队和业务方能够快速了解系统在容灾场景下的表现。
  - 此外，除了面向业务方的视图，还提供了面向中台的视角，确保技术中台团队也能从另一个维度进行观测与协作。
- **业务线逃逸预案的指定与观测：**
  - 在容灾体系中，提前制定好各业务线的"逃逸预案"至关重要，即在极端情况下如何进行服务降级、流量迁移或数据隔离等操作。这张大盘也承载着对这些预案指定与下发状态的观测能力，确保预案能够及时生效，从而有效止损。

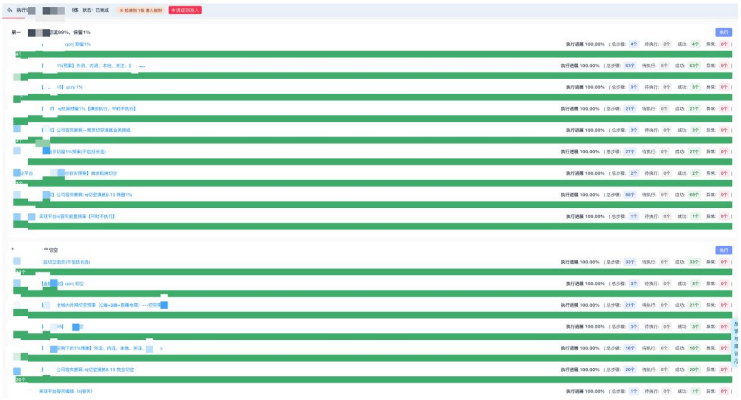
通过这种"容灾可观测"的实践，小红书构建了一个实时、分层的监控体系，确保在 X 机房断网这样的极端场景下，团队能够迅速发现问题、评估影响、及时采取行动，从而保障业务的连续性和用户的体验。

全局预案大盘与执行监控

小红书容灾演练实践

X机房断网演练-容灾中控

全局预案大盘



通过"容灾中控"系统统一管理与执行。画面核心是一个"全局预案大盘"，展示了演练各环节的执行状态，多数项目已成功完成，进度显示为 100.00%。这帮助了解各业务线在流量切换或业务降级过程中，命令是否真实生效并执行成功，也为复杂预案的透明化执行提供了支撑。

- **业务降级与流量切换的精细化验证**
    - 在机房断网这样的灾难场景下，核心要解决的问题在于，各业务线在执行流量切换（切流），或是业务降级（例如将数据库主库切换到备用机房）时，相关指令是否能精准下发，并确保操作的真实性与成功率。
    - **不同业务线的差异化应对：**
        - **功能型业务（如社区、电商）：** 这类业务相对简洁，通常只需通过流量切换和数据库主库切换（DB 切主）即可实现从故障机房的优雅逃逸。
        - **复杂型业务（如搜推广）：** 这类业务不仅涉及效果（如广告推荐），其日常部署也并非简单地按照“三活即 300%容量”来配置。因此，在进行故障逃逸时，必须结合容量降级策略，确保当某个机房切空时，剩余的两个机房能够有效承载相应的业务容量，避免因容量不足导致服务中断。

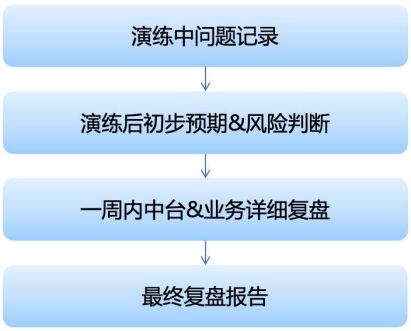
- **应对复杂性：**不同业务线预案的复杂性和个性化差异，普遍包含十几到二十个原子级预案。

- **“全局预案大盘”的核心价值**
  - 为了有效管理与监控这些复杂的原子预案，构建了"全局预案大盘"。它作为容灾中控系统的可视化核心，提供了一个统一的视图，帮助实时观测每个业务线预案的整体执行进展。
  - 通过大盘，可以第一时间发现并识别演练中出现的问题：
    - 包括业务降级指令的执行失败。
    - 流量切换操作的异常。
    - 数据库主库切换过程中发生的故障。
  - 一旦发现问题，系统会自动记录并及时通知相关同学，以便迅速协作，定位问题根源，并进行处置，确保演练流程的顺畅与有效性。

X 机房断网演练复盘流程

小红书容灾演练实践

X机房断网演练-复盘



容灾演练，尤其像模拟机房断网这类演练，具有较高的价值。因此，我们每次都会组织详细的复盘工作，将经验转化为系统韧性提升。

演练复盘流程如下：

- **演练中问题记录**
  - 在演练进行过程中，我们会实时记录所有发现的问题、异常情况以及观察到的行为。

- 这些记录不仅限于技术层面的系统问题或平台故障，还包含流程中存在的问题，例如操作步骤不清晰、协同效率低下等。
- 同时，我们也会关注人的因素，识别是否存在组织架构上的缺失，例如特定角色（如应急响应中的 GA 专家）的空缺，这都需要在后续补充完善。

- **演练后初步预期与风险判断**
  - 演练结束后，我们会要求各方向的团队成员，通常在 1 到 2 小时内，进行初步的评估与判断。
  - 目的是确认本次演练结果是否符合预期，以及是否存在可能导致业务中断的容灾风险点，这些风险点需要标记并优先处理。
- **一周内中台与业务详细复盘**
  - 基于初步的判断结果，我们会在一周内召集中台团队和各业务线的技术负责人，进行详细的复盘会议。
  - 此次会议旨在分析问题根源、影响范围，评估已采取应对措施的有效性，并规划后续的改进措施（Action Items），确保问题得以闭环解决。
- **最终复盘报告**
  - 完成所有分析和讨论后，我们会形成一份针对本次演练的最终复盘报告。
  - 该报告将指出本次演练是否成功通过，并阐述各业务线的表现是否达到预期目标，以及所有发现的问题、根因分析和改进计划。
  - 这份报告最终将递交给公司的管理层，确保高层了解风险状况和改进方向。

此次分享旨在总结小红书在过去容灾工作中的实践经验。后续，我们将展望未来的发展方向，持续提升系统的容灾能力。

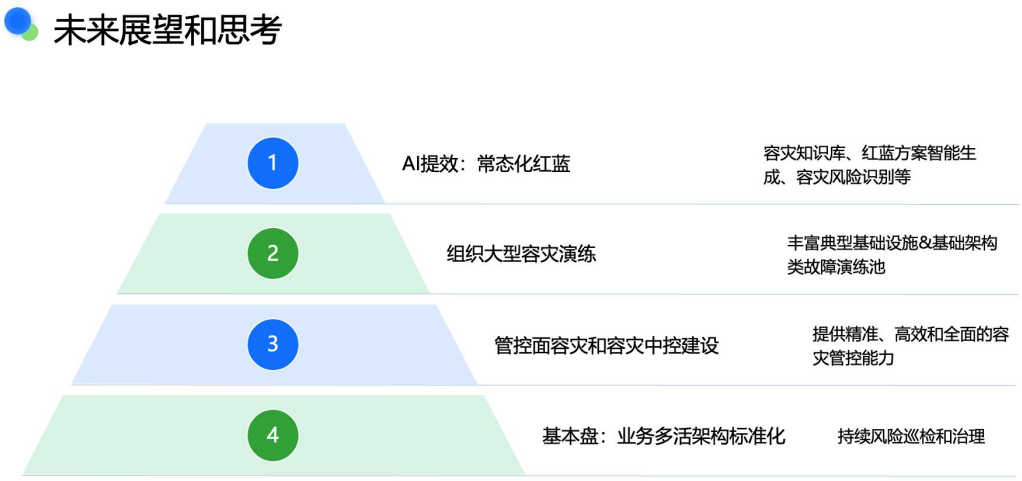
（三）总结及展望

总结回顾

04

未来展望和思考

未来展望与持续优化方向



旨在与大家共同探讨 SRE 在风险管理与容灾能力建设方面的未来展望和深度思考，以期通过技术创新、组织演练、管控平台建设以及架构标准化，持续提升我们系统的整体韧性。

- **基本盘：业务多活架构标准化**

- 这是容灾体系的基石。多活架构是容灾逃逸的基础。我们将持续推进业务多活架构的标准化，并进行风险巡检与治理，确保其有效性和稳定性。

- **管控面容灾和容灾中控建设**
    - 在多活架构之上，我们正建设统一的容灾管控平台与中枢系统。它将提供管控能力，确保在真实故障场景下，指令能够迅速下发，整体故障切换与恢复流程能够顺利执行，这是保障业务连续性的关键环节。
- **组织大型容灾演练**
    - 基础设施类的大规模故障并非频繁发生，这可能导致团队对容灾风险的警觉性下降。因此，我们将定期组织大型容灾演练，丰富典型基础设施与基础架构故障演练池。
    - 定期的演练能帮助团队保持对风险的意识，并模拟真实故障场景下的应急响应，避免在真正灾难来临时，因紧张情绪而导致操作失误。
    - 演练的设计将考虑演练地点、具体课题、蓝军攻击策略等细节。
- **AI 提效：常态化红蓝**
    - 我们正探索如何利用人工智能提升容灾治理的效率与智能化水平。
    - **具体应用：**
        - 利用 AI 构建容灾知识库，沉淀历史故障经验与处置流程，促进容灾知识的传播与共享。
        - 智能生成红蓝演练方案，优化演练设计与执行。
        - 持续利用 AI 进行容灾风险识别，借助大型模型的经验与业界实践，提供不同视角的风险评估，辅助发现潜在容灾风险。
        - 目标是让业务团队能够在一个爆炸半径可控的灰度环境中，自主进行常态化的红蓝攻防演练，提升系统的自愈能力与韧性。

通过这四个层面的规划与实践，我们致力于构建弹性和自适应能力的容灾体系，为业务的持续稳定运行提供保障。

5.4.7 蚂蚁大促场景下全链路压测体系构建与保障实践

SRE Elite 精选原因

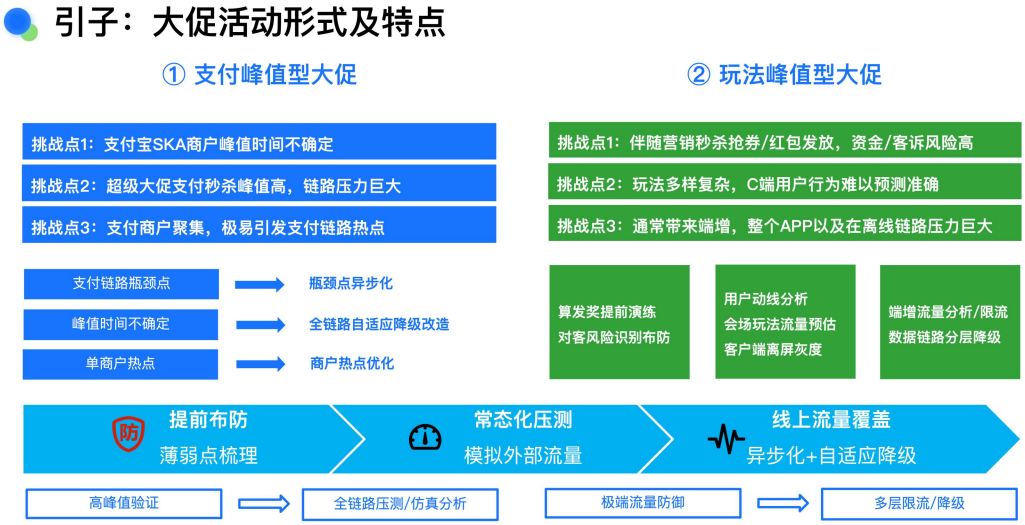
蚂蚁集团承载着支付宝、花呗、借呗等国民级金融产品，服务全球超过 10 亿用户。在双十一、五福等大促场景下，系统面临极为严苛的技术挑战：支付峰值瞬时爆发，需在毫秒级内完成亿级交易的事务一致性保障；热点商户流量高度聚集，极易在底层资源形成热点效应导致局部过载；营销玩法复杂多变，用户行为难以精确建模，资金安全与客诉风险压力巨大。

面对这些挑战，蚂蚁 SRE 团队构建了一套以全链路压测为核心的大促保障体系。本案例基于真实的双十一、五福等世界级流量洪峰场景，分享了从活动分级评估、容量风险识别、全链路压测验证，到压测熔断与风险防控的完整实践经验。其中，基于 Trace 的流量染色打标技术、影子表数据隔离方案、秒级压测熔断机制等核心能力，对其他需要在生产环境进行高保真压测的金融、电商业务同样具有重要的借鉴价值。

（一）背景及设计原则

蚂蚁大型活动保障架构概览

大促活动形式及特点



场景分类

- 在互联网行业的稳定性保障实践中，支撑大型活动的系统挑战往往集中体现在两大典型场景：
    - **支付峰值型大促场景**：以极致的并发处理能力应对瞬时流量洪峰，确保资金流水在海量请求下依然保持事务的强一致性。
    - **玩法峰值型大促**：侧重于大规模、高实时性的数据计算与状态更新，要求架构具备极强的横向扩展能力与响应时延控制。

支付峰值型大促：攻坚超高并发与确定性

在蚂蚁的业务语境下，支付是核心基石，而支付峰值型大促的难点在于对“极值”与“热点”的控制。

- 挑战点：
    - 重点 SKA 商户的峰值爆发时间具有不确定性，系统必须时刻准备应对突发流量；
    - 秒杀场景下的极致瞬时峰值，对整个支付链路的吞吐能力提出了严苛要求；
    - 众多商户请求在短时间内高度聚集，极易在底层资源上形成热点效应，导致局部过载。
- SRE 应对策略：
    - 瓶颈点异步化：针对支付链路中的同步阻塞环节进行异步化改造，提升系统整体承载力；
    - 自适应降级：通过全链路自适应降级机制，动态应对峰值时间的不确定性；
    - 热点优化：实施商户级热点隔离与调度优化，从底层规避热点效应对全局稳定性的影响。

玩法峰值型大促：应对复杂逻辑与用户行为

相比支付型，以“五福”或互动氛围图为代表的玩法型大促，其复杂性源于营销逻辑的多样化与 C 端行为的不可预测性。

- 挑战点：
    - 复杂的发券、红包与抢购逻辑涉及频繁扣减，资金安全防线与客诉风险压力巨大；

- 营销玩法多变，用户动线难以精确建模，APP 容器及离线数据链路常因用户端增量而面临压力。

- SRE 应对策略：
  - 风险布防：针对算发奖逻辑提前演练，建立针对客诉风险的自动化识别与布防体系；
  - 流量治理：深入分析用户动线与会场流量，利用离屏灰度技术平滑用户冲击；
  - 分层兜底：对端增流量实施精准限流，并对非核心数据链路进行分层降级，确保核心玩法不中断。

SRE 核心保障体系：从被动防御到闭环治理

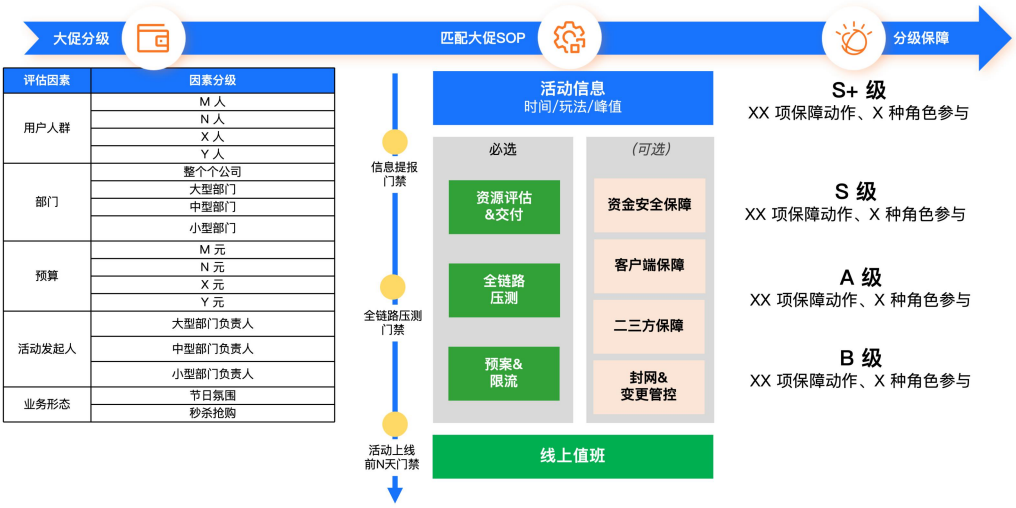
针对上述两类挑战，我们通过“布防–压测–覆盖”三位一体的闭环流程，构建起大促稳定性的底层逻辑。

- 提前布防与薄弱点梳理：
  - 图标中的红色“防”字盾牌象征着我们在大促早期介入，通过架构审查深入挖掘系统内部隐患，将风险消灭在萌芽状态。
- 常态化压测与仿真验收：
  - 利用仪表盘式的监控验证体系，模拟真实的外部流量压力，通过容量验收确定系统的真实水位与性能边界。
- 线上流量覆盖与动态韧性：
  - 结合心电波形般的实时监控，实施异步化改造与自适应降级，在高并发环境中利用多层限流与全链路压测分析，确保极端流量下业务依然具备极强的韧性。

正是通过这套“前置防护–模拟验收–动态响应”的完整链条，我们才能在亿级支付波动与复杂营销玩法交织的环境下，实现大促稳定性的全链路高可用保障。

活动分级评估与标准化保障 SOP

蚂蚁大促分级及活动保障 SOP



评估：量化驱动的活动分级 为了在大促这种高并发、高压力的场景下实现精准的资源分配与风险管控，我们构建了一套基于权重的多维度评估矩阵，将活动从触发初期就纳入标准化的管理轨道：

- **多维评估因子**：系统性地考量用户覆盖规模、发起部门的行政层级、营销预算投入、活动发起人的职权范围以及业务形态（如节日氛围渲染或高爆发的秒杀抢购）等核心指标；
- **加权计分机制**：通过预设的算法对各项因素进行加权求和，以此确定活动分值，并将活动精确划分为 B 级、A 级、S 级乃至 S+ 级（如双十一、五福等国民级活动）；
- **定级意义**：这种分级方式不仅通过科学的量化手段为后续资源投入提供了决策依据，更在组织内部达成了一致的优先级共识。

执行：标准化操作流程 (SOP) 与门禁管理

一旦活动的等级被锁定，系统将自动匹配对应大促 SOP，通过“必选+可选”的模块化动作，确保保障工作既能守住稳定性底线，又能兼顾交付效率：

- **稳定性基石（必选动作）**：
  - **资源适配**：覆盖从初期的容量评估到最终的资源交付全链路，确保底层基础设施与业务需求精准匹配；

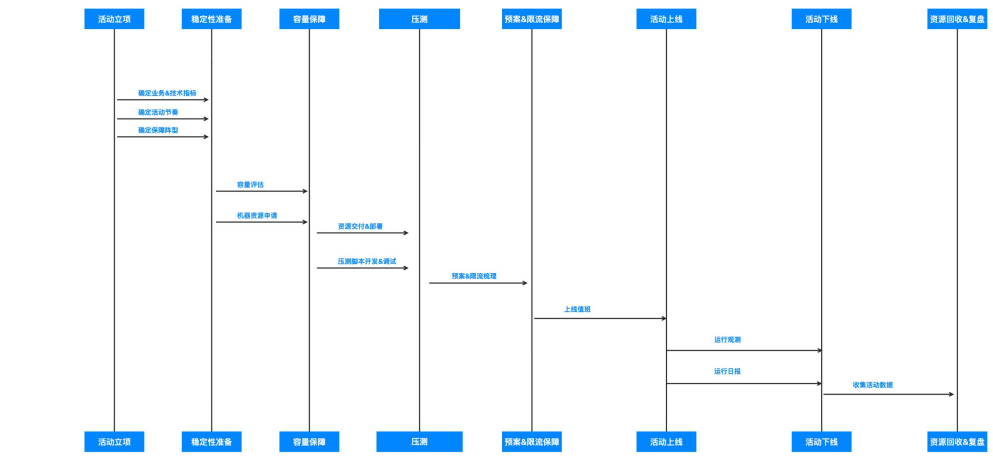
- **压力验证**：执行全链路压测以摸清系统瓶颈，在此基础上配置精细化的预案与限流策略，确保系统在极端流量下不可被压垮；
- **底座支撑**：强制要求线上值班机制，通过跨角色的实时响应保障业务连续性。

- **弹性扩展（可选动作）**：
  - 根据活动复杂度，按需引入资金安全保障、客户端特化保障及二三方依赖专项梳理；
  - 在特定高等级时段执行严苛的封网与变更管控，通过冻结全站生产变更来排除人为干扰因素。
- **强管控门禁**：在时间轴上设立信息提报、全链路压测、上线前 N 天等关键管控门禁，确保每一个关键节点都满足准入条件，杜绝任何带病上线的可能。

目标：科学化的全链路保障框架

通过这套“前置分级、匹配 SOP、分级保障”的标准化体系，蚂蚁集团将原本碎片化的运维动作整合成为了一个有机整体。我们不再盲目投入资源，而是利用这套科学的框架，将保障动作与活动风险进行深度绑定。

蚂蚁大促保障整体流程



大促保障的整体流程

蚂蚁集团的大促保障是一场典型的"技术战争"，为了确保亿级流量冲击下的系统稳如磐石，我们构建了一套横跨多个职能部门、高度协同的标准作业程序（SOP）。这张架构图以时序为轴，深度解构了从活动立项到资源回收的全生命周期保障链路。

筹备阶段：活动立项与稳定性准备

在项目启动之初，首要任务是明确业务与技术双重指标，梳理整体活动节奏。由于大促涉及作战室制度，我们必须预先确定保障阵型，将 SRE、开发、质量、客权、财务等职能部门整合进统一的指挥体系，确保信息的高度对称与决策的快速传导。

资源配置：容量评估与交付

基于业务预估进行严谨的容量评估，随后发起机器资源申请。这一阶段的重点在于资源的按时交付与环境部署，为后续的实战演练提供坚实的基础设施底座。

容量验证：全链路压测

这是保障流程中最核心的环节之一。通过开发与调试压测脚本，我们在真实环境中模拟极致流量高峰，通过不断的"摸高"测试，验证系统承载上限并发现架构瓶颈。

防御加固：预案与限流保障

基于压测获取的容量边界，我们进一步梳理限流策略与应急预案。这要求我们将复杂的系统依赖转化为可量化的控制手段，确保在极端场景下能通过熔断、降级等手段保住核心业务。

实战运行：活动上线与观测

在活动期间，SRE 团队进入高强度值班状态，利用实时运行观测系统与运行日报机制，对业务指标进行秒级监控，确保任何微小的异常都能被立即捕捉。

复盘沉淀：活动下线与价值闭环

在大促落下帷幕后，重点转向资源回收与数据收集。通过对活动期间技术表现的深度复盘，我们将实战经验沉淀为可复用的技术资产。

尽管整套流程涵盖了从前置防护到后期复盘的多个关键节点，但由于大促保障的复杂性，容量评估与全链路压测始终是支撑这套 SOP 的技术支柱。接下来，我们将重点聚焦这两大核心能力，探讨如何在复杂的分布式环境中实现精准的容量确定性。

容量风险识别与管理

什么是容量风险?

在SRE（Site Reliability Engineering，站点可靠性工程）领域中，“容量风险”（Capacity Risk）指的是：系统当前或未来可能因资源容量不足，无法满足用户需求或业务增长，从而导致性能下降、服务不可用或用户体验恶化的潜在风险。

什么是容量风险?

在 SRE 的视角下，容量风险并不只是简单的“资源不够”，它是一个动态的供需失衡过程。我们将容量风险定义为：系统在当前运行状态或未来业务预期的背景下，因基础设施资源储备不足，无法承载用户请求或业务流量的爆发式增长，进而引发系统性能下降、服务不可用甚至用户体验彻底恶化的潜在威胁。

根源：基础设施的瓶颈

我们可以将系统资源想象成一个装满液体的水桶，当业务压力激增，底层资源的瓶颈就像桶身的漏洞一样，让系统的稳定性不断流失：

- **计算资源与连接枯竭**：CPU 过载会导致处理响应变慢，而数据库连接数耗尽则直接封死新请求的入口，让系统陷入停滞
- **存储与分发瓶颈**：磁盘空间不足往往会触发服务强制中断，而 CDN 带宽达到峰值则会拖慢内容交付速度，使全球业务受阻

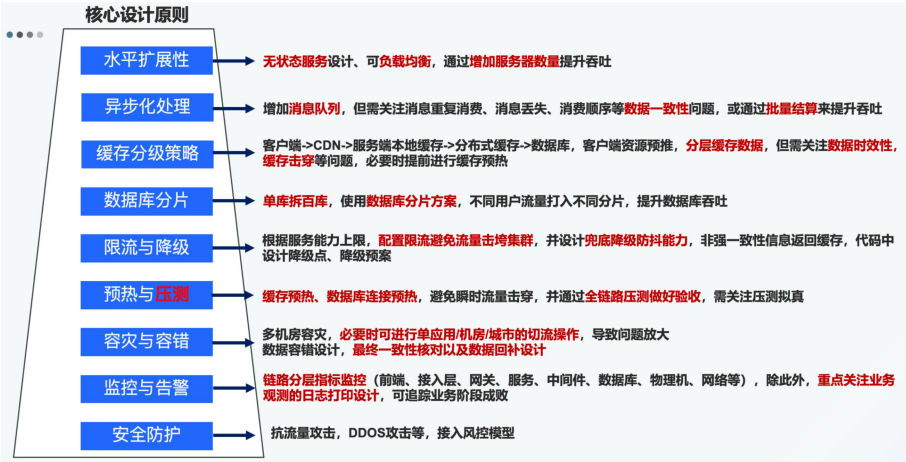
影响：从资源到业务的传导链

- **性能衰减**：系统处理效率大幅降低，用户操作反馈变得极其迟缓
- **服务不可用**：这是最严重的状态，就像我们常看到的"微博热搜导致后端崩了"或"服务 502"，用户被彻底阻断
- **体验持续恶化**：即使服务没有完全宕机，频繁的报错与加载失败也会摧毁用户对产品的信任

从资源瓶颈到业务坍塌，这种由内而外的因果关系构成了容量风险的全貌。我们之所以需要建立严密的容量保障体系，正是为了在"水桶漏水"之前，通过精准的度量与预判，将这些可能导致系统崩溃的隐患消除在萌芽状态。

容量风险的本质：如何承接高并发流量？

容量风险的本质：如何承接高并发流量？



九大高并发系统核心设计原则

水平扩展性 无状态服务设计、可负载均衡，通过增加服务器数量提升吞吐

异步化处理 增加消息队列，但需关注消息重复消费、消息丢失、消费顺序等数据一致性问题，或通过批量结算来提升吞吐

缓存分级策略 客户端 → CDN → 服务端本地缓存 → 分布式缓存 → 数据库，客户端资源预推，分层缓存数据，但需关注数据时效性、缓存击穿等问题，必要时提前进行缓存预热

数据库分片 单库拆百库，使用数据库分片方案，不同用户流量打入不同分片，提升数据库吞吐

限流与降级 根据服务能力上限，配置限流避免流量击垮集群，并设计兜底降级防抖能力，非强一致性信息返回缓存，代码中设计降级点、降级预案

预热与压测 缓存预热、数据库连接预热，避免瞬时流量击穿，并通过全链路压测做好验收，需关注压测拟真

容灾与容错 多机房容灾，必要时可进行单应用 / 机房 / 城市的切流操作，避免问题放大，数据容错设计，最终一致性校对以及数据回补设计

监控与告警 链路分层指标监控（前端、接入层、网关、服务、中间件、数据库、物理机、网络等），除此以外，重点关注业务观测的日志打印设计，可追踪业务阶段成败

安全防护 抗流量攻击、DDoS 攻击等，接入风控模型

（二）体系设计及关键流程体系设计

全链路压测体系概述

02

蚂蚁全链路压测体系

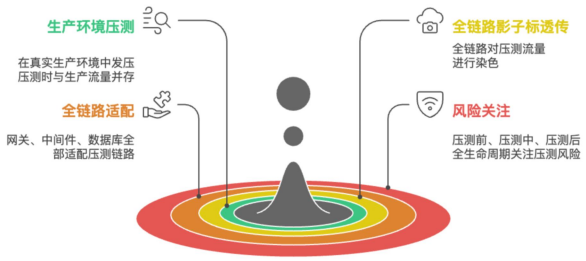
全链路压测的定义与技术特性

全链路压测定义

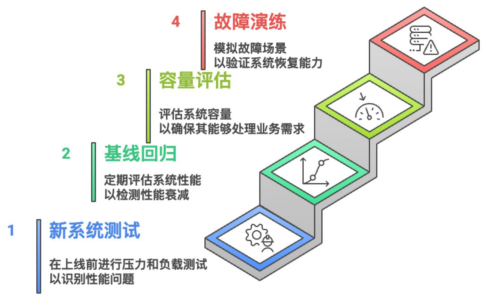
全链路压测是模拟真实业务场景流量，对完整系统链路（前端→后端→数据库等）进行高并发压力测试，发现性能瓶颈与单点故障，确保系统在峰值（如大促）下稳定运行。通过流量染色、影子表等技术实现零业务影响

—— From Ling-1T

蚂蚁全链路压测特点



典型使用场景



核心定义：全链路压测的本质

全链路压测通过模拟真实的业务高峰流量，对从网关接入到后端应用，再到中间件与数据库的完整系统链路进行高并发测试。

- **核心逻辑**：其本质是在复杂的分布式架构中，通过高仿真的压力负载，深度挖掘系统的性能瓶颈与脆弱的单点故障
- **技术底座**：借助"流量染色"与"影子表"等核心隔离技术，确保压测流量在生产环境中被精准识别与路由，实现对线上真实业务的零影响

蚂蚁全链路压测的四大技术特性

- **真实环境发压**：直接在生产环境执行压测，让流量在真实的硬件、网络与配置环境下奔跑，彻底告别实验室环境与生产环境由于差异导致的"测不准"问题
- **全链路深度适配**：从最前端的流量接入层到微服务集群，再到复杂的中间件节点与底层数据库，每一环都必须具备识别压测标的能力，确保压力负载能透传至系统最深处
- **影子标透传技术**：通过统一的染色协议，使压测标识在跨语言、跨协议的服务调用链中保持不丢失、不篡改，为后续的数据隔离提供最可靠的逻辑依据
- **全生命周期风险防控**：作为技术风险部的核心关注点，压测并非简单的"暴力测试"，而是必须建立起涵盖压测前准入、压测中实时熔断以及压测后数据清理的全生命周期准则，将对生产系统的安全承诺贯穿始终

典型应用场景

- **新系统上线测试**：在全新业务上线前进行极限负载测试，提前识别并消除潜在的性能隐患，确保服务"出生"即具备抗压能力
- **基线回归评估**：在版本迭代过程中，定期通过标准化的压测流程评估系统性能，及时发现代码变更带来的性能衰延，防止隐患随应用发布进入生产环境
- **核心容量规划**：以全链路压测数据为准绳，精确评估现有集群的承载量级，为弹性扩容、资源编排以及大促期间的容量决策提供真实有效的依据
- **故障演练与韧性验证**：在压测的高负载状态注入故障场景，验证系统在极端压力下的自愈能力、降级策略以及监控告警的灵敏度，从而全方位提升系统的健壮度

综上所述，全链路压测不仅是一套测试工具，更是一套严密的风险管理体系。它让我们可以从容应对亿级设备与千万用户带来的性能挑战，确保系统在大促等关键时刻能够稳如泰山。

全链路压测平台架构全景

蚂蚁全链路压测平台架构介绍

压测平台的四层架构体系

以上为蚂蚁集团全链路压测平台的架构全景，它不仅是支撑双十一等大促容量保障的基石，也是我们后续探讨所有技术细节的逻辑起点。整个平台经过多年的沉淀与演进，将复杂的压测全生命周期抽象为了四个核心层级：

压测管理平台：面向用户的"指挥中心"

作为最上层的操作门户，管理平台承载了从资源筹备到结果回溯的完整业务流。尽管系统经历了长期的迭代，但核心功能始终聚焦于高效协同与确定性交付：

- **项目与场景管理**：支持压测脚本的编写与生命周期管理，并能够针对不同的业务闭环快速构建压测场景，精细化管理压测所需的影子数据与监控指标
- **资源与执行管控**：涵盖了流量配置、压力机状态监控及其动态扩缩容，确保在压测任务启动前，所有施压资源已就绪且处于受控状态

压测管控与压力机模块：高效分发与稳定施压的底层支撑

这是压测流量的"心脏"，负责将管理层的控制指令转化为实地的流量冲击：

- **智能调度体系**：管控模块通过压力机注册与心跳检查机制，实时掌握执行节点的健康状况，并将压测任务精准分发至各压力机，实现监控指标的实时聚合汇总
- **执行单元保障**：压力机内部实现了脚本部署、执行与数据上报的自动化闭环，对于 SRE 而言，保障施压端自身的稳定性是确保压测数据真实可靠的前提

压测风险模块：生产环境的"安全底座"

在蚂蚁的稳定性体系中，风险防控是极具特色的核心所在。在大规模全链路压测中，我们必须构建起多重防御线，以防压测流量对生产环境造成不可逆的干扰：

- **准入与熔断机制**：通过严格的压测窗口管理与白名单审核确保"合法准入"；利用压测前置校验排除环境隐患；而自动化熔断策略则是在系统负荷超出预设阈值时，能够瞬间切断流量，保障核心业务的绝对安全

这套架构涵盖了从资源管理、任务调度、执行反馈到安全防护的全生命周期，体现了大规模分布式系统在容量保障上的深度思考。在明确了整体架构后，我们接下来将深入探讨其中最关键的技术细节，首先便是支撑全链路流量隔离的基础——全链路打标技术。

蚂蚁全链路染色打标技术

蚂蚁全链路压测平台核心技术介绍_全链路染色打标

基于 Trace 的流量标识透传机制

关于蚂蚁全链路压测平台，其实现生产环境下安全压测的核心基石在于**全链路染色打标**技术。这张幻灯片清晰地对比了生产流量与压测流量在系统中的差异化流转路径。

核心机制：基于 Trace 的全链路上下文透传 为了确保压测流量在复杂的分布式架构中不“迷路”，影响生产，我们利用全链路可观测性技术的 SOFATracer，将“压测标”嵌入到 Trace 上下文中；这种设计能够保证标识跟随请求横跨不同的服务节点与技术栈，实现无损的逻辑传递。

备注：SOFATracer 是一个用于分布式系统调用跟踪的 SDK 组件，通过统一的 traceId 将调用链路中的各种网络调用情况以日志的方式记录下来，以达到透视化网络调用的目的。这些日志可用于故障的快速发现，服务治理等。该 SOFATracer 类似于社区中 OpenTelemetry 的 SDK，在蚂蚁内部，所有的分布式应用都会嵌入这个 SDK。作为链路跟踪的基本的数据来源。

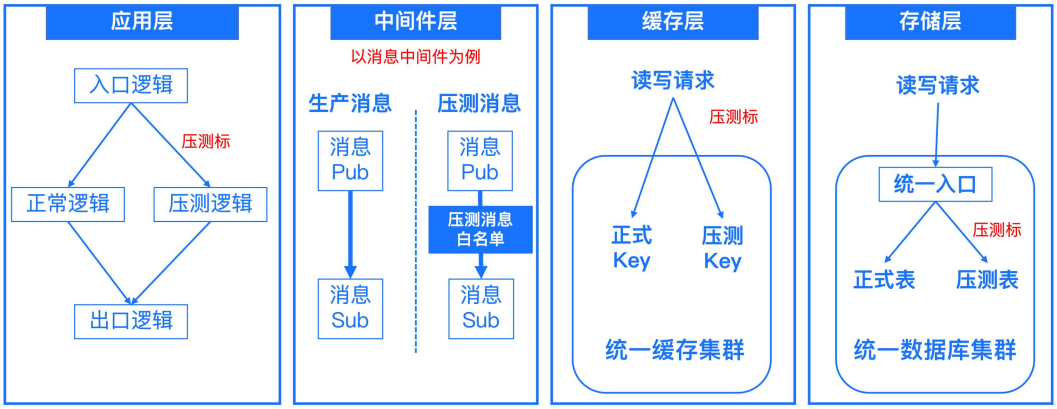
各层级的精细化识别与隔离处理：

- **流量接入层：职责在于“标”的透传**
    - 当压力机模拟真实的业务请求向系统发压时，接入层首先识别入参中的压测标识，并负责将该状态注入到当前请求的全局上下文中，开启后续的链路染色过程。
- **应用服务层：负责业务逻辑的路由判断**
    - 应用系统通过中间件提供的底层工具类实时拦截流量，精准识别当前请求是否携带压测标，从而在业务代码执行过程中，决定是否进入特定的压测逻辑分支或 Mock 处理。
- **中间件与框架层：执行流量的“接纳”与“控制”**
    - 无论是消息中间件还是通信框架，在感知到压测标后，会根据预设的策略决定是对该请求进行限流降级、正常放行，还是路由到特定的测试信道，从而保护生产环境不受异常压测行为的影响。
- **数据存储层：实现物理与逻辑的双重隔离**
    - **缓存层**：为了避免压测数据污染生产热点，系统会自动将压测流量的读写操作分流至特定的压测 Key 空间，确保生产缓存集群的纯净度。
    - **数据库层**：这是压测隔离的最后一道防线，由数据库中间件识别压测标后，将所有增删改查动作重定向至预先准备的“影子表”中。

正如大家在图中看到的，这种从入口到数据库的全链路打标机制，是实现体系化隔离的前提。只有把“标”打稳了、透传透了，我们才能在不中断线上业务的前提下，放心地在生产环境进行高保真的全链路压测演练。

压测隔离技术

蚂蚁全链路压测平台核心技术介绍_压测隔离



全链路压测面临的最核心挑战在于，如何在真实生产链路上进行高仿真测试的同时，确保压测产生的流量与数据不会对业务生产数据造成污染，且不影响真实用户的交互体验。接下来我们将深入剖析蚂蚁全链路压测平台中，基于"压测标"实现的压测隔离技术方案。

应用层：逻辑流转的动态识别

- **流量识别与判定**：当流量进入系统时，应用层入口逻辑会迅速识别请求头中是否携带"压测标"，这是整个隔离机制的触发源点
- **分支化处理**：系统通过逻辑分流模型，确保带有压测标的请求进入专门的"压测逻辑"分支，而普通用户的请求则继续在"正常逻辑"中运行，从而实现了在不改变应用部署拓扑的前提下，完成业务逻辑的精准隔离

中间件层：异步请求的安全受控

- **消息并行分发**：在处理异步通信时，系统支持生产消息与压测消息并行流转，确保压测产生的消息不对生产消费者产生干扰
- **白名单校验机制**：为了防止压测消息在复杂的分布式环境中发生扩散或误投递，中间件层引入了严谨的"压测消息白名单"机制，只有经过授权并符合受控规则的消息方可进行流转，从根源上规避了潜在的系统污染风险

缓存层：基于键值的逻辑隔离

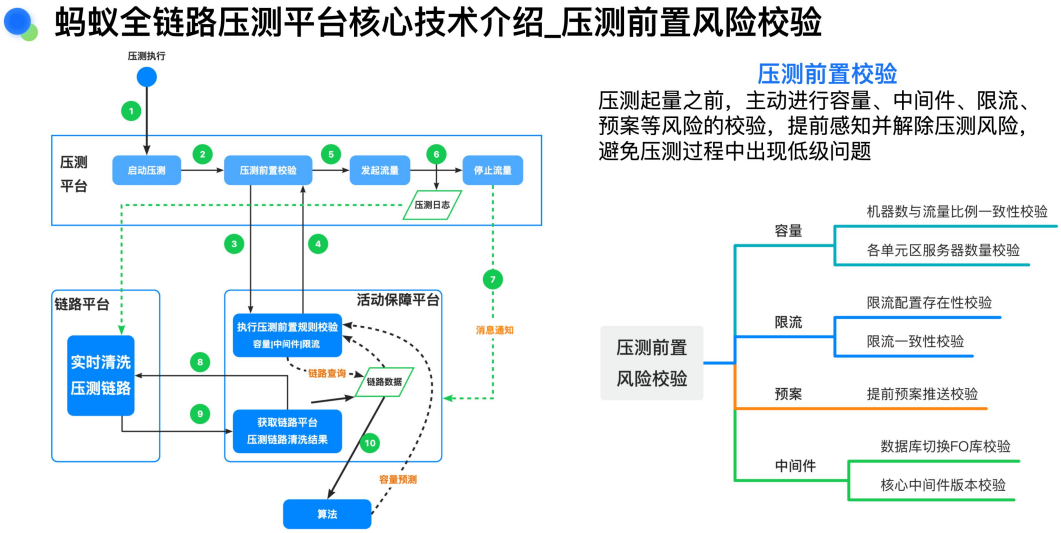
- **Key 级路由模型**：在读写请求进入统一缓存集群时，系统会根据压测标识对缓存键值进行动态转换

- **影子 Key 策略**：所有压测流量均会被映射至"影子 Key"上进行数据操作，而正常的生产逻辑则保持对"正式 Key"的读写，这种方式确保了在同一个物理缓存集群中，压测流量与生产流量能够井水不犯河水

存储层：数据落地的物理与逻辑屏障

- **影子表技术**：存储层作为数据一致性的最后防线，通过统一入口接收读写请求，并依据压测标识对 SQL 执行路径进行即时重定向
- **物理集群内的逻辑解耦**：在统一数据库集群内部，系统将压测数据路由至专门的"压测表"（即影子表），实现了生产数据与压测数据在底层物理介质上的逻辑隔离，彻底避免了脏数据流入生产表的风险

压测前置风险校验机制



核心理念：从"敢压"到"稳压"

为了避免在起量过程中触发低级错误或造成非预期的生产影响，我们构建了压测前置风险校验机制。其核心逻辑在于：在正式发起流量负载之前，系统必须主动对容量、中间件、限流及预案等关键要素进行自动化校验，提前感知并解除潜在风险；如果规则校验未通过，系统将直接执行阻断逻辑，要求运维人员在解除风险或确认跳过之后，方可继续压测。

系统交互：多平台联动

这一套复杂的校验流程并非孤立存在，而是依托于"压测平台、活动保障平台、链路平台"三方的高效协同：

- **压测平台作为执行中枢**：负责压测任务的启停、流量发起及日志记录，并在任务启动瞬间触发前置校验请求
- **活动保障平台作为调度大脑**：它是为大促场景定制的治理中心，向下联通限流、预案、配置等底层平台，向上支撑压测平台的规则执行请求，确保各项校验逻辑能够闭环
- **链路平台作为数据底座**：通过对生产与压测链路的实时分析，利用特定的压测标识进行"链路清洗"，并将清洗后的精准数据提供给算法模块，进行更科学的容量预测

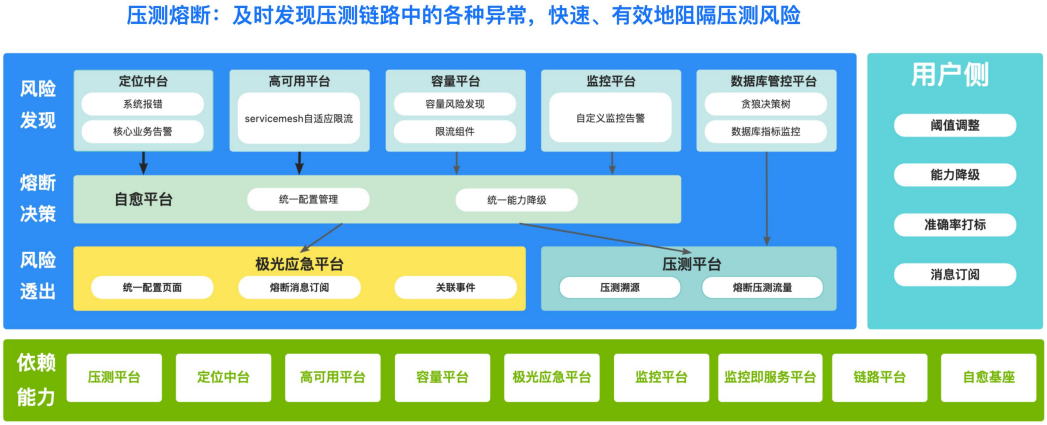
校验维度：四大关键检查

目前，我们将压测前的"健康检查"细化为四个关键维度，确保系统在承受冲击时具备足够的韧性：

- **容量水位校验**：严格比对机器数量与流量比例的一致性，核查各分单元（Unit）的服务器部署数量，防止因扩容未对标导致的局部崩溃
- **限流防控校验**：重点检查限流配置是否存在以及各节点配置是否达成一致，确保大流量进入时，防护篱笆已严丝合缝
- **预案推送校验**：确认大促预案是否已提前下发并生效，防止在故障发生时，应急手段因配置延迟而失效
- **中间件安全性校验**：深度扫描核心中间件的版本差异，并特别针对数据库FO（Failover）库的切换状态进行准入核验

压测熔断机制

蚂蚁全链路压测平台核心技术介绍_压测熔断



核心理念：压测风险的"自动刹车"

压测熔断机制是蚂蚁 SRE 体系在 2023 至 2024 年间重点打磨的核心功能，其本质是为了解决在大规模全链路压测过程中，如何实时感知各类系统异常并确保风险不外溢的关键难题。通过这套机制，我们能够在异常指标超过预设阈值的瞬间，通过自动化闭环迅速关停压测场景，从而在保障生产环境绝对安全的前提下进行容量验证。

能力层：多维度风险发现

由于蚂蚁的基础设施极其复杂，我们整合了多个专业平台的异常感知能力，形成了一套全方位的风险监控网：

- **垂直化的发现平台：**
    - 定位中台：负责实时捕捉系统报错与核心业务告警，并精准识别异常是否由压测流量引起
    - 高可用与容量平台：基于 ServiceMesh 的自适应限流及容量风险组件，监控服务端的负载极限
    - 数据库管控平台：利用"贪狼决策树"对数据库底层指标进行深度巡检
- **API 化能力集成：**各个平台各司其职并对外开放 API，确保当监控指标发生抖动时，异常信号能第一时间汇聚至决策中心

决策层：基于自愈平台的统一指挥

为了避免各平台碎片化处置带来的协作干扰，我们引入了自愈平台作为熔断决策的核心大脑：

- **统一收口与分发**：自愈平台对接底层所有监控平台的信号，执行统一的配置管理与降级决策
- **自动化响应闭环**：一旦决策生效，指令将通过极光应急平台进行消息推送，并由压测平台执行流量截断，极大程度减少了在高压环境下人工操作的迟滞与误判

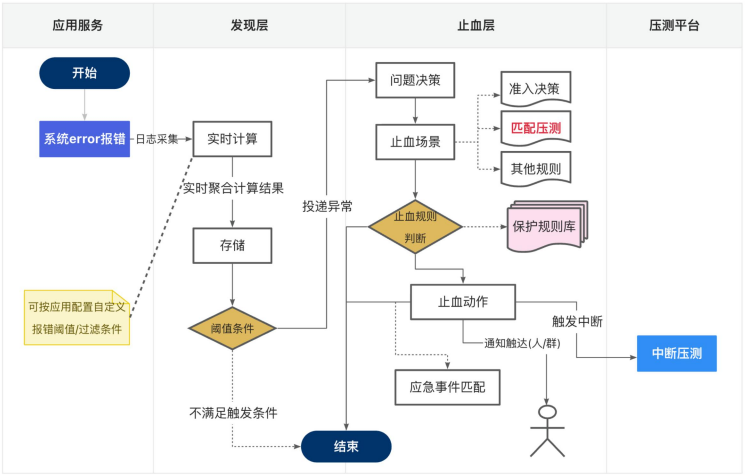
用户侧：精细化差异化配置

考虑到不同业务系统对于异常的容忍度存在显著差异，我们为用户提供了高度灵活的交互能力：

- **阈值自定义**：系统报错的基准因业务而异，如果采用"一刀切"的判定标准，会导致某些高频业务在压测开始即触发误熔断，因此用户可以根据实际业务水位调整熔断阈值
- **手动配置管理**：支持开关配置、能力降级以及准确率打标，确保熔断逻辑既能实现全自动防御，也能兼顾人工干预的灵活性

压测熔断举例

蚂蚁全链路压测平台核心技术介绍_压测熔断_举例



系统error熔断：监测应用的系统报错日志量级，超过特定阈值则会触发熔断

熔断场景：被压测应用的系统报错量级上涨

熔断时效：秒级，20秒之内

熔断要求：应用需要接入标准日志框架，并配置熔断报错阈值

- **发现层**
  - 通过实时日志采集，将应用服务产生的系统级 error 报错进行秒级聚合与实时存储。

- 利用预设的自定义报错阈值与过滤条件，对压测链路上的流量表现进行实时计算，一旦发现系统报错量级出现异常上涨，即刻触发决策逻辑。

- **止血层**
  - 在异常投递后，系统将结合准入决策、保护规则库以及当前的压测上下文进行深度计算。
  - **压测溯源技术：**这一步至关重要，自愈平台通过识别压测流量标识，能够精准判定当前的故障是由压测任务引发的，而非正常的业务波动。

- **压测平台**
  - 基于上述决策结果，自愈平台向压测平台发送中断信号，执行自动化止血动作。
  - 在执行熔断的同时，系统会同步启动通知触达机制与应急事件匹配，确保相关负责人在第一时间掌握故障详情。

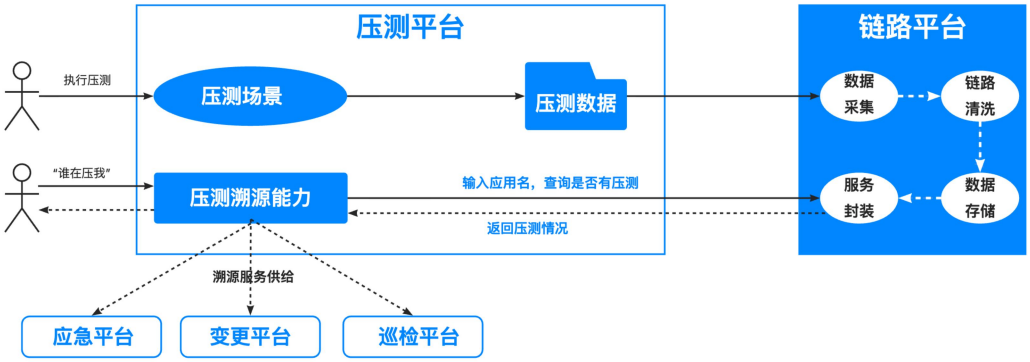
- **响应效率的质变：**
  - 依托于高性能的实时计算能力，熔断时效从早期的分钟级演进到了秒级响应。
  - 目前的响应延迟严格控制在 20 秒之内，这意味着在压测引发的风险发生雏形时，系统就已经完成了从感知到中断的全过程。

- **极低的接入成本：**
  - 只要是遵循蚂蚁标准中间件体系与标准日志打印框架的应用，均可实现轻量化接入。
  - 这种基于标准化框架的操作优势，使得我们能够在大规模复杂微服务场景下，依然保持熔断流程的高度顺畅与可靠。

压测溯源定位技术

蚂蚁全链路压测平台核心技术介绍_压测溯源定位

压测溯源：通过应用 appName 查询该应用当前的压测情况，快速获取某一应用被哪个场景压测，便于进行压测来源判断，异常情况下能够及时应急处理



在压测的过程中，假设有一个系统在报错，然后你怎么知道你这个系统是否被压测流量压到了？所以就需要压测溯源定位的这一个功能。

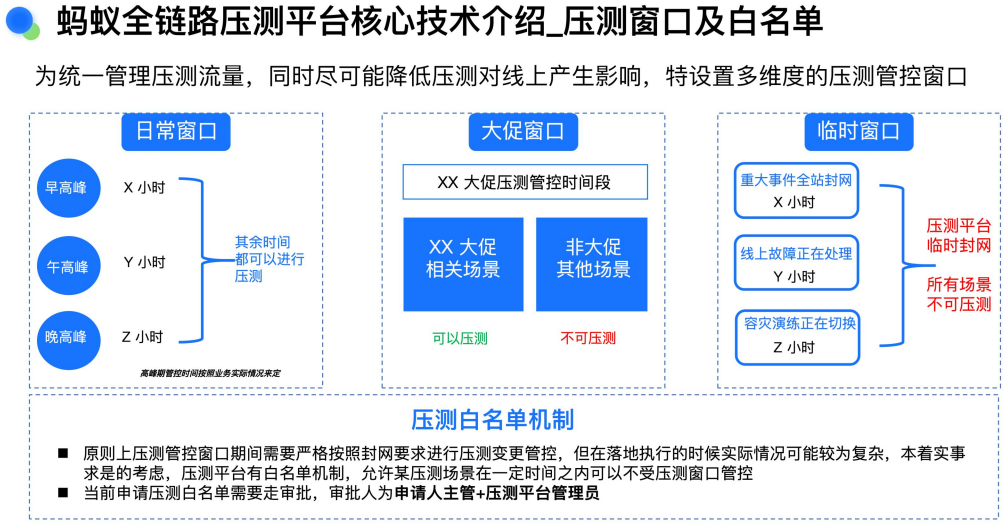
压测溯源通过应用 AppName 查询该应用当前的压测情况，快速获知某一个应用被哪个场景压测，便于进行压测来源判断。在异常情况下，能够更及时地进行应急处理。

具体流程是：

用户在执行压测时，会在压测场景里产生一批压测相关数据，这些数据会同步到右侧的链路平台。链路平台主要负责数据采集与链路清洗，清洗完成后进行存储，并对外提供服务接口，供压测平台调用。

当我们需要主动或被动地查询某个应用“到底有没有被压测”时，就通过该能力去查询链路平台。链路平台会返回结果：例如“确实存在压测”。

这项能力可以复用到多个平台中，最常见的是应急平台和变更平台。因为压测本身也可以视作一种“变更”，线上出问题时，我们可以基于压测溯源能力快速判断：问题是否与压测相关、是否存在压测流量影响。



压测行为必须为线上环境的关键时间节点和业务稳定性让步。为了实现这一目标，我们构建了一套精细化的“管控窗口”与“白名单”体系。

三大压测管控窗口

- **日常窗口（Routine Windows）：**
    - 基于业务流量的波峰波谷规律，我们将早高峰、午高峰及晚高峰定义为禁压期；
    - 在高峰时段内，压测平台会通过技术手段直接阻断任务的发起，确保生产环境的系统资源全力支撑真实用户访问。
- **大促窗口（Big Promotion Windows）：**
    - 在双十一、五福开奖等高敏感的大促期间，全站的关注度与资源配置均处于顶格保障状态；
    - 此时段采取分级管控：唯有经过报备且专门为大促预热设计的业务场景允许执行压测，其余非相关场景一律处于禁压状态，防止非必要流量对核心业务稳定性造成干扰。
- **临时窗口（Emergency Block）：**
    - 这是一种面向突发状况的“强制熔断”机制。当线上应用出现重大故障、全站进入紧急封网状态，或正在执行机房故障转移等容灾演练时，压测平台将立即封锁。

- 特别是在机房切流过程中，流量会跨地域向备用机房高度集聚，此时若叠加压测流量极易导致系统崩溃，因此必须通过临时窗口实现全站禁压。

压测白名单

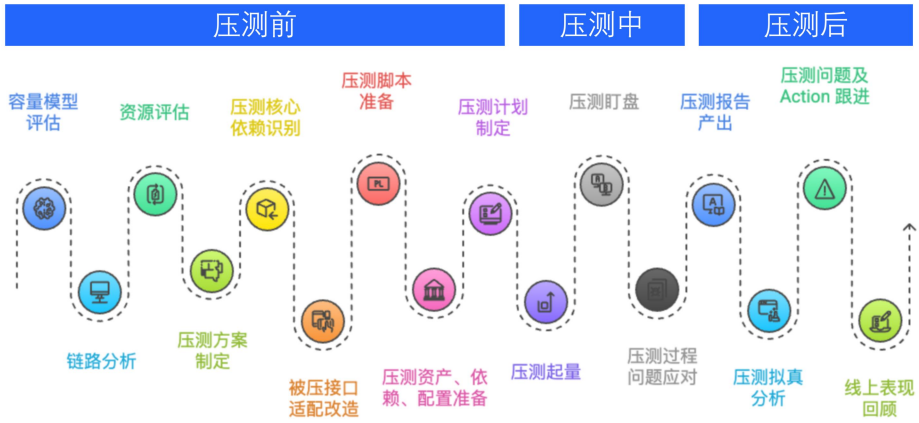
- **避灾出口：**虽然窗口期设置了严格的管控规则，但考虑到实际研发节奏中可能存在紧急回归或线上验证的需求，我们保留了白名单作为特许场景的出口。
- **严格审批制：**为防止权限滥用，白名单的申请有着极高的门槛。申请人必须明确说明压测原因，并提交至其主管以及压测平台管理员进行联合审批；只有在确保风险可控、相关干系人知情且达成共识的前提下，才会被授予特定时限内的压测权限。

03

蚂蚁大促全链路压测实战

全链路压测基本流程

全链路压测基本流程



全链路压测基本流程：从预防到闭环的稳定性命脉

即便拥有了完善的监控平台与技术架构，若缺乏严谨的流程管理与机制约束，依然难以从根本上防控系统崩溃的风险；因此，我们定义了一套覆盖预案、执行与复盘的全生命周期标准流程。

压测前：重在预判与前置改造

这是整个流程中最为关键的筹备阶段，旨在通过深度的业务梳理排除潜在的系统性风险。

- **模型与链路评估**：首要任务是基于预期的业务目标进行精准的容量建模，并针对待压测链路开展自动化或人工深度分析，明确上下游的拓扑关系及其依赖节点的就绪状态
- **资源与方案制定**：根据链路分析结果对全链路进行横向的资源评估，执行必要的扩缩容操作，并明确压测范围、核心指标与执行排期
- **识别核心依赖**：在漫长的业务链路中，必须精准识别出那些决定生死的核心依赖项，确保在压测过程中对这些"关键少数"进行重点观测
- **适配改造与脚本就绪**：针对识别出的不支持压测、无法透传流量标的特殊接口进行代码级改造，并配置相应的压测脚本，以确保流量能够真实且无损地模拟生产场景
- **资产隔离与环境准备**：

- **数据隔离**：严禁混淆压测资产与生产资产，必须对压测涉及的金额、代金券、优惠券等敏感核销资源进行物理或逻辑层面的严格隔离
- **防范风险**：通过标准化配置，杜绝因向生产环境注入压测流量而导致真实预算被误消耗等生产事故的发生

压测中：稳步起量与实时监控

执行阶段的核心在于"循序渐进"与"全盘掌控"。

- **梯次起量**：严格按照既定的压测计划逐步释放压力，观察系统在不同并发梯度下的水位表现
- **实时盯盘与异常处置**：由专人负责实时监控系统各项核心指标，一旦检测到性能拐点或异常报错，迅速响应并针对瓶颈进行即时处理

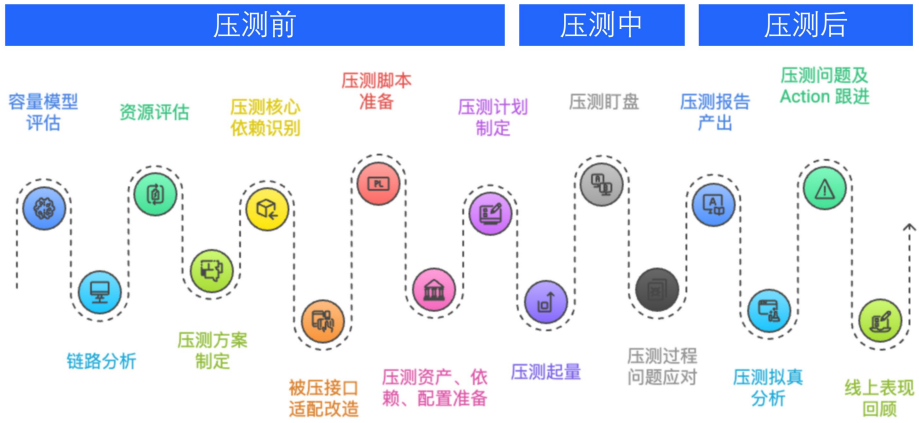
压测后：数据回溯与闭环跟进

压测的终点并非流量停止，而是问题的闭环。

- **报告产出与拟真度分析**：汇总压测数据得出结论，并深度对比压测表现与生产环境真实水位之间的差异，分析环境拟真度对结果的影响
- **问题闭环管理**：针对压测中暴露的短板制定具体的 Action 并分配负责人跟进，确保所有技术债在下一次业务高峰前得到妥善解决
- **线上表现回顾**：将压测预测值与实际生产表现进行联动回放，通过这一环扣一环的闭环机制，我们才能在亿级设备与动态流量的冲击下，为系统稳定性构筑起坚实的技术屏障

全链路压测的 15 个关键节点

全链路压测基本流程



标准化协同工程的全生命周期

全链路压测并非简单的流量灌入，而是一场跨团队、跨系统的标准化协同工程，其核心生命周期被严格拆解为由 15 个关键节点组成的三个阶段。

一、压测前：评估、识别、改造与准备（10 个节点）

- **容量模型评估**：明确系统容量上限、瓶颈形态与风险边界
- **链路分析**：梳理业务请求路径与关键链路，确认观测与定位路径可用
- **资源评估**：核对压测所需 CPU/内存/网络/存储等资源是否充足，避免“资源不足导致误判”
- **压测方案制定**：定义场景、流量模型、指标口径、停止条件与回滚预案
- **压测核心依赖识别**：识别关键依赖（DB/缓存/MQ/第三方等）与其容量/限流策略
- **被压接口适配改造**：对被压接口做隔离/降级/限流/影子能力等适配，保证数据与影响可控
- **压测脚本准备**：脚本与参数化配置就绪，可复用、可回放、可对账
- **压测资产、依赖、配置准备**：压测账号、权限、环境、依赖版本、配置项全部核对到位
- **压测计划制定**：明确时间表、负责人、盯盘分工、沟通机制与应急流程

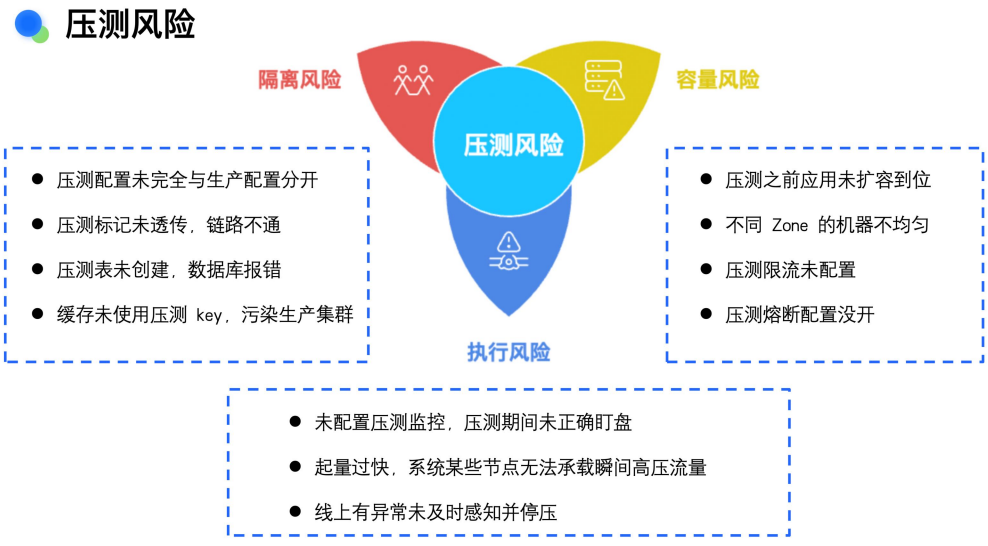
- **压测起量**：按阶梯/波次起量策略执行（起量节奏、拐点观察、阈值触发）

二、压测中：集中盯盘与问题处置（2 个节点）

- **压测盯盘**：集中核心人员看板巡检（业务指标 + 中间件/资源指标），实时捕捉异常抖动
- **压测过程问题应对**：按预案快速止损与调整（限流/降级/扩容/回滚/暂停起量等）

三、压测后：结论产出与闭环验证（4 个节点）

- **压测报告产出**：沉淀容量结论、瓶颈证据链、风险点与改进建议
- **压测拟真分析**：对比真实高峰与压测表现，评估“量是否压的上去”，“压得准不准”，目前也还在推进过程中。（这是重点难点）
- **压测问题及 Action 跟进**：将瓶颈转成任务，明确 owner/DDL/验收标准，闭环推进
- **线上表现回顾**：结合后续线上高峰复盘，验证优化是否有效、结论是否可信



核心原则：严防"压测"变"事故"

确保压测活动本身不会对线上生产环境造成负面冲击，我们不仅要通过压测摸清系统底座的容量，更要前置性地识别并规避各类潜在风险。我们将压测风险抽象为三个关键维度，构建起一套完整的风险防控 Checklist：

隔离风险：严控生产环境的"纯净度"

- **配置与链路隔离**：确保压测配置与生产配置实现逻辑上的彻底分离，避免因代码习惯导致压测标记无法在全链路中完整透传，防止压测流量误入生产路径；
- **存储与缓存防护**：严禁在压测中使用生产环境的 Key 或直接操作生产表，必须确保压测影子表提前就绪，防止因缓存污染或数据混杂导致生产数据库异常；
- **核心目标**：从逻辑与物理两个层面切断压测流量对生产数据的干扰路径，保障生产业务的绝对纯净。

容量风险：确保系统防御机制"不裸跑"

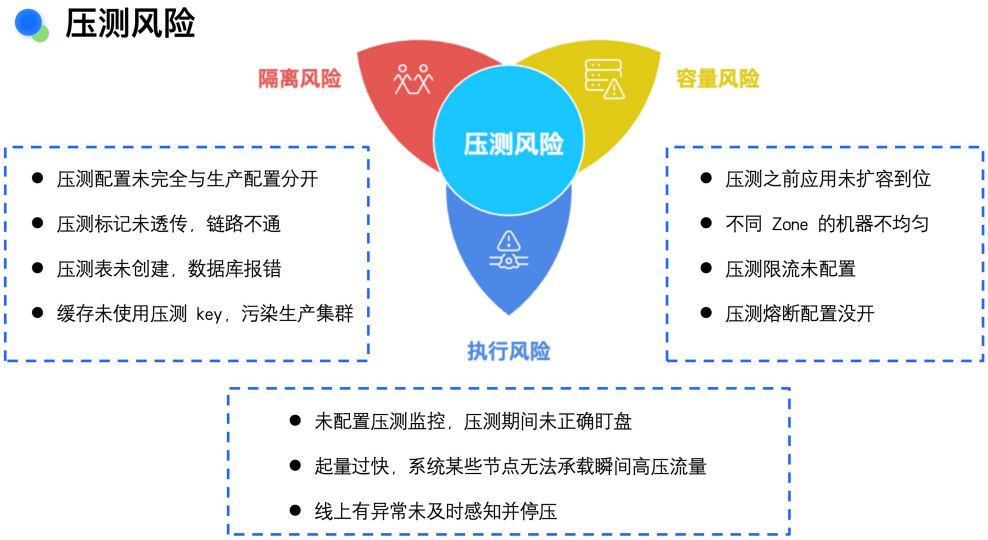
- **资源就绪评估**：建立严格的扩容核查机制，防止在资源未扩容到位或 Zone 间机器分配不均的情况下盲目起量；
- **防御机制配置**：
    - **动态限流**：必须配置压测专属限流策略，防止压测流量挤占正常的生产带宽；
    - **熔断机制**：明确压测熔断的阈值与开关状态，确保在极端压力下系统能自动止损；
- **核心目标**：通过资源预补偿与防御机制的预配置，为压测过程提供稳固的弹性和兜底能力。

执行风险：保障压测过程的"可控性"

- **可观测性保障**：必须配置压测专项监控大盘，严禁在缺乏观测能力或盯盘缺失的情况下进行操作，确保异常情况能被第一时间感知；
- **阶梯式起量策略**：针对 Java 体系下 JVM 预热的特性，应采取由浅入深的提量策略，严禁在系统未充分预热前直接施加高强度脉冲压力，防止瞬间峰值流量直接击坏系统节点；
- **风险熔断机制**：建立压测期间的“紧急停压”决策流程，一旦线上出现异常感知，必须立即采取行动防止影响范围扩大。

只有建立起“隔离严密、容量对等、执行可控”的三维防护体系，我们才能在亿级规模的流量冲击下，既能精准测出系统的极限，又能确保生产环境的稳如泰山。

压测前置准备：风险防控 Checklist



做全链路压测，最重要的底线是：**不能因为压测引发线上问题**。围绕这条底线，风险主要分为**隔离风险、执行风险、容量风险** 三类（与图一致）。

隔离风险

隔离风险的核心是：**压测流量/数据没有和生产彻底分开**，导致“压着压着把生产搞脏、搞乱、搞挂”。

- **压测配置未完全与生产配置分开**：生产与压测配置混用，压测行为直接作用到生产路径。
- **压测标记未透传，链路不通**：框架本来支持透传，但由于代码习惯或链路改造不一致，标记丢失，导致压测流量在链路中“变回生产流量”。
- **缓存未使用压测 key，污染生产集群**：缓存 key 未区分压测/生产，压测写入覆盖生产缓存，可能引发生产集群异常甚至连锁问题。
- **压测表未创建，数据库报错**：压测所需表结构/隔离库表未准备，压测流量落到生产库表，造成报错或脏数据。

执行风险

执行风险的核心是：**压测过程不可控、不可观测、不可及时止损**，即使问题已经发生也没被发现，导致影响扩大。

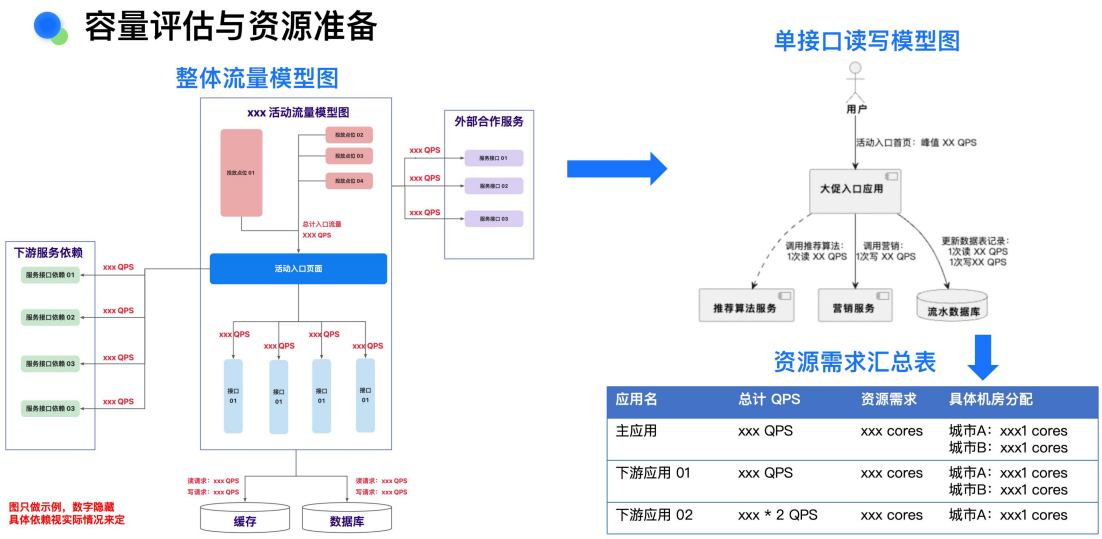
- **未配置压测监控，压测期间未正确盯盘**：没有压测专用监控或盯错盘，关键指标无人看，风险极高。
- **起量过快，系统某些节点无法承载瞬间高压流量**：尤其是常见的 Java/JVM 系统，若未预热、未触发 JIT、连接池/线程池/缓存未就绪，脉冲式起量可能直接把节点打崩。更稳妥的方式通常是**先缓慢爬升到稳定量级，再做脉冲验证**。
- **线上有异常未及时感知并停压**：压测过程中已经出现异常，但未被及时发现并停止压测，导致影响范围扩大（典型执行风险）。

容量风险

容量风险的核心是：**明知道要扩容/要保护，但扩容不到位、保护没配置、缺少检查机制**，最终压测挤占生产资源。

- **压测之前应用未扩容到位**：评估已得出要扩容，但没扩到位，且缺少压测前的检查/门禁机制，直接开压。
- **不同 Zone 的机器不均匀**：实例分布不均导致局部 Zone 先顶不住，出现“整体资源看着够、局部先崩”的现象。
- **压测限流未配置**：压测流量没有限流保护，直接挤占生产流量，造成生产请求被拖慢或失败。
- **压测熔断配置没开**：熔断需要开关与阈值配置，未启用时一旦异常放大，系统更容易雪崩式扩散。

容量评估与资源准备



幻灯片 23

压测之前，必须先把“流量怎么来、怎么拆、会放大到哪里、最后需要多少资源”算清楚。我们通常会产出三样东西（与图一致）：1）**整体流量模型图**→2）**单接口读写模型图**→3）**资源需求汇总表**。资源交付到位后，才进入后续压测（脚本/表/执行等）。

---

1) 整体流量模型图（从入口到依赖，一次画清楚）

目标是把“入口流量”拆成“页面/接口 QPS”，并把**下游依赖与放大比**也画进去。

- **确认入口流量与投放点位**：入口有哪些投放点位（投放点位 01/02/03/04），汇总得到**总计入口流量（XXX QPS）**。
- **拆活动入口页的接口**：流量进到“活动入口页面”后，系统内部有哪些**接口 01/02/03/...**，先全部拆出来。
- **按比例分摊到页面/接口**：每条链路、每个接口的占比怎么来？
  - 可以基于**历史经验**
  - 或找**产品/运营**拿页面点击率、转化率等数据
  - 最终把每个接口对应的 **QPS** 估出来
- **补齐下游依赖与放大比**：
  - 下游服务依赖（服务接口依赖 01/02/03...）
  - 外部合作服务（服务接口 01/02/03...）

- 注意“**放大比**”：一次请求可能触发多次调用（例如 1 次入口请求 → 下游调用 N 次），要在图上标清。

- **标出缓存/数据库读写压力与隔离点**：
  - 缓存、数据库分别有多少读/写 QPS
  - 是否有隔离（压测/生产 key、库表隔离）
  - 是否需要扩容（容量/连接数/IO/热点）

这张图的作用：把“流量从哪里来 → 经过哪些接口 → 打到哪些依赖/缓存/DB → 放大到多少”一次说清楚。

---

2) 单接口读写模型图（把“一个接口”拆到可算资源）

整体图画完后，要把每个关键接口进一步“算细”，形成**单接口读写模型**：

- **从接口入口再拆一遍流量**：这个接口在不同路径上的占比、峰值 QPS。
- **拆清关键节点与读写次数**：一次请求会：
  - 读缓存几次 / 写缓存几次
  - 查库几次 / 写库几次
  - 调下游服务几次（含重试、并发、批量等）
- **把依赖调用关系画出来**：哪个依赖在关键路径、哪个是放大源头、哪个可能成为瓶颈。

这一步的意义：把“QPS”变成“**每请求成本**（读写次数/依赖调用次数）”，否则资源评估会很虚。

---

3) 资源需求汇总表（把结论变成可交付的资源单）

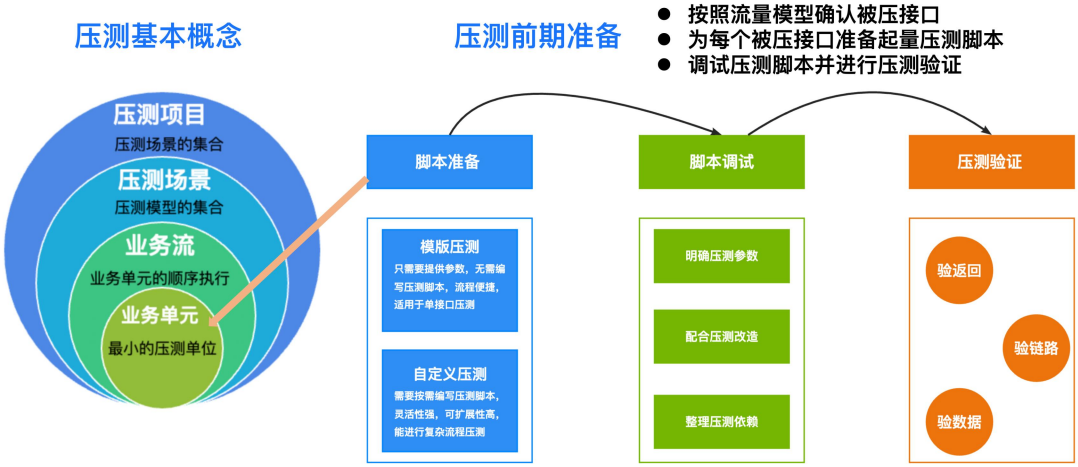
最后把各接口模型汇总成一张“**资源需求汇总表**”（与图右下角一致）：

- 字段建议保持：**应用名 / 总计 QPS / 资源需求（cores 或实例数）/ 具体机房分配（城市 A、城市 B…）**
- 汇总对象覆盖：主应用 + 下游应用/依赖（必要时含外部合作服务的配额/限流策略）
- 输出目的很明确：**把资源需求提上去，尽快扩容交付**

- 交付顺序：**资源交付完 → 才能压测**（否则压测结论不可信，甚至会把线上打出问题）

压测脚本准备与层级管理

压测脚本编写、调试、验证



压测层级的深度解构

在蚂蚁的压测实践中，我们对压测对象进行了逻辑抽象，构建了从宏观支撑到微观执行的四层体系：

- **压测项目（Project）**：作为最顶层的管理维度，通常对应一次大型促销活动或特定团队的年度计划
- **压测场景（Scene）**：由多个脚本组合而成的集合，旨在模拟真实业务中的某种功能模块
- **业务流（Business Flow）**：具体的业务链路，如首页加载流或任务列表流，它描述了用户在系统中的行为轨迹，往往涉及多个接口间的参数传递与上下文复用
- **业务单元（Business Unit）**：整个体系中最核心、最精细的原子单位，通常与单个接口一一对应，是构建所有压测脚本的基石

压测前期准备：脚本从"编写"到"就绪"

基于上述层级，我们确立了标准化的前期准备流程，确保流量能够按照预设模型精准起量：

- **脚本准备的"双模式"驱动：**

- **模板压测**：针对单接口压测，用户只需提供核心参数，无需编写任何代码，极大提升了常规压测的效率
- **自定义压测**：提供高度的灵活性与可扩展性，用于支持复杂的业务逻辑和具有前后置依赖关系的流程压测
- **调试阶段的精细化管控**：在脚本正式上线前，必须明确压测涉及的全部参数，整理复杂的外部依赖，并配合业务侧进行必要的压测代码改造

压测验证：三位一体的质量把关

脚本调试完成后，必须经过严格的压测验证才能进入实战环节。我们强调不仅要"跑得通"，更要"跑得对"，这涉及对系统行为的深度校验：

- **验返回**：确保接口响应结果符合预期逻辑，排查由于压测流量导致的逻辑报错
- **验链路**：核实压测流量是否真实触达了所有预期的服务节点，确保没有请求在中间层丢失
- **验数据**：这是全链路压测中最关键的闭环，需要验证底层数据库（DB）和缓存（Cache）的读写行为是否符合预期，确保数据偏移和入库策略执行准确

由于全链路压测对链路覆盖率的要求极高，目前我们正尝试引入 AI 技术，针对这种高度标准化的压测资产进行生成与配置管理，以进一步提升压测脚本的产出提效。只有在脚本调试就绪、验证无误且资源配置到位的前提下，真正的性能压测才具备开启的基础。

压测配置与数据资产管理

压测配置与资产准备

压测配置：一般分为全局配置和业务自定义配置。全局配置一般用于通用内容配置，例如访问地址、环境信息等内容；业务自定义配置一般实现在业务接口中，在压测期间会推送对应的压测开关、压测特定配置等，来保证全链路压测能走通

压测数据资产：用于业务流或业务单元脚本中需要用到的压测数据，一般是压测用户资产；在执行压测时，压测中心会将压测数据分发到各台压力机中，用于压测任务的执行

核心理念：以配置与数据为基石

全链路压测绝非仅仅是向系统施加压力，其核心挑战在于如何让压力流量在复杂的分布式架构中，既能像真实业务一样精准流转，又能与生产数据实现完美的逻辑隔离。如果配置出现偏差导致链路无法触达，压测的有效性将大打折扣。为此，我们构建了"配置管理"与"资产准备"双驱动体系，通过压测标（Traffic Tagging）的原子化传递，实现对生产环境的保真模拟与安全防护。

压测配置（Pressure Test Configuration）：链路通达的控制器

- **全局配置管理**：定义通用的环境变量与基础访问地址，下发全局一致的业务参数，确保各系统对压测流量的认知保持高度同步
- **业务自定义配置**：在接口层面动态推送功能开关，激活特定的压测逻辑；针对特定业务场景调整配置参数，从代码层面确保逻辑链路能够根据预设路径顺畅执行

压测数据资产（Pressure Test Data Assets）：流量模拟的燃料

- **资产构造与隔离**：严禁使用生产真实用户资产进行压测，必须建立一套完整的流程来构造专属的压测用户、营销资产及模拟金额
- **自动化分发机制**：由压测中心统一调度，将清洗后的数据资产实时分发至各压力机节点，支撑高并发场景下的真实行为模拟

技术落地：压测标全链路穿透逻辑

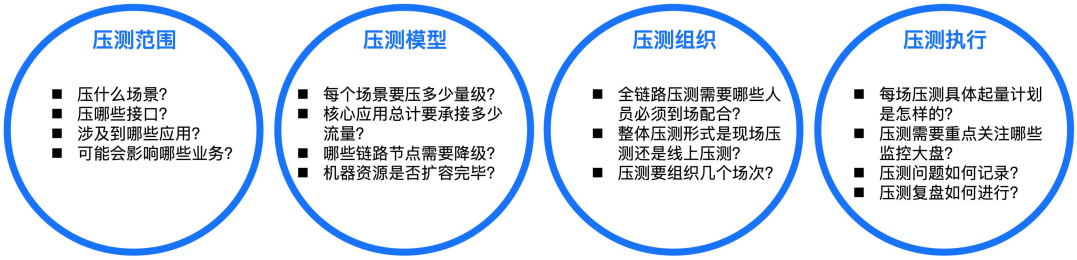
- **流量接入层：身份识别与打标**
    - 系统需在入口处精准识别"压测用户资产"，并注入统一的"压测标"，这是后续所有隔离逻辑的触发标准
- **应用服务层：逻辑兼容与配置读取**
    - 服务节点通过拦截器识别压测标记，动态读取业务定义的压测配置，启用专门的兼容逻辑以适配压测流量
- **中间件层：隔离、白名单与挡板策略**
    - 中间件负责执行核心的隔离任务，通过白名单机制识别合法压测流；对于不支持压测的特殊组件，则通过预设的"压测挡板"进行流量截断或降级处理，确保护航生产安全
- **数据存储层：影子库表与缓存路由**

- **缓存端**：透传压测标记，实现压测 Key 的独立读写，避免污染生产缓存
- **数据库**：执行严格的压测路由逻辑，确保所有写入操作落入影子表或特定压测数据库中，实现物理或逻辑上的数据隔离

压测计划制定与执行策略

压测计划制定

由于全链路压测涉及到的范围广、链路长、人员众多，压测负责人必须制定详细的压测计划，以便压测过程顺利、压测问题及时解决，助力全链路压测尽快达成既定目标



核心背景：不打无准备之仗

全链路压测是一项系统性工程，具有范围广、链路长、参与人员多等复杂逻辑，作为压测负责人，必须在正式启动前制定详尽的压测计划；这份计划不仅是行动指南，更是全链路压测的"作战手册"，旨在确保过程有序、风险可控，并最终达成业务预期的性能指标。

压测范围：划定战场边界

- **场景与接口**：明确本次压测覆盖哪些业务场景，细化到具体承载流量的接口，确保不遗漏核心节点
- **应用影响面**：梳理压测涉及的底层应用及其上下游依赖，勾勒出完整的流量流向大盘

压测模型：量化压力指标

- **流量规模**：精准盘点每个场景需触达的 QPS 量级，并计算单一核心应用在多接口组合下需承载的总流量值
- **资源保障**：确认机器资源是否扩容完毕，针对非核心链路节点，需明确在高峰期是否执行降级策略或通过限流手段来保障系统整体硬度，而非盲目扩容

压测组织：建立作战方阵

- **人员值班**：确定哪些核心开发、运维及业务人员必须实时在岗，以应对压测期间的突发异常
- **形式与频次**：明确是进行线上实战压测还是线下模拟演练，并提前规划好需要开展的压测轮次与场次

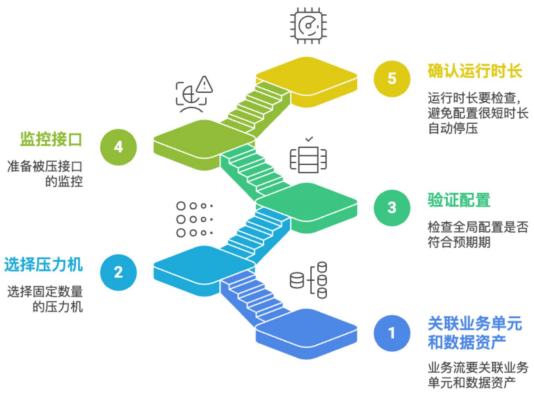
压测执行：细化落地过程

- **起量策略**：制定具体的起量计划，是采用百分比平滑起量、梯度起量，还是针对特定场景进行瞬时脉冲起量
- **观测与闭环**：定义监控大盘的关键指标指标，建立压测问题的记录与分发流转机制，确保每一个瓶颈都能被发现、被记录、被解决

在将压测单元与数据资产准备就绪、并完成压力机选型验证后，我们即可按照这套“作战手册”进入实操阶段，通过标准化、流程化的计划引导，在亿级设备与千万用户规模的复杂环境下，实现从流量发起、监控观测到复盘改进的全链路闭环管控。

压测执行与实时盯盘

压测执行及盯盘



压测执行的标准化操作流程

在压测执行及盯盘阶段，SRE 的核心任务是确保每一个施压动作都精准落地，并实时捕捉系统在高负载下的细微颤动，是一套严谨的“起压、观测、验证”标准化操作流程。

压测执行的阶梯式标准路径

- **资源与链路对齐**：
    - **业务单元关联**：首先必须将业务流准确关联至具体的业务单元与底层数据资产，确保施压覆盖的链路与生产环境的逻辑架构高度一致；
    - **负载能力预置**：选择足够数量的压力机集群，避免因施压端性能瓶颈导致观测数据失真，保证流量源头的稳定性。
- **配置验证与防护**：
    - **全局参数核对**：在正式起压前，针对并发数、QPS 阈值、持续时长等全局配置进行最后检查，确保压测能够维持足够的运行时长，从而充分暴露内存溢出或垃圾回收机制引发的长尾问题；
    - **监控视图准备**：建立针对被压接口的实时监控 Dashboard，将响应时间、吞吐量和错误率等关键指标前置，进入“盯盘”状态。

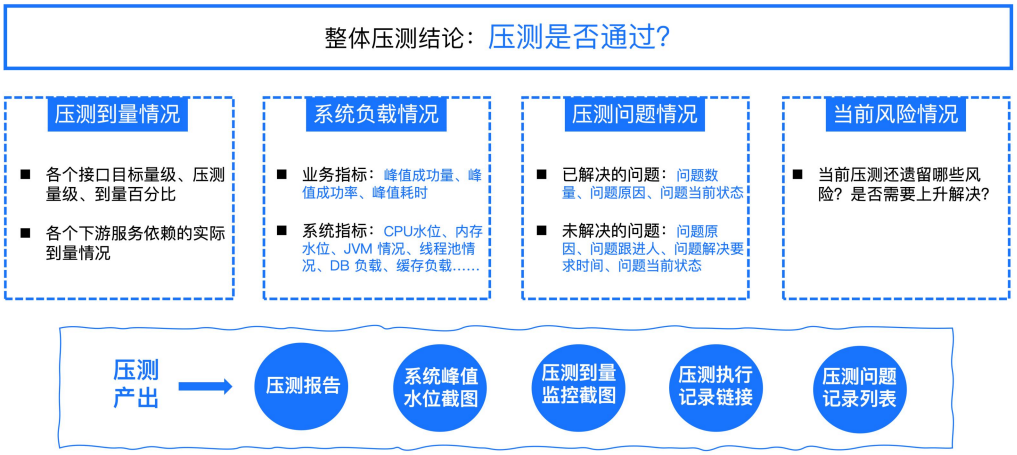
执行现场的动态观测与交付逻辑

- **阶梯提量与流量拟合**：
    - 在执行过程中，我们通常采取百分比递增、脉冲式冲击或特定顺序造数后再验证等策略，观察系统在不同流量梯度下的水位表现；
    - **全链路水位监测**：当压测达到峰值时，需要全面穿透基础设施层与中间件层，密切关注计算资源、存储 I/O、数据库连接池及各类中间件的抗压状态，识别性能拐点。
- **最终结果评估与交付**：
    - 压测不仅仅要获得一组性能数字，更需要针对“目标是否达成”给出一个确定的答案，即系统是否在预定容量下保持了预期的稳定性；
    - **量化回答核心诉求**：最终的通过标准必须锚定在实际业务量级上，通过复盘系统水位与观测到的瓶颈，确定当前架构是否具备支撑亿级流量的能力。

通过这套从底层资产关联到高位监控盯盘的五步走策略，我们将不确定的压力演练转化为可预期的、平台化的质量保障过程，为后续的容量规划提供了真实可靠的数据支撑。

压测结果评估与准出标准

压测准出



压测准出不仅是终点，更是决策的基点

- **压测结论的终极目标**：明确回答"压测是否通过"，并基于详尽的数据模型评估系统是否具备承载预期业务流量的能力
- **降本增效的量化准绳**：在追求高可用的同时，通过压测深度榨取硬件性能，如果系统在设计峰值下负载水位过低，则需触发资源缩容流程，确保每一份算力都被充分利用

四大评估维度：全方位审视系统成色

- **压测到量情况**：
  - 核心在于对比预期目标量级与实际达成量级，精确计算各接口的到量百分比
  - 必须穿透全链路，验证下游服务与各层级依赖是否均已负载到预设的饱和度
- **系统负载表现**：
  - **业务韧性**：复盘峰值成功率、吞吐量与耗时走势，确保在高压下用户体验不产生断崖式下跌
  - **资源镜像**：严密监测 CPU、内存、JVM 堆栈及 DB 负载，以此判定系统是处于健康区间还是强弩之末
- **问题闭环管理**：

- **存量清理**：梳理已修复问题的根因与执行路径，防止故障复现
- **动态追踪**：针对未解决的问题，必须落实到具体责任人并在限定时间内给出补救措施

- **风险前瞻预测**：
  - 对遗留的隐患进行分级，明确是否需要触发上升机制申请上层决策，杜绝带病上线

产出交付：标准化报告沉淀技术资产

- **全链路压测报告**：这是整个 SRE 工作的最终交付实物，作为面向技术负责人与业务方、传达系统稳态的关键媒介，报告必须具备数据自洽性
- **证据链闭环**：整合水位截图、到量监控、执行日志与问题清单，形成一套可追溯的标准化附件

这套从“原始指标采集”到“深度风险评估”再到“标准化文档交付”的工作流，不仅是为了给本次压测画上句号，更是为了在下一次迭代中，能够有据可依地回答出系统负载能力的真实底牌。接下来，我们将探讨在这个过程中，AI 如何赋能压测实现更高效的演进。

（三）总结及展望

AI 压测探索与展望

04 未来已来：AI 压测初探

核心主题：AI 赋能下的稳定性保障新范式

- **技术演进**：标题中"未来已来"不仅是愿景的宣告，更标志着 AIOps 已经跨越了概念蓝图，正式进入 SRE 稳定性保障的实战环节
- **压测重塑**：从传统的基于固定脚本的 Load Testing 转向 AI 驱动的智能化压测，通过自动生成动态流量模型、实时异常感知和参数自适应调节，彻底打破了传统性能评估的局限性
- **初探精神**：作为第四章节的开篇，这一页聚焦于我们在 AI 压测领域的初步探索与技术方案雏形

技术视角下的章节定位

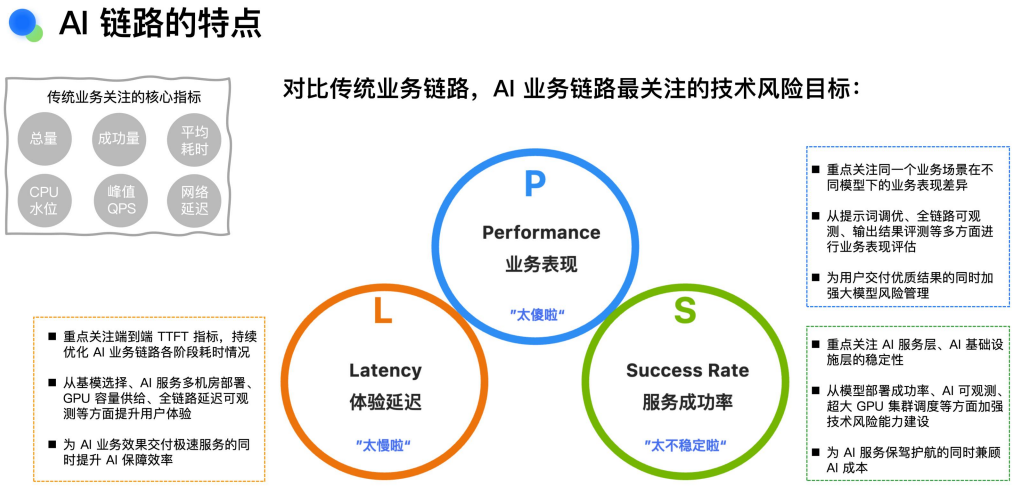
- **环节定位**：序列标识"04"清晰地指明了在整个高可用架构演进路径中，AI 智能化能力的引入是继自动化运维之后的关键跃迁
- **设计哲学**：极简的视觉风格旨在引导并聚焦听众的注意力，暂时剥离复杂的架构图和监控指标，从而深入探讨如何利用大模型与机器学习算法解决业务逻辑愈发复杂、流量波峰难以预测的压测痛点

AI 压测的探索维度

- **智能化流量生成**：摈弃简单的回放机制，利用 AI 模拟真实用户的行为特征，生成更贴近生产环境的非线性随机流量
- **自愈式压测控制**：在压测过程中实时监测系统水位，AI 能够通过反馈回路自动调整并发速率，在不压垮系统的前提下精准探测瓶颈
- **多维结果研判**：通过智能化异常检测模型（Anomaly Detection），系统能从海量监控指标中自动识别出细微的性能劣化，而非仅仅依赖硬编码的告警阈值

步入第四章，我们将从“AI 压测初探”出发，详细解构 SRE 团队如何通过算法驱动的测试手段，在亿级设备接入与千万级并发场景下，构建起一套比人类经验更敏锐、比传统脚本更具韧性的性能验证体系。

AI 链路的 P-L-S 模型



观察维度的变迁：从传统指标向 AI 特性对齐

在传统业务场景中，SRE 的关注点通常集中在总量、成功量、平均耗时、CPU 水位、峰值 QPS 以及网络延迟这六大核心指标上；然而，随着业务逻辑步入 AI 时代，原有的监控模型已不足以捕捉大模型链路的波动，我们需要将视角从基础资源层跃迁至业务表现层，通过构建 P-L-S（Performance, Latency, Success Rate）三维模型，重新定义 AI 生态下的技术风险目标。

P-L-S 模型：定义 AI 链路的三大核心命题

- **P (Performance) —— 解决“模型太傻”的业务表现问题：**
    - AI 业务的本质在于输出质量，我们要关注同一业务场景在不同模型下的表现差异，避免模型因认知局限或生成逻辑错误导致的评估不佳；
    - 通过提示词（Prompt）优化、模型风险管理以及多维度的结果评测，确保在交付优质生成结果的同时，实现全链路的业务质量观测。
- **L (Latency) —— 解决“体验太慢”的响应延迟问题：**
    - 在 AI 推理场景下，用户对耗时的感知极其敏锐，我们必须建立起端到端的全链路延迟观测体系，重点攻坚 **TTFT**（首字延迟）这一关键指标；

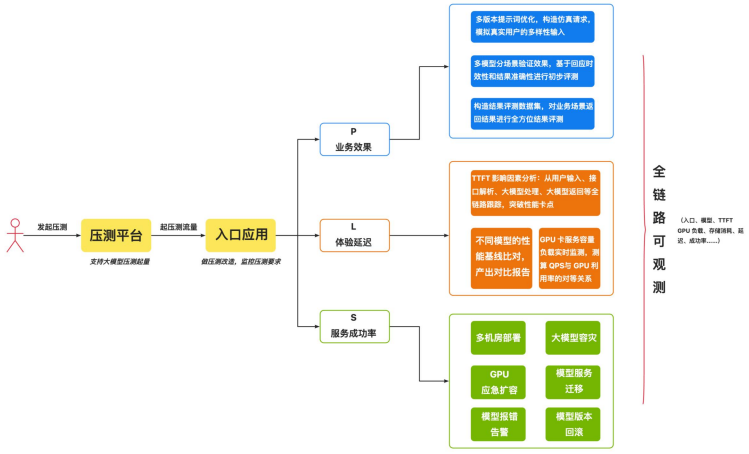
- 在底层基模选择、多机房协同部署、GPU 算力保障等环节持续发力，以基础设施的供给侧优化支撑业务效果的极速交付，消除长时间不可响应的问题。

- **S (Success Rate) —— 解决“系统太不稳”的服务成功率问题：**
  - AI 链路的稳定性不仅取决于应用层，更深受底层大片 GPU 集群调度、基础设施鲁棒性的影响，任何微小的波动都可能演变为服务无响应；
  - 我们需要加强对 AI 可观测性与模型部署成功率的建设，在保障超大集群稳定运行的同时，精细化管理 AI 算力成本，为业务持续可用保驾护航。

正是通过对 P-L-S 这三个核心维度的深度解析，我们能够更精准地勾勒出 AI 链路的风险全景图，从而在复杂的算法、激增的算力需求与波动的用户预期之间，寻找到一套属于 AI 原生时代的高可用保障策略。

AI 全链路压测架构方案

AI 链路压测实践



一、AI 链路压测的核心范式：从传统指标向 PLS 模型演进

在蚂蚁集团推进 AI 业务转型的过程中，我们针对大模型（LLM）的特性，构建了这套全链路压测架构，其核心逻辑在于将压测重点从传统的“CPU/内存”转向更贴合 AI 业务本质的 PLS 评估模型：

- **P (Precision/Quality) 业务效果：**

- 我们重点关注多版本提示词（Prompt）在不同压测流量下的表现，通过构造仿真请求与评测数据集，验证大模型在不同场景下返回结果的准确性与时效性；
- 诚然，业务效果的自动化评测是目前最具挑战性的环节，我们正在不断探索如何更精准地量化模型产出的质量。

- **L (Latency) 体验延迟：**
  - AI 场景下的延迟受模型版本与硬件性能双重驱动，我们通过对首字延迟（TTFT）的全链路跟踪，深入分析从用户输入、接口解析到模型推理各阶段的耗时。
  - 压测平台会详细产出不同模型的性能基线对比报告，明确 QPS 增长与 GPU 资源消耗之间的线性关系，从而突破性能卡点。
- **S (Stability/Success rate) 服务成功率：**
  - 侧重于基础设施的稳健性，包括 GPU 负载的实时监测、千万级请求下的多机房容灾能力、以及模型服务的自动迁移与故障回滚机制；
  - 在大模型服务出现异常时，系统需确保具备快速扩容与告警响应的兜底能力。

二、架构支撑层：让压测具备端到端的穿透力

- **压测平台与入口应用：**
  - 压测平台承担“起量”职责，模拟真实业务的高并发冲击；入口应用则同步进行压测改造，以确保流量标签能完整传递至后端模型服务。
- **全链路可观测性：**
  - 方案的最右侧通过纵向观测链条，将入口应用、模型计算、GPU 负载、存储消耗、TTFT 等零散指标串联起来；
  - 目前我们已基本实现 L（延迟）与 S（成功率）的深度覆盖，力求在复杂的大模型调用链路中，实现每一毫秒、每一个 Token 成本的可追溯。

这套体系反映了我们在 AI 运维领域的演进思路：不仅要测试系统能否“抗得住”流量，更要验证由于硬件负载波动或提示词差异带来的模型表现衰减。虽然 P 维度的业务评测尚在深水区探索，但通过这套 PLS 架构，我们已经能够为 AI 产品提供从容量评估到稳定性校验的全生命周期保障。

5.4.8 腾讯游戏高可用保障体系实践

SRE Elite 精选原因

腾讯游戏运营着超过 300 款自研与代理游戏，日活用户数以亿计。游戏行业对高可用性有着极为苛刻的要求：竞技类游戏延迟需控制在 20ms 以内，帧率低于 30FPS 即会引发明显卡顿；单机承载能力远低于传统互联网应用，CPU 利用率超过 60%就可能导致玩家投诉；活动期间流量可在一分钟内暴增 8-10 倍，极易触发雪崩效应。

面对这些挑战，腾讯游戏构建了一套贯穿全生命周期的高可用保障体系。本案例系统分享了从设计阶段的 TDR 技术评审实现高可用"左移"，到测试阶段的分层混沌实验与全链路压测结合，再到上线阶段的生产环境动态权重压测方案，以及运营阶段的 AI 辅助故障定位等完整实践。这套体系化的方法论对于构建大规模、高并发、强实时性业务的稳定性保障具有重要参考价值。

（一）背景及设计原则

游戏行业高可用性挑战

01

游戏行业高可用挑战

游戏行业在构建高可用系统时，面临一系列独特的挑战。这些挑战源于游戏本身的特性和运营规模。

- **业务数量庞杂且高度异构：**
    - 累计运营的自研与代理游戏超过 300 款，这意味着 SRE 团队需要应对数百个独立产品线，每个产品可能都有其独特的技术栈、架构设计和特定需求。
    - 这种业务的庞大数量和异构性，使得构建统一、高效且具备高可用能力的整体架构变得异常复杂。
- **对实时性与流畅度的高标准要求：**
    - **高实时性要求：** 游戏对毫秒级的延迟高度敏感。哪怕是微小的延迟，都可能直接影响玩家的游戏体验，甚至导致玩家流失，这对服务器响应速度和网络传输效率提出了极致要求。
    - **高流畅度要求：** 流畅度是衡量游戏体验的核心指标。当游戏帧率低于 30FPS 时，玩家会明显感知到卡顿。对于 MOBA 或 FPS 这类快节奏游戏，理想帧率应稳定在 60FPS 以上，任何波动都可能导致玩家操作失准，甚至影响竞技结果。
- **单机承载能力显著偏低：**
    - 相较于传统互联网应用，游戏逻辑的复杂性导致其单机承载能力远低于预期。

- 这意味着在重大活动保障等场景下，需要快速、大规模地弹性扩容，一次性可能需要扩容数十万核心的服务器资源。
- 服务器性能也高度敏感，CPU 利用率一旦超过 60%，就可能引发玩家卡顿投诉，因此对资源管理和优化提出了更高要求。

- **高频且规模巨大的流量洪峰冲击：**
  - 游戏行业经常会在特定活动节点，如周年庆、新版本上线或节假日期间，面临瞬时并发流量的几何级增长。
  - 在短短一分钟内，连接数可能暴增数倍。这种突发性的流量洪峰，极易触发“雪崩”效应，特别是对于像游戏匹配这样的全局核心模块，大量的重试请求可能导致整个服务瘫痪。

综上所述，游戏行业的高可用挑战绝非线性叠加，而是多维交织，需要 SRE 团队在架构设计、容量规划、弹性伸缩、故障预防与快速恢复等方面投入巨大精力，以确保在亿级设备和千万用户规模下，为玩家提供无缝、流畅且稳定的游戏体验。

业务数量多，异构性强

腾讯游戏目前运营着超过 300 款游戏，从日活跃用户数达数千万乃至过亿的《王者荣耀》这类 MOBA 游戏，到《和平精英》等战术竞技品类，再到 RPG、卡牌、休闲等多元游戏类型，业务线横跨多个领域。

这种业务的异构性，使得每一款游戏都拥有独特的技术架构和业务逻辑。就像是指挥一支由步兵、骑兵、弓箭手组成的混合军团，每个兵种都有其特定的作战方式和保障需求。面对如此多样化的场景，我们无法奢望通过一套“银弹”式的通用解决方案来解决所有问题。

因此，在腾讯游戏的高可用性建设过程中，这种固有的异构性显著提升了整体的复杂性，这对 SRE 团队而言是巨大的挑战。我们必须寻求创新和定制化的策略，去应对不同业务深层且特有的保障需求。

低承载能力

游戏服务器的单机承载能力仅为普通互联网应用的 5%-10%，这源于：

复杂的游戏逻辑： 游戏服务器需要处理复杂的战斗逻辑、实时状态同步和物理计算，这些高密度计算任务和毫秒级的响应要求，使得单机承载能力远低于传统互联网应用。这带来了三个挑战：精确验证业务容量、确保故障场景下的快速扩容能力、构建高效的弹性扩容机制。

大规模扩容要求： 以"和平精英"、"三角洲行动"等大型游戏为例，活动高峰期单个业务可能需要一次性扩容数十万核心资源。这对扩容的规模、速度和架构弹性提出了较高要求——系统需要能够在短时间内完成大规模水平扩展，以应对激增的玩家流量。

容量规划的困境： 游戏业务快速迭代、持续升级，玩家行为模式复杂多变，导致资源需求难以精确预测。这使运维团队陷入两难：资源预估过高造成浪费，预估不足则可能引发服务中断。如何平衡资源利用率与服务稳定性，是游戏运维的核心挑战。

高实时性要求

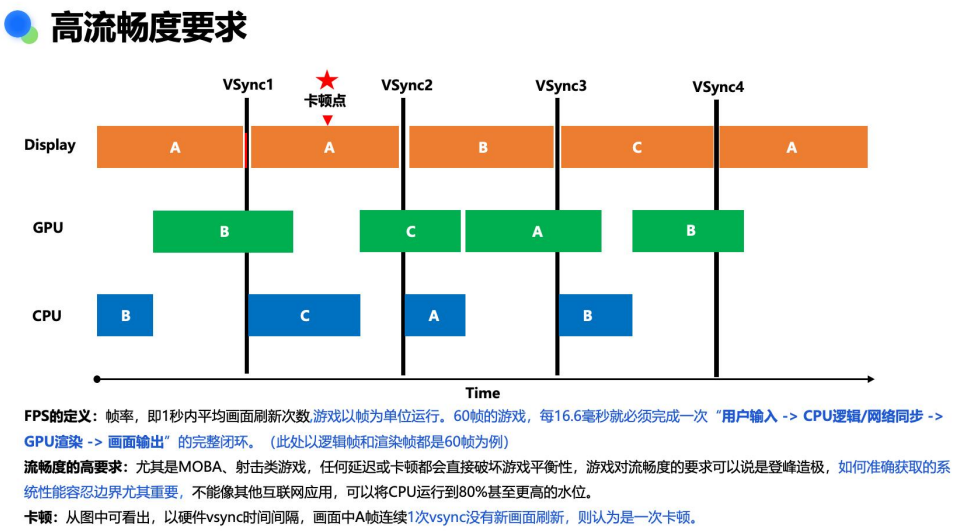
- **总体概述：** 诸如 MOBA 和 FPS 这类竞技游戏，对网络延迟的容忍度极低，操作延迟直接关联到游戏胜负与用户体验留存。因此，SRE 需思考的核心问题是：如何精准界定业务网络延迟的容忍边界，并利用分布式部署架构，实现全国乃至全球玩家在同一个服务下流畅竞技的体验。
- **极限敏感阈值：** 竞技类游戏对服务器响应延迟的门槛极为苛刻，要求延迟必须控制在 **20ms** 以下。与之对比，即使是休闲游戏，其延迟容忍阈值也仅为 50ms 左右；而电商交易可容忍 100ms 的延迟；网页浏览的延迟则可达 200ms。
- **体验直接关联：** 延迟的不稳定性会直接导致玩家操作失误和误判，从而产生严重的挫败感，这不仅显著影响用户留存率，甚至可能波及付费意愿。值得注意的是，即便系统内部如服务器 CPU 利用率超过 60%，也足以引发用户投诉，这从侧面反映了系统内部资源消耗对实时性与用户满意度的间接影响。
- **雪崩风险：** 超时故障可能引发连锁反应，触发"雪崩效应"。大量玩家因断线而同时尝试重连，会加剧服务端的压力，可能导致系统崩溃。在毫秒级的延迟波动面前，当前监控工具在预测和应对方面存在一定局限性。

幻灯片右侧的条形图“不同业务场景延迟敏感度对比”直观地展现了各类业务对延迟容忍度的差异：

- **网页浏览：** 延迟 ≤ 200ms
- **电商交易：** 延迟 ≤ 100ms

- **休闲游戏：** 延迟 ≤ 50ms
- **MOBA/FPS：** 延迟 ≤ 20ms

高流畅度要求



高流畅度要求，是游戏及其他交互式应用的核心性能指标，远超电影等传统内容形式。电影通常以每秒 24 帧（FPS）呈现，而现代游戏，如我们常见的 60 帧或 120 帧游戏，对画面的刷新速度有着极致的追求。

为了实现 60 帧的流畅体验，系统必须在每 16.6 毫秒内完成一个完整的闭环周期，这个周期包含了从用户输入、CPU 逻辑处理与网络同步、GPU 画面渲染，直至最终画面输出的全过程。这意味着在短短十几毫秒内，所有环节都需高效协同，不容许任何微小的延迟。

在 MOBA 或射击类游戏中，任何帧的处理卡顿或延迟，都会直接影响游戏的平衡性和玩家体验。与多数互联网应用相比，游戏对流畅度的要求更高。因此，准确识别系统性能的容忍边界很重要，不能像对待其他互联网应用那样，容忍 CPU 长时间运行在 80%甚至更高的利用率，因为这可能导致不可接受的延迟。

卡顿的直观定义与帧生产管线解析：

我们通过一个时序图来清晰地阐释卡顿的发生机理，以及 CPU、GPU 和显示器（Display）之间的帧处理流程。

- **垂直同步（VSync）机制：**
    - 图中的 VSync1、VSync2 等标记代表显示器的垂直同步信号，指示显示器刷新画面的时间点。

- 在理想情况下，一个新的画面帧应该在每个 VSync 信号到来时准备就绪并被显示。

- **帧处理流程：**
  - **CPU (逻辑处理)：** 负责处理游戏逻辑、用户输入、网络同步等，图中以蓝色条表示。例如，在 VSync1 和 VSync2 之间，CPU 首先处理帧 C，然后处理帧 A。
  - **GPU (渲染)：** 负责根据 CPU 的指令进行画面渲染，图中以绿色条表示。例如，在 VSync2 和 VSync3 之间，GPU 渲染帧 A。
  - **Display (显示)：** 负责将渲染完成的帧呈现给用户，图中以橙色条表示。
- **卡顿点的识别：**
  - 图例展示了一个典型的卡顿场景：在 VSync1 时刻，红色的五角星明确标识了卡顿点。
  - 正常情况下，VSync1 到来时，显示器应呈现一个新的画面帧。然而，在卡顿发生时，即使经过了一个完整的 VSync 周期，显示器仍然在显示旧的画面（如图中的帧 A）。
  - 卡顿的根本原因可能是多方面的，包括 CPU 未能及时完成逻辑处理、GPU 渲染队列阻塞，或是后端服务响应缓慢等。

性能边界的探索与流量洪峰应对：

为了确保高流畅度，我们对系统性能的容忍边界有着极高的要求。通过混沌工程等方法，我们不仅仅是为了发现系统的故障点，更重要的是精确地探测这个性能边界究竟在哪里。这意味着我们希望找到在保证系统不卡顿的前提下，又能实现资源利用率最优的状态。这对于应对瞬时流量洪峰尤其关键，比如游戏在周年庆或限时活动期间可能遇到的爆发性用户访问。

高频流量洪峰挑战

在周年庆、新版本上线、限时活动等特殊时期，系统登录量可在 1 分钟内骤增 8-10 倍，极易引发崩溃。根据"木桶效应"原理，找出并补齐系统性能短板——尤其是关键模块的承载瓶颈，是维系稳定的关键。

核心挑战场景：

- **整点突发登录：** 活动开启瞬间，登录请求量 1 分钟内飙升至日常 10 倍以上，对弹性伸缩能力构成巨大挑战。
- **节日活动冲击：** 周年庆、节假日期间，同时在线用户数达到日常 2 倍，系统并发处理能力面临极限考验。
- **新玩法局部过载：** 新内容上线后，玩家集中涌向特定场景，局部压力迅速超出设计承载上限。
- **雪崩效应风险：** 流量峰值时，匹配系统和跨服模块因不当重试机制触发"雪崩"，可能导致整个区服瘫痪。

典型流量峰值数据：

- ① 节假日期间 PCU 提升 1 倍
- ② 新版本发布 1 分钟内登录暴增
- ③ 断线重连产生二次峰值冲击
- ④ 周年庆登录达到平日 8-10 倍

流量波动趋势： 右下方曲线图直观展现了流量随时间的剧烈波动，呈现明显的锯齿状起伏。峰值处的橙色标注线代表特定活动时间点，流量飙升后虽有回落但仍高于基线水平。这种高频、瞬时暴增的特性，意味着单个大型活动可能需要额外准备数十万台服务器资源。

应对高频流量洪峰，需要有针对性的策略和资源储备。

游戏高可用性建设的解法

游戏高可用性建设的核心，在于推动 SRE 从被动应对转向主动掌控系统稳定性。

当前面临的挑战： SRE 团队应对系统故障时，常陷入"匆匆忙忙、连滚带爬"的被动状态，与理想的"从容应对、游刃有余"形成鲜明对比。

核心解决方案：全链路压测 + 混沌工程

- **全链路压测：** 模拟真实流量和负载，提前发现系统瓶颈与潜在风险，验证系统承压能力。
- **混沌工程：** 主动引入受控故障，验证系统在异常场景下的韧性和恢复能力，提升鲁棒性。

工作模式的演进： 通过这两大方法的结合，实现 SRE 工作模式的根本转变： * 从"被动应对故障"到"主动拥抱故障" * 从事后补救到事前预防 * 从手忙脚乱到从容管理

这不仅提升了系统可靠性，也极大改善了 SRE 团队的工作体验和效率。

（二）体系设计及关键流程体系设计

腾讯游戏高可用保障体系

02

腾讯游戏高可用保障体系

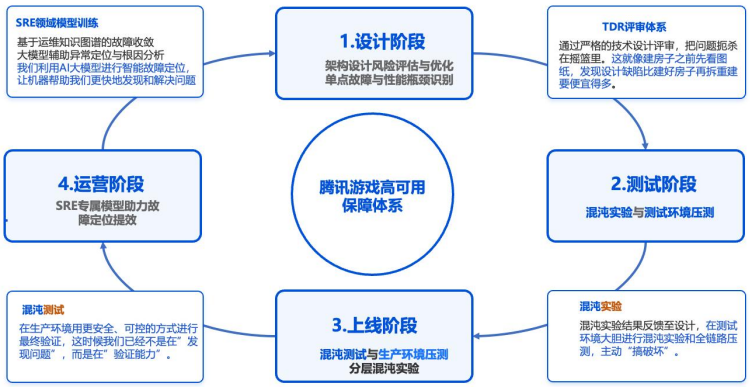
高可用保障体系的核心建设方向：

- **韧性架构设计：**如何从系统设计初期就融入弹性伸缩、异地多活、容灾备份等机制，确保服务在局部故障下仍能持续运行。
- **全链路可观测性：**构建统一的监控、日志、追踪系统，实现对业务和基础设施的全面洞察，从而快速发现潜在问题并定位故障根源。
- **自动化运维能力：**通过自动化部署、弹性伸缩、故障自愈等技术，减少人工干预，提高运维效率和故障恢复速度。
- **风险管理与混沌工程：**定期进行故障演练和混沌工程实践，模拟各种异常场景，验证系统的鲁棒性和故障恢复能力。
- **应急响应机制：**建立高效的应急响应流程、预案管理和团队协作机制，确保在故障发生时能够迅速止血、恢复服务。

这套体系的最终目标是保障腾讯游戏在亿级用户规模下，服务的持续稳定运行，为玩家提供流畅的游戏体验，并最大限度地降低业务中断带来的损失。

从设计到运营、体系化建设游戏高可用体系，贯穿游戏全生命周期。

从设计到运营，体系化建设游戏高可用体系，贯穿游戏全生命周期



四个阶段形成了一个完整的闭环，每个阶段都有明确的目标和技术手段，构建了完整的高可用体系。

从设计到运营，我们构建了一个贯穿游戏全生命周期的高可用体系，力求将稳定性融入产品基因，而非事后补救。这一体系以“腾讯游戏高可用保障体系”为核心，通过四个紧密相连的阶段形成一个持续优化的闭环。

1. 设计阶段：将问题扼杀在摇篮里 在系统开发之前，我们便通过严谨的 TDR（技术设计评审）体系，对架构方案进行深入的风险评估与优化。此阶段旨在识别并解决潜在的单点故障和性能瓶颈，确保系统在设计之初就具备高可用基石。这就像盖房子前仔细审查图纸，发现设计缺陷的成本远低于房子建好后再拆改。这是一种“左移”的策略，将高可用性考量前置，防患于未然。

2. 测试阶段：在非生产环境主动“搞破坏” 进入开发和测试阶段，我们大胆地引入混沌实验和全链路压测，在非生产环境中主动模拟各种故障场景。通过“混沌实验”，我们旨在发现系统在异常情况下的薄弱环节并验证其容错能力，并将实验结果及时反馈至设计团队进行优化。同时，全链路压测则确保系统在预期的极端负载下依然能够稳定运行。

3. 上线阶段：生产环境下的最终验证 系统即将上线时，我们会在生产环境中进行更加安全、可控的混沌测试。与测试阶段不同，此阶段的目的并非发现新问题，而是在真实环境下对系统的高可用设计与容错机制进行最终验证。我们通过分层级的混沌实验，逐步提升测试的深度和广度，以确保系统能够抵御真实世界中可能发生的各类故障，验证其真正的韧性。

4. 运营阶段：AI 助力故障定位提效 在游戏正式运营后，我们积极拥抱 AI 技术，通过 SRE 专属模型来提升故障定位与解决效率。我们利用成熟的 SRE 领域模型训练，基于运维知识图谱进行故障收敛，并通过大模型辅助异常定位和根因分析。这种智能化的手段能

显著减少 MTTR（平均恢复时间），确保当故障发生时，我们能以最快的速度发现、定位并解决问题，从而保障游戏的持续稳定运行。

这四个阶段，从早期的预防到后期的响应，每个环节都有其明确的目标和技术手段，共同构建了一个贯穿游戏全生命周期、并持续迭代的高可用体系。

设计阶段：腾讯游戏 TDR 技术设计评审

03

设计阶段：腾讯游戏TDR技术设计评审

高可用能力左移的关键机制

在设计阶段，我们实施了腾讯游戏 TDR（Tencent Data Report 或特定技术设计评审）的技术评审，这正是实现高可用能力“左移”（Shift Left）的关键机制。

鉴于腾讯游戏产品数量庞大且持续增长，同时各游戏的技术水平参差不齐，我们迫切需要制定一套统一的技术标准。这套标准旨在确保所有游戏在设计之初就符合高可用左移的能力要求。

将高可用性考量提前到设计阶段，就像在建筑蓝图阶段发现问题并进行修改，其成本远低于建筑完成后再进行拆除重建。通过在设计阶段介入，能够确保将可靠性保障融入系统 DNA，避免后期高昂的修复成本与潜在的服务中断风险：

- **设计阶段介入：**在正式开发前，通过技术评审介入方案和架构设计。
    - **优点：**能够及早发现并解决潜在的可靠性缺陷，从系统根基处强化高可用能力。
    - 在"图纸阶段"调整设计，修改成本较低。
- **后期修复：**如果高可用问题在系统上线后才暴露。

- **困境：**此时需要投入较多资源进行改造甚至重建，对业务连续性造成较大影响。
- 在系统建设完成后再修改，代价较高。

通过实施这种前置的技术评审机制，我们能够高效、经济地提升整体系统的可靠性水平，这是我们在复杂游戏生态中确保业务连续性的核心策略。

TRD 评审关键要素

TDR 评审基于多年在架构优化、可扩展性、容灾设计、容量评估及异常处理机制等方面的经验，总结出 200 余项评审子项。目标是在系统设计的早期阶段，即"左移"地发现并规避潜在的单点故障风险、并发瓶颈与性能隐患，从而构建系统的容灾能力。

为此，TDR 评审聚焦于五大关键要素：

- **架构说明**
  - 梳理系统整体结构与层次
  - 明确核心组件及其交互关系
  - 剖析数据流与控制流
- **可扩展性**
  - 评估系统水平/垂直扩展能力
  - 识别扩展瓶颈与应对策略
  - 审视扩缩容成本与复杂性
- **容灾设计**

    - 检验多活部署与异地容灾方案
    - 验证数据备份与恢复机制
    - 考量故障隔离与自动转移能力
- **容量评估**
    - 预估系统吞吐、并发与存储需求
    - 分析资源水位与容量规划
    - 评估容量动态调整策略
- **异常处理机制**
    - 定义错误类型与错误码规范
    - 设计重试、超时与降级策略
    - 建立告警、日志与可观测性体系

通过这些评审，确保在删档测试等关键阶段，系统能够经受真实场景的考验，满足韧性和稳定性的要求。

TRD 评审流程和标准

在系统设计和开发过程中，通过"TDR 评审流程和标准"确保系统的高可用性和稳定性。包含了一套评审机制，并辅以"红黄绿灯"标准来量化风险等级。

评审流程

评审工作分为以下四个步骤：

- **1. 需求评审阶段提交 TDR 文档：** 在项目初期，开发团队需提交一份详细的技术设计文档，这份文档不仅涵盖了系统的宏观架构、核心业务流程，还特别强调了容灾设计，力求在源头保障系统健壮性。
- **2. 多领域专家联合评审：** 汇集来自 SRE、DBA、后台开发、客户端开发、测试等多个领域的专家，共同对设计文档进行会审。这些专家结合实战经验，从多维度视角挖掘潜在风险，确保评审的全面性。针对核心业务或重要项目（例如"和平精英"这类产品），会邀请最高级别的专家参与评审。
- **3. 问题发现与修复：** 评审的核心在于识别问题。检查设计中可能存在的单点故障、容量瓶颈、以及异常处理流程中的潜在缺陷，确保这些问题被提前发现，并推动团队在设计层面进行优化。
- **4. 设计优化与再评审：** 根据评审中提出的建议，开发团队对原有设计进行迭代优化。关注系统容灾能力的强化，并在优化完成后，再次进行评审，确保建议已有效落地，从而形成一个持续改进的闭环。

"红黄绿灯"标准衡量风险等级

为了直观地衡量评审中识别出的风险，引入了"红黄绿灯"的标准化风险等级系统：

- **红灯：严重风险**
    - **描述：** 这代表了不可接受的严重级别风险，例如存在单点故障，或可能直接导致业务中断的重大缺陷。
    - **处理机制：** 如果存在被标记为“红灯”的问题，项目团队必须优先修复，否则我们将在品质层面强制要求产品不能上线，坚决杜绝带着严重风险运营。
- **黄灯：中低风险**
    - **描述：** 这类风险级别相对较低，不构成致命威胁，例如系统承载能力可能偏低，或是存在优化空间。
    - **处理机制：** 对于“黄灯”问题，我们强烈建议进行优化，虽然在特殊情况下，即使不修复也可以上线（“如果硬要上，那也是没关系的”），但我们期望团队能将其纳入后续迭代计划。
- **绿灯：通过**
    - **描述：** 表示当前设计没有发现明显风险，符合上线标准。

- **处理机制：** 标记为"绿灯"的项目，意味着其技术设计已满足高可用和稳定性要求，可以推进后续开发和发布流程。

TRD 评审收益

技术设计评审（TDR）的收益，倡导在系统设计阶段即“左移”融入高可用性建设，以此践行“预防胜于治疗”的 SRE 核心理念。

TDR 评审的实施，从技术和业务层面带来了一些优化：

技术收益：

- **提前发现系统风险：** 在设计阶段即识别并规避单点故障、并发瓶颈及性能隐患，确保系统上线后能稳定应对高负载挑战。
- **强化容灾能力：** 通过多活架构设计提升关键模块的抗风险能力，构建快速故障切换机制，并保障数据在任何故障情境下的最终一致性。
- **提升系统可扩展性：** 验证系统水平扩容能力，设计弹性伸缩机制以适应负载变化，并精准化容量评估，实现资源的高效利用。

业务收益：

- **保障上线质量：** 通过前置拦截严重风险，阻止带有高风险的设计上线，从而降低线上重大故障的发生率。
- **降低运维成本：** 减少故障处理时间与人力投入，提升运维团队效率，使团队能将更多精力聚焦于预防性维护和系统优化。

- **提升用户体验：**确保服务持续高可用性，特别是在游戏等高要求场景下，保障流畅稳定的体验，最终提升用户满意度。

总结来说，通过 TDR 评审在设计期提前介入高可用性建设，能够将预防作为核心策略，这不仅在技术层面提升了系统的稳定性与弹性，也在业务层面带来了质量保障、成本降低和用户体验提升的价值。

基于这种"左移"的实践经验，将"预防胜于治疗"的理念贯穿始终，可以在较大的业务体量下，降低整体风险。接下来的测试阶段，将是混沌工程发挥作用的时机，以进一步验证和提升系统的韧性。

测试阶段：混沌实验与全链路压测

04

测试阶段： 混沌实验和测试环境 全链路压测

一个从未经历过故障演练的系统 VS 一个经历过各种故障演练并能快速恢复的系统

测试阶段聚焦于两种高级测试方法，在专用测试环境中进行，避免对生产环境造成负面影响。

- **核心测试方法：**
    - **混沌实验（Chaos Engineering）：**这并非简单的故障注入，而是一种通过有计划地、主动地在系统内部制造混乱和故障，来探究系统在真实世界中的弹性表现，从而识别潜在弱点和未知行为的系统性方法。
    - **全链路压测（Full-Link Stress Testing）：**我们的目标是模拟真实用户流量，对整个系统调用链进行极限压力测试。这不仅是为了评估系统在峰值负载下的性能阈值，更是为了发现瓶颈、验证组件间的协同效应，并确保系统在高并发场景下的稳定性。

目标是构建一个能够应对各种挑战的弹性系统。一个从未经历过故障演练的系统，当面对突如其来的故障时，往往会手足无措，甚至可能导致业务停摆。相比之下，经历过各种故障演练并能快速恢复的系统表现会更好。因此，有必要在系统上线运行后，持续对其进行故障模拟，即便有时会触发预期的故障恢复流程，其带来的经验和提升也超过未进行过演练的系统。

回溯混沌工程的发展历程，自 2008 年萌芽至今，它早已超越了单一工具的范畴，升华为一套成熟的系统级方法论。在当下，这套方法论已深度融入我们游戏高可用体系的构建与实践之中。通过持续的混沌实验和全链路压测，我们力求将系统从一个对未知故障可能崩溃的状态，转变成为一个能够快速响应、自我修复并从故障中弹性恢复的健壮体系。

从 Netflix 到腾讯游戏：混沌工程的演进与适配

- 2008 年：Netflix 系统故障
- 这次长达三天的数据库故障，成为 Netflix 系统弹性提升的催化剂，促使公司深刻反思并寻求解决之道。
- 2010 年：Chaos Monkey 诞生
- Netflix 率先开发了混沌工程的标志性工具 Chaos Monkey，通过随机终止生产环境实例，主动探寻系统脆弱点，推动工程师构建更具韧性的系统。
- 2012 年：Simian Army 开源
- Netflix 将混沌工程工具集 Simian Army 开源，其中包含一系列“混沌猴子”（如 Chaos Monkey），自此混沌工程理念开始走向开放社区，吸引更多开发者参与。
- 2015 年：混沌工程原则发布
- 混沌工程从工具层面升华至方法论层面，其核心理念和实践原则被正式确立，为行业提供了统一的指导框架。
- 2019 年：国内实践兴起
- 混沌工程在国内大型互联网公司中实现全面落地，成为构建高可用系统不可或缺的标准实践。
- 2022 年至今：混沌工程与 AI 结合探索
- 混沌工程正积极探索与人工智能的深度融合，预示着其未来发展将迈向更智能、更自动化的方向。

总结而言，混沌工程自 2008 年 Netflix 故障事件之后，已从单一工具演进为一套系统的工程方法论，实现了从被动应对故障到主动发现潜在缺陷的范式转变。如今，它已成为保障全球大型分布式系统稳定性的必备实践，并在游戏高可用架构中发挥着举足轻重的作用。

理念基础及定义

混沌工程的理念基础与核心原则

混沌工程是一门在分布式系统上进行实验的学科，通过主动探索的方式，提前发现系统缺陷，进而增强系统的稳定性和可靠性。其核心理念是在风险可控的前提下，进行有序的故障注入。

在理念基础与定义方面，着重强调以下几点：

- **混沌工程的定义**：在分布式系统上进行实验，旨在建立对生产环境中各种故障的抵御信心。通过主动注入故障，探索系统边界，以增强系统的稳定性和可靠性。
- **核心目标**：通过主动注入如网络延迟、服务不可用、资源耗尽等各类故障，观察系统反应与自愈能力，从而建立对系统韧性的信心，确保系统能够抵御生产环境中的故障考验。
- **理念转变**：将传统的"避免故障发生"转变为"拥抱故障，构建韧性系统"。这是从被动等待问题出现再解决，向主动防御与验证系统容错能力的转变。

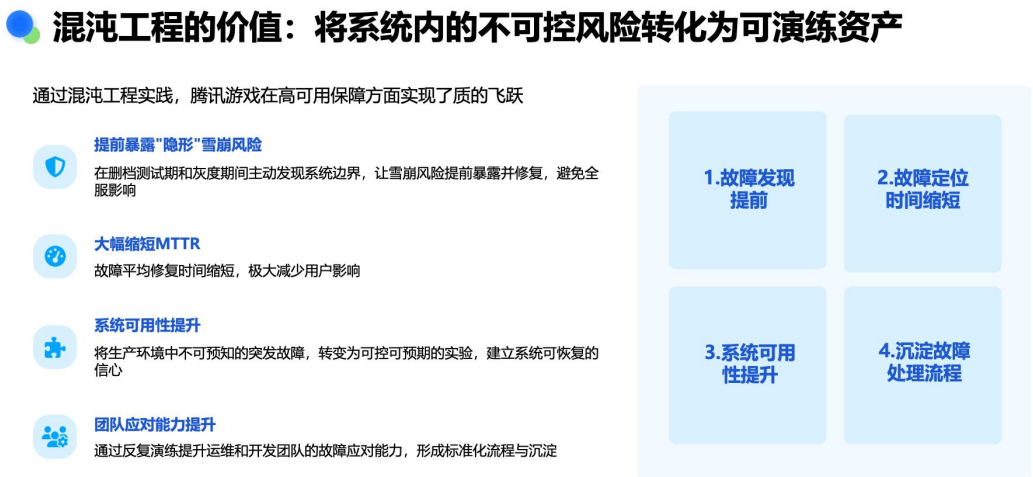
- **游戏场景适配性**：考虑到游戏业务对可用性的要求较高，混沌工程能够在大型在线游戏这类复杂分布式系统中，提前发现潜在的脆弱点，从而保障服务稳定性。

遵循混沌工程的实践原则至关重要，这些原则是混沌工程理论的基石，业界普遍采纳：

- **假设建立**：根据系统的正常行为建立明确的假设，随后通过实验来验证或证伪这些假设，以理解系统行为。
- **真实事件**：模拟真实的故障事件，而非仅仅关注假设性的极端场景，确保实验结果具有实际参考价值。
- **生产环境实验**：在真实生产环境中进行实验，以确保实验结果的真实性和有效性，从而获得对系统韧性的评估。
- **自动化实验持续运行**：将实验流程自动化，实现实验的持续运行，而非一次性验证。实验将伴随系统演进而不断迭代和进行。
- **可控爆炸半径**：实验的影响范围需严格可控，确保不会对生产环境造成较大影响或业务中断。

实践主要基于这些通用原则，并针对游戏场景进行了适配，以确保混沌工程在游戏业务中有效应用。

混沌工程的价值：将系统内不可控的风险转化为可演练的资产



混沌工程实践的显著成果

混沌工程的推行，在高可用保障方面带来了一些提升，具体体现在：

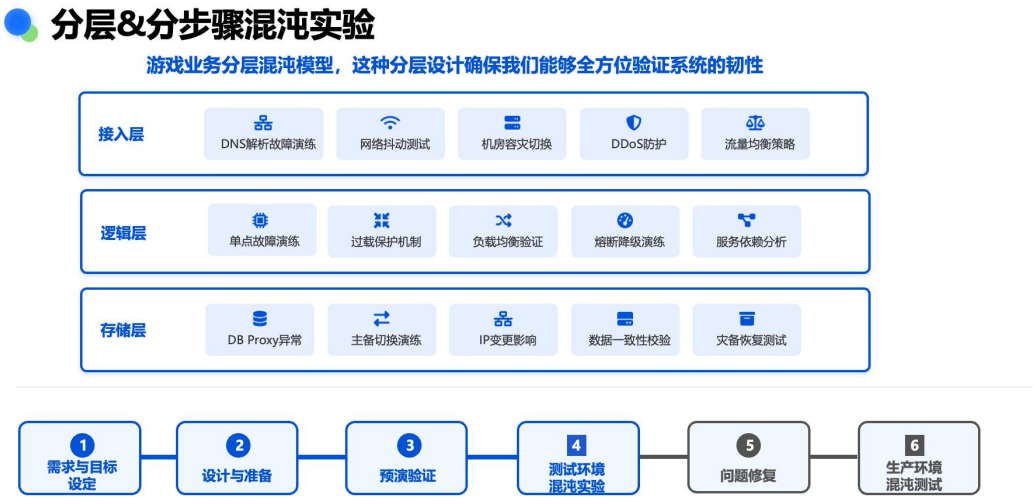
- **提前暴露雪崩风险：**
    - 在删档测试和灰度发布阶段，主动探测并识别系统边界，让潜在的雪崩风险在早期阶段被发现并修复。
    - 这有效避免了这些风险在生产环境中爆发，对服务造成大范围影响。
- **缩短 MTTR (Mean Time To Recovery)：**
    - 通过较高频次的故障注入和演练，团队对各类故障处理更加熟悉，缩短了故障的平均修复时间。
    - 这意味着即使发生故障，也能较快恢复，从而降低对用户的影响。
- **系统可用性提升：**
    - 混沌工程将生产环境中不可预测的突发故障，转化为可控、可预期的实验场景。
    - 通过反复验证系统的韧性与恢复能力，增强了团队对系统自我修复和高可用性的信心。
- **团队应对能力提升：**
    - 持续的故障演练提升了运维和开发团队在故障面前的应变能力，促进了标准化故障处理流程的形成和经验沉淀。
    - 这种实践强化了团队的实战经验，为未来可能出现的各类问题做好了准备。

混沌工程对故障处理流程的提升

通过实践混沌工程，故障处理的效率和质量得到了一定优化，主要体现在以下四个方面：

- **故障发现提前：** 混沌工程主动暴露系统脆弱点，使得故障能够被更早地识别。
- **故障定位时间缩短：** 较频繁的故障注入实践，让团队对各类故障模式及其定位方法更加熟悉，加快了故障定位速度。
- **系统可用性提升：** 持续的演练和修复，不断增强了系统的韧性，从而提升了整体可用性。
- **沉淀故障处理流程：** 实践中积累的经验和方法被沉淀下来，形成可供 AI 等工具利用的标准流程，进一步支持自动化运维。

分层与分步骤混沌实验



首先，构建了游戏业务的分层混沌模型，这种设计理念的核心在于通过层级划分，较全面地验证系统的韧性。

- **接入层：** 识别影响用户访问系统的故障场景，确保用户请求的顺畅与稳定。
    - DNS 解析故障演练，模拟域名解析异常对用户访问的影响。
    - 网络抖动测试，评估系统在网络不稳定情况下的表现。
    - 机房容灾切换，验证在主用机房故障时，系统能否无缝切换到备用机房。
    - DDoS 防护，测试系统抵御分布式拒绝服务攻击的能力。
    - 流量均衡策略，验证负载均衡机制在不同负载下的有效性。
- **逻辑层：** 专注于核心业务逻辑处理的健壮性，保障业务功能的高可用。
    - 单点故障演练，模拟关键组件失效对业务的影响。
    - 过载保护机制，测试系统在流量洪峰下的自我保护能力。
    - 负载均衡验证，确保请求能够均匀地分发到各个服务实例。
    - 熔断降级演练，验证系统在依赖服务异常时能否自动熔断或降级，避免雪崩效应。
    - 服务依赖分析，深入理解服务间的依赖关系，识别潜在的级联风险。
- **存储层：** 确保后端数据存储的可靠性和一致性，它是业务连续性的基石。
    - DB Proxy 异常，模拟数据库代理故障对数据访问的影响。

- 主备切换演练，验证数据库在主备模式下的切换流程和数据一致性。
- IP 变更影响，评估 IP 地址变化对数据库连接和业务的影响。
- 数据一致性校验，确保在各种故障场景下，数据依然保持一致。
- 灾备恢复测试，全面验证数据备份和灾难恢复的能力。

其次，混沌实验的实施遵循六步骤流程，以确保实验的安全性和有效性。需要强调的是，在测试环境中进行的是"混沌实验"，而只有在经过充分验证和修复后，才能在生产环境中进行"混沌测试"。

1. **需求与目标设定：** 清晰定义混沌实验的目的、范围和预期验证点。
2. **设计与准备：** 规划详细的实验方案，包括故障注入方式、观测指标和回滚计划。
3. **预演验证：** 在受控的沙箱环境中小范围运行实验，验证故障注入机制的正确性与安全性。
4. **测试环境混沌实验：** 在非生产环境中全面执行混沌实验，暴露系统弱点并进行充分的故障模拟。
5. **问题修复：** 根据测试环境中发现的问题，进行根因分析并实施相应的修复措施。
6. **生产环境混沌测试：** 在修复完成后，经过严谨评估，在生产环境中进行有限、受控的混沌测试，最终验证系统的韧性。

例如，在接入层，我们可能会模拟像近期 AWS DNS 解析异常这类故障场景，通过模拟 DNS 解析故障来验证系统的韧性。

接入层混沌实验设计

接入层混沌实验概述作为游戏服务流量的入口，接入层的故障可能引发较大的影响。因此，实验目标集中于验证负载均衡系统与过载保护机制的有效性，并探讨在接入层故障情境下，系统如何展现其容错能力与降级机制。

三类实验设计：

- **DNS 解析故障实验**
    - **典型场景模拟：**
        - DNS 解析超时。
        - 域名被劫持。
    - **实验方法：**通过修改本地 hosts 文件、模拟 DNS 劫持，或注入本地 DNS 服务故障等方式进行。
    - **成功标准（兜底机制）：**验证系统能否成功启用 **httpDNS** 作为备份解析链路，或在极端情况下，利用客户端预埋的 **IP 地址**实现服务兜底。此实验将着重测试业务在域名被解析至广告页或无效地址等异常情形下的正常响应能力。
- **网络抖动实验设计**
    - **模拟延迟：**在数据包传输中注入 20-200 毫秒的随机网络延迟。
    - **丢包模拟：**注入 1%-10% 的随机数据包丢失率。
    - **验证目标：**
        - 评估在网络延迟环境下，玩家的游戏体验流畅度。
        - 验证断线重连功能的成功率。
    - **特别说明：**考虑到游戏对延迟的敏感性，此类实验的验证往往需要专业的测试人员协同，结合主观体感判断，以确定不同类型游戏对网络延迟和丢包的容忍阈值。
- **机房不可用（关机模拟）实验设计**
    - **模拟场景：**模拟整个机房宕机，具体操作为批量终止某个模块服务器上的进程，或模拟批量关机。需明确，在大型云基础设施中，直接关闭整个机房不具备实施性，因此采取“关机模拟”方式。
    - **验证目标一：**跨机房流量调度的有效性与自动化扩容机制的响应速度。

- **验证目标二（灾备切换）：**检验系统能否在主用机房异常时，自动且快速地将流量切换至其他正常运作的机房。

在右侧，我们将上述实验设计具体化为可操作的演练剧本，包含场景、工具、指标和预期：

- **DNS 解析故障演练**
  - **演练场景：**分别模拟 DNS 服务异常和域名解析被劫持两种情况。
  - **预期评估：**
    - 验证 **httpDNS** 能否成功启用，作为第一道备份解析链路。
    - 进一步验证在 LocalDNS 和 httpDNS 均异常的情况下，系统能否有效启用 **预埋 IP** 机制进行最终兜底。
- **网络抖动演练**
  - **核心工具：**利用 **UE Net PktLag** 工具注入网络延迟，并使用 **Net PktLoss** 工具模拟丢包。
  - **关键指标：**
    - 掉线率应低于 10%。
    - 重连成功率需高于 95%。
    - 最终通过人工体验确认游戏流畅度符合预期。
- **机房不可用演练**
  - **演练剧本：**
    - 模拟单个机房出现异常的场景。
    - 模拟接入层服务器宕机。
  - **预期与验证：**
    - 预期流量能够自动切换至备用机房。
    - 切换耗时需严格控制在 3 分钟以内。

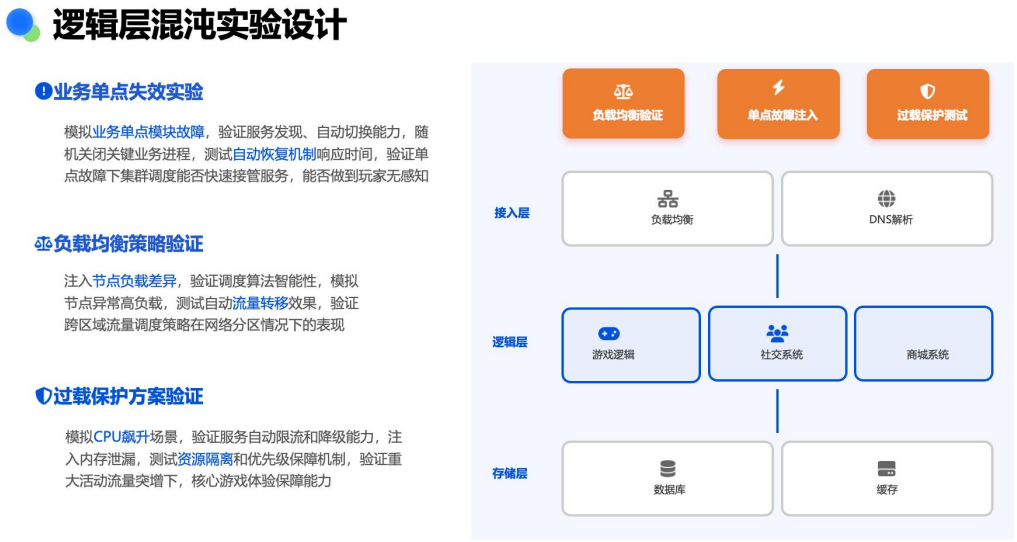
接入层-蓝鲸混沌实验案例

本页展示蓝鲸混沌平台在游戏接入层的具体应用案例，通过模拟网络故障，验证系统在异常条件下的韧性与用户体验边界。

- **混沌实验场景：** 针对游戏接入层设计，主要注入了网络延迟和丢包，此外还支持乱序等故障类型。
- **延迟注入：**
  - **操作：** 注入 200 毫秒的网络延迟。
  - **配置：** 在蓝鲸混沌平台的参数配置界面中，明确指定延迟时间为 200ms，作用对象为单个实例，注入持续时间为 300s，协议为 TCP。
  - **流程：** 通过并行网关，可与其他故障注入操作同时执行，实现复杂场景模拟。
- **丢包注入：**
  - **操作：** 注入 20% 的网络丢包率。
  - **配置：** 参数配置界面中，明确指定丢包百分比为 20%，作用对象为 eth1 网卡，可指定目标 IP、端口和协议，并配置是否强制覆盖现有规则以及注入时间 300s。
  - **流程：** 与延迟注入类似，通过并行网关执行，增强实验的并行性与组合性。
- **结果验证与用户体验：**

- **验证机制：** 利用自动化测试工具，通过多轮次的混沌实验，持续验证不同延迟和丢包率下系统的表现。
- **临界点识别：** 在实验中，我们寻找系统能够接受的性能边界。当丢包率达到一定阈值时，系统将无法正常连接，例如左下角的示例所示，游戏登录时会出现“连接服务器没有响应”的错误提示，明确表示已触及系统容忍的极限。
- **目标：** 通过这种反复测试，精准定位系统的错误容忍边界，为后续的系统优化和应急预案提供数据支撑。

逻辑集成混沌实验设计



系统“逻辑层”的混沌实验设计，这是 SRE 实践中验证系统韧性与稳定性的重要一环。核心目标是：**通过主动注入故障，验证并提升系统的容错能力和弹性伸缩性。**

幻灯片左侧列举了三大类逻辑层混沌实验，旨在模拟真实世界的各种异常场景，以检验系统的应变能力：

- **业务单点失效实验**
  - **目标与操作：** 我们会像外科医生一样，随机“关闭”关键业务进程，模拟核心业务模块的局部故障。这就像是在一个繁忙的工厂里，突然让一条重要的生产线停止工作。
  - **验证点：** 实验将聚焦于系统对故障的发现速度和自动恢复机制的响应时间。我们尤其关注：在单点故障发生时，集群调度系统能否在**玩家无**

感知的前提下，快速接管服务并实现平滑过渡。这不仅考验技术的鲁棒性，更关乎用户的体验。

- **负载均衡策略验证**
    - **目标与操作：**针对负载均衡，我们会通过创建节点异常高负载的情境，注入显著的节点负载差异。这好比在一条高速公路上，突然制造人为的拥堵。
    - **验证点：**实验将验证当前调度算法的智能性，评估其在复杂负载下的**自动流量转移效果**。我们还会进一步分析在网络分区等极端情况下，跨区域流量调度策略能否依旧表现出色，确保服务的连续性。
- **过载保护方案验证**
    - **目标与操作：**模拟系统过载，通过注入CPU飙升、内存泄漏等典型过载场景，测试系统的自我保护能力。
    - **验证点：**验证服务的**自动限流和降级能力**，确保在流量洪峰下，非核心功能能够被降级，从而保障核心功能的正常运行。同时，资源隔离和优先级保障机制的有效性也是验证重点，以确保在大规模活动流量突增时，**核心游戏体验不被打断**。

幻灯片右侧的**分层架构图**直观地展示了这些实验的应用场景和系统关注点：

在架构图顶部，三个醒目的橙色方块，分别代表着：*** 负载均衡验证**（天平图标）：对应着我们对流量分配和调度智能性的探索。*** 单点故障注入**（闪电图标）：象征着我们主动制造局部故障，以发现系统的脆弱点。*** 过载保护测试**（盾牌图标）：代表着我们对系统自愈与防御能力的检验。

这三类实验将主要作用于以下分层架构：

- **接入层 (Access Layer):** 这是用户请求进入系统的第一道关卡，包含：
    - **负载均衡:** 负责将外部请求有效地分发到后端服务。
    - **DNS 解析:** 确保用户能够正确解析到服务地址。
- **逻辑层 (Logic Layer):** 系统的“大脑”，承载核心业务逻辑，也是本次混沌实验的焦点区域，由以下模块构成：
    - **游戏逻辑:** 系统的核心玩法与交互。
    - **社交系统:** 用户间的互动与关系管理。
    - **商城系统:** 游戏内的交易与物品管理。

- **存储层 (Storage Layer):** 系统的“记忆”，负责数据的持久化与高速存取，包括：
    - **数据库:** 存储核心业务数据。
    - **缓存:** 提升数据访问速度，减轻数据库压力。

通过连接线，可以看到各层服务之间的依赖关系和数据流。在这样的架构下进行混沌实验，能够较全面、准确地评估系统在真实故障场景下的表现，提升整个系统的韧性，确保业务的连续性和用户的流畅体验。

逻辑层-蓝鲸混沌实验案例

逻辑层-蓝鲸混沌实验案例

此幻灯片展示了如何运用蓝鲸混沌工程平台，对核心业务逻辑层，特别是 **DS 战斗服务器**，进行高负载下的韧性测试与边界探索。

核心目标是：构建抗故障的能力，保障业务的连续性。

混沌实验概览

在实验过程中，我们通过蓝鲸混沌平台注入了高 CPU 利用率故障，具体操作为“逻辑层混沌-战斗服注入高负载”，其手段是“混沌工程-CPU 利用率高”，目标实例为 1 台。幻灯片左侧的“实时状态跟踪卡片”清晰显示了实验当前处于“执行中”状态，并提供了“暂停执行”和“终止”选项，确保实验的可控性。

实时监控与结果观测

实验进行时，我们通过“蓝鲸监控-CPU 负载面板”实时观测目标服务器的 CPU 使用情况。该面板展示了服务器大量 CPU 核心（如 cpu47, cpu16, cpu10, cpu19 等）的详细利用率数据，实验期间大部分 CPU 核心的使用率稳定在 **75%-85%**之间，其中部分核心甚至达到了 94.9%（cpu47）。

混沌工程的价值：量化系统边界

这次混沌实验的核心价值在于：**不仅能发现潜在问题，更重要的是能够量化系统的边界。**

- **识别性能阈值上限：**通过模拟持续的高 CPU 负载，我们能够精准判断在何种压力下，业务仍能保持可接受的用户体验。这远比传统方法更精准，传统方式难以让机器持续精确运行在 85%或 85.5%的 CPU 利用率。
- **指导容量规划：**结合后端重连数据和用户体验反馈，我们能够清晰地了解不同业务模块（例如游戏业务 A 和业务 B）分别能容忍的 CPU 利用率上限（是 85%还是 90%），这直接服务于**容量建设指导**，帮助我们更合理地规划资源，确保系统在极端压力下的稳定运行。

业务在高压下的表现

幻灯片右下方的游戏画面截图，直观地展示了在高 CPU 负载冲击下，业务层的实际运行情况。尽管服务器承受着较大的压力，用户界面可能仍能保持基本功能，但可能会出现帧率下降或卡顿等现象，这正是通过实验需要记录和评估的真实用户体验。

通过这种方式，可以理解和优化系统的行为，从而保障用户体验和服务质量。

存储层混沌实验设计

存储层混沌实验的关键设计点

1. **DB Proxy 异常模拟**
    - **注入场景**
        - 进程终止（服务宕机）
        - 高负载导致的响应迟缓
        - 内存溢出（OOM）
    - **验证目标**
        - 数据库连接池在代理故障发生时能否实现自动切换与重连。
        - 业务逻辑侧配置的连接超时及重试机制能否有效地应对此类瞬态或持续性异常，确保服务的连续性。
2. **主从切换演练**
    - **注入场景** 模拟 Primary（主）数据库节点故障，强制触发自动主从切换流程。
    - **核心关注点**
        - **数据同步延迟**：评估主从切换对业务数据一致性的影响及其容忍度。
        - **从库承载能力**：验证 Replica（从）节点在晋升为 Primary 后，能否稳定承载并处理主库级别的读写流量。

- **故障复盘**：深入分析切换过程中可能出现的数据不一致问题，并制定相应的修复策略。

3. **IP 变更与网络异常**
    - **注入场景**
        - 网络抖动：模拟链路不稳定引发的网络连接质量波动。
        - 延迟：增加数据传输的时延。
        - 丢包：模拟网络中数据包的丢失。
    - **影响分析**
        - **游戏应用影响**：分析网络质量波动对游戏客户端与服务器间交互造成的影响。
        - **数据存储操作影响**：评估网络异常如何干扰数据写入、读取等存储操作，以及系统如何应对。
4. **数据一致性保障关键点**
    - **事务隔离与补偿**：确保关键业务操作在任何故障场景下，通过合适的事务隔离级别和有效的补偿机制，仍能维持数据的高度一致性和准确性。
    - **避免长事务阻塞**：识别并优化可能导致长时间运行的事务，以避免在故障发生时，这些事务阻塞其他关键操作或使故障恢复过程复杂化。
    - **存储层故障恢复策略**：制定并定期验证在存储层发生各类故障时的详细恢复策略，确保能够快速、有效地将系统恢复到正常运行状态。
    - **实验数据回滚与清理**：建立严格的安全措施，确保混沌实验的执行不会对生产环境的真实数据造成永久性破坏，并能自动化地进行实验数据的回滚与清理。

流程图解读：混沌实验注入点与数据流向

流程图直观展示了混沌实验在存储层架构中的故障注入机制及数据流向：

- **混沌故障注入点（红色小圆点）**
    - **主从切换（图标：带左右箭头的圆形）**：直接在“Primary DB”上进行故障注入，触发主从切换。
    - **网络扰动（图标：WiFi 信号）**：模拟数据传输路径上的网络不稳定。

- **Proxy 故障注入（图标：闪电符号）**：针对“DB Proxy A”或“DB Proxy B”进行故障注入。
- **正常数据流（蓝色小圆点）**
  - 数据通常从“游戏应用层”发出，通过“DB Proxy A”或“DB Proxy B”，最终到达“Primary DB”进行操作。在 Replica DB 1 和 Replica DB 2 的图标中也带有数据流，表示它们会持续接收到 Primary DB 同步的数据。

总而言之，通过这些有针对性的混沌实验，不仅能够发现存储层潜在的薄弱环节，还能验证并强化其在面临真实世界复杂故障挑战时的应变能力，从而提升系统的整体韧性与可靠性。

存储层-蓝鲸混沌实验实践

混沌实验流程：故障注入与 DBHA 切换 * 实验概览：此次混沌实验模拟了存储层故障，旨在验证 DBHA 的自动切换能力。实验开始于 2025–10–11 09:30:01+0800，并在 45s 后结束。 * **故障注入方式：**通过 iptables 规则隔离 Redis Master 节点，模拟其不可用，从而触发 DBHA 的容灾演练。 * **切换监控：**系统持续监测 DBHA 的切换状态，确保自动故障转移按预期进行。

切换过程中的告警与预期表现 * **告警检测：**在 DBHA 切换期间，系统捕获到“前端数据存档失败”的短暂告警，表现为 db–error 告警 2 次，并记录了详细的错误日志，如 failed to save 和 result=–103 等关键信息。 * **预期实现：**尽管出现了后端错误，这正是混沌实验

预期中的表现。告警的存在表明监控系统运行正常，并及时捕捉到了切换期间的短暂数据写入异常。 * **DBHA 切换详情：**日志信息清晰地展示了 DBHA 从 switch_start_time: 2025-10-11T01:20:09Z 到 switch_finished_time: 2025-10-11T01:30:09Z 的完整过程，包含了调度、状态检查和服务查询等步骤。

业务正常无感知

- **前端业务无影响：**最核心的验证是，尽管后端存储层经历了故障注入和 DBHA 切换，前端业务逻辑（以游戏界面为例）仍然能够正常运行。这意味着用户对后端切换过程中的短暂写入失败无感知。
- **前端容忍机制：**业务层设计了容忍机制，如初次写入失败后，在 10 秒或 30 秒后重试。这样，如果 DBHA 能在短时间（例如十几秒内）完成切换，用户体验便不会受到影响。
- **应用层告警：**应用层仍然会记录在切换瞬间数据写入失败的告警，但这些告警在前端重试机制和 DBHA 快速切换的协同下，不会转化为用户可见的业务中断。

为什么混沌工程需要结合全链路压测

混沌工程作为评估系统韧性的工具，其有效性有时会因测试环境的局限性受到影响。传统上，混沌实验往往在预发布或测试环境中进行，但在这些环境中，常常面临一个核心挑战：真实用户流量的缺失。例如，在游戏场景中，登录、匹配、开局等关键的用户行为流量几乎为零。这导致一个问题：混沌工程的重要前提——"稳态指标"无法有效设定。

稳态指标是衡量系统健康状态的基础，如果无法获取系统在正常运行或高负载下的基线指标，那么即使注入故障，也难以量化评估系统是否仍然处于健康状态，或是已进入异常。在这种情况下，混沌实验的效果评估往往只能依赖于人工判断，难以实现自动化和闭环。

为此，探讨了将全链路压测与混沌工程结合的必要性，以应对以下关键挑战：

- **挑战 1：如何高度还原用户行为模式？** 真实系统的韧性评估需基于高度仿真的用户行为，例如复杂的登录、匹配、开局序列，以确保测试结果的准确性。
- **挑战 2：如何定量评估混沌工程的实验效果？** 缺乏量化指标，将使我们难以客观判断系统在故障注入后的表现，以及优化措施的有效性。

针对这些挑战，我们提出了以下解决思路：

- **全链路压测：** 模拟真实的用户行为模式，如登录、匹配、开局等复杂序列，以便在预发或测试环境中生成规律且稳定的“稳态指标曲线”。这条曲线代表了系统在模拟真实高负载下的健康运行状态，为后续的故障注入提供了可靠的基线。
- **混沌工程：** 在全链路压测所生成的稳定指标曲线的基础上，注入预设的故障（例如网络延迟、服务宕机、资源耗尽等）。通过观察相关指标（CPU 利用率、内存、响应时间、错误率等）在故障注入后的变化趋势，我们可以精准评估系统的韧性和抗故障能力。

这种结合带来的效果是显著的：

- **效果 1：** 全链路压测能够生成可靠且稳定的指标曲线，这些曲线可以作为混沌工程中“稳态指标”设定的权威依据。
- **效果 2：** 在压测过程中同时注入混沌（故障），我们可以观察指标的健康度变化，从而实现对系统韧性的量化评估，从“正常”到“受损”状态的转变将清晰可见。
- **效果 3：** 根据指标所体现的健康度表现，也就是系统在故障下的稳定程度，我们可以驱动和指导架构的迭代优化，确保关键业务流程在各种异常场景下都能保持高可用性。

测试环境全链路压测和混沌工程结合的四个步骤

聚焦于如何在测试环境中将全链路压测与混沌工程结合，通过四个关键步骤，提升系统的韧性与稳定性。

核心流程由四个环节构成：

- **压测协议编排**：此阶段我们模拟真实用户的操作路径，如登录、匹配、开局等复杂业务流程，将其转化为可执行的压测协议。
- **稳态基线采集**：在模拟用户行为的基础上，我们会持续采集系统在此正常运行状态下的各项性能指标，构建起一条反映系统健康状况的“基线曲线”。
- **故障注入**：此环节我们引入混沌工程实践，主动、可控地向运行中的系统注入各类故障，以探究系统在非稳态下的表现。
- **指标波动分析**：在故障注入后，系统各项指标必然发生波动，我们会紧密监测、分析这些变化，以此评估系统面对故障时的响应能力，判断压测结果是否符合预期。

关键技术点：

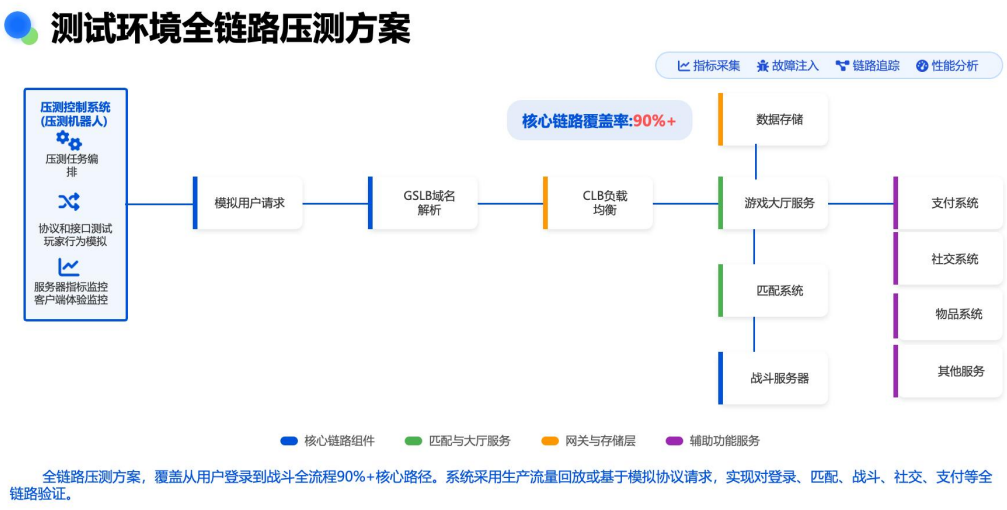
- **全面的稳态指标体系**：为准确评估系统性能，我们建立了包含匹配成功率、TPS/QPS（每秒事务数/每秒查询数）、开局成功率及进程资源利用率在内的多维度指标体系。
- **精细的压测协议编排**：我们不仅能模拟单一场景，更支持如登录-匹配-开局这类复杂的多场景组合，以最大限度地还原真实用户路径。

- **可控的故障注入策略**：我们定义了明确的故障类型，涵盖资源类（CPU、内存等）、网络类（延迟、丢包等）和依赖服务类（服务不可用等），并配备了自动终止机制以确保实验受控且安全。
- **智能的自动化验证与修复**：我们利用 AI 技术智能分析异常指标，并能自动推荐容错方案，力求实现故障发现、分析到修复的自动化闭环管理。

实施方法论：

1. **压测机器人编排典型业务流量**：针对游戏等具体场景，我们使用压测机器人精心编排包括登录、匹配队列、开局及战斗等在内的所有关键业务流程。
2. **生成稳态基线曲线**：在预发布（Pre-Stage）环境中，我们回放真实的业务流量，实时采集各类性能指标，进而描绘出系统在正常负载下的稳态基线。
3. **设计混沌实验方案**：在开展混沌实验前，我们严格定义故障假设、明确实验的范围、确定故障注入点，并制定详细的止损策略，确保实验的可控性与安全性。
4. **自动化闭环验证**：在稳态基线之上，我们主动注入故障，并持续监控各项指标的波动情况，一旦发现异常，系统将自动触发告警，实现验证的自动化闭环。

测试环境全链路压测方案



测试环境全链路压测方案是 SRE 实践中至关重要的一环，旨在通过全面、端到端的压力测试，验证系统在各种压力下的性能和稳定性。

核心目标：验证系统在极端负载下的稳定性和性能

- **全链路覆盖：**确保压测范围覆盖**90%以上**的核心业务路径，例如用户登录、匹配系统、战斗流程、社交互动以及支付功能等关键环节，模拟真实用户行为，深入探究系统瓶颈。
- **流量生成多元化：**
    - **生产流量回放：**利用历史生产数据，回放真实用户请求模式，高度还原线上场景，提升压测的准确性。
    - **模拟协议请求：**基于协议规范和接口定义，生成结构化压测流量，灵活配置请求参数，应对特定场景和功能测试。

压测过程控制与监控

- **压测控制系统（压测机器人）：**作为压测任务的核心调度者和执行者，负责：
    - **任务编排：**自动化管理压测任务的启动、停止、暂停和恢复，确保压测过程可控。
    - **协议与接口测试：**精准模拟各业务接口的协议交互，验证其在并发压力下的响应能力。
    - **玩家行为模拟：**模拟真实用户的操作路径和行为模式，例如登录、创建角色、进入大厅、发起匹配、参与战斗等，使压测更具真实性。
- **多维度监控体系：**在压测过程中，整合多方数据，提供全面的性能洞察：
    - **服务器指标监控：**实时采集 CPU 利用率、内存消耗、网络 I/O、磁盘 I/O 等服务器资源指标，发现资源瓶颈。
    - **客户端体验监控：**关注客户端的帧率（FPS）、响应时间、延迟等用户体验指标，从终端视角评估系统表现。

核心链路与系统架构

此方案覆盖了从用户请求到后端数据处理的全链路组件：

- **流量入口与分发：**
    - **模拟用户请求：**作为压测流量的始发点。
    - **GSLB 域名解析：**全球负载均衡，负责将用户请求智能路由到合适的区域或数据中心。
    - **CLB 负载均衡：**负责流量分发，将请求均衡地发送到后端服务实例。

- **核心业务服务：**
    - **游戏大厅服务（绿）：** 提供游戏入口、房间管理、用户数据管理等核心功能。
    - **匹配系统（绿）：** 负责玩家匹配，构建游戏对局。
    - **战斗服务器（蓝）：** 处理实时游戏逻辑和战斗过程。
- **数据存储（橙）：** 支撑各服务的数据持久化需求，保障数据一致性和完整性。
- **辅助功能服务（紫）：**
    - **支付系统：** 处理交易和充值服务。
    - **社交系统：** 提供好友、聊天、公会等社交互动功能。
    - **物品系统：** 管理游戏内虚拟物品。
    - **其他服务：** 泛指其他支撑性业务服务。

压测结果分析与洞察

为了充分利用压测数据，本方案集成了一套强大的分析工具链：

- **指标采集：** 统一收集各项性能指标，为后续分析提供原始数据。
- **故障注入：** 通过模拟网络延迟、服务宕机、资源耗尽等故障场景，验证系统的韧性和容错能力。
- **链路追踪：** 全面记录请求在分布式系统中的调用路径和耗时，精准定位性能瓶颈和异常点。
- **性能分析：** 结合指标、链路和故障数据，深入分析系统性能表现，识别瓶颈并提出优化建议。

通过这一全面的全链路压测方案，我们能够系统性地发现和解决潜在的性能问题，确保游戏系统具备更强的可扩展性和稳定性，为玩家提供流畅、稳定的游戏体验。

测试环境全链路压测 + 蓝鲸混沌实验结合实践

本节将探讨在测试环境中，如何将全链路压测与蓝鲸混沌实验相结合，以验证系统韧性并构建一套实践方法，从而提升系统的可靠性和高可用性。

- **全链路压测建立稳态基线：**
    - 首先，利用压测工具对 **Login 接口**执行全链路压测，以创建一个稳定的 **TPS 曲线**，图中所示约为 7 TPS。
    - 这条曲线（稳态指标）将作为后续所有混沌实验的性能基准，提供系统稳定运行状态的量化视图。
- **蓝鲸混沌平台注入故障：**
    - 接下来，通过"蓝鲸混沌工程平台"向负责 Login 功能的 **"lobby"模块**注入 **CPU 故障**。
    - 此步骤旨在模拟生产环境中可能出现的各类系统异常，通过主动破坏行为，评估系统在故障情境下的表现阈值。
- **稳定指标在故障注入后出现异动：**
    - 在混沌故障注入后，实时监控 Login 接口的 TPS 指标，并观察到明显的下降。
    - 具体来看，在 **2025/10/11 21:20:49** 这一时间点，系统 TPS 出现明显异常，原有的稳态指标被打破，清晰地描绘出系统在特定压力下的边界所在。通过多次注入和逐级加压，可以获取影响稳态指标的临界阈值。

- **用户体验受到影响：**
    - 从用户视角来看，当系统 TPS 出现异常时，游戏端会弹出 **"连接服务器无效，请尝试重新连接"** 的错误提示。
    - 这展示了底层技术异常如何影响业务功能和用户的实际体验，体现了系统韧性对业务连续性的重要性。

这项实践的核心价值在于：

- **全链路压测与混沌工程的结合，形成验证系统韧性的闭环。**在缺乏压测的测试环境中，通常无法获得连续可观测的指标，而这种结合能够使异常变得可量化、风险可控。通过**"稳态—异常—影响"** 的全流程分析，能够验证系统的韧性与故障自愈能力。
- **"有指标才能判断异常，混沌工程需要有基准指标对比才能量化影响。"** 这一核心理念贯穿始终，强调了量化分析在有效混沌实验中的基础地位。

"压测+混沌" 的结合，能够从技术指标到业务影响实现较全面的验证，是构建高可用系统的重要实践。

上线阶段：混沌测试、混沌安全、生产环境全链路压测方案

05
上线阶段： 混沌测试、混沌安全、生产环境全链路压测方案

上线阶段的验证很重要，聚焦于三个相互关联的实践：

混沌测试（Chaos Testing），这是一种主动探测系统脆弱性的方法。通过在分布式系统中故意引入受控的故障，例如模拟网络延迟、服务宕机、资源耗尽等场景，在生产环境

的真实压力下验证系统的弹性。这能够揭示在传统测试中难以发现的潜在缺陷，优化故障恢复机制，确保系统在高压和异常条件下的稳定运行。

其次，运用 **混沌安全（Chaos Security）** 的理念，将混沌工程的实践扩展到安全领域。通过模拟真实世界的安全攻击，例如 DDoS 攻击、SQL 注入、未授权访问等，来测试系统安全防御机制的韧性与有效性。此举旨在发现并加固那些可能被攻击者利用的安全漏洞和响应盲点，从而提升系统的整体安全防御能力。

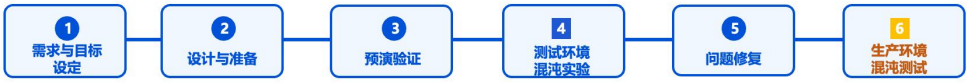
最后，是 **生产环境全链路压测（Production Environment Full-Link Pressure Test）**。脱离真实环境的压测数据往往无法完全反映系统的真实性能瓶颈。因此，这一实践要求在真实的生产环境中，通过模拟高并发或特定业务场景下的流量，对整个系统链路——包括前端、后端服务、数据库、缓存、消息队列等所有组件——进行端到端的压力测试。其核心目标在于评估系统在高负载下的性能表现、稳定性、可扩展性以及故障恢复能力，以确保系统能够承受预期的流量冲击，并验证各个环节的承载能力和协同工作情况。强调"生产环境"确保了测试数据的真实性，而"全链路"则保障了测试的全面性。

通过这三项实践的结合，旨在构建一个即使在面对极端情况和未知挑战时也能保持高可用、高性能和高安全性的弹性系统，为用户提供良好的服务体验。

混沌实验 vs 混沌测试

混沌实验 vs 混沌测试：区别与理念转变

对比维度	混沌实验（Chaos Experiment）	混沌测试（Chaos Testing/Verification）
执行环境	测试环境或预生产环境	生产环境
主要目的	发现问题	验证已知的容灾能力
风险接受度	较高，可接受一定程度的系统故障	极低，需严格控制风险，不影响用户体验
执行频率	开发迭代阶段，特性上线前	定期验证，系统变更后
主要关注点	学习与发现，揭示潜在问题	验证与确认，确保系统韧性持续有效



混沌实验与混沌测试在目标、风险假设和实施环境上存在显著差异，共同构成系统韧性提升的完整生命周期。

通过对比表格，我们可以清晰看到以下几个核心维度：

- **执行环境：**
    - **混沌实验**通常在**测试环境或预生产环境**中进行，其目的是在可控风险下探寻系统的未知脆弱点。
    - **混沌测试**则旨在**生产环境**中验证系统已知的容灾能力，确保其在真实负载下的有效性。
- **主要目的：**
    - **混沌实验**的核心是**发现问题**，揭示系统中潜藏的故障模式和行为。
    - **混沌测试**则着重于**验证已知的容灾能力**，确认系统在面对特定故障场景时，是否能按预期保持稳定。
- **风险接受度：**
    - **混沌实验**的风险接受度相对**较高**，可接受一定程度的系统故障，因为其主要目标是学习和发现。
    - **混沌测试**的风险接受度则要求**极低**，必须严格控制，确保不影响用户体验和服务连续性。例如，像游戏匹配系统这种一旦崩溃可能导致千万用户无法登录的关键模块，大规模的破坏性演练绝不可能直接在生产环境进行。
- **执行频率：**
    - **混沌实验**主要在**开发迭代阶段和特性上线前**进行，作为质量保障的早期环节。
    - **混沌测试**则在**功能上线后或系统变更后**进行，作为一种持续性的验证机制。
- **主要关注点：**
    - **混沌实验**侧重于**学习与发现**，主动探索系统的薄弱环节。
    - **混沌测试**则专注于**验证与确认**，保障系统韧性持续有效。

从混沌实验走向混沌测试，代表着系统韧性建设从探索性阶段迈向成熟与常态化验证。这种循序渐进的生命周期，我们可以通过以下六个步骤来理解：

1. **需求与目标设定：** 首先，我们需要明确系统要达到的韧性目标以及希望通过混沌活动解决的关键问题。

2. **设计与准备：** 在此阶段，我们会详细规划混沌实验或测试的场景、选定注入故障的方法与工具，以及确定如何衡量实验结果。

3. **预演验证：** 在执行大规模混沌之前，通常会进行小规模的预演，在受控环境中对设计方案和工具进行初步验证，确保其安全性和有效性。

4. **测试环境混沌实验：** 随后，在测试或预生产环境中执行多样的混沌实验，主动发现潜在问题和系统脆弱点，从中学习并迭代改进。例如，对于一个匹配服务，我们会在这里搭建完整的测试环境，注入模拟流量，并进行各种故障演练。

5. **问题修复：** 针对混沌实验中暴露出的问题，团队会及时进行修复和优化，提高系统的容错能力。

6. **生产环境混沌测试：** 经过层层验证与修复后，我们才会在生产环境对已修复、确认具备容灾能力的场景进行小范围、低风险的混沌测试，以验证这些能力在真实世界中的有效性。这通常意味着只影响一小部分机器或用户，且具备快速回滚机制。

总之，明确混沌实验与混沌测试的区别很重要，它指导如何在不同环境、以不同目的、采取不同风险策略来推进混沌工程实践，构建一个更具弹性、更可持续运行的系统。

混沌测试安全保障：三重防护机制

本幻灯片聚焦于"混沌测试安全保障（三重防护机制）"，阐述在进行混沌测试时，如何通过多层保障机制确保测试过程的安全可控，避免对生产环境造成较大影响，并维持业务连续性。

混沌测试的安全原则

- 最小化影响原则：实验范围需严格控制在最小必要限度内，降低潜在风险。

- 分阶段执行：混沌测试应遵循渐进式策略，优先从非核心系统开始，逐步扩展至核心系统，在实践中积累经验并验证有效性。
- 快速终止机制：设定清晰的触发阈值，一旦达到或超出这些阈值，混沌测试应能自动及时中止。
- 多人 Check 机制：针对生产环境的演练方案，需由 SRE、开发、测试等多方共同确认，以确保方案的全面性与安全性。

爆炸半径控制策略

- 故障隔离：利用现有的架构隔离机制（如"大区"或"集群"）限制故障实验区域，将潜在影响范围限定在特定边界内。
- 流量分流：实验应仅影响较小比例的流量（例如 0.1%或更低），降低对真实用户的影响。
- 多层防护：建立多重自动化防护措施与止损机制，确保在故障发生时能够及时介入并快速恢复。

执行前置条件

- 经过验证：核心前提是，所有容灾能力与故障恢复措施需在测试环境中通过混沌实验的验证后，方可启动生产环境的测试。
- 演练预案：制定完整的实验方案与回滚预案，为可能出现的突发状况做好准备。
- 全程监控：混沌测试期间，需对系统关键指标和用户体验进行实时监控，以便及时发现并响应异常。
- 专人值守：混沌实验全程应由专人负责监控和响应，确保突发情况能被及时处理。

混沌测试的爆炸半径控制模型 幻灯片右侧展示了“混沌测试的爆炸半径控制模型”，一个由同心圆构成的可视化模型：

- “最小影响域”位于最内层，代表混沌实验实际作用的最小范围。
- “混沌实验边界”是其次一层，表示混沌测试的实际操作边界。
- “生产环境边界”位于最外层，涵盖整个生产系统。

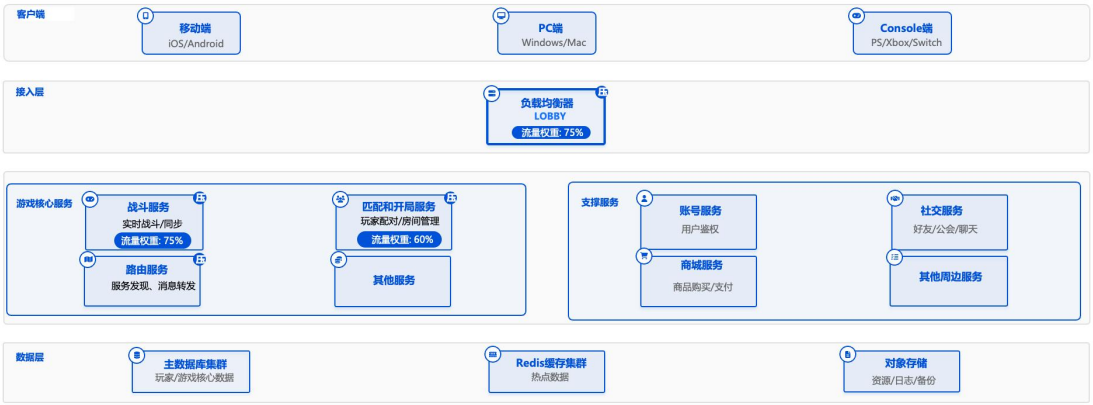
此模型周围的关键控制点包括：

- 多层防护：以盾牌图标象征，强调防御机制。
- 全程监控：以眼睛图标表示，强调实时洞察。

- 流量分流：以分叉箭头图标展示，强调精细化控制。
- 快速回滚：以循环箭头图标呈现，强调快速恢复能力。

总结而言，通过多层防护、资源隔离、流量控制等综合手段，旨在确保混沌实验的可控性，从而保障业务的连续性。

基于动态权重调整的生产环境全链路压测方案



幻灯片展示的是一套专为游戏行业定制的“基于动态权重调整的生产环境全链路压测方案”。其核心理念在于，通过在生产环境的各个关键环节灵活配置流量权重，我们得以进行精确且安全的压测，而非简单地注入虚拟流量。

该方案的亮点在于，我们为每个核心模块设计了可配置的流量权重机制，以实现局部的流量注入与精细化控制。具体而言：

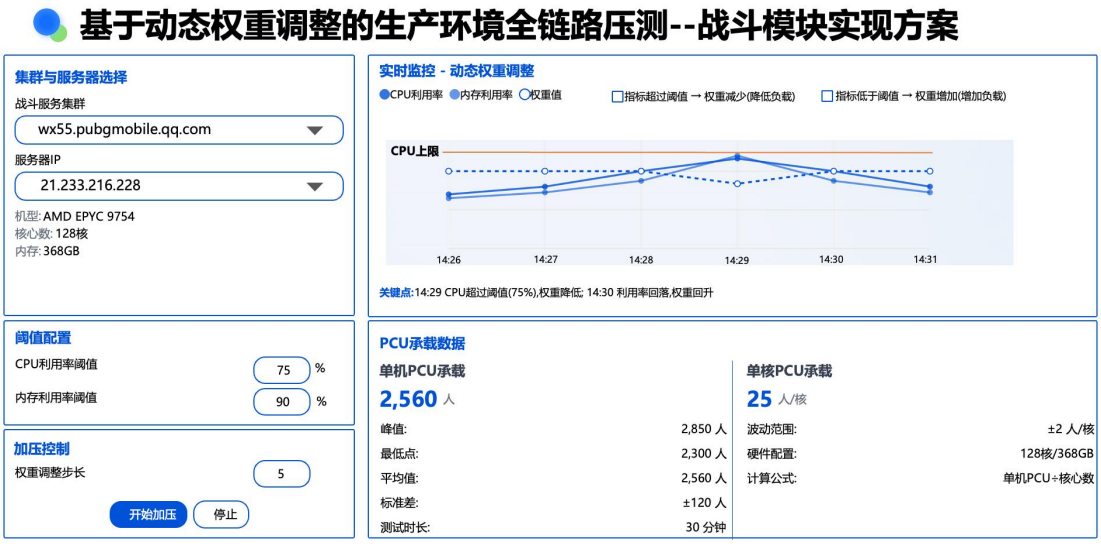
- **多客户端接入**：系统支持包括移动端（iOS/Android）、PC 端（Windows/Mac）以及 Console 端（PS/XBOX/Switch）在内的多种客户端类型，确保压测覆盖所有用户入口。
- **接入层流量控制**：
  - 通过负载均衡器（LOBBY）作为统一接入层，我们在此处配置了75%的流量权重，使其能够差异化地处理流经各机器的流量。这意味着我们可以设定特定服务器上的流量与其余服务器不同，从而实现流量的精细化分配与控制。
- **游戏核心服务压测**：

- 针对“战斗服务”与“匹配和开局服务”等游戏核心业务，我们也设置了特定的流量权重（分别为 75%和 60%）。
- 以战斗模块为例，我们可以精确配置指定集群或机器的流量分配，实现局部加压，以模拟真实业务负载。

- **服务发现与消息转发**：“路由服务”负责底层的服务发现与消息转发，是连接各服务、保障通信流畅的关键。
- **多元支撑服务**：
  - 账号服务：负责用户鉴权与管理。
  - 商城服务：处理商品购买与支付流程。
  - 社交服务：支持好友互动、公会管理与聊天功能。
  - 其他周边服务：涵盖了游戏生态中的其他非核心但不可或缺的业务模块。
- **分层数据存储**：
  - 主数据库集群：存储玩家与游戏核心数据。
  - Redis 缓存集群：用于高速存取热点数据。
  - 对象存储：承载资源文件、日志以及备份数据，优化数据存储效率与可靠性。

整体而言，这套分层架构方案通过在客户端层、接入层、应用服务层以及数据层各环节的流量权重动态调整，确保在不影响生产环境核心业务的前提下，对系统承载能力进行全链路、精准、安全的验证。

基于动态权重调整的生产环境全链路压测---战斗模块实现方案



我们的核心目标是在不影响线上服务的前提下，精准评估系统的真实承载能力，并确保压测结果高度贴近真实用户行为。

该方案构建了一个智能化的实时压测系统，其关键功能和优势如下：

- **动态负载调整：**
    - 系统能够根据服务器 CPU 和内存的实时利用率，动态调整压测的负载权重。
    - 通过设置 CPU 和内存的利用率阈值以及权重调整步长，实现精细化的压测管理，确保压测过程中的资源利用率始终处于可控范围之内。
    - 当 CPU 或内存利用率超过预设阈值时，系统会自动降低权重，以减少压测负载，避免对生产环境造成负面影响。
    - 当资源利用率回落到阈值之下时，系统会相应增加权重，继续探测系统的承载上限。
- **实时监控与可视化：**
    - 我实时监控图表直观展示了 CPU 利用率、内存利用率和权重值的变化趋势。
    - 横轴表示时间轴，展现了压测不同阶段的资源波动。

- 图表清晰标注了关键事件，例如“14:29 CPU 超过阈值(75%)，权重降低”和“14:30 利用率回落，权重回升”，生动地呈现了系统动态调整的过程。

- **精准承载能力数据分析：**
  - 通过这种动态调整机制，系统能精确地测算出在达到资源阈值时，单个服务器和单个 CPU 核心能够承载的并发玩家数（PCU）。
  - 表格详细展示了：
    - **单机 PCU 承载**：平均 2,560 人，峰值 2,850 人，最低点 2,300 人，标准差 ±120 人，测试时长 30 分钟。
    - **单核 PCU 承载**：25 人/核，波动范围 ±2 人/核。
    - 这些数据基于当前服务器的硬件配置（128 核/368GB）进行计算，为容量规划提供了坚实的数据基础。
- **解决传统压测与线上表现不符的问题：**
  - 传统压测常出现测试结果与线上实际承载能力不符的情况，例如，压测时单机承载 5000 人，但实际线上仅能支持 2000 人。
  - 本方案通过在生产环境进行动态调整和实时反馈，有效解决了这一痛点，确保压测结果高度还原真实线上性能。
- **持续更新容量数据：**
  - 每次版本发布或配置变更后，系统都能快速更新并提供最新的容量承载值，确保容量规划始终基于最准确的数据，为未来的系统扩展提供可靠依据。

这种“前置防护–平台化工具–分级组织”的三层运维体系，利用动态权重调整的压测方案，极大地提升了我们在亿级设备与千万用户规模下，实现从预防到响应、从发布到复盘的全链路高可用保障能力。

运营阶段：混沌工程与 AI 故障定位

为什么需要 SRE 领域大模型

- **混沌工程：AI 模型的训练场** 混沌工程的价值不仅在于验证系统的韧性，还在于为 SRE 领域的大模型提供训练数据，其在演练过程中产生的故障数据，可作为 AI 模型训练的素材。
- **AI 故障定位的实现路径** 通过较多次可控的故障注入，AI 模型可学习并理解在各种业务表现异常下，是何种故障原因导致。当真实的业务表现发生时，AI 可根据其学习经验，倒推并定位到根本原因。
- **两者相辅相成，提升运维能力** 混沌工程与 AI 故障定位互为补充，推动系统韧性和智能化运维能力的提升。

为什么 SRE 领域迫切需要 AI 赋能：构建专属大模型势在必行

当下 SRE 团队在日常工作中常常面临多重挑战，导致 MTTR（平均恢复时间）优化遭遇瓶颈。通过构建 SRE 领域大模型，旨在实现从"人找信息"到"信息找人"的转变。

- **当前 SRE 故障定位的痛点：**
  - **数据孤岛与信息过载：** 面对海量的监控指标、日志、链路追踪、知识库以及历史故障报告等多源异构数据，SRE 工程师在故障定位过程中，需同时处理繁杂且分散的信息，这使得有效的关联分析较为困难，增加了问题排查的复杂性和耗时。

- **专家经验瓶颈：**故障诊断，尤其是针对复杂度较高的系统问题，往往依赖资深 SRE 专家的个人经验与洞察力。这种对"个人经验"的依赖，限制了团队整体的响应能力，也使得经验难以有效沉淀和规模化复用，一旦专家流失，会对团队造成较大风险。
- **响应速度的限制：**随着业务对 MTTR 的要求越来越高，传统的人工排查模式已接近响应速度的上限。为了进一步优化 MTTR，需要一个"智能辅助工具"，能够理解系统上下文并进行实时推理，从而提升故障响应的效率和准确性。

- **通用大模型的局限性：**
  - **领域知识的缺失：**现有的通用大模型对于 SRE 领域的专业术语、监控指标、日志信息、架构拓扑以及业务交互流程，往往缺乏准确的理解。例如，对于特定的业务错误码或系统组件名称，通用模型可能无法识别其含义，导致判断不准确。
  - **"幻觉"现象的风险：**由于对 SRE 领域知识的有限理解，通用大模型在提供故障诊断或运维建议时，容易生成不准确、不相关甚至是具有误导性的"幻觉"内容，这对关键系统运维而言，存在一定的风险。
- **破局点：**
  - 正是基于上述挑战及通用模型的局限性，我们必须构建一个能够深度理解 SRE 领域知识的专属大模型。这一模型将能够主动分析并推荐潜在的故障信息、提供精准的诊断建议，从而实现运维模式从被动式“人找信息”向主动式“信息找人”的根本性转变，显著提升故障处理效率与系统可靠性。

构建 SRE 领域模型的四大核心

构建 SRE 领域模型的四大核心环节，这是一个系统工程，核心在于领域知识的注入，而非从零开始训练。

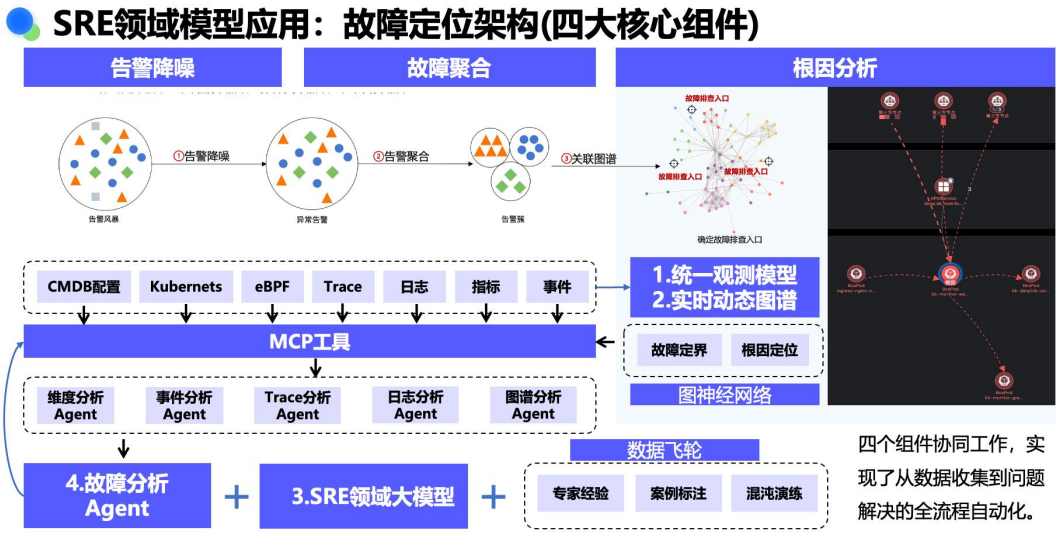
训练大致分为以下四个步骤：

1. **基石选择：**
    - 首先是选取一个开源或商用基础大模型作为 SRE 领域模型的基座。可能会选择 Deepseek 或混元这类模型作为起点。
2. **领域数据的准备与治理：**
    - 将混沌实验数据、监控指标、知识库、以及历史故障数据等 SRE 相关数据，作为大模型训练的素材。在数据准备过程中，需要进行清洗、去噪和标准化，并建立数据间的关联性，以确保数据的质量和模型训练的有效性。
3. **知识注入：领域适应的关键步骤：**
    - 知识注入是核心环节，旨在将 SRE 领域的专业知识融入基础模型。
        - **持续预训练：** 利用 SRE 领域文本和数据对模型进行持续预训练，使其掌握 SRE 领域知识与经验。
        - **指令微调：** 通过指令样本对模型进行微调，使其更好地理解和执行 SRE 专业任务。

- **人类反馈强化学习：** 引入 SRE 专家的反馈机制，对模型的输出进行评估，用于持续优化模型表现。
- **环境反馈强化学习：** 结合混沌工程演练的结果，对模型输出进行反馈评估并持续优化，确保模型在真实或模拟环境中的有效性。

4. **评估与部署：**
   - 最后一步是建立评估机制，并确保模型能够满足生产环境的需求。建立评估集，以验证模型在故障识别、问题解答等 SRE 核心任务上的性能，确保其能够达到生产级别的要求。

SRE 领域模型应用：故障定位架构（四大核心框架）



这是故障定位的核心架构，主要由四大组件协同工作，实现从数据收集到问题解决的流程。

一、统一观测模型与实时动态图谱

- 统一观测模型：整合了来自 eBPF、Trace、日志、指标等多种数据源，构建了一个观测模型，确保所有相关数据都能被聚合处理。
- 实时动态图谱：在此基础上，将这些观测指标间的关联关系以图谱的形式动态呈现，旨在实现告警的有效收敛。当故障发生时，往往伴随着"告警风暴"，海量的告警信息令人难以迅速识别真正的故障根源。通过图谱，能够看到告警间的联系，快速定位关键路径。

二、SRE 领域大模型

- 该模型的核心作用是驱动故障分析的自动化进程。
- 在故障分析过程中，它会调用专门的“故障分析 Agent”来自动化执行各项分析任务。

三、故障分析 Agent - 这个 Agent 是 SRE 领域大模型的重要组成部分，它负责接收模型指令，并执行具体的故障分析操作，从而实现故障定位的自动化。

这四大组件紧密协作，形成了一个强大的故障定位体系，显著提升了故障排查的效率与准确性。

SRE 领域模型应用：AI 故障诊断案例

SRE 领域模型应用：AI 故障诊断案例，重点阐述了 AI 在 SRE 故障诊断中的具体实践，以“某大型游戏业务开局失败 AI 定位案例”为例，展现了故障诊断流程、SRE 领域模型的能力和知识储备，以及实际案例的诊疗成果。

AI 驱动的故障诊断流程

- **告警收敛与故障识别：** 初始阶段，系统产生的大量告警信息将通过 SRE 知识图谱进行智能聚合并进一步分析，从而精准识别并确认“故障”事件的发生。
- **AI Agent 诊断：** 一旦故障确立，AI Agent 将被激活，它通过调用 SRE 领域专属的大模型对故障进行深度诊断。
- **工具辅助分析：** 在诊断过程中，若有必要，AI 将自动调用 MCP 等自动化运维工具平台，获取更详尽的数据支持。

- **LLM 整合与报告生成：** 诊断结果和辅助工具的分析数据，将由大语言模型（LLM）进行深入整合，形成结构化根因分析报告。
- **结果推送：** 最终的根因分析报告将通过企业微信等渠道，及时推送给相关运维人员，以便快速响应与处理。

SRE 领域模型能力与知识储备

SRE 领域模型被定位为“游戏服务器故障定位专家”，其核心能力聚焦于处理游戏拉起对局导致的超时场景。该模型具备七项关键能力：

- **多维度数据关联分析：** 能够综合分析对局数、大区、版本、集群、可用区、子模式等多方面数据。
- **多模式并行故障分析：** 支持同时处理不同类型的故障模式。
- **自动化决策树故障排查：** 运用决策树机制，实现故障排查过程的自动化。
- **系统日志模式识别与安全漏洞检测：** 从海量日志中识别异常模式，并检测潜在的安全漏洞。
- **Redis 性能与连接异常分析：** 专注于 Redis 相关的性能瓶颈和连接异常问题。
- **网络事件关联分析：** 关联分析网络事件，评估其对系统造成的影响。
- **结构化报告生成：** 将复杂的诊断过程和结果，以清晰、结构化的报告形式呈现。

该模型依赖于七项核心知识储备：

- **游戏匹配与拉起对局流程：** 深刻理解游戏核心业务流程。
- **游戏服务器架构：** 熟悉 DS 版本、集群拓扑、BusID 映射等服务器架构细节。
- **DS 版本合法性校验规则：** 掌握 DS 版本合法性判断的标准。
- **系统日志异常模式库：** 拥有识别常见系统日志异常模式的能力，如 nsicd 用户认证异常。
- **云服务可用区故障传播模型：** 理解云服务中可用区故障的传播机制。
- **Redis 连接池管理与性能特征：** 掌握 Redis 连接池的优化与性能监控。
- **网络事件类型与影响范围模型：** 了解不同网络事件的类型及其潜在影响范围。

模型通过 Prompt 系统，利用提示词（Prompt）机制与大模型进行交互，引导模型按既定诊断流程，一步步完成故障定位。同时，系统并行运行 AI 诊断与人工编排流程，通过对

比两者结果进行 AB 测试，不断优化模型性能、修正潜在的“幻觉”，并调整提示词策略，以期最终实现完全自动化的故障诊断。

案例分析：某大型游戏业务开局失败 AI 定位

在某个大型游戏业务开局失败的案例中，AI 诊断得出了以下根因分析结论：

- **主要故障根因：**
    - 可用区级故障集中在南京地区，其中南京可用区（2 区, 3B, 1 区）服务器故障占比高达 58%，显示出显著的区域性问题。
    - 子模式故障方面，两个核心子模式（2 /24）合计贡献了 20 个故障。
    - 特定子模式（2/79/24 20）的故障占比超出了预设阈值。
- **分布特征分析：**
    - 故障主要分布在南京 2 区（42.6%）、南京 3 区（31.9%）和南京 1 区（21.3%）。
    - 主要故障集群中，无单一集群占比超过 20%，表明故障并非由单一集群引发。

（三）总结与展望

腾讯游戏高可用保障体系总结出四大核心经验，旨在构筑全生命周期、多层次的防御体系，从架构设计源头到线上混沌验证，实时守护游戏服务的稳定运行。

一、经验一：高可用能力必须左移

高可用能力左移，其核心在于将可靠性设计前置到软件开发生命周期的早期阶段。通过 TDR（Tencent Design Review）评审机制，将高可用性评估深度嵌入到研发流程之中，在架构设计之初便进行多维度全面审视。这种早期介入和严格把关，能够极大地识别并规避潜在的架构缺陷和单点风险，从而有效减少九成以上可能影响高可用的故障风险，避免问题在后期上线后才暴露，彼时修复的成本和影响将成倍增加。

二、经验二：混沌演练需分层实施，严格区分测试与生产环境

混沌演练是验证系统韧性的重要手段，但必须策略性地分层进行，并严格区分测试环境与正式生产环境。

- **分层与渐进式验证：**混沌演练应从接入层开始，逐步深入到存储层，分批次、渐进式地验证系统的抗故障能力。这种由表及里、循序渐进的方式，确保了演练范围和深度的可控性。
- **环境隔离的重要性：**必须明确区分“混沌实验”和“混沌测试”。
    - **混沌实验：**专指在测试环境或预发布环境进行的混沌演练，其目的是允许团队在相对安全的环境中探索和发现系统弱点，并在不影响用户体验的前提下进行充分的问题修复与优化。
    - **混沌测试：**针对生产环境进行的，其前提是系统已经过充分的混沌实验验证，且具备强大的自愈和回滚能力。在生产环境进行任何操作都必须经过严谨的规划、充分的验证，以确保安全性和业务的连续性。在生产环境中的操作，尤其是一些未经充分验证的变更，是绝对不允许的，以防止对业务造成不可逆的影响。

三、经验三：全链路协同是构建高可用体系的关键

构建真正韧性的高可用体系，并非单一技术或实践所能达成，它需要多项 SRE 实践深度融合、协同发力，以形成整体效益。

- **全链路压测：**作为验证系统容量和稳定性的基石，它能模拟高并发场景，提前发现性能瓶颈。
- **流量控制与限流算法：**在系统面临异常流量或局部过载时，能够智能地对流量进行调度和限制，确保核心服务的可用性。

- **混沌实验：**用于主动探测系统在各种异常条件下的行为，验证容错和自愈机制。

当这些实践相互配合、形成合力时，它们共同支撑起一个强大而富有弹性高可用体系，而非孤立地各自发挥作用。例如，如果没有充分的全链路压测数据支撑，混沌实验的有效性将大打折扣，因为无法准确模拟真实负载冲击下的系统反应。

四、经验四：拥抱 AI，让 AI 成为故障定位的超级助手

在 SRE 实践中，人工智能技术展现出巨大潜力，尤其是在故障排查和定位方面。

- **知识沉淀与结构化：**将历史故障处理经验、问题排查 SOP（标准操作规程）以及监控告警规则等非结构化知识，通过智能工具进行高效的抽取、归纳与结构化。
- **SRE 领域大模型应用：**结合这些结构化知识和专门训练的 SRE 领域大模型，能够对海量的监控数据、日志信息进行更智能的分析和关联。

通过 AI 的辅助，SRE 团队可以显著提升故障诊断的自动化和智能化水平，将故障定位时间有效缩短高达 75%，极大地提高了故障响应效率，将 SRE 从繁琐的人工排查中解放出来，聚焦于更深层次的系统优化与创新。当然，AI 工具和模型也需要持续的打磨和优化才能发挥其最大效用。

结束语与致谢

“ 拥抱混沌，方能赢得秩序 "

感谢您的聆听，希望腾讯游戏的实战经验能为您的高可用体系建设提供启发！

最后一个结语，送给广大 SRE 同仁，这也是我们腾讯游戏在高可用保障中深刻体会到的哲理：拥抱混沌，方能赢得秩序。

5.4.9 小米混沌工程与全链路压测实践

SRE Elite 精选原因

小米作为全球领先的消费电子和智能制造企业，其互联网服务承载着亿级设备和千万级日活用户。在电商大促、新品发布、米粉节等关键业务场景中，系统需要应对瞬时流量洪峰的冲击，对稳定性和弹性提出了极高要求。

面对这些挑战，小米中间件团队从"蓝军"视角出发，构建了混沌工程与全链路压测双轮驱动的韧性保障体系。混沌工程方面，团队实现了覆盖基础硬件层、容器层和应用层的无侵入式故障注入能力，并创新性地将 AI 技术融入故障场景识别与实验规划，故障分类准确率达 90%以上。全链路压测方面，团队自研了基于 JDK21 协程和 Faas 架构的分布式压测引擎，具备百万级并发能力，通过真实流量录制实现高拟真压测，问题排查效率从小时级提升至分钟级。

本案例系统分享了从故障注入技术选型、AI 智能化演进、压测引擎架构设计到一站式观测联动的完整实践经验，对于需要构建大规模分布式系统韧性保障能力的团队具有重要参考价值。

（一）背景及设计原则

本文将从中间件技术的视角，深入探讨我们在公司内部如何实践混沌工程和全链路压测，以期提升系统的稳定性与弹性。

蓝军与红军：SRE 角色分工

在 SRE 领域，我们通常将团队角色划分为“蓝军”与“红军”，以构建和维护高可用、高韧性的系统。

蓝军：主动发现系统风险与韧性短板

蓝军的核心职责是系统地探测和发现分布式系统链路中的潜在风险点，这些风险可能来源于：

- **拓扑结构问题：** 例如不合理的网络配置或服务依赖。
- **容器资源问题：** 包括资源抢占、耗尽或配置不当。
- **应用层问题：** 如代码缺陷、业务逻辑漏洞。
- **压测水位问题：** 如系统在特定负载下的性能瓶颈。

为了赋能蓝军有效地“引路和筑路”，我们通过中间件视角构建了两大平台：

- **混沌工程 (Chaos Engineering):**
    - 通过在生产环境中主动注入故障、模拟异常场景，来验证系统对各种意外情况的容错能力和快速恢复能力。它帮助我们发现那些在常规测试中难以暴露的深层次问题。
- **全链路压测 (Full-link Stress Testing):**
    - 旨在模拟高并发、大流量的用户请求，对从用户前端到所有后端服务的完整调用链进行端到端测试，全面评估系统在高负载下的性能表现、瓶颈和稳定性。

红军：快速识别与定位线上问题

红军的主要任务是识别和诊断线上运行中出现的问题。蓝军主动"发现问题"，红军精准"诊断问题"。

- **可观测性平台 (Observability):**
    - 是红军最重要的抓手。通过收集、聚合和分析系统的日志 (Logs)、指标 (Metrics) 和链路追踪 (Traces) 数据，可观测性平台使我们能够深入理解系统内部状态，快速定位故障根源，并进行性能优化。

总而言之，蓝军利用混沌工程和全链路压测，主动发现和验证系统的弱点；红军则通过强大的可观测性平台，实时监控和快速响应线上突发事件。这种“一攻一守”的协同机制，是确保我们系统韧性和可靠性的基石。

（二）体系设计及关键流程

混沌工程核心能力与实践原则

核心能力

1. 故障注入(Fault Injection)
故障注入指在系统运行时有意引入各种故障场景,如硬件故障、容器故障、网络故障、服务故障等，观察系统在这些故障情况下的表现

2. 监控和度量(Monitoring and Metrics)
在注入故障的同时,需要对系统的关键指标进行监控和度量，如吞吐量、延迟、错误率、恢复时长等，以评估系统在故障情况下的可靠性和自愈能力

3. 实验分析(Experiment Analysis)
对实验结果进行分析，识别系统中存在的弱点和瓶颈,并提出改进措施。

4. 智能化(Intelligent)
通过自动化工具来执行故障注入、监控和分析

混沌工程作为提升系统韧性的关键实践，概念虽提出已久，但在实际落地中能真正有效践行的团队并不多。我们在推进混沌工程时，秉承与社区保持一致的理念，主要从以下四个核心能力方面来构建。

核心能力

- **1. 故障注入 (Fault Injection)**
    - 作为混沌工程最核心、最本质的环节，故障注入的核心思想是在系统运行时，有意识地、可控地引入各种预设的故障场景。
    - 这些故障可能涵盖硬件故障、宿主机或容器故障、网络故障（如延迟、丢包）、以及服务层面的故障等。
    - 我们的目标是在生产环境中执行故障注入，但需严格控制风险，确保对业务的影响最小化。通过此举，我们能够观察系统在真实故障下的行为模式，从而评估其健壮性。
- **2. 监控和度量 (Monitoring and Metrics)**
    - 在故障注入的同时，我们必须对系统的关键指标进行实时的监控与精确的度量。

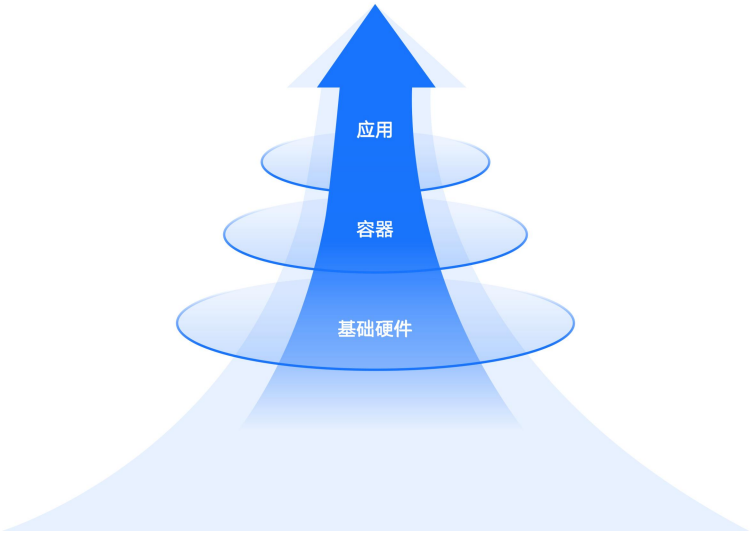
- 这些关键指标包括但不限于系统的吞吐量、请求延迟、错误率，以及从故障发生到系统恢复正常所需的时间等。
- 通过对这些数据的细致分析，我们能够准确评估系统在 SRE 语境下，面对故障时的可靠性表现和其自动愈合的能力，判断故障注入实验是否符合预期，以及故障影响范围和持续时间。

- **3. 实验分析 (Experiment Analysis)**
  - 完成故障注入与监控后，需要对实验结果进行深入分析。
  - 这份分析通常会以报告的形式呈现，其核心目的在于识别系统中存在的薄弱环节和性能瓶颈。
  - 更重要的是，基于这些发现，我们需要提出具体的改进措施和优化建议，以持续提升系统的韧性。
- **4. 智能化 (Intelligent)**
  - 将人工智能技术融入混沌工程流程是提升效率和覆盖面的重要方向。
  - 这包括通过自动化工具来执行故障注入、实现智能化的监控策略以及自动化分析实验结果。
  - 我们正在积极探索将 AI 技术应用于这一领域，以期自动化整个混沌工程流程，从而大幅提高韧性测试的效率和广度，减少人工干预，并有望发现更复杂的潜在问题。

我们目前的故障注入实践主要从三个层面展开，以确保其全面性和有效性。

故障注入

故障注入



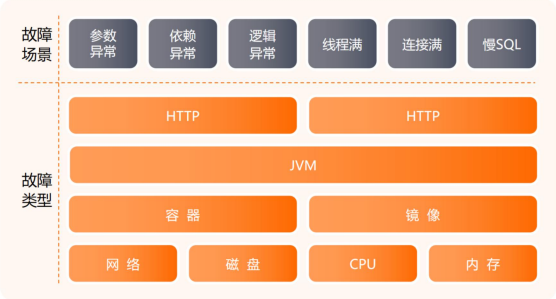
故障注入可以针对系统的不同抽象层次进行深入实践，这些层次由下至上，涵盖了从基础设施到业务逻辑的整个系统架构：

- **基础硬件层**：这是系统运行的基石，包括物理服务器、网络设备、存储系统等。在此层级进行故障注入，旨在模拟最底层的硬件故障，例如：
    - CPU 异常：模拟 CPU 性能下降或过载。
    - 网络中断或延迟：模拟网络连接的不可靠性或链路故障。
    - 机房级故障：模拟整个数据中心或单个机房的电力中断、环境异常等。通过在基础硬件层注入故障，可以有效评估系统在面对底层基础设施冲击时的稳定性和恢复能力。
- **容器层**：作为应用层与基础硬件层之间的桥梁，容器环境（如 Docker 容器、Kubernetes Pod）承载着应用程序的运行。在容器层注入故障，可以模拟容器运行时的各种异常情况，例如：
    - 容器资源耗尽：模拟 CPU、内存、磁盘 I/O 等资源被过度消耗。
    - 网络分区：模拟容器间或容器与外部服务间的网络通信障碍。
    - 容器崩溃或重启：模拟容器意外停止或频繁重启的场景。
    - 镜像拉取失败：模拟在部署或扩展时，容器镜像无法正常获取的问题。在此层级进行故障注入，能有效验证编排系统（如 Kubernetes）的自愈能力和容错机制。

- **应用层**：这是最贴近用户和业务逻辑的层面。在应用层注入故障，能够模拟与业务相关的各种软件缺陷和运行时问题，例如：
    - 业务逻辑错误：模拟代码中的缺陷，导致业务流程中断或数据异常。
    - 服务间调用失败：模拟微服务架构中，一个服务调用另一个服务时出现的超时、拒绝或错误响应。
    - 接口或参数异常：模拟应用程序接收到无效输入或输出不符合预期的场景。
    - 高并发或慢请求：模拟瞬时流量洪峰导致的服务响应延迟或饱和。

    鉴于大多数生产环境问题往往源于应用层，这一层级的故障注入尤其普遍且关键，能直接反映系统对业务连续性的保障能力。

故障注入



典型注入故障场景

我们从业务视角深入剖析，来审视系统可能出现的各种异常情况，并依此进行故障注入，以提升系统的韧性与稳定性。

我们将聚焦于应用层或业务逻辑层面常出现的指标和异常，它们是保障业务连续性的关键考量点：

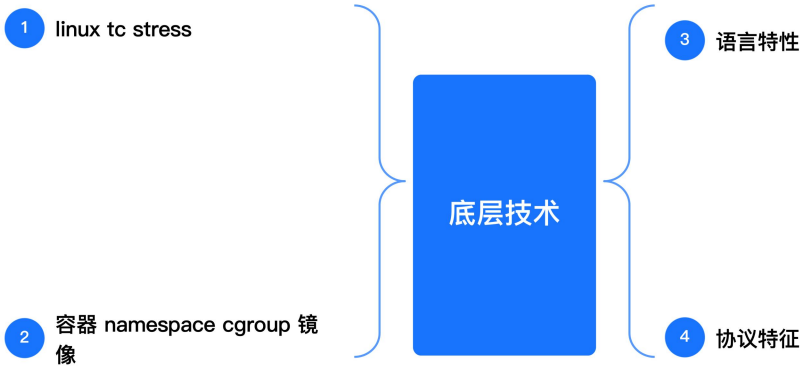
- **参数异常：** 模拟输入参数的不正或缺失，例如用户输入数据校验失败、API 调用参数类型错误等，这类问题可能导致业务流程中断或数据处理错误。
- **依赖异常：** 深入模拟外部服务、组件或资源出现故障的情形。无论大型企业还是初创公司，在处理依赖瓶颈方面，都或多或少遇到挑战。我们深知，尽管服

务等级协议（SLA）力求完善，但在复杂的分布式系统中，依赖故障（如数据库不可用、第三方服务响应慢、消息队列阻塞等）仍是常见且难以完全避免的“软肋”。因此，我们会在依赖环节进行重点注入，以验证系统对外部依赖失效的鲁棒性。

- **逻辑异常：** 旨在模拟应用程序内部业务逻辑的错误，例如代码缺陷、条件判断失误、业务流程设计不合理等，这些问题会导致业务功能异常或处理结果不符合预期。
- **线程池饱和：** 模拟系统线程资源耗尽，导致无法处理新的请求，例如在高并发场景下，业务处理线程池达到上限，新的请求被拒绝或阻塞。
- **连接池饱和：** 模拟数据库连接池或外部服务连接池耗尽，无法建立新的连接，例如在短时间内出现大量数据库连接请求，导致连接池资源耗尽。
- **慢 SQL：** 模拟数据库查询性能瓶颈，导致 SQL 执行缓慢，例如复杂查询没有命中索引、数据库负载过高，这类问题会严重影响业务响应时间。

这些指标均是业务上线后经常出现的潜在问题源。因此，我们的故障注入平台将横跨基础设施、容器和应用这三个技术层级，去模拟和验证业务场景下这些核心指标的弹性和恢复能力。

故障注入



具体而言，故障注入的底层技术考量主要集中在以下四个关键点：

1. **Linux tc 与 stress 工具**
    - **核心思想：** 掌握 Linux 底层技术特征是开展故障注入的基础。
    - **硬件层故障注入：**

- **网络、磁盘、CPU、内存等硬件层面：** 参照 Linux 的 tc（流量控制）和 stress 工具进行故障模拟。
  - **tc（Traffic Control）：** 经典且强大的内核协议栈工具，通过在流量出网卡前的队列层面进行拦截，可实现：
    - 网络延迟模拟
    - 网卡丢包模拟
    - 数据包损坏模拟
  - **stress：** Linux 中一个很早就引入的插件，通过消耗系统资源（如 CPU、内存）来模拟资源吃紧场景。
    - CPU 高负载模拟
    - 内存耗尽模拟

2. **容器技术（namespace、cgroup、镜像）**
   - **核心原则：** 故障注入应具备无侵入性，尤其在生产环境中。
   - **容器化背景：** 鉴于当前服务普遍采用容器化部署（如 Docker），我们利用 Linux 结合 Docker 的技术原理。
   - **具体手段：** 依托 namespace（命名空间）、cgroup（控制组）和定制镜像等技术点，实现无侵入性的故障注入。
     - **namespace：** 提供进程、网络等资源的隔离能力。
     - **cgroup：** 细粒度控制 CPU、内存、I/O 等资源，模拟资源限制故障。
     - **镜像：** 构建预置故障注入工具或配置的定制容器镜像。
3. **语言特性**
   - **应用层关联：** 这项技术主要与应用层相关。
   - **多语言兼容：** 针对特定编程语言（如 Java），通过利用其无侵入性的字节码增强技术，实现应用层的故障注入。
     - **Java 字节码增强：** 在不修改源代码的情况下，在运行时修改或增强应用程序的行为，以注入延迟、异常等故障。
4. **协议特征**

- **通信协议：** 针对不同类型的通信协议（如 RPC 私有协议、gRPC 协议或其他各类协议，包括当前流行的大模型协议），利用其自身特征进行故障注入。
- **无侵入注入：** 深入理解协议细节，能够帮助我们开展无侵入性的协议层面故障注入。
- **RPC/gRPC 协议：** 模拟 RPC 请求/响应的延迟、错误码、连接中断等。
- **其他协议：** 根据具体协议规范，在请求/响应或连接建立过程中进行故障模拟。

综上，我们底层故障注入技术的主要实现就是围绕这四个核心点展开。

监控度量

监控度量

本图展示了混沌工程实验的监控数据，旨在评估实验效果并验证系统恢复能力。

- **实验概览：** 本次为 CPU 压力测试实验，运行时长约 43 秒，实验状态显示"已恢复"，表明系统在故障注入后成功恢复正常运行。
- **报告名称：** 7-stress-any-202404241407，表明这是一次在 2024 年 4 月 24 日执行的通用压力测试报告。
- **创建时间：** 2024-04-24 14:07:22。

- **实验名称：** tress_cpu_0424_1-复制一复制，推断为一次 CPU 压力测试的副本。
- **故障类型：** CPU，明确了故障注入的类型。
- **运行时间：** 42751ms（约 42.75 秒），表示实验的持续时间。
- **实验状态：** 已恢复，表示在故障注入后，系统成功地恢复了正常运行。
- **实验日志：** 从 2024-04-24 14:05:26 的“未执行”到 14:06:39 的“执行中”，最终在 14:07:22 达到“已恢复”状态，整个过程的时间戳清晰地记录了实验的生命周期。
- **实验对象：** podName: 932536-948b9f4fb-6pq5c，表明实验是针对 Kubernetes 环境中特定 Pod 进行的。

- **核心度量指标图表：** 幻灯片下半部分展示了实验期间关键资源（CPU 和内存）的动态变化图表。所有图表的横轴均表示时间（从 14:04 到 14:28），纵轴表示不同的度量指标。被监控的 Pod 名称统一为 932536-948b9f4fb-6pq5c-91050-0-932536。
  - **CPU Usage (CPU 使用率) - 左上角图表：**
    - 纵轴范围：0% 到 125%。
    - 内容：实验前期（约 14:04-14:05）CPU 使用率接近 0%。在 14:06 左右，CPU 使用率骤升至 100%并维持一段时间后迅速回落。约 14:26 后，CPU 使用率再次飙升至接近 100%。这可能指示实验中的两次 CPU 压力注入或服务负载的异常波动。
  - **CPU 使用率【100%最高】- 右上角图表：**
    - 纵轴范围：0% 到 125%。
    - 内容：该图表趋势与左上角 CPU Usage 图高度一致，可能强调了实验的目标是使 CPU 达到 100%的利用率。图中显示在 14:06-14:
  - **Memory Usage (内存使用率) - 左下角图表：**
    - 纵轴范围：0% 到 125%。

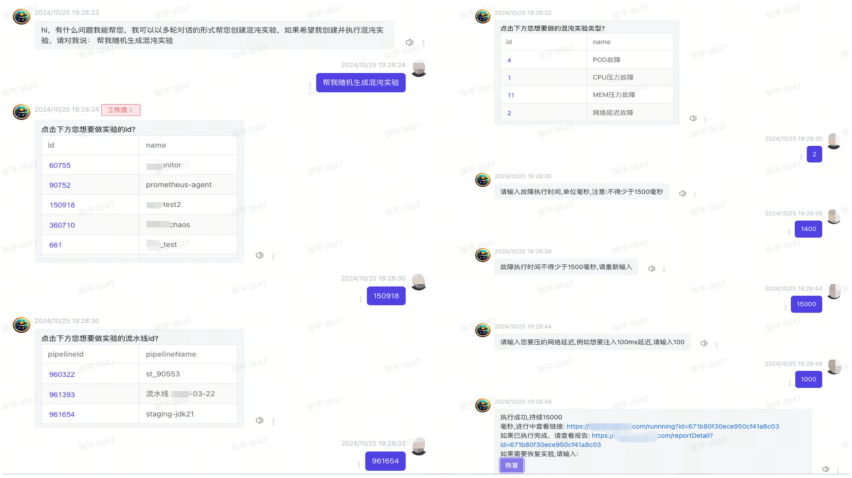
- 内容：实验期间，内存使用率维持在较低水平，没有出现大幅波动，表明 CPU 压力测试未显著影响内存资源。

- **Memory Usage (内存使用率) - 右下角图表：**
  - 纵轴范围：0% 到 125%。
  - 内容：此图表与左下角内存使用率图表趋势相同，进一步确认了内存资源在实验期间的稳定性。

通过这些监控度量，我们可以直观地判断混沌实验是否达到预期效果，评估系统在遭受诸如 CPU 压力这类故障时的行为模式，以及其从异常状态中恢复的能力，确保整个系统链路的韧性和可控性。

混沌工程的智能化

智能化



在混沌工程领域，我们历时约一个季度进行 AI 方向的探索，主要围绕三个环节展开：场景与节点选择、混沌实验实施、效果评估。

用 AI 帮我们“找该打谁”——场景与节点识别

- 先让大模型“读懂代码”：基于语法树等结构化特征，分析代码依赖关系和潜在风险/瓶颈，把有问题的系统、链路、节点列出来，作为优先混沌对象。
- 再让模型“读懂生产”：结合线上调用链和历史故障数据，识别哪些节点在真实环境中出过事、因为什么出事，这些都会被标记成必须重点验证的点。

- 由于故障类型非常多，我们把故障抽象成二十来类（比如依赖链路问题、架构设计问题等），在大模型做理解的基础上，用小模型做分类训练，最终故障识别准确率能做到 90%+，同时模型规模只有几 B 级，成本很可控。

用 AI 帮我们“排练怎么打”——实验规划与注入

- 第一阶段，我们尝试过纯对话式的“随机选节点做混沌”，实验发现结果不够稳定、价值有限。
- 第二阶段（今年初开始），我们把前面“代码 + 调用链 + 故障分类”的结果交给模型，让它自己按思维链的方式去规划一个季度要做的混沌实验：例如一个 Q 要做 8 次，它会给出每次建议打哪些系统、哪些链路、对应什么故障类型。
- 模型只负责规划和生成方案，我们要求必须人工 review 一轮：不合理就退回重生成，合理就“一键通过”，后续调度、通知和落地流程自动跑。这一块把原来全靠人脑整理“打谁”“怎么打”的工作大幅减轻了。

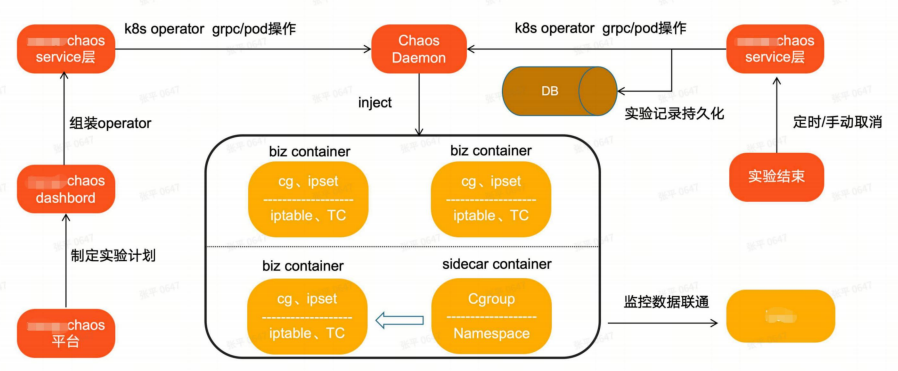
用 AI 帮我们“算这仗打得值不值”——效果评估与报告整理

- 在效果评估上，我们也在让模型参与混沌实验结果的梳理和总结，包括对实验过程、影响范围和指标波动的归纳。
- 相比前两块，这部分技术难度相对可控，更多是在已有数据和报告的基础上做自动化整理和提炼，进一步减少人工写报告的时间

同时我们也做了一个对话式交互的智能混沌实验平台，旨在通过自然语言引导用户创建和执行混沌实验，降低门槛， 如上图所示，

混沌工程平台核心工作流程

核心工作流程



混沌工程平台核心工作流程可以概括为**控制面与数据面协同，构建故障注入与验证能力。**

1. 数据面：故障注入核心

- **Chaos Daemon**：通过 DaemonSet 方式部署，确保每个节点仅运行一个实例。
- **无侵入性注入**：以无侵入的方式，通过操纵业务容器（biz container）的底层资源控制机制进行故障注入：
    - **资源限制**：通过 cgroup（Control Group）对 CPU、内存、I/O 等资源进行限制，模拟资源耗尽场景。
    - **网络故障**：利用 ipset、iptables 和 TC (Traffic Control) 模拟网络延迟、丢包、连接拒绝等网络异常。
- **Sidecar 容器的混沌支持**：鉴于微服务架构中 Sidecar 模式的广泛应用，Chaos Daemon 同样支持对 Sidecar 容器进行注入，通过 cgroup 和 Namespace 等机制，模拟 Sidecar 自身的故障，例如资源隔离或网络代理问题，从而全面评估服务网格、API 网关等组件的韧性。

2. 控制面：实验编排与管理

- **Chaos Platform 与 Chaos Dashboard**：用户通过 Chaos Platform 入口，在 Chaos Dashboard 上直观地制定混沌实验计划，包括选择目标、定义故障类型、设置注入时长和频率等。
- **Chaos Service Layer**：接收并解析来自 Chaos Dashboard 的实验计划，将其转化为可执行的 Operator 指令。
    - **Kubernetes 交互**：通过 gRPC 或 Pod 操作与 Kubernetes (k8s) Operator 进行通信，调度和管理混沌实验的生命周期。这包括实验的部署、启动、监控和终止等。
    - **实验控制**：负责定时或手动取消正在进行的混沌实验，确保实验的可控性和安全性。
- **监控与持久化**：
    - **监控数据联通**：在整个实验过程中，平台会持续收集各类监控数据，包括业务指标、系统性能数据和故障注入效果，为实验结果的分析提供依据。
    - **DB (Database)**：所有实验记录、配置和结果都会被持久化存储在数据库中，方便后续的审计、复盘和知识积累。

3. 核心目标：提升服务韧性

此流程旨在通过端到端、自动化、无侵入的故障注入能力，使服务能够在微服务 2.0 甚至 3.0 阶段，面对日益复杂的 Sidecar 场景时，进行有效的故障演练，从而持续提高系统的故障抗性，确保业务的连续性和稳定性。

02
全链路压测

全链路压测的定位与频率

全链路压测是 SRE 领域的核心实践之一，它模拟真实用户场景和流量模式，对系统所有依赖组件进行连续、高并发的压力测试，以发现系统瓶颈、验证系统容量、评估系统稳定性，并确保系统在高负载下的正常运行。

- **频繁的发布需求**：我们有大量的发布活动，例如今晚即将进行的一场，这使得我们必须频繁地进行全链路压测。
- **常态化压测**：为应对高频率的发布需求，确保系统持续稳定，我们几乎每个月都会进行多次全链路压测，将其作为常态化的 SRE 工作流程。

全链路压测的四大能力

核心能力

高拟真流量：真实流量录制，百万级并发 压测引擎

全场景编排：多协议全场景覆盖，全链路串并结合

一站式服务：报告详细，打通观测平台，问题快速排查

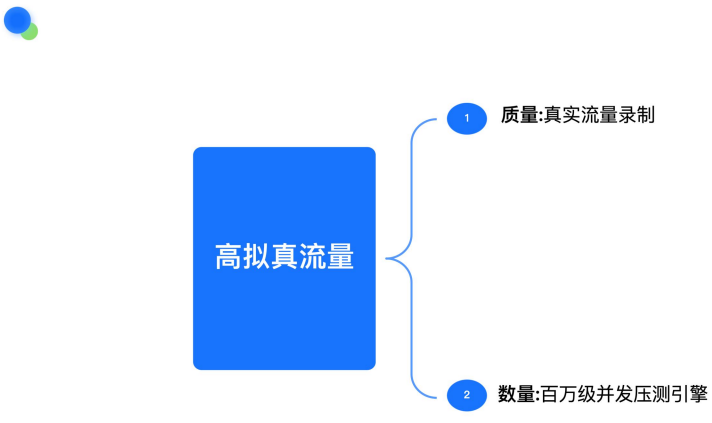
全链路压测是我们 SRE 实践中的核心能力，它从四个关键维度支撑着我们对系统稳定性的极致追求。

- **全场景编排：**
  - 我们的目标是解决复杂业务场景的精准模拟问题，这更像是一种精密的仿真艺术。
  - 鉴于线上业务链路极其复杂多变，传统的压测方法往往难以做到100%还原真实场景。
  - 因此，我们设计的编排能力必须足够可靠，以尽可能逼近或重构线上环境的复杂性。
- **高拟真流量：**
  - 我们致力于实现流量的“真实性”，使其与生产环境的流量特征高度贴近。
  - 这意味着我们在生成压测流量时，会精心设计其结构、速率和行为模式，使其最大限度地模拟用户真实请求，从而有效反映系统在实际负载下的表现。
  - 虽然旨在高度拟真，但压测流量与生产流量在具体目的和控制机制上仍有区分，比如压测流量可控、可调整，用于发现和验证性能瓶颈，而生产流量则直接承载业务。

- **一站式服务：**
    - 我们的全链路压测平台，不仅仅是一个工具集，更是一个整合了整个企业技术生态的服务枢纽。
    - 它与我们的持续集成/持续部署 (CI/CD) 流水线深度融合，实现了自动化触发和一体化管理；
    - 同时，也与可观测性系统（包括指标、日志、链路追踪等）全面打通，确保压测过程中及结束后，我们能获得完整的性能数据、异常告警和调用链分析，从而实现全方位的监控和问题定位。
    - 这种端到端的集成，旨在提供无缝、便捷的压测体验，将各个环节紧密联动起来，最大限度地提升效率并减少人工干预。
- **全栈自研：**
    - 从底层的架构设计到上层的用户界面，我们所有代码均系自主研发。
    - 这意味着我们拥有对整个压测平台和技术栈的完全掌控能力。
    - 这种自研模式不仅赋予我们高度的定制化和迭代能力，可以根据内部业务需求灵活调整和优化，也确保了我们能够深入理解和解决在压测过程中出现的任何技术挑战。

正是通过这四个维度的协同发力，我们的全链路压测系统能够有效地帮助团队发现潜在的性能瓶颈和稳定性风险，为业务的持续优化和高可用性保障奠定坚实基础。

高拟真流量与一站式服务实践



在核心能力方面，我们着重于应对流量问题。自 2022 年起，所有压测流量均通过录制生产环境流量并进行清洗后获得。为实现真实流量的录制，我们主要采取了以下两大措施：

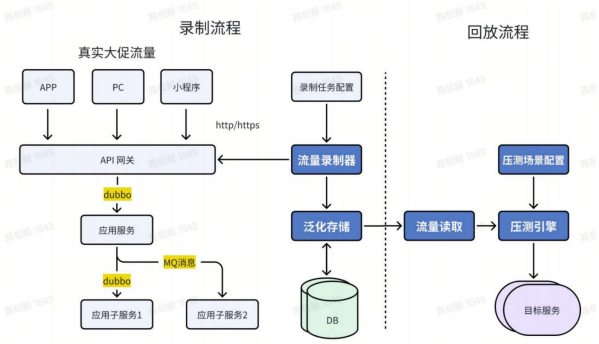
- **网关统一流量采集**：在统一网关层进行流量采集，确保所有入口流量均被准确记录。
- **录制流量清洗**：对采集到的原始流量进行清洗和处理，以去除敏感信息或冗余数据，优化其在压测环境中的可用性。

未来，我们将基于这些能力，进一步发挥“高拟真流量”、“全场景编排”和“一站式服务”的核心优势：

- **高拟真流量**
  - **真实流量录制**：系统能够捕捉并回放实际生产环境中的用户流量，确保测试的真实性和有效性。
  - **百万级并发压测引擎**：系统具备强大的并发处理能力，能够模拟百万级用户并发，进而进行高强度的性能与压力测试。
- **全场景编排**
  - **多协议全场景覆盖**：系统支持 HTTP/HTTPS、TCP/UDP 等多种通信协议，并能覆盖各类复杂的业务场景，提供全面的测试能力。
  - **全链路串并结合**：系统在场景编排时，能够灵活组织测试步骤，支持串行与并行执行，以模拟真实系统中复杂的交互与依赖关系。
- **一站式服务**
  - **报告详细**：系统能够生成详细、全面的测试报告，提供丰富的数据与洞察，协助用户理解测试结果。
  - **打通观测平台**：系统与现有监控及观测平台深度集成，测试数据与系统指标可无缝传输至观测平台，便于统一分析与问题定位。
  - **问题快速排查**：通过详细报告与集成观测平台，加速故障诊断与问题排查的效率，显著提升问题解决速度。

高拟真流量：真实流量录制与回放

高拟真流量：真实流量的录制 + 回放



录制

通过在网关配置流量录制器，对多种协议流量进行泛化解析，实现完整属性提取并存储；

回放

配置选择已存储流量进行压测，压测时可对录制流量进行重新编排，修剪；

录制环节：无侵入式的数据捕获

- **数据采集的无侵入性：**
  - 我们的方案以 Java 技术栈为主，在网关层进行流量采集，这使得我们能够便捷地利用语言特性实现无侵入式采集。我们仅针对协议本身的数据进行标准化捕获，而非应用数据，以规避对业务逻辑的干预。
  - 通过泛化解析 HTTP/HTTPS、Dubbo、MQ 消息等多种协议，我们能够提取流量的完整属性，并将其存储于泛化存储和数据库中。
  - 这种设计显著降低了对系统性能的损耗，确保采集过程的高效稳定。
- **录制流程图解：**
  - **流量来源：** 汇集来自 APP、PC 和“小程序”等多个前端应用的真实流量。
  - **API 网关：** 所有流量首先在 API 网关处聚合，并此处处理 HTTP/HTTPS 协议。
  - **服务调用链：** 流量从 API 网关流向“应用服务”，可能通过 Dubbo 协议调用“应用子服务 1”，或通过 MQ 消息流转至“应用子服务 2”。
  - **流量录制：“流量录制器”**在 API 网关处捕获流量，并将采集到的原始数据传输给**“泛化存储”**。

- **数据持久化：“泛化存储”**与**“DB（数据库）”**双向交互，确保所有流量数据的可靠存储。

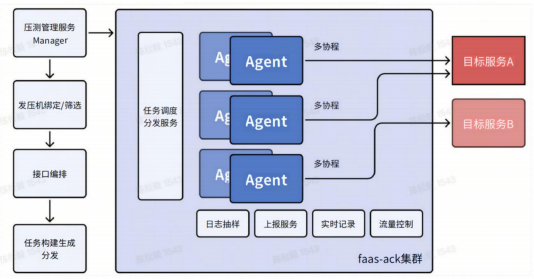
回放环节：精准的压测与场景验证

- **流量回放的灵活性与效能：**
  - 回放阶段允许我们从已存储的流量数据中进行精确筛选与配置，以便进行定制化的压测。
  - 在此过程中，我们具备对录制流量进行重新编排和修剪的能力，以适应不同的测试需求和场景验证。
  - 通过模拟生产环境的负载和行为，我们能够高效执行压力测试或进行场景验证，确保系统在高并发下的韧性。
- **回放流程图解：**
  - **数据源：“泛化存储”**作为回放流量的数据中枢，数据由**“流量读取”**组件提取。
  - **压测配置：“压测场景配置”**模块指引**“压测引擎”**。
  - **模拟请求：“压测引擎”**整合从“流量读取”获取的数据流，精准地向**“目标服务”**发起模拟请求，以验证其性能和稳定性。

最终，该系统通过**高拟真流量录制**，捕获生产环境的真实数据流，并通过**灵活的流量回放机制**，在测试环境中精确模拟真实负载，以此强化了系统的韧性与可靠性。

分布式压测引擎架构与能力

高拟真流量：分布式压测引擎的高性能、高可用



Faas架构

faas云原生压测机集群， 通过事件驱动，实现分布式任务执行的全托管， 用顺序、 分支、并行等方式编排分布式任务，动态启停， 自由扩缩容；

协程

基于jak21协程能力的压测引擎，轻量化执行高并发任务，最大支持百万级并发能力；

一、核心理念与架构概览

分布式压测引擎聚焦于高拟真流量生成能力，确保压测结果的精准性，同时追求高性能与高可用性。架构分为三个核心区域：

- **流程控制区：** 压测管理服务负责任务统一管理、发压机资源分配、接口编排及任务分发。
- **核心执行区（Faas-ACK 集群）：** 任务调度分发服务接收压测任务，多个 Agent 实例利用协程并行处理能力对目标服务发起压力，同时提供日志抽样、结果上报、流量控制等辅助功能。
- **目标服务区：** 支持多个目标服务的并行压测。
- **左侧流程控制区：**
    - **压测管理服务 (Manager)：** 作为整个压测任务的入口和中枢，负责任务的统一管理。
    - **发压机绑定/筛选：** 根据具体的压测需求，智能选择并分配合适的发压机资源。
    - **接口编排：** 提供灵活的接口编排能力，支持定义和组合压测所涉及的接口调用顺序和复杂逻辑。
    - **任务构建生成分发：** 最终将构建好的压测任务打包并分发到实际的执行单元。
- **中间核心执行区 (faas-ack 集群)：**
    - **任务调度分发服务：** 接收并高效调度来自管理服务的压测任务。
    - **Agent 单元：** 多个独立的 Agent 实例构成了压测任务的核心执行者。每个 Agent 都充分利用“多协程”的并行处理能力，协同对目标服务发起压力。
    - **下方附属功能模块：** 涵盖了日志抽样、结果上报服务、实时数据记录及流量控制等关键辅助功能，确保压测过程的全面性与数据的准确性。
- **右侧目标服务区：** 清晰展示了压测目标，例如 Agent 1 和 Agent 2 的协程指向“目标服务 A”，而 Agent 3 的协程则指向“目标服务 B”，体现了引擎对多个目标服务并行压测的能力。

二、技术支柱：Faas 架构与协程技术

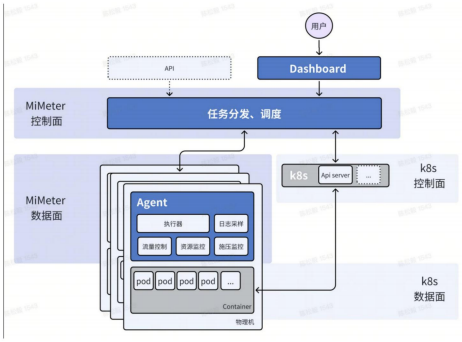
为了实现高性能与高可用，我们的分布式压测引擎主要依赖于两大技术支柱：

- **Faas 架构（云原生压测机集群）：**
    - **事件驱动：** 任务执行采用事件驱动模式，而非传统的常驻进程，极大地提升了资源利用率。
    - **全托管：** 分布式任务的执行被完全托管，简化了用户操作和运维负担。
    - **灵活的任务编排：** 支持顺序、分支、并行等多种复杂的任务编排方式，以应对多样化的压测场景。
    - **动态启停与扩缩容：** 压测任务和关联资源能够根据需求动态启停，并实现容量的自由伸缩，完美解决了压测流量巨大导致的自身弹性扩缩问题。
- **协程（Goroutine）技术应用：**
    - **基于 JDK 21 协程能力：** 作为底层的加速器，压测引擎利用 JDK 21（或此前的虚拟线程 Preview 版）提供的协程能力。早在 JDK 21 正式发布前，我们已率先将 Java 虚拟线程（Loom 项目）引入平台，并随后升级至 JDK 21 正式版。
    - **轻量化执行高并发任务：** 协程的轻量级特性使得引擎能够高效地执行海量并发任务，显著提升了并发处理能力。
    - **支持百万级并发：** 明确指出，得益于协程技术，引擎的最大并发能力可达百万级别，充分展现了其卓越的性能。

通过上述流程层、架构层与技术层的精妙设计，我们的分布式压测引擎得以在亿级设备与千万用户规模下，实现高拟真流量的并发生成，为业务稳定运行提供了坚实保障。

百万级并发压测能力实践

分布式压测引擎：百万级并发的秘密



自研 Faas 平台

- 动态扩缩容，秒级响应
- 施压机状态管理，心跳控制

超强 Agent

- 行业首个 jdk21 协程，更轻量化
- thread-per-request
- 高压下压测数据异步分析
- 资源动态监控

一、核心能力：百万级并发的秘密武器

- **自研 Faas 平台：动态扩缩容与精细化管理**
    - **动态扩缩容，秒级响应：** 平台具备弹性伸缩能力，可根据压测任务需求在秒级内迅速调整资源配比，确保在高并发场景下的资源供给及时性与充足性。
    - **施压机状态管理，心跳控制：** 通过实时心跳机制，系统能够有效监控所有施压机的运行状态与健康状况，并进行统一管理，为稳定压测提供坚实基础。
- **超强 Agent：轻量化、高并发与异步分析的融合**
    - **行业首个 JDK21 协程，更轻量化：** Agent 率先引入 JDK21 虚拟线程技术，显著降低了线程模型开销，以更轻量级的并发原语提高资源利用率，从而在有限资源下支持更高的并发量。
    - **Thread-per-request：** 采用经典的请求处理模型，将每个请求映射至独立的线程（或协程），在保证请求隔离性的同时，简化了处理逻辑并提升了并发处理能力。
    - **高压下压测数据异步分析：** 即使在高负载压测场景下，Agent 也能以异步方式高效收集并分析压测数据，避免对核心压测任务造成阻塞，确保数据完整性与分析效率。

- **资源动态监控：** Agent 内置实时资源监控能力，能够动态捕捉自身 CPU、内存、网络等关键资源使用情况，为压测过程中的性能瓶颈诊断与资源优化提供数据支撑。

二、技术架构：分层协作，保障高效运行

幻灯片左侧的架构图清晰展现了分布式压测引擎的模块构成与协作机制，核心划分为控制面与数据面：

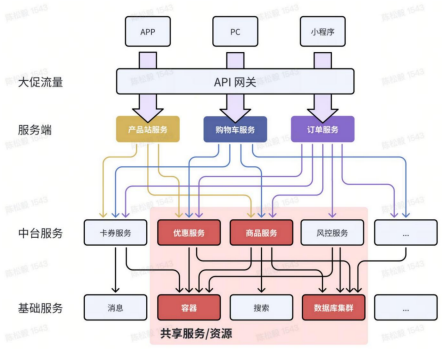
- **MiMeter 控制面：统筹全局的智慧大脑**
  - **Dashboard：** 作为用户与系统交互的统一入口，提供直观易用的界面，方便用户配置、管理和查看压测任务。
  - **API：** 对外提供标准化的编程接口，承载 Dashboard 与后端服务之间的通信，实现命令下发与状态查询。
  - **任务分发、调度：** 核心控制模块，负责接收用户提交的压测任务，并根据预设策略与资源状况，将其智能分派给数据面的 Agent 实例，并进行全局调度与协调。
- **MiMeter 数据面：实际执行压测的强劲引擎**
  - **Agent：** 分布式部署在 PaaS 基础设施上的核心执行单元，负责承载具体的压测逻辑。
    - **执行器：** 负责运行压测脚本或模拟用户行为，生成实际的请求流量。
    - **日志采集：** 实时收集压测过程中 Agent 产生的各类日志信息，为问题排查提供依据。
    - **流量控制：** 根据压测配置，精确控制请求的发送速率与并发度。
    - **资源监控：** 持续监控 Agent 自身的资源消耗，确保其稳定运行并及时发现异常。
    - **施压监控：** 实时监测压测性能指标，如请求成功率、响应时间等。
- **PaaS 基础设施：构建 Agent 运行的容器化环境**
  - **多个 pod：** Agent 实例以 K8s Pod 的形式部署，天然具备高可用与弹性伸缩能力。

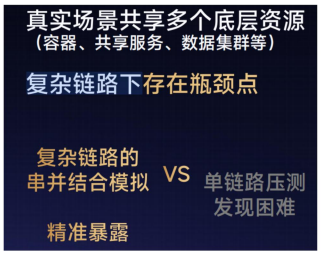
- **Container：** 每个 Pod 内部封装一个或多个容器，运行 Agent 及其依赖。
- **物理机：** 容器最终运行在底层的物理服务器上，提供计算与存储资源。

- **K8s 控制面：统一资源调度的基石**
  - **K8s Api server：** 作为 Kubernetes 集群的管理接口，MiMeter 控制面的任务分发、调度模块通过此接口与 Kubernetes 集群进行交互，实现 Agent Pods 的动态创建、调度、管理与生命周期控制。
- **K8s 数据面：承载 K8s Pods 的运行时环境**
  - 实际承载 Kubernetes Pods 的基础设施，对应于我们的物理资源池，确保 Agent 能够被高效部署并稳定运行

全场景编排与复杂链路仿真

全场景编排：复杂链路的串并结合模拟





全场景编排通过串联与并行结合的模拟方式，精准定位系统性能瓶颈，这在微服务架构下的压力测试中至关重要。

核心挑战与方案

- **全场景编排的重要性：**在高并发业务场景下，不能仅关注单一服务或链路，而需从全局视角进行编排与模拟。
- **复杂链路的挑战：**真实场景中的服务调用链条往往极为复杂，涉及诸多共享底层资源（如容器、共享服务、数据库集群等），极易形成瓶颈。

- **串并结合模拟的价值**：提出通过“复杂链路的串并结合模拟”更有效地发现瓶颈，以弥补传统“单链路压测”在复杂场景下发现瓶颈的不足。
- **精准暴露瓶颈**：此模拟方法的终极目标是实现系统性能瓶颈的“精准暴露”，从而支撑面向问题的优化。

如图所示，架构图（左侧）细致描绘了“大促流量”下的系统概览，包含分层的服务体系：

- **访问层**：作为用户入口，通过 API 网关承接来自 APP、PC、小程序等请求。
- **服务端**：网关之下，核心业务服务如产品站、购物车、订单服务在此层。
- **中台服务**：提供卡券、优惠、商品、风控等细粒度业务能力。
- **基础服务**：为上层服务提供基础支撑，包括消息、容器、搜索、数据库集群等。

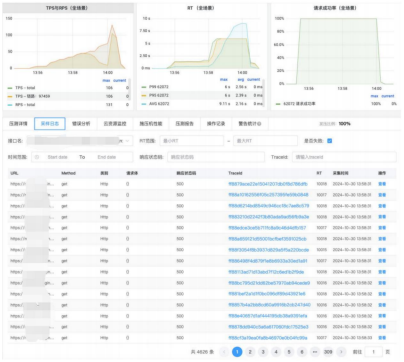
各服务间的调用关系清晰可见：API 网关请求分发至服务端，服务端调用中台服务，中台服务则调用基础服务。在基础服务层，被红框标注的“容器”、“搜索”和“数据库集群”明确标识为**共享服务/资源**，这表明它们是多个上层服务共同依赖的关键资源，也是潜在的性能瓶颈点。图左侧的“大促流量”标识，进一步强调了模拟高并发场景的重要性。

- **真实场景的特点与问题**：
    - 真实场景中，多个底层资源（如容器、共享服务、数据库集群等）常常被共享。
    - 在复杂链路下，瓶颈点普遍存在。
- **模拟方法对比**：
    - **复杂链路的串并结合模拟**对比**单链路压测**。
    - 单链路压测在复杂场景下发现瓶颈存在“发现困难”的局限性。
    - 串并结合模拟则能实现瓶颈的“精准暴露”。

全场景编排实际上是一种仿真技术。从技术实现角度而言，其核心在于如何运用有向无环图（DAG）来支撑整个编排场景。在小米的实践中，编排场景极为复杂，可能涉及串行、并行乃至串并行结合，甚至需要融入体内仿真等高级特性。这种复杂性对性能提出了极高的要求。我们主要通过有向无环图来构建并支撑复杂场景下串并行机制所需的数据结构。

一站式联动观测与问题排查

一站式服务：联动观测系统，问题快速排查



本页聚焦于"一站式服务：联动观测系统，问题快速排查"，旨在通过集成化观测系统，实现对问题的高效定位与解决。

左侧展示了一个复杂的数据分析界面，详细呈现了联动观测系统的具体实现，右侧总结了该系统的核心能力、关键技术与显著效益。

核心能力与效益

- **对压测流程进行抽样：** 能够针对压力测试数据进行深度分析，准确快速地定位潜在问题。
- **问题排查时效：** 大幅缩短故障恢复时间（MTTR），将问题排查效率从“小时级”提升至“分钟级”。
    - 在实际应用中，观测系统能够在 30 秒内识别出异常，从而实现分钟级的快速响应。
- **效率提升：** 单个项目在压测阶段可节省约 450 小时，显著降低了测试成本与时间投入。

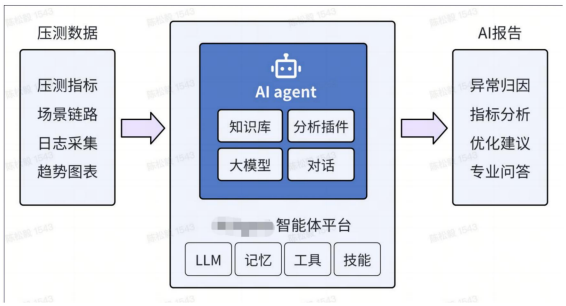
关键技术手段

- **日志采样：** 从海量日志数据中提取关键信息，聚焦核心事件，提升分析效率。
- **流量染色：** 通过标记特定流量，追踪其在系统中的传播路径与行为模式，深入洞察业务流程。

- **数据隔离：** 将与问题相关的数据从庞大数据集中有效分离，确保分析聚焦，减少干扰。
- **AI 分析：** 运用人工智能技术对采集数据进行高级分析，自动识别异常模式与潜在问题，提高预警能力。

一站式异常报告的 AI 智能解析

一站式服务：异常报告AI智能解析



构建压测报告解析AI助手

- 一键分析
- 专家标准
- 对话式问答

"一站式服务：异常报告 AI 智能解析"，核心内容是利用 AI 技术，尤其是结合大模型和智能体平台，实现对异常报告的智能解析，从而构建一个用于压测报告解析的 AI 助手。

混沌工程和全链路压测的执行周期长、迭代频繁，对工程师的时间和精力消耗较大。为此，我们引入 AI 技术辅助压测分析，以提升效率、降低人工成本。

AI 赋能压测，加速问题定位：

- **痛点：** 在压测工作中，我们发现 30% 甚至高达 60% 的时间并非用于执行压测本身，而是耗费在压测后的问题定位上。这一过程不仅效率低下，且常常令人身心俱疲。
- **AI 价值：** 我们力求借助 AI 解决此痛点，以期显著缩短问题查找时间。面对百万乃至千万级别流量的压测，一次性成功绝非易事。AI 的介入，将有效提升我们在迭代压测中发现问题的效率。

AI 智能解析的实现路径：

- **全生态数据打通：** AI 要有效运行，必须与整个系统生态深度融合，实现主数据贯通。这意味着 AI 能够在整个链路中精准定位问题节点，例如在交易链路中

迅速识别是商品服务或订单服务导致压测失败。此举将极大简化问题诊断流程，避免团队成员各自为战、逐一排查系统。

- **深度指标分析与评估：** AI 将对整个拓扑结构中各节点的关键指标进行深入分析，包括：
    - **资源水位：** CPU 使用率。
    - **性能表现：** 各环节的响应时间与吞吐量。
    - **数据库状态：** 数据库连接、查询性能等。
    - **SRE 策略执行：** 熔断、限流和降级策略是否按预期执行并生效。

  AI 将基于这些评估，自动识别潜在问题并给出明确的诊断结论。

AI 助手的核心能力：

- **输入数据全面：** 接收压测指标、场景链路、日志采集和趋势图表等多元数据。
- **AI Agent 核心处理：**
    - **组件：** 深度融合知识库、分析插件、大模型与对话能力，确保分析的广度和深度。
    - **平台支撑：** 基于智能体平台构建，充分利用大语言模型（LLM）、记忆管理、工具调用和特定技能等能力。
- **AI 报告价值输出：** 提供异常归因、指标分析、优化建议和专业问答等核心功能。

未来展望：

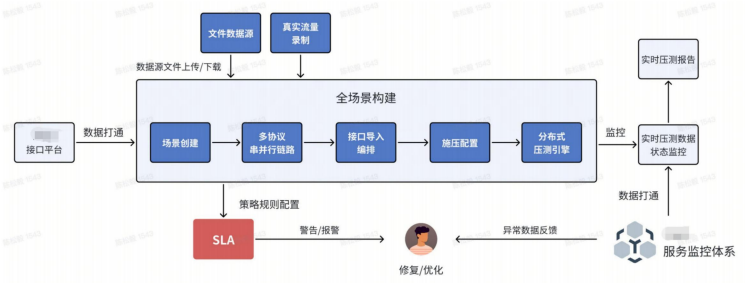
尽管当前这项工作仍在积极推进中，但我们充满信心，相信 AI 辅助压测的模式能有效降低至少 30% 的工程师疲劳程度，显著改善工作体验。因为凭借我们以往的经验，压测的整个周期非常考验团队的耐力与定力。

AI 助手三大关键特性：

- **一键分析：** 简化复杂报告分析流程，实现自动化、高效化。
- **专家标准：** AI 助手将参照领域专家知识进行分析，确保结果的专业性与准确性。
- **对话式问答：** 提供交互式问答界面，使用户能更直观地获取和理解分析结果。

基于 SLA 的全场景性能验证闭环

核心流程



该流程图的核心思想在于通过自动化工具和实时反馈机制，确保系统在面临高并发压力时的稳定性与可靠性，其核心环节可细分为数据准备、场景构建、压测执行与监控、以及 SLA 驱动的反馈优化，形成了持续改进的闭环：

- **数据准备阶段：精准模拟真实流量**
  - 我们通过两种主要途径获取压测所需的数据源：
    - **文件数据源**：直接从文件中导入预设的测试数据。
    - **真实流量录制**：通过录制线上真实流量，高度还原用户行为模式，确保压测的真实性和有效性。
  - 这些数据源随后通过“数据源文件上传/下载”机制，无缝导入到后续的“全场景构建”流程中。
- **全场景构建阶段：灵活编排复杂场景**
  - 这是整个流程的核心环节，旨在构建多样化、高复杂度的压测场景：
    - **接口平台**：与“场景创建”模块通过“数据打通”机制紧密连接，确保接口信息的准确导入与管理。
    - **场景创建**：基于导入的接口，SRE 专家可以灵活地定义和构建具体的压测场景。
    - **多协议串并行链路**：支持多种通信协议，并允许用户配置串行或并行执行的链路，以模拟真实的业务流程依赖和并发行为。

- **接口导入编排**：进一步细化接口的导入与执行顺序，确保业务逻辑的正确性。
- **施压配置**：集中管理压测参数，如并发用户数、持续时间、ramp-up 策略等，确保了压测负载的可控性。
- **分布式压测引擎**：作为压测的实际载体，它能够分发压测任务，执行大规模的压力测试，模拟亿级用户、千万级设备场景下的峰值流量。

- **压测执行与监控：实时洞察系统健康**
  - 由“分布式压测引擎”执行的压测数据，通过“监控”通道实时传输至“实时压测数据状态监控”模块。
  - “实时压测数据状态监控”负责聚合、分析并展示各项性能指标，并最终形成**实时压测报告**，为 SRE 专家提供即时的性能概览。
  - “服务监控体系”与“实时压测数据状态监控”通过“数据打通”密切协作，提供全面的系统健康视图。
- **SLA 驱动的反馈优化：构建韧性系统**
  - 压测结果会与我们 SRE 团队关注的 **SLA（Service Level Agreement）**进行严格对比。
  - 任何不符合 SLA 指标的情况，系统将立即触发**警告/报警**，通知运维或 SRE 人员。
  - SRE 人员收到告警后，将协同“服务监控体系”提供的**异常数据反馈**，快速进行问题的**修复/优化**。
  - 整个流程形成一个持续优化的闭环，确保系统在性能和稳定性上不断迭代提升。

该体系通过精细化的数据准备、场景构建、实时监控与智能反馈，帮助团队在业务发布前充分暴露潜在的性能瓶颈与稳定性风险，并基于 SLA 驱动持续优化系统韧性，确保其能够在预期压力下稳定运行。

（三）总结及展望

本文系统阐述了小米在混沌工程与全链路压测领域的实践经验。

混沌工程方面，我们构建了覆盖基础硬件层、容器层和应用层的故障注入能力，通过 Linux tc/stress、容器技术、字节码增强及协议特征等手段实现无侵入式故障模拟。结合 AI

技术，我们实现了故障场景智能识别（准确率 90%+）、实验自动规划与效果智能评估，大幅提升了韧性测试效率。

全链路压测方面，我们打造了具备四大核心能力的自研平台：基于真实流量录制的高拟真流量生成、支持复杂串并行链路的全场景编排、融合 CI/CD 与可观测性的一站式服务，以及基于 JDK21 协程和 Faas 架构的百万级并发压测引擎。通过日志采样、流量染色和 AI 智能解析，问题排查效率从小时级提升至分钟级。

两大实践相辅相成：混沌工程主动暴露系统薄弱环节，全链路压测验证系统容量与稳定性，共同构成小米 SRE 韧性保障体系的核心支柱，为亿级设备、千万用户规模下的业务稳定运行提供坚实保障。

正所谓：平时多"折腾"，战时少"折腾"。唯有在风平浪静时主动拥抱混乱，方能在惊涛骇浪中从容不迫。

最后，祝各位 SRE 同仁：系统韧性满满，告警寥寥无几；压测一次通过，大促高枕无忧！

5.4.10 小红书基于流量地图的活动保障实践与 AI 探索

SRE Elite 精选原因

小红书以重大活动保障为切入点，围绕流量地图、容量预估、风险前置识别、常态化压测验收等能力，构建了从“看得清、算得准、调得快”到自动化落地的体系化保障方案。

在此基础上，团队进一步将 AI 引入告警分析、历史故障复盘和稳定性指标度量等场景，把活动保障与技术风险治理连接起来，对需要同时建设流量治理、压测验收和智能化风险控制能力的 SRE 团队具有较强参考价值。

（一）背景及设计原则

活动保障现状与挑战

小红书业务概览

小红书作为超大型应用，承载着多元化业务：

- **社区内容与笔记：** 核心社交与分享互动平台。
- **直播业务：** 实时互动和内容形式的拓展。

- **电商平台：** 商业变现和服务闭环能力。
- **强大的推送算法与技术支撑：** 为上述业务提供个性化推荐和高效运营支撑。

在这样的背景下，如何确保这些复杂业务在运营活动高并发、高压力的场景下保持高可用、高性能和高稳定性，是我们需要从容量规划、高可用架构设计、监控告警体系、应急响应机制以及故障演练等多个 SRE 核心维度进行全面审视的课题。

多元化业务活动

小红书平台业务多元，每年大量活动在带来巨大流量的同时，也对平台稳定性保障提出了严峻挑战。

- **CNY（农历新年活动）**
    - 小红书已连续三年赞助春节联欢晚会，每年春晚前一个月，技术风险部都会牵头联合各业务线的研发及质量保障团队，提供全面的稳定性保障。
    - 活动包括"看春晚"、"又见爱人"、"见笑啦 2024"及"繁花来了"等，其中"又见爱人，年末真心话"每晚 8 点独家回应热议，涉及明星或 KOL 参与，旨在引发讨论和互动。
- **社区**
    - 小红书核心是社区，因其特有的社区属性在内容生态上吸引了大量用户。去年"TT 难民"涌入小红书，带来巨大流量机遇的同时也带来了前所未有的技术压力。

- "马路生活节"鼓励用户发掘并分享身边美好，将线上社区的影响力延伸到线下实体消费场景。

- **直播**
  - 小红书直播被用户评价为"最有人味的直播"，每次直播都能带来意想不到的流量高峰。此前的亚运会和赵露思直播都曾达到超过 30 万的峰值在线人数（PCU）。
  - 直播内容涵盖商业带货和娱乐性互动，并能产生社会热点效应。
- **电商**
  - 小红书深度布局电商，推出友好市集等特色电商模式。电商活动通常伴随巨大流量，特别是在 618、双 11 等大型购物节期间带来显著的流量增长。

无论是春节活动还是各业务线特有的活动，小红书都建立了一套完整的活动保障能力和标准操作流程（SOP）。

活动备战 SOP

活动备战 SOP 的核心痛点在于**低效&重复**。我们将活动备战的关键工作细分为以下阶段：

一、活动初期规划与评估 * 绘制活动地图： 涵盖所有相关业务线与核心业务模块，清晰界定活动范围与影响。 * **核心入口拆解：** 各业务线及模块负责人深入拆解并识别承接活动流量的核心入口。 * **服务 QPS 预估：** 针对核心入口及其依赖的关键服务，进行精确的

每秒查询次数（QPS）预估。* **服务容量预测：** 基于 QPS 预估结果进一步进行容量预测，涵盖所有相关基础设施，为资源申请和扩容提供数据支撑。

二、风险管理与资源准备 * **活动风险识别：** 全面识别潜在风险，如新增接口的兼容性问题、关键依赖服务的变化等。* **服务预案配置：** 针对识别出的各类风险，前置配置应急预案，包括洪峰预案和各种故障场景下的降级预案。* **资源申请与扩容：** 根据容量预测结果完成资源申请和横向扩容，并规划活动结束后的资源回收。* **多轮压测验收：** 进行多轮全面的压力测试，不仅验证流量预估和资源配置，更对应急预案进行实战演练和功能验收。

尽管 SOP 相对成熟，但深层次的挑战在于整套流程仍然偏重于依赖**人工经验**进行判断和执行，存在经验不足、人为失误或效率低下等风险，这是我们未来需要持续优化和自动化的方向。

备战现状与挑战

备战的核心问题链条为：活动流量洪峰是多少？→ 哪些业务参与？→ 对哪些服务扩容、扩多少？→ 风险识别与稳定性加固。在实际备战过程中，我们面临三大核心挑战：

- **强依赖经验：**
    - 活动备战高度依赖少数资深技术人员的经验。对于经验不足的新员工，执行与落地的难度增大，备战效率和准确性受限于个体，难以标准化和传承。
- **资源利用低：**

- 由于对经验的过度依赖及保守策略，往往预留远超实际需求的资源。例如某次活动中，最终实际资源使用量仅为预估量的 30%，造成显著的资源浪费。

- **人力投入大：**
  - 备战流程包括 SOP 制定、系统配置及验收等，需要大量技术人员投入。例如某一年为应对最大规模活动，投入了近千名工程师参与备战，凸显了当前流程自动化程度不高的现状。

这三大挑战相互关联，如何实现更高效、更经济、更少人为干预的活动备战，是我们亟需解决的核心问题。

（二）体系设计及关键流程

流量地图建设方案

建设背景与动机

面对活动保障中投入资源巨大但效率低下的挑战，我们提出了构建**流量地图**的设想，并将其纳入去年的重点规划。流量地图在 SRE 实践中能够：

- **可视化服务间调用关系与数据流向：**清晰呈现系统内部的复杂依赖架构。
- **辅助故障排查与根因分析：**快速定位潜在问题点，提高故障响应效率。
- **支持容量规划与性能优化：**通过流量模式分析为资源分配和性能改进提供数据支撑。

- **发现"风险"依赖**：识别不明显但对系统稳定性有关键影响的隐藏依赖。

其核心在于通过一套可观测性工具将整个系统运行脉络具象化，将传统的活动保障模式转化为更高效、更具韧性的 SRE 实践。

活动保障全流程

活动保障全流程分为五个关键步骤：

Step 01: 地图编排

构建全面、精确的业务拓扑视图和流量地图：* 基于各业务域的链路数据，自动生成单业务的详细拓扑结构。* 将多个单业务域的拓扑整合，绘制出全局级的流量地图。* 尤其关注涉及核心保障（重保）的跨域服务和所有新增 API。

Step 02: 流量预估

基于活动特性和系统数据，精准预测活动期间的流量峰值 QPS：* 深入剖析活动环节，识别不同业务区域的流量爆发潜力及相应系数。* 借助链路数据，分析服务间调用关系，获取流量基准系数。* 将活动预估流量与日常背景流量叠加，自动预测各服务的 QPS 压力。

Step 03: 容量预估

评估现有系统容量并规划扩容资源：* 基于预估 QPS，结合线上服务的容量水位及性能拟合曲线进行精确计算。* 利用智能算法自动推导出活动所需的精确服务扩容资源量。

Step 04: 风险识别

主动识别并评估潜在系统风险：* 自动巡检链路依赖及调用系数的潜在变化环节。* 对风险点依据高可用架构规范进行全面排查。

Step 05: 活动限流 & 压测

通过限流和压测验证并固化系统稳定性保障：*根据预估 QPS 推荐并自动批量配置限流值。*在压测过程中提供实时容量洞察数据，确认系统极限承载能力。*提供快速扩容机制，根据业务需求迅速增加资源。

这五个步骤整合形成的活动保障能力，核心目标可概括为三个层面：**看得清**（全流程透明化、可视化）、**控得住**（风险识别和控制能力）、**调得快**（响应自动化与批量化）。

活动保障建设思路

核心思想：以"流量地图"为驱动，构建精准且响应迅速的活动保障体系。

一、看得清：提升活动的可见性和前瞻性

- **流量地图日常保鲜与活动定制：** 日常持续维护各业务域的拓扑结构与依赖关系，确保流量地图的实时性和准确性；活动筹备阶段基于日常积累进行针对性微调和编排，规避从零开始构建的低效。
- **前置风险识别与协同：** SRE 团队在活动前主动识别新变量及潜在风险；通过平台化汇聚多方数据，打通信息孤岛，确保所有相关方协同保障。

二、算得准：确保预估的准确性和策略的合理性

- **精细化流量与资源预估：** 基于历史数据、指标及算法模型，开展入口流量、各服务流量及服务所需资源的精准预估，消除评估盲点。
- **策略匹配度分析与优化：** 将流量预估结果与现有限流配置进行精确匹配分析，持续摸清系统弹性容量的承载上限，优化扩容策略。

三、调得快：实现响应的自动化与批量化

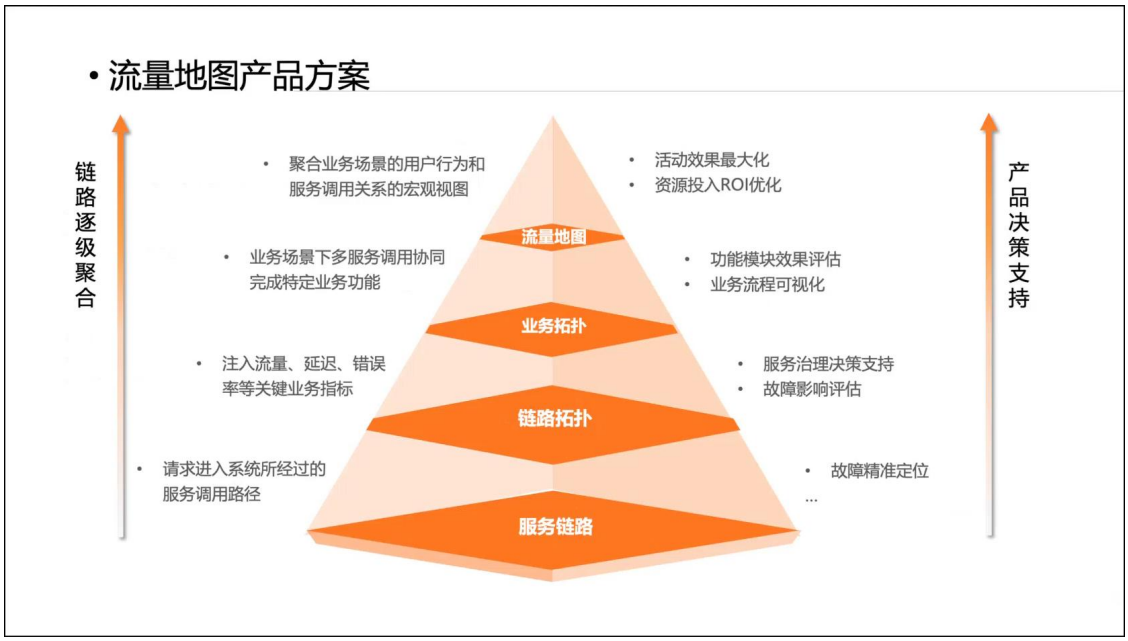
- **自动化资源与限流配置：** 以服务到可用区粒度批量交付扩容清单，实现资源快速弹性调整；当限流配置效果不达预期时批量调优。
- **自动化测试与验证：** 将拆解后的 API 和模拟流量自动导入压测平台，规避人工操作带来的效率瓶颈和潜在失误。

四、体系架构支撑

- **底座能力：** 强大的可观测性数据底座，覆盖业务监控、服务链路追踪以及各类配置与元数据。
- **核心模块：** 流量地图产品能力（地图编排、流量模型、业务拓扑、业务打标等），以及基于流量地图衍生的数据预估能力（入口流量预估、服务流量预估、服务资源预估等）。
- **上层支撑：** 支撑流量拆解、容量预估、风险感知、限流管控、活动观测、流量评估、资源交付、业务诊断、压测验收和数据沉淀等关键环节。

这种架构确保了从底层数据采集到中层能力构建，再到上层场景支撑的完整性与高效性。

流量地图产品方案



流量地图产品方案通过分层聚合，实现对复杂分布式系统的全面可见性管理。核心是从细粒度服务调用到宏观业务视图的逐级聚合过程：

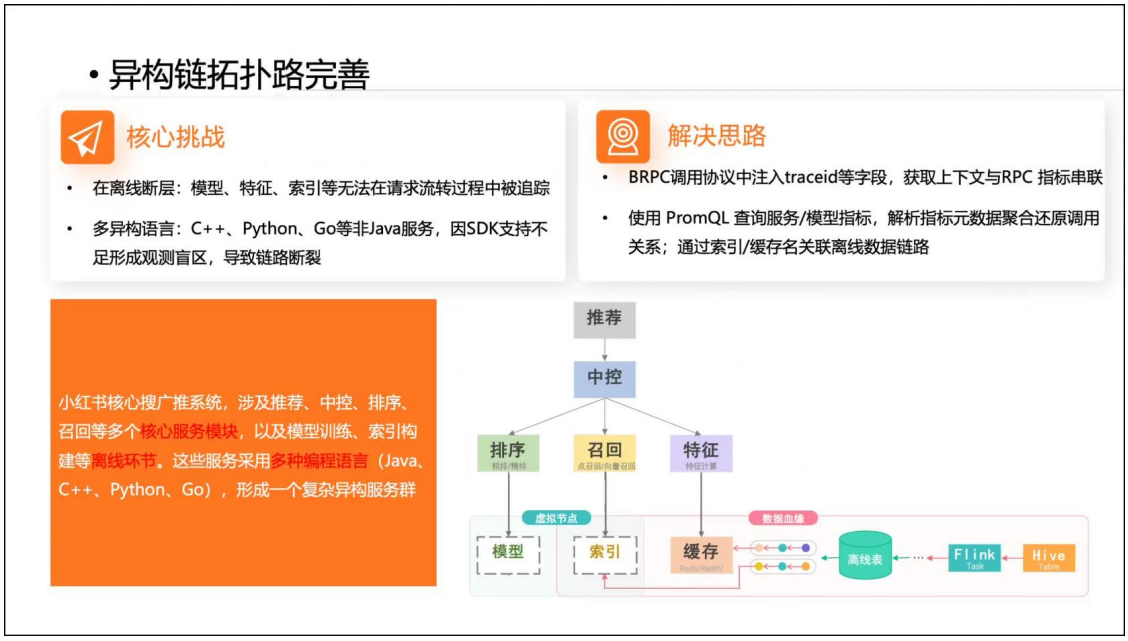
链路逐级聚合（从底层到顶层）：

- **服务链路：** 最底层、最细粒度的信息，记录每一次请求在不同服务间的流转。要求底层链路数据完善准确，为上层提供可靠数据基础。
- **链路拓扑：** 在服务链路基础上，聚合注入流量、延迟、错误率等关键业务指标，形成服务粒度的聚合视图。
- **业务拓扑：** 按照特定的业务入口或业务标签对链路拓扑进一步聚合，形成业务场景下多服务调用协同完成特定业务功能的视图。
- **流量地图：** 将多个业务拓扑依据活动流程进行深度聚合，形成全局的、能够精确描绘用户行为的流量地图，提供整个系统最高层面的视图。

各层级的产品决策支持：

- **服务链路层** → 故障精准定位
- **链路拓扑层** → 服务治理决策支持 / 故障影响评估
- **业务拓扑层** → 功能模块效果评估 / 业务流程可视化
- **流量地图层** → 活动效果最大化 / 资源投入 ROI 优化

异构系统链路追踪



在小红书核心的推搜广系统中，分布式链路追踪面临两大核心挑战：

核心挑战

- **在离线断层：** 推搜广系统包含推荐、中控、排序、召回等在线服务模块，还涉及模型训练、特征计算、索引构建等大量离线环节。离线组件在请求链路流转中难以被有效追踪，导致端到端可见性缺失，影响数据预估准确性。

- **多异构语言：** 系统中广泛使用 C++、Python、Go 等非 Java 语言开发的核心服务。现有 SDK 对异构语言支持不足，导致链路断裂，形成观测盲区。

解决思路

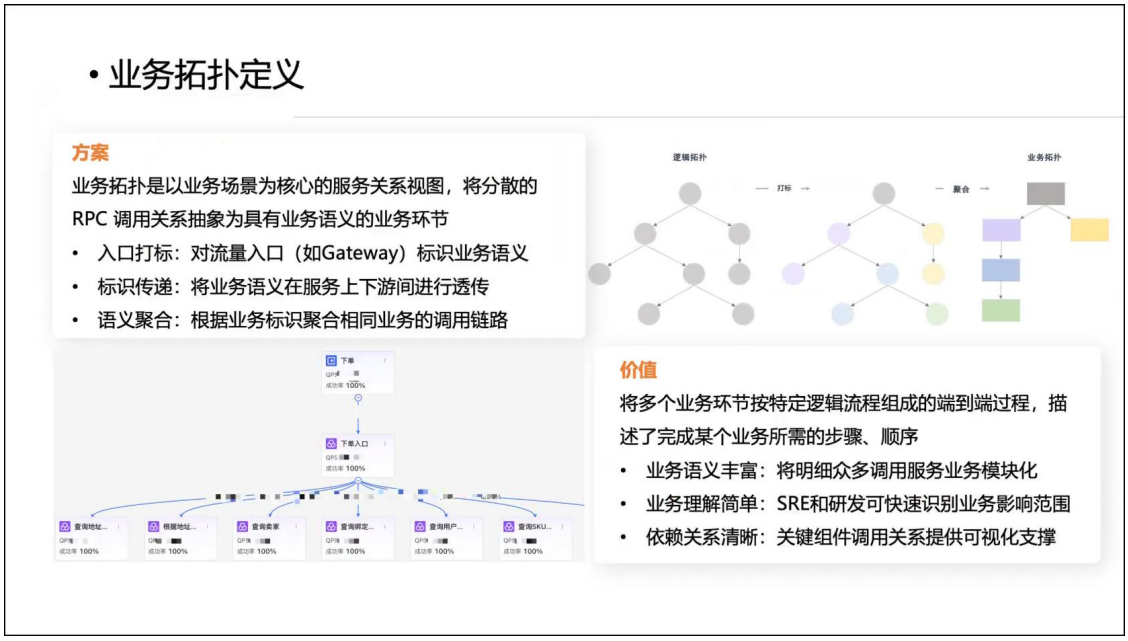
- **BRPC 协议注入 TraceID：** 在多语言异构链路中，通过 BRPC 等 RPC 调用协议统一注入 TraceID 及上下文信息，获取调用数据和 RPC 指标，实现异构语言服务间的调用串联。
- **PromQL 关联离线数据链路：** 对在线服务、模型、特征等模块利用 PromQL 查询指标数据，解析元数据还原服务间调用关系；通过索引名或缓存名（如 Redis/RedKV），将在线请求链路与离线数据链路（离线表、Flink 任务、Hive 表）进行关联，形成完整闭环追踪。

系统架构示意

- **业务主链路：** 推荐流量入口 → 中控服务 → 排序（粗排/精排）；中控服务 → 召回（点召回/向量召回）；中控服务 → 特征（特征计算）
- **虚拟节点与资源依赖：** 排序模块依赖"模型"，召回模块依赖"索引"
- **缓存与数据血缘：** 模型和索引数据来源于缓存（Redis/RedKV），进一步关联到离线表，并流向 Flink (Task)和 Hive (Table)，揭示完整的数据血缘关系

通过这一体系，我们可以清晰追踪从推荐入口到离线数据源的完整链路，实现闭环追踪。完成链路数据采集后，下一步便是基于此定义和构建业务拓扑。

业务拓扑定义



一、方案：构建业务视角的服务关系视图

业务拓扑以业务场景为核心，**将纷繁复杂的 RPC 关系抽象并聚合为具有明确业务语义的业务环节**。核心实现步骤包括：

- **入口打标（Tagging）：** 针对流量入口进行标记，赋予特定业务语义标识。
- **标识传递（Propagation）：** 将业务语义标识在整个调用链路的上下游服务之间透传。
- **语义聚合（Aggregation）：** 基于统一业务标识，将具有相同或相关业务含义的调用链路进行聚合，形成完整的业务环节并赋予业务语义化命名，如"下单"、"支付"等。

演进过程：从底层服务交互构成的**"逻辑拓扑"**，通过**"打标"**，经**"聚合"**，最终形成清晰的**"业务拓扑"**。

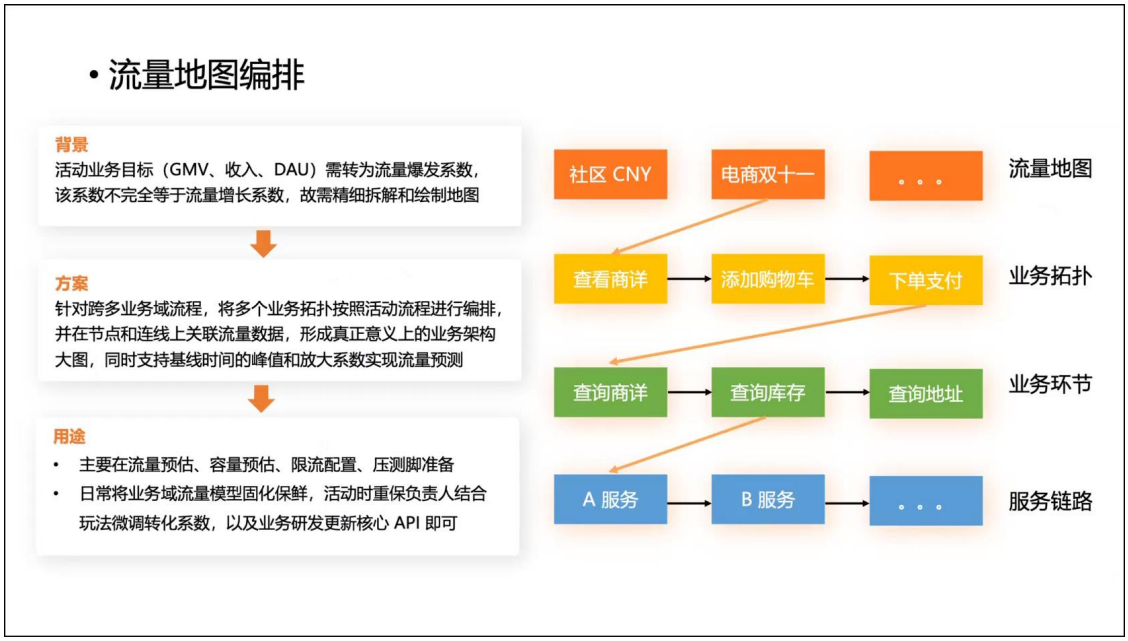
二、价值：提升业务理解、故障定位与协作效率

- **业务语义丰富：** 将数量众多的底层服务调用模块化、抽象化，转化为易于理解的业务组件。
- **业务理解简单：** SRE、研发乃至业务同学都能站在同一视角，快速识别技术问题可能影响的业务范围。
- **依赖关系清晰：** 提供关键组件间调用关系的可视化支持，极大简化问题排查难度。

以"下单"为例：作为核心请求入口，其 QPS 和成功率等指标一目了然；"下单"之下的"根据地址查询运费"、"查询卖家信息"、"查询 SKU 库存"等业务环节被清晰展现，每个环节背后整合了一组底层服务依赖。当出现指标异常时，能够快速、准确判断整个业务影响面。

当前业务拓扑主要聚焦于单一业务域。在面对大型促销活动等跨业务域场景时，还需进一步拓展其能力——这正是流量地图编排所要解决的问题。

流量地图编排



一、背景

我们需要将活动的业务目标（如 GMV、收入或 DAU）精准转化为"流量爆发系数"。这与常规的"流量增长系数"有本质区别，要求精细化拆解并绘制流量地图，深刻理解流量的来源、流向和潜在冲击点。

二、方案：跨业务领域的流量编排与预测

核心思想是将跨多个业务领域的流程聚合，按照特定活动流程进行编排，形成统一的流量地图：*** 业务拓扑编排：** 将一系列相互关联的业务拓扑按活动流程组装，每个节点和连接关联实际流量数据。*** 构建业务架构大图：** 形成全面反映业务流程和技术栈的业务架构大图。*** 流量预测能力：** 支持基于历史数据基线和放大系数，实现对未来流量的精确预测。

三、用途

- 流量预估、容量预估、限流配置、压测脚本准备。

四、动态管理与持续优化

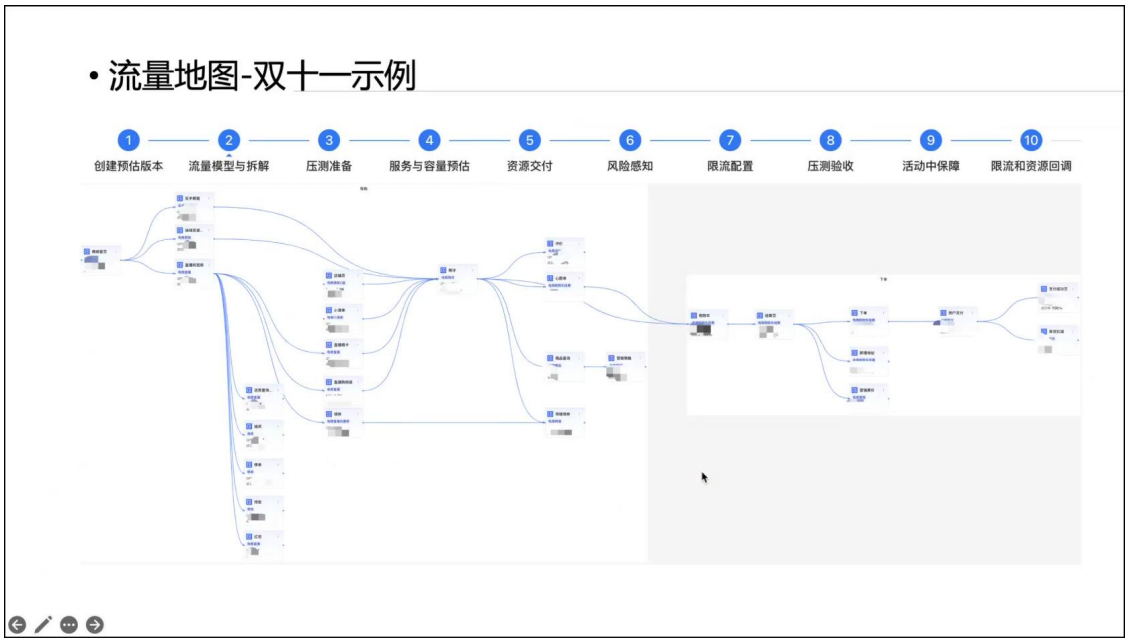
- **日常运营：** 将业务领域的流量模型固态化，保持数据新鲜度和准确性。
- **活动应对：** 结合负责人反馈对转化系数微调，根据业务研发团队更新的核心 API 进行调整。

五、分层结构

- **第一层：流量地图** — 不同业务或活动维度的宏观流量视图，如"社区CNY"、"电商双十一"。
- **第二层：业务拓扑** — 用户在特定活动下的核心业务流程，如"查看商详→添加购物车→下单支付"。
- **第三层：业务环节** — 支持业务拓扑中每个步骤的具体业务操作，如"查询商详"、"查询库存"、"查询地址"。
- **第四层：服务链路** — 支持业务环节的底层微服务调用链。

通过这种分层方法，可以从宏观活动流量一直追溯到微观服务调用细节，为 SRE 团队提供全面洞察力。

流量地图双十一示例



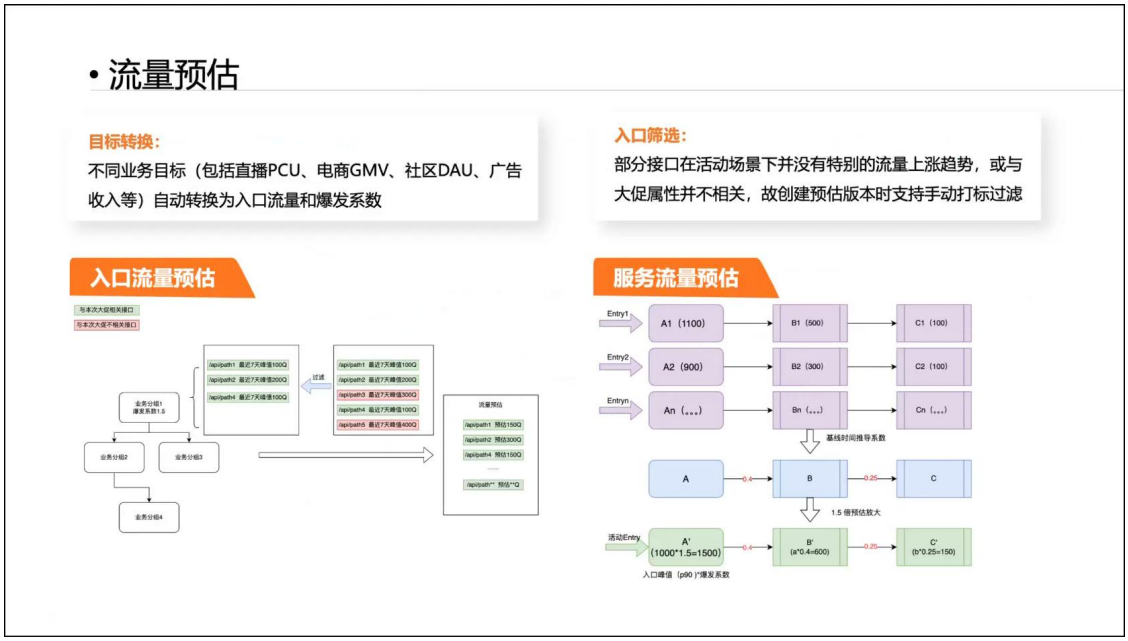
以"双十一"为例，展示流量地图在实际大促中的应用。

核心流量地图以节点和连线形式展示了从用户浏览到完成支付的完整交易链路：

- **用户侧互动与商品浏览：** 用户可能通过直播间进入，查看直播商品、抽奖、查询榜单；查看商卡、领券，加入购物车；进入商城后流量集中于"商城首页"，进一步流向"店铺首页"、"商品详情"。"商品详情"环节伴随调用图片服务、评论、搜索、直播间伴随、问问客服、抢购、优惠、红包等辅助服务。
- **交易核心流程：** 加入购物车 → 结算页（调用新增地址、费用计算等服务）→ 下单 → 用户支付 → 发行成功页、库存扣减 → 物流环节。

有了完整的流量地图，下一步最重要的工作便是基于此进行精确的流量预估。

流量预估



核心预估流程

- **业务目标转化：**将直播 PCU、电商 GMV、社区 DAU 或广告收入等多元化业务目标，通过算法转化为"入口流量"和"爆发系数"，量化高层业务目标对系统请求量的影响，明确流量放大倍数（如 1.5 倍、2 倍、3 倍）。
- **入口流量筛选与预估：**
  - **接口分类与过滤：**并非所有接口在活动中都会经历显著流量上涨，预估版本支持手动标记和过滤不相关接口，避免冗余资源预留。
  - **爆发系数应用：**针对与大促相关的接口，结合其所属"业务分组"应用相应爆发系数（如"业务分组 1 爆发系数 1.5"）。
  - **历史数据过滤与计算：**收集"最近 7 天峰值 100Q"等原始 API 数据，经过滤处理后生成单个 API 的流量预估值。

服务流量预估模型

- **链路聚合：**依据单个业务请求的完整 trace，将完成相同核心业务请求的关联链路聚合。通过分析一段时间内（几万到几十万笔）的请求数据，拟合计算出服务间（如 A 到 B、B 到 C）的流量转化系数。
- **级联预估：**
  - **基线流量推导：**以历史基线时间的流量数据为基础，结合基线时间推导系数调整。

- **活动流量计算：** 将入口峰值(p90)乘以爆发系数（如 1000 * 1.5 = 1500）。
- **后端服务传导：** 活动入口流量按既定的服务间转发比例（如 A'到 B'为 0.4，B'到 C'为 0.25）传导至后端服务，推导出各节点预估流量（B'为 600QPS，C'为 150QPS）。

这一系统化方法将高层级业务目标精准映射到每一层服务组件的 QPS 指标，为资源规划和容量保障提供坚实的数据基础。

流量预估与容量规划工作流

- 用户可灵活设定"基线时间"和"预估系数"等全局参数。
- 针对不同业务场景（如"商家登录"、"买手分销"），支持精细化配置"预估开关"、"流量占比"、"放大系数"及"预估 QPS"。
- 在资源交付阶段，全面覆盖 HTTP 接口、RPC 接口、Redis、SQL、MQ TOPIC、Mongo 和 Elasticsearch 等资源类型的容量规划。
- 界面实时呈现入口 HTTP 服务、活动放大系数、基准平均 QPS、活动预估 QPS 及事件发生时间等关键数据。
- "批量生效"和"调试"功能确保了高效的批量操作与验证能力。

核心优势在于系统能够计算出：核心入口的预估 QPS、每个入口所依赖服务在特定时间段内的基准 QPS、以及活动期间每个服务预估的 QPS。通过这种方式获得精确且数据驱动的预估结果，而非依赖主观的人工估算。

跨域流量处理

跨域流量是备战跨业务活动时容量规划的关键环节。

核心挑战： 识别和处理不同业务域间的重复流量，支持用户灵活调整入口流量占比，避免容量预估偏差。

分析： * **依赖组件的去重：** 某个模块可能同时被多个业务域依赖（如直播购物袋，既是直播机房的后端，也是电商业务的核心场景），其贡献流量需去重处理，避免重复计算导致资源浪费。 * **中台组件的累加模式：** 被多个业务域调用的中台组件，其流量应采用累加模式，不同业务域贡献的流量叠加计算以反映真实容量需求。

方案： * **识别交叉依赖接口：** 系统智能识别活动关联业务线中的交叉依赖接口并突出显示，配置界面中带有"多方依赖"标签及可调整百分比输入框。 * **基于历史数据推荐与手动调整：** 系统提供基于历史数据的流量占比推荐，用户可根据实际情况手动调整（如从100%调整为 30%或 40%），后续流量预估随之更新。

流量地图的保鲜： 我们构建一套可复用、可持续维护的流量地图，在日常运营中持续更新，而非仅在活动备战时临时构建。

地图配置保鲜

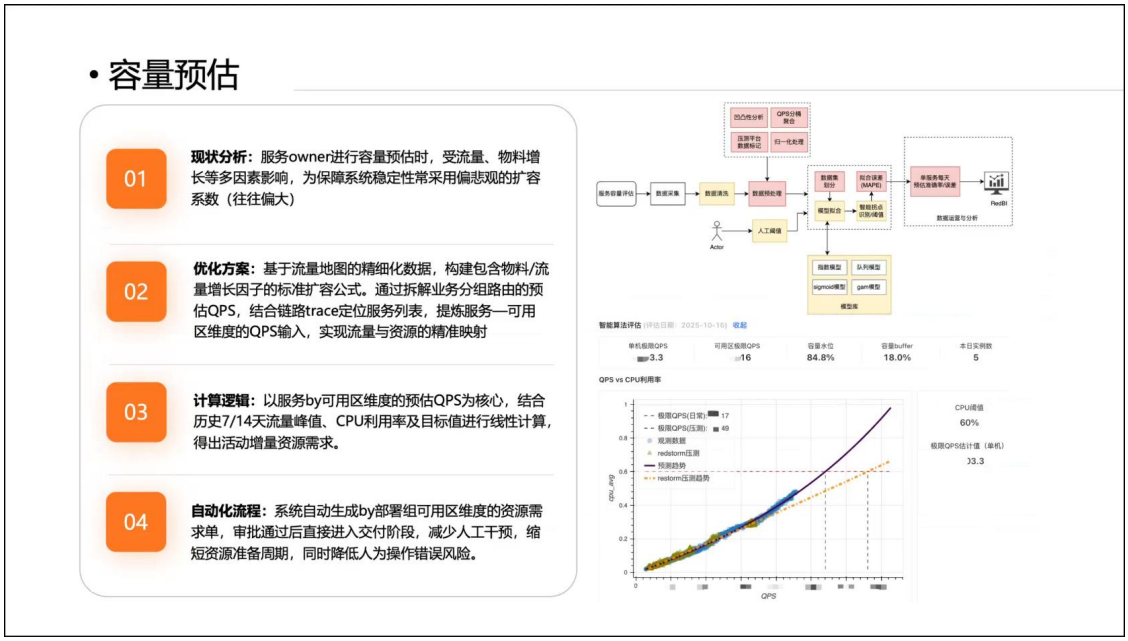
流量地图在实际维护中面临三大常见困境：

- **新需求接口的及时展现问题：** 关键活动期间，尚未上线的新接口如何及时纳入流量地图。
- **低流量高波动接口的估算挑战：** 日常流量极低但活动期间可能爆发增长的接口（如抽奖活动），如何确保预估值反映潜在峰值。
- **业务迭代带来的接口感知与更新延迟：** 接口常常发生变化或新增，流量地图如何实时感知并及时更新。

优化策略：

1. **新生效 API 的自动化异常提醒与周期性周知：**
    - 建立周期性巡检机制，识别并提醒未进行业务打标的新增 API，以周为维度通知相关 SRE 和研发负责人进行打标操作。
    - 从源头规范接口注册，确保每个核心接口拥有健全的业务元数据基线。
2. **流量地图对新增入口的智能展示与手动调权能力：**
    - 允许用户在流量预估阶段对特定接口进行手动权重调整，尤其在活动前接口尚未上线或流量特性与日常迥异的情况下。
    - 配置界面支持标识"新入口"、设置流量百分比、输入权重数值，并支持"批量生效"等功能，实现对流量入口的细粒度控制。

容量预估



一、现状分析

服务 Owner 进行容量预估时，受流量、物料增长等复杂因素影响，往往倾向于采用偏悲观的扩容系数，导致严重的资源浪费。

二、优化方案

基于精细化的流量地图数据，构建包含物料/流量增长因子的标准化扩容公式：* 拆解业务分组路由，精准预估 QPS。* 结合全链路 Trace，定位服务列表。* 提炼服务与可用区维度的 QPS 输入，实现流量与资源的精准映射。

三、计算逻辑

以服务在可用区维度的预估 QPS 为基础，不仅局限于 CPU 与 QPS 的简单计算，更兼顾服务的耗时与成功率：* **预估 QPS：** 结合历史 7/14 天的流量峰值、CPU 利用率，设定合理目标值进行线性计算。* **多维度指标考量：** 除 QPS 和 CPU，还考虑耗时、成功率等关键指标。* **数据处理流程：** 采集多维度指标数据 → 清洗剔除异常尖刺 → 根据不同指标特性选择算法进行模型拟合（如 Sigmoid 模型）。* **产出：** 单机极限 QPS、可用区极限 QPS、容量水位、容量 buffer、本日实例数。

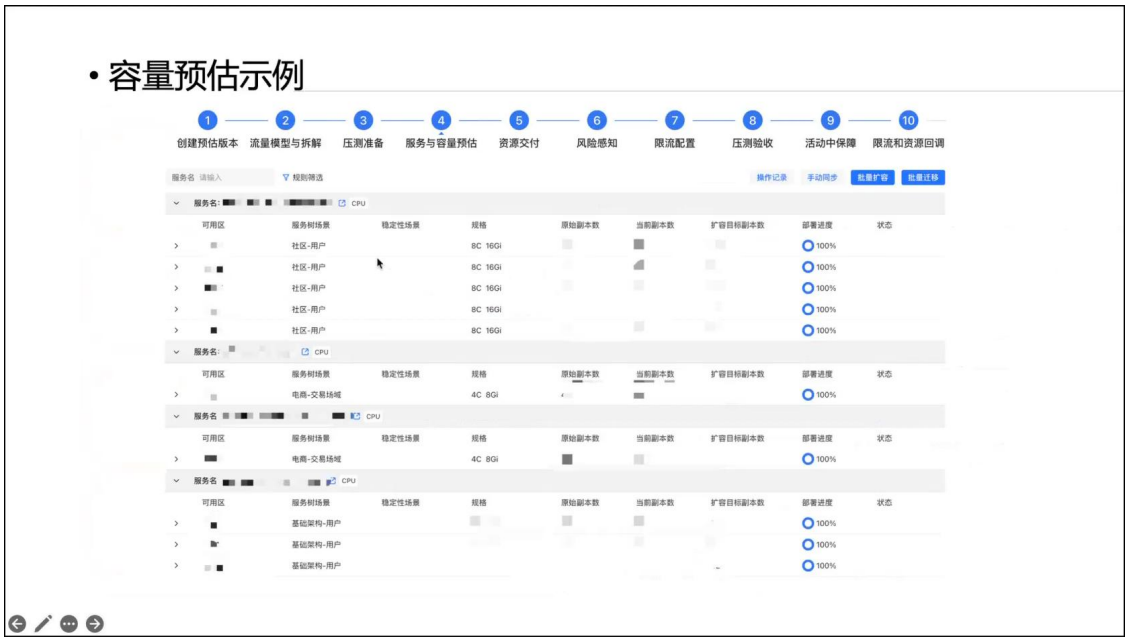
四、自动化流程

- 系统自动生成按部署组、可用区维度的资源需求单。
- 需求单审批通过后直接进入交付阶段，大幅减少人工干预。

五、智能容量预估

- **模型库：** 包含指数模型、队列模型、Sigmoid 模型、GAM 模型等。
- **评估机制：** 通过拟合误差（MAPE）评估模型精度，结合凹凸性分析识别智能拐点，利用压测平台数据标记确保模型基于真实负载数据训练。
- **评估数据示例（2025-10-16）：** 单机极限 QPS: 3.3，可用区极限 QPS: 16，容量水位: 84.8%，容量 buffer: 18.0%，本日实例数: 5。
- **QPS vs CPU 利用率图表：** 展示极限 QPS（日常: 17，压测: 49）、观测数据、Redstorm 压测数据及预测趋势。

容量预估与管理流程



在 10 步全流程的"服务与容量预估"阶段，系统提供了直观的管理界面：

- **便捷筛选与操作：** 通过服务名称搜索和"规则筛选"聚焦特定服务，支持"操作记录"、"手动同步"、"批量扩容"和"批量迁移"等自动化操作。
- **详尽的服务扩容信息：** 表格展示各服务的可用区、服务树场景（如"社区-用户"、"电商-交易场域"）、稳定性场景、规格（如"8C 16Gi"）、原始副本数、当前副本数、扩容目标副本数、部署进度和状态。

当用户确认数据后，系统支持执行"批量扩容"操作——不再需要独立申请资源、单独计算各服务扩容需求并手动执行，极大简化了容量管理流程，有效规避高峰期人工操作可能引入的误差和延误。

限流管控配置

限流管控配置是确保系统在高并发场景下稳定性和可用性的核心，包含三个关键能力：

1. 限流配置数据获取： * **利用流量地图精确预估各核心入口的 QPS。** * **通过深入业务环节的 RPC 接口，实现限流值的精细化预估。**

2. 限流配置自动生成： * **提供多维限流配置组合，根据不同维度（用户、业务场景、接口类型）定义规则。** * **自动生成各种场景下的限流计算模式，配置规则按优先级从高到低匹配。**

3. 限流配置智能调优： * **快速识别限流配置的准确度，分析当前限流值与预期值的偏差。** * **支持批量调优操作，通过动态调整限流阈值实现持续优化与自适应。**

限流配置界面展示了实际管理功能，包括业务分组、路由 Path、edith 集群等筛选维度，表格列出近 7 天峰值 QPS、活动预估 QPS、线上限流值及限流阈值调整目标，涵盖"下单"、"买手分销"、"买手橱窗"、"交易老接口"、"修改地址"等业务场景，支持"批量触发"操作。系统能够主动识别限流配置的潜在异常，并建议相应调整。

常态化压测验收

背景： 传统压测流程痛点突出——业务流量拆解高度依赖人工，QA 团队需导出 API 文件进行压测评审，用例筛选与脚本生成耗时费力。

方案： * 将预估的 API 接口与预期 QPS 数据统一导入压测平台，自动生成用例集合与压测脚本并自动化执行。 * 每次压测完成后沉淀时间范围与水位数据，便于后续追溯与分析。 * 期望在活动启动前仅需完成一次全链路压测即可达到验收标准，极大节省人力成本。

价值： * 精准获取流量达成率和资源使用情况。 * 深入分析系统瓶颈，明确性能短板。 * 自动产出限流调整与容量建议。 * 快速响应性能问题，显著提升系统稳定性。

数据示例： 表格展示了"Method"、"原始预估 QPS"、"当前实际 QPS"和"差异原因"的对比数据。部分请求差异原因为"平台 QPS 未带过来"（自动化平台未完全导入预估数据），另有部分请求实际 QPS 远高于预估且标记为"人为修改"（实际操作中存在人工干预调整）。

风险前置识别

核心理念是在运维流程早期主动识别潜在风险，而非仅依赖事后定位。

背景痛点： 当业务架构或流量放大系数变化时，系统往往无法快速感知，问题仅能在压测环节被发现，甚至可能将风险带入线上。例如某次"618"大促期间，应用链路变化导致活动流量激增 30 倍。

目标： 通过链路数据识别相关链路变化，对涉及的服务和接口依据高可靠性规范进行巡检，在问题暴露前引导业务方治理。

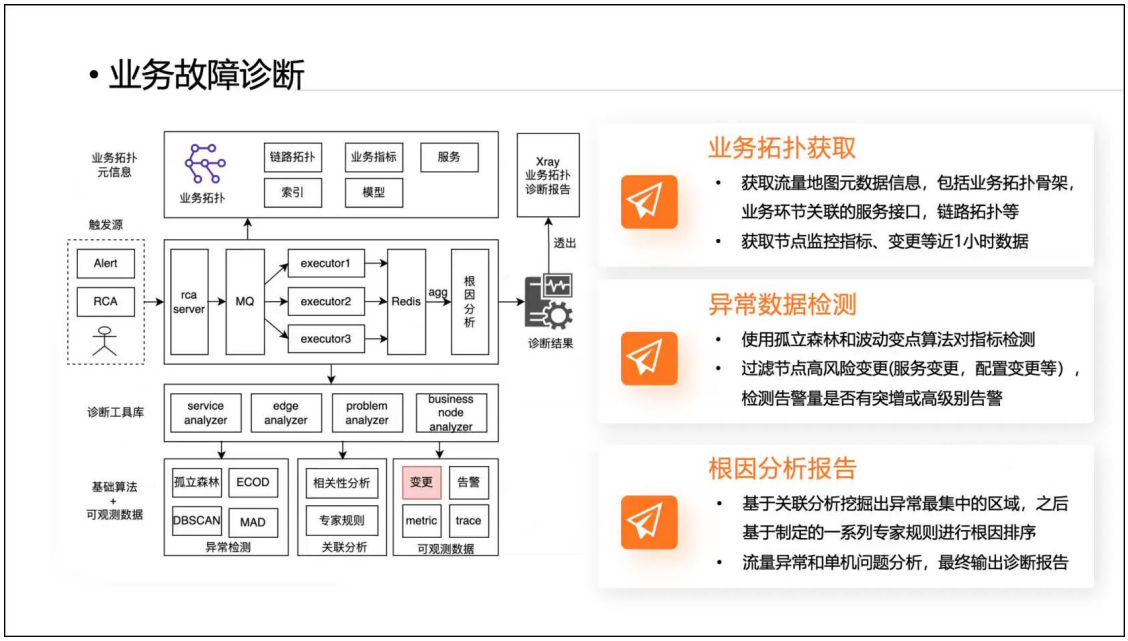
风险识别覆盖两大场景：

架构变化： * **入口层改造：** 识别新增入口并获取全链路架构信息。 * **内部服务依赖变化：** 关注调用关系和强弱依赖变化，评估潜在连锁反应。

流量变化： * **外部因素导致流量比例变化：** 因上游放量、部署结构调整等导致各区域流量比例偏离，需识别入口区域间流量比例。 * **内部服务迭代导致流量放大系数变化：** 服务迭代引发调用流量的转换或放大系数改变，需精确识别服务间转化系数。

风险检查结果表展示了应用名、服务名、检查类型、检查结果及描述，例如多条"接流风险检查"结果显示为"失败"，描述包括"存在切流风险(未获取到上游信息)"或"存在切流风险(非 Java 服务)"等具体问题。

业务故障诊断



当业务指标发生异动时，系统迅速锁定受影响的业务拓扑，基于业务维度进行智能诊断，明确故障的业务影响范围。通过提前了解影响面可快速启动预案止损，止损后进一步精细化故障定位，整体提升故障处理效率。

系统架构与处理流程：

- **业务拓扑元信息：** 包含业务拓扑、链路拓扑、业务指标、服务、索引和模型，提供全面的诊断上下文。
- **触发源：** 告警自动触发、用户主动发起根因分析、人工触发。
- **处理流程：** 触发源通过 rca server 进入 MQ，分发至多个 executor 并行处理，经 Redis 缓存和 agg 聚合后进行根因分析。
- **诊断工具库：** 包含 service analyzer、edge analyzer、problem analyzer 和 business node analyzer。
- **基础算法 + 可观测数据：**
  - 异常检测：孤立森林、ECOD、DBSCAN、MAD 等算法。
  - 关联分析：相关性分析和专家规则。
  - 可观测数据：变更、告警、metric 和 trace。
- **诊断结果：** 在 Xray 系统中呈现为详细的业务拓扑诊断报告。

核心诊断步骤：

- **业务拓扑获取：**获取流量地图元数据（业务拓扑骨架、业务环节关联服务接口、链路拓扑），以及近一小时的节点监控指标和变更信息。
- **异常数据检测：**使用孤立森林和波动变点算法检测指标异常模式，过滤高风险变更（服务变更、配置变更），检测告警量突增或高级别告警。
- **根因分析报告：**基于关联分析确定异常最集中区域，根据专家规则排序根因优先级，通过流量异常和单机问题分析输出详细诊断报告。

活动保障落地效果

通过流量地图驱动的活动保障体系，我们在三个关键维度取得了显著成效：

备战效率革命性提升

- 在去年"双十一"中成功将备战所需人力成本**降低了 90%**，得益于活动保障平台的自动化和智能化能力。
- 整个平台在部署、故障发现与恢复、演练流程等方面实现了质的飞跃，核心在于自动化工具链与标准化操作流程。

资源利用率显著优化

- 在本年度某关键活动中首次成功实现了**"零扩容"模式**，在不增加任何基础设施资源的情况下应对了整个活动的流量高峰与负载压力。
- 背后是更为智能的资源调度策略、精确的容量规划能力，以及对闲置资源的有效回收机制。

保障模式创新性革新

保障模式从"经验驱动"转变为"数据驱动"，实现从"全面保障"到"重点保障"，最终升级为"自动化操作"：

- **数据驱动与智能算法：**通过深度分析链路层面的数据，结合智能算法精确计算流量峰值、预测所需资源，实现科学前瞻的保障策略。
- **从全面到重点保障：**能够前置识别新功能、新玩法引入的潜在风险点及核心服务接口变化，将保障重心集中于关键区域。
- **自动化操作：**限流、扩缩容和压测等操作从人工干预全面升级为可自动执行的流程，有效规避人为失误。

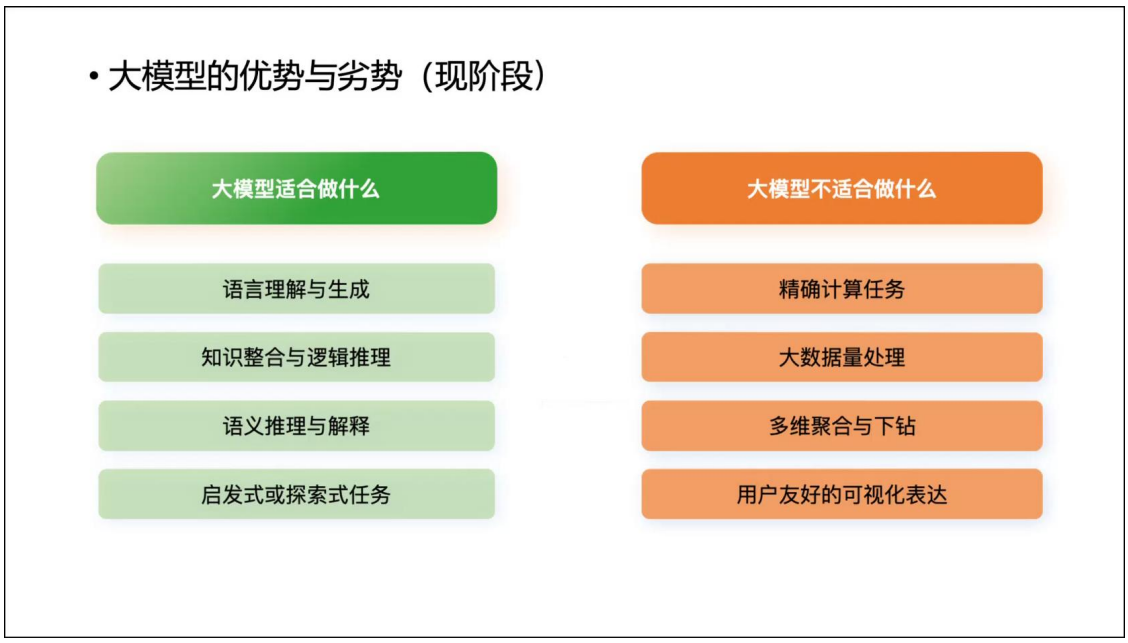
在流量地图活动保障体系取得成效的基础上，我们进一步探索了 AI 技术在技术风险管理领域的应用。

（三）总结及展望

技术风险 AI 探索

本章聚焦于如何将人工智能与机器学习技术应用于技术风险的管理、预测、识别、评估和缓解过程中，涵盖风险预测、异常检测、故障诊断等具体实践，以及 AI 在技术风险管理领域的未来展望。

大模型的优势与劣势



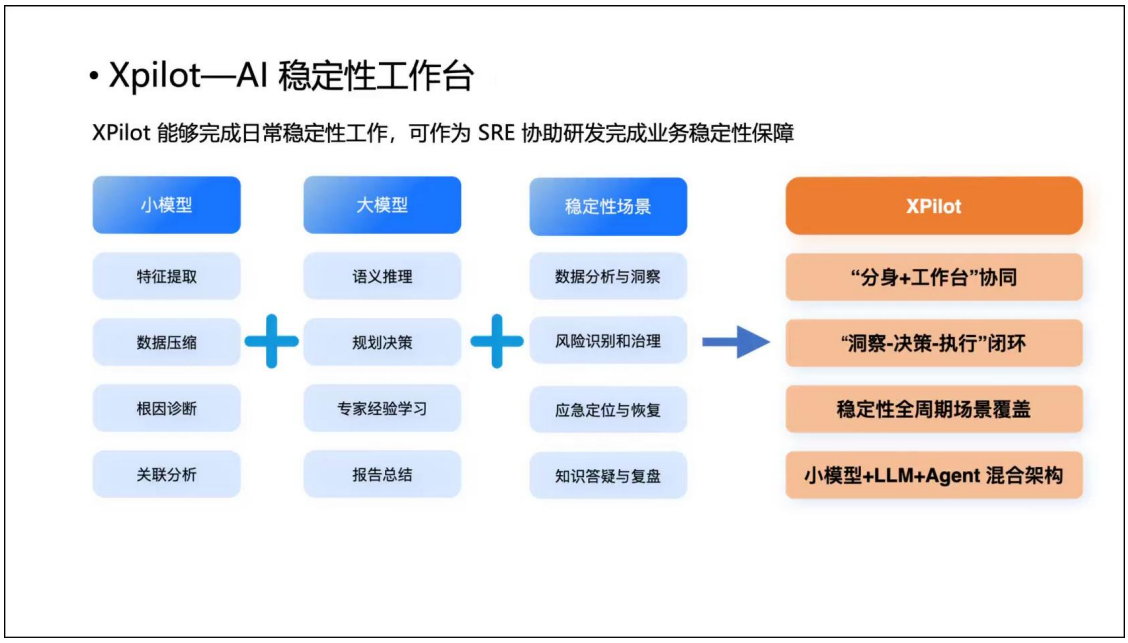
在将 AI 引入 SRE 实践之前，必须清醒认识大型语言模型（LLMs）的适用边界。

大模型适合做什么： * **语言理解与生成：** 文本创作、内容总结、多语言翻译等，显著降低用户交互门槛。 * **知识整合与逻辑推理：** 从海量信息中高效学习整合知识，进行一定程度的逻辑推理。 * **语义推理与解释：** 理解深层含义，对抽象或复杂概念进行清晰解释。 * **启发式或探索式任务：** 在需要创新思维、联想能力或初步探索未知领域时表现卓越。

大模型不适合做什么： * **精确计算任务：** 涉及严格数值运算时容易出现"幻觉"或计算错误。 * **大数据量处理：** 并非为高效处理大规模结构化或非结构化数据集而设计。 * **多维聚合与下钻：** 这是 BI 工具和数据分析平台的核心功能，大模型难以胜任。 * **用户友好的可视化表达：** 不具备直接生成复杂图表或交互式数据可视化的能力。

运维和可观测性领域的数据量通常非常庞大，因此不能简单地将小模型方案直接替换为大模型，而必须审慎评估适用场景，构建更健壮高效的解决方案。

Xpilot：AI 稳定性工作台



基于上述认识，我们构建了 Xpilot——一个采用"小模型与大模型"相结合混合架构的 AI 稳定性工作台。

技术架构与核心能力：

- **小模型：基础数据处理与事件分析的核心**
  - 特征提取：从海量监控数据中精准捕获关键特征。
  - 数据压缩：有效处理和压缩数据，降低存储与传输成本。
  - 根因诊断：自动分析故障日志与指标，快速定位根本原因。
  - 关联分析：识别不同系统组件或指标间的相互关系。

- **大模型：高级智能推理与决策引擎**
    - 语义推理：深入理解自然语言描述的故障信息和运维文档。
    - 规划决策：基于实时数据与专家知识，智能制定故障处理和风险规避策略。
    - 专家经验学习：从历史故障案例和 SRE 经验中持续学习。
    - 报告总结：自动生成故障复盘报告、系统健康分析报告。
- **稳定性场景覆盖**
    - 数据分析与洞察、风险识别和治理、应急定位与恢复（缩短 MTTR）、知识答疑与复盘。

核心价值： * **"分身+工作台"协同：** 作为 SRE 的"数字分身"自动处理重复性任务，同时提供工作台支持人工干预和团队协作。 * **"洞察-决策-执行"闭环：** 从底层指标和日志中洞察系统状态，基于智能分析进行决策，通过自动化工具执行落地。 * **小模型+LLM+Agent 混合架构：** 深度融合特定任务优化的小型模型、大型语言模型以及智能代理，构建高度智能化的运维平台。

AI 稳定性场景 1：告警数据分析

第一项 AI 稳定性场景聚焦于告警数据的深度分析——如同邀请一位"可观测性专家"对告警系统进行全面体检。

痛点： 人工维护告警配置耗时耗力，难以保障配置的完整性和准确性。

AI 赋能的分析 SOP：

- **数据聚合与洞察：**
    - 自动收集服务已有告警事件数据。
    - 依据指标、等级和通知方式等维度深度挖掘告警未覆盖区域或配置缺陷。
    - 分析服务与接收人分布，识别可优化环节。
- **智能诊断与优化建议：**
    - 提供具体的配置优化建议，生成结构化告警分析报告。

分析示例（2025-10-09 11 至 2025-10-10 11）：

- 告警总数 19 条，响应率 89.47%，已恢复 100%，P2 级别 100%，通知方式 100%为邮件。
- **关键洞察：**
    - **单一规则过度触发：** "[PromQL 告警][RedSchedule 任务执行失败]" 触发 17 次占 89.47%，可能过于敏感或存在配置偏差，需优化以减少"告警疲劳"。
    - **告警接收人负载不均：** 89.47%的告警集中由少数接收人处理，需重新均衡配置。
    - **告警数据源单一：** 所有告警均源自 Xray 数据源，建议拓宽数据采集渠道。

AI 稳定性场景 2：历史故障报告分析

第二项场景展示了 AI 驱动的"故障助手"如何对历史故障数据进行深度分析。

核心要素： * **Agent：** 故障助手 * **用户：** 研发、业务 POC、SRE * **场景：** 定期自动化对各业务线的故障事件进行统计分析与通晒，聚焦不同等级和类型的故障模式及频率。

AI 分析 SOP： * 基于评判标准对故障指标进行达标情况说明，支持明细数据智能分析和总结。 * 自动生成故障数据分析报告，给出风险洞察及针对故障频发指标组合的优化建议。

报告示例——执行摘要核心洞察：

- **变更风险管控：** 45.45%的故障由变更引发，其中 50%在灰度阶段发现，需加强变更过程中的监控和测试。
- **智能运维成效：** X-ray 智能告警覆盖 36.36%的故障，人工发现占 63.64%，需进一步提升智能告警的覆盖率和准确性。
- **MTTR 优化重点：** 搜索广告场景 MTTR 严重超标（5837 分钟），主因为代码 bug 导致的长时间止损，建议优化代码质量和应急响应机制。
- **业务线风险：** 外流广告场景故障数量最多占 36.36%，需重点关注。

稳定性达标情况： * P2+故障控制：总数 0 个（达标，标准:<4 个/月）。 * MTTR 达标：9 个场景达标，2 个场景未达标（标准:≤20 分钟）。

AI 稳定性场景 3：稳定性指标度量

传统人工统计稳定性指标往往导致数据在说服力、完整性和关联性上有所欠缺。我们利用 AI 为每个业务线提供综合的稳定性指标度量体系，深入揭示指标背后的计算逻辑、问题根源，并给出具体的治理和优化方案。

一、变更管控

AI Agent 化身"变更助手"，提供事件分析与拦截诊断、合规报告与封网信息查询。

- **度量指标（公司最近一周整体变更合规率）：** 459 次变更中 454 次达标，整体合规率 98.91%。大多数业务线（如第一、第二、第四业务线）合规率达 100%，部分业务线存在少量不合规变更，是 Agent 重点关注的优化方向。

二、风险治理

AI Agent 扮演"风险顾问"，提供风险现状与历史故障分析、解决路径辅助。

- **度量指标（各业务线近一周风险治理情况）：**
  - V1 风险治理完成率整体较高（多数业务线 91%-100%）。
  - V2 风险治理完成率（深度风险治理）差异显著：第四业务线 100%，但第一业务线 33.33%、第三业务线 0%、第五业务线 23.08%等显示"不达标"，表明深层次风险治理仍有显著提升空间。

三、强弱依赖治理

AI Agent 作为"高可用专家"，提供业务强弱依赖现状分析、治理辅助与演练方案创建。

- **度量指标（社区业务线强弱依赖标记率概览）：** 共 1,837 个调用关系，已标记 1,762 个，标记率 95.92%。"社区私信群聊"等场景标记率 100%（达标），体现了对系统高可用性的持续投入。

通过这套基于 AI 的稳定性指标度量体系，我们获得的不仅是一组组数据，更是一套能够告诉你哪里存在问题、如何着手解决、以及如何提升系统韧性的完整方案。

6 上线后持续优化工作

6.1 用户体验优化

解释什么是用户体验优化之前，我们先来解释一下什么是用户体验。

感官、情感、认知、行为等各个方面，一个好的用户体验应该是用户感到舒适、方便、快捷、高效和满意的。

而用户体验优化则是指通过维护系统的高可用性、减少系统的错误率、加快用户的响应速度等方面来提升用户在系统上的使用体验。

在 SRE 中，用户体验优化是一个重要的目标，它能够提高用户的满意度和信任度，增大用户对业务的黏性和忠诚度，从而增加用户的留存率和转化率，提高产品或服务的竞争力和市场占有率。

6.1.1 基于用户端的直接用户体验优化

基于用户端的直接用户体验优化，主要是针对用户直接接触到的产品或服务的方面进行优化，包括用户卡顿（页面加载速度、响应时间）、用户网络等方面，以提高用户的使用体验和忠诚度。

1. 面向用户端的优化措施

2. 用户端卡顿优化：通过提高客户端的性能（优化代码、减少资源消耗、改进算法等）、压缩和合并前端资源以及使用缓存和预加载等方式来提高页面加载速度，减少用户等待时间和卡顿现象；

3. 用户端网络优化：通过使用网络传输加速技术（包括使用 TCP 拥塞控制算法、调整 TCP 窗口大小、使用 TCP 连接池等）和双通道加速技术来提高数据传输的效率、减少用户网络的延迟以及增强通信的稳定性；如当用户处在弱网情况下，能利用双通道加速技术（如 Wi-Fi 和 LTE 同时发包的方式）来降低用户的网络时延和增强用户通信的稳定性，就可以减少用户流失。

4. 用户端其他优化：除卡顿和网络优化这种较为通用的优化之外，还有一部分是基于业务逻辑，对用户请求的反馈提示的优化。例如：触发了排队机制的提示信息，服务器满员的提示信息等。这些信息有效提示，可以引导用户等待或者引导用户到其他有资源的途径，让用户感受到系统仍然在工作，避免流失。

6.1.2 基于系统端的间接用户体验优化

基于系统端的间接用户体验优化，主要是针对用户无法直接接触到的产品或服务的方面进行优化，包括系统的性能、架构、容灾等方面，以保证系统的鲁棒性、健壮性和可用性，提高用户的信任度和满意度。

1. 面向系统端的优化措施

2. 基于鲁棒性的优化，提升系统健壮性：通过优化系统的容量规划、容错机制、架构设计（如弹性和可伸缩性）和负载策略（如负载均衡、优雅降级和限流策略），确保系统

在高负载和故障情况下仍能正常运行，防止因系统故障而对用户造成影响；例如：系统突发大量请求增加时，系统能够自动扩展和收缩资源以适应变化的负载，且系统能够实施限流和排队机制，限制并发请求的数量，并将请求放入队列中进行有序处理，避免系统过载和崩溃。

3. 基于幂等性的优化，提升系统可用性：通过将系统关键操作设计为幂等操作，并且在关键操作中，实施幂等性检查机制，如果关键操作失败或中断，可以使用事务和回滚机制来确保操作的幂等性，避免数据不一致或重复操作；如系统接收到重复请求时，系统可以检查请求 ID 或其他标识符，并判断是否已经处理过该请求；如果已经处理过，则可以忽略重复请求或返回相同的结果；或者如果系统操作失败或中断时，系统可以自动回滚到之前的状态，避免数据不一致或重复操作，并且通过系统自动重试后，能够被正确的处理。

在 SRE 中，用户体验优化措施实施之后，我们还需要一系列的度量指标来衡量和评估优化的效果：

1. 用户满意度（User Satisfaction）：通过用户反馈、调查或评级来衡量用户对系统或服务的满意程度，这可以通过定期的用户调查、NPS（Net Promoter Score）或用户反馈收集来评估。

2. 用户可用性（User Availability）：通过监测系统的正常运行时间、故障时间和恢复时间来衡量系统或服务在用户可访问性方面的表现，关键指标包括平均故障间隔时间（MTTF，Mean Time Between Failures）和平均故障恢复时间（MTTR，Mean Time To Recover）。

3. 系统错误率（Error Rate）：通过监测错误请求的百分比、错误码的数量或错误率来衡量系统或服务在处理用户请求时发生错误的频率，较低的错误率通常表示更可靠的系统。

4. 系统或者页面响应时间（Response Time）：通过监测平均响应时间、最大响应时间和百分位数（如 P99. 来衡量系统或服务对用户请求的响应速度，较低的响应时间通常意味着更好的用户体验。

5. 用户反馈响应时间（User Feedback Response Time）：通过监测用户反馈的处理时间、解决问题的时间和用户满意度来衡量团队对用户反馈的响应速度。

6. 客户端性能状态指标（Client Perf Status Metrics）：通过监测客户端关键路径上各动作的状态和发生时间，了解和衡量用户端操作和运行是否符合设计预期，主动确认是否存在优化的可能性。

除了以上通用型指标，还建议对用户使用的关键路径进行监控，关键路径监控是指关注那些对用户体验有决定性影响的系统组件或交互流程。这些路径是用户完成主要任务所必须的，比如电商网站的结账流程或者社交网络的信息发布流程。

关键路径监控通常包括：

(1) 端到端事务跟踪：追踪用户请求从开始到结束的全过程，确保每个步骤都按预期工作。

(2) 性能瓶颈识别：识别并优化那些延迟用户操作的关键组件。

(3) 可靠性保障：确保关键路径上的服务具备高可用性和容错能力。

(4) 实时警报：当关键路径上的性能下降或出现错误时，实时通知团队进行干预。

(5) 影响评估：分析关键路径性能问题对用户体验和业务目标的影响。

通过关键路径监控，SRE 团队可以更全面地理解用户体验，并采取针对性的优化措施。这些措施不仅仅关注技术性能指标，而且着眼于用户行为和感知，同时可以更为集中火力，获得更好的优化效果。

6.2 运维琐事的日常管理及优化

6.2.1 运维琐事的介绍

运维琐事是指运维处理一些手动性、重复性、可以被自动化的、被动响应的、没有持久价值的工作，而且琐事与服务呈线性关系的增长。运维琐事包括系统监控、故障排查、配置管理、容量规划、数据备份等。当然并不是每件琐事都有以上全部特性，但是每件琐事都满足其中的一个或多个特点。下面对运维琐事的特点进行介绍。

1. 手动性

例如对某个业务的服务进行停服处理，需要登录服务器，然后执行停服命令。如果是通过在作业平台上执行停服指令脚本，可以减少登录服务器的限制；如果是通过流水线作业来串联起该脚本指令，那么效率会更高。那这里手动登录服务器执行停服命令就被认为是琐事。

2. 重复性

如果某件事是第一次做，甚至是第二次做，都不应该算作琐事。琐事就是不停反复做的工作，例如业务的停服更新。如果你正在解决一个新出现的问题或者寻求一种新的解决办法，不算做琐事。

3. 可以被自动化

如果某个需求，除了业务运维可以手动处理外，也可以通过作业流程工具进行自动触发某个任务来解决该需求，就说明该需求可以被自动化。例如业务停服更新完起服后，业务运维需要检查服务是否都正常启动好了，这个需求就可以通过增添流程节点，检测起服日志中是否含有 success 等关键字来自动化判断服务进程是否都启动正常。

所以如果一个需求可以通过流程自动解决，或者是流程优化消灭了该类需求，那么这类需求就算作琐事。

4. 被动响应

处理琐事是处理那些突然出现的、被动去响应的工作，而不是主动安排的工作。处理紧急告警是琐事，我们可能永远无法完全消灭这类工作，但是我们也得必须努力减少这类需求，减少对运维的打扰。

5. 没有持久价值

如果在你完成某项工作后，这类工作的状态没有发生改变，这类工作就很可能是琐事。如果我们做了能给这类工作带来永久性改进效果的优化（例如操作步骤减少了），那么你做的事情就不是琐事。例如我们梳理了业务的核心配置文件，然后将其接入了进程配置管理平台，来达到平台化管理业务配置文件的目的，就不是琐事。

6. 与服务呈线性增长

如果在工作中所涉及的任务与服务的大小、数量呈线性增长关系，那这项任务就可能属于琐事，例如申请服务器这个操作，每个业务都有搭建新服的需求，都需要申请服务器，那么申请服务器的工作会随着服务业务的数量呈线性增长，那么我们可以通过在每个业务的流程里，自动嵌入申请服务器的节点，自动申请服务器来达到优化这个琐事的目的。

综上所述运维琐事的 6 个特点，可以发现如果对琐事不加以管理，运维很可能会被这些运维琐事淹没，影响工作执行质量和执行效率，也造成运维人才能力的提升空间受限，长期下去消耗工作热情。

6.2.2 运维琐事的质量管理

运维琐事的质量管理是指通过一些管理措施和方法来确保运维琐事工作的高效性和稳定性，来降低系统故障率、减少人为故障、提高系统的可用性和稳定性。通过建立规范的操作流程、使用自动化工具、培训与管理人员、持续改进等手段，可以提高运维琐事的质量管理水平，确保系统正常运行和业务的连续性。

运维琐事的质量管理包括以下方面：

(1) 流程管理：通过规范的运维流程，包括工作分工、操作步骤、审批机制等。确保每个运维操作都有明确的责任人和标准化的执行流程。例如我们每一次发布变更都需要有完整的操作 checklist 一样。

(2) 自动化工具：引入自动化工具和脚本来处理重复性和繁琐的运维任务，避免手工操作。通过将操作内容自动化，来屏蔽操作人员能力的差异性，可以减少运营故障的发生。

(3) 技能培训与人员管理：为运维人员提供必要的培训和技能提升机会，使其具备应对不同琐事的能力。同时，合理分配人员资源，确保每个运维任务都有专人负责。例如定期对团队内成员进行平台工具的标准化操作方案的培训。

(4) 持续改进：定期回顾运维工作的过程，例如对操作任务的成功率的统计，对每一项任务进行分析，发现问题并制定改进措施，因为任何一个问题都会传递到业务运维，需要业务运维分析并解决，通过持续改进来提高运维琐事的质量。

6.3.3 运维琐事的效率管理

运维琐事的效率管理是指通过标准化、工具自动化、OnCall 需求分层、优先级管理和持续改进等手段，来实现高效的运维琐事管理，为组织带来更高的价值和效益。

运维琐事的效率管理包括以下方面：

(1) 标准化和流程优化：建立规范的运维流程和操作步骤，确保每个操作都按照统一的标准执行，通过流程优化，消除冗余的步骤和无效的等待时间，提高工作效率。

(2) 工具自动化：通过自动化工具平台的能力，减少人工干预和手动操作，提高操作速度、降低人为错误，达到释放运维精力用于更复杂和有价值的工作。例如通过 CMDB 平台（运维配置管理数据库）来管理业务的硬件等资源、通过进程配置管理平台来管理业务的核心配置文件和进程启停信息、通过监控平台来管理业务的服务器资源使用情况、通过作业管理平台来批量管理业务的任务模板等。

(3) oncall 需求分层：OnCall 是指业务运维团队将工作时间划分为若干个固定的时段，每个时段都会有专门的运维人员负责值班处理需求，从而达到可以随时响应业务需求，确保服务的连续性和稳定性，oncall 在进行响应需求时，需要对需求进行分层处理，例如根据需求的紧急程度进行分层、根据需求的工作范围进行分层、根据需求的所属平台进行分层等。通过分层后的需求，就可以分配到不同工种来解决，实现问题的高效处理。

(4) 优先级管理：通过 OnCall 团队一线的需求梳理后，对每一项需求进行优先级管理，让团队一线通过标准化文档操作指引优先去处理紧急且重要的事情，避免运维人员工作精力分散，提高工单的流转率。

(5) 持续改进：定期回顾运维工作的执行情况，例如对工单的类型比例分析、工单的耗时比例分析，这种数据驱动的策略来发现问题，例如发现哪一类需求比重比较高，耗时比较长的情况。持续优化，提高运维琐事的效率。

(6) 文档沉淀：由于相关琐事较多，需要通过文档进行知识管理及文档的标准化，便于团队成功共享、查询和复用，

(7) AI 提效：利用人工智能技术来提高运维工作的效率，减少重复性和低价值的工作。具体措施非常广泛，现阶段，部署聊天机器人作为一线支持，对接文档支持库，处理常见的查询和简单问题，减轻人工客服的负担，是在提效上面非常值得探索，且可落地性强的方向。

6.3 业务全生命周期工具建设

业务全生命周期主要包含研发期，上线期，运营期以及长尾期阶段，在每个阶段都需要有能提升运营效率的通用化工具。建设的工具包含版本控制，监控和告警，日志管理，CI/CD，数据备份，自动化测试，容器编排，业务巡检等工具。

在 SRE 中，业务全生命周期工具建设是非常重要的一部分，通过使用这些工具和技术，SRE 团队可以更加高效地管理和优化业务的整个生命周期，提高业务的各个阶段的可靠性和稳定性，降低成本和风险。

6.3.1 研发期工具建设

业务研发期是指产品研发阶段，这个阶段需要使用一些工具来辅助研发和管理。在业务研发期，工具建设可以提高系统的可靠性、可用性、可扩展性以及研发效率，以支持产品的研发和快速迭代。

主要包括如下几类：

(1) 版本控制工具：例如 Git，用于管理代码的版本、分支、合并和协作等。研发期版本控制工具是必要的，帮忙研发人员更高效的进行项目的开发。

(2) 自动化构建工具：例如 Jenkins，用于自动化构建、测试和部署代码，提高开发效率和质量。研发期 SRE 人员参与较少，自动化构建功能实现研发人员版本自动化部署能力。

(3) 代码质量工具：用于分析代码质量、安全性和可维护性等，帮助开发团队提高代码质量和可维护性，确保代码不会有安全的漏洞。

(4) 代码审查工具：用于进行代码审查和评审，提高代码质量和可维护性，可提高代码的健壮性，减少程序可能出现的 BUG。

(5) 测试工具：用于进行单元测试、集成测试和 UI 测试等，提升测试的效率。

效果评估：

(6) 提高开发效率：使用自动化构建、测试和部署工具可以减少手动操作，提高开发效率和质量。

(7) 提高代码质量：使用代码质量工具和代码审查工具可以发现和修复代码中的问题，提高代码质量和可维护性。

(8) 提高团队协作效率：使用项目协作工具可以提高团队协作效率和沟通效果，减少沟通成本和误解。

(9) 提高测试效率：使用测试工具可以更多的覆盖测试用例和场景，并且提高测试效率。

6.3.2 上线期工具建设

业务上线期是指产品上线阶段，这个阶段需要使用一些工具来辅助上线。上线期工具可以提高系统的可靠性、可用性和可扩展性。以支持产品稳定上线和运营。

主要包括如下几类：

(1) 监控和告警工具：用于监控业务的运行状态和性能指标，并在出现问题时发出告警。

(2) 日志管理工具：用于收集、存储和分析业务的日志数据，业务出现问题能及时发现。

(3) 自动化部署工具：用于自动化部署业务代码到生产环境。

(4) 故障排除工具：用于快速定位和解决业务故障。

(5) 容器编排工具：用于管理和编排容器化的业务应用。

(6) 上线风险检查工具：用于保障业务的健康状态工具集，工具可以定时巡检，保证业务实时的稳定性。

效果评估：

(7) 提高上线效率：使用自动化部署工具可以减少手动操作，提高上线效率和质量。

(8) 提高系统可靠性：使用监控工具和日志管理工具可以及时发现和解决系统中的问题，提高系统的可靠性和可用性。

(9) 降低业务风险：使用上线检查工具可以保障业务上线环境的稳定性，降低上线风险问题的发生。

6.4.3 运营期工具建设

业务稳定运营期是指产品上线后的运营阶段，这个阶段需要使用一些工具来辅助运营和管理。在业务稳定运营期，建设工具来提高系统的可靠性、可用性，以支持产品的稳定运营和业务增长。

主要包括一下几类：

(1) 监控和告警工具：用于监控业务的运行状态和性能指标，并在出现问题时发出告警。

(2) 日志管理工具：用于收集、存储和分析业务的日志数据，业务出现问题能及时发现。

(3) 自动化部署工具：用于自动化部署业务代码到生产环境。

(4) 故障排除工具：用于快速定位和解决业务故障。

(5) 容器编排工具：用于管理和编排容器化的业务应用。

(6) 健康巡检工具：用于保障业务的健康状态工具集，工具可以定时巡检，保证业务实时的稳定性。

(7) 成本分析与优化工具：用于运营期业务的成本分析和成本优化。

(8) 安全巡检工具：用于运营期安全风险的检查，预防安全事件。

效果评估：

(9) 提高系统可靠性：使用监控工具，日志管理工具，健康巡检工具，安全巡检工具，可以及时发现和解决系统中的问题，提高系统的可靠性和可用性。

(10) 提高系统性能：使用负载均衡和容器编排相关工具可以提高系统的性能和可扩展性，支持高并发和大流量的访问。

(11) 提高运维效率：使用自动化运维工具可以减少手动操作，提高运维效率和一致性。

6.4.4 下线期工具建设

业务下线期是指产品不再提供服务，准备关停服务。这个阶段需要使用一些工具来操作下线的流程。在业务下线期间，使用工具长久保存用户数据并对相关资源进行回收。

主要包括以下几类：

(1) 数据备份工具：用于重要数据永久备份，业务退市后数据库一般是需要长久备份，用于后续数据的查询需求。

(2) 资源回收工具：业务下线后，为了节省成本，需要尽快对相关资源进行回收。包括但不限于服务器资源，CDN 资源，COS 资源。需对业务相关的资源全部清理回收处理。

(3) 遗留成本检查工具：为了确保业务相关所有资源都及时回收，需要工具对涉及到的相关资源使用情况进行检查。确保所有资源都得到回收和释放。

(4) 权限检查工具：业务下线后，对应业务的相关权限，都需要工具来检查，例如 AKSK，周边平台的权限都因彻底回收。

效果评估：

(5) 数据可追溯：业务下线后，相关的重要数据都能查询。如果外网用户需要查询数据核对情况，均能查询到用户数据。

(6) 下线业务无运营成本：业务相关资源全部回收和释放，下线后业务成本为 0。

6.4 运营成本分析及优化

6.4.1 运营成本分析及优化的必要性

运营成本是任何产品都会涉及到的问题，其重要性是不言而喻的。通过运营成本分析，我们便可知一个产品的成本有哪些部分构成，各部分是否都是合理的，是否需要做优化。持续的成本优化是保障一个产品健康发展的有效手段之一，需要贯穿到整个产品的全生命周期来执行。

6.4.2 运营成本实时监控

由于误操作，程序应用代码错误，黑客攻击大量，而导致使用使用资源飙升，成本失控的情况屡见不鲜，因此，一定要对运营成本进行实时监控，避免无谓浪费。

1. 建立成本监控系统：

利用现有的监控工具或开发专门的成本监控系统，实时追踪云服务使用情况、服务器资源占用、带宽消耗等关键成本指标。

将运营成本数据与业务指标（如用户活跃度、交易量等）关联，以便从业务角度理解成本变化的影响。

2. 设置预警机制：

为各项成本指标设定阈值，当成本超出正常波动范围时，系统自动触发预警通知相关负责人。

预警机制应包括多级告警，以应对不同程度的成本波动情况。

3. 细化成本分类：

将成本分解到更细的粒度，如按服务模块、按功能点、甚至按代码提交进行成本追踪，以便更精确地定位成本波动的来源。

4. 每日账单监控：

账单自动化获取，配置云服务提供商或财务系统的 API，实现每日账单数据的自动获取，并与监控系统对接。如实现难度较大，SRE 起码养成每日登陆系统查看账单习惯，或者订阅邮件账单

6.4.3 运营成本分析及优化的指标

1. 财务指标

财务指标是运营成本分析及优化的重要指标，只有在财务的成本数据体现了成本降低，才是有效的成本优化。

财务指标包含：

1. 单用户成本

单 PCU 成本：业务总运营成本除以业务最高在线人数（PCU）

单 DAU 成本：业务总运营成本除以业务日活跃人数（DAU）

单 MAU 成本：业务总运营成本除以业务月活跃人数（MAU）

2. 单用户成本增长率

在业务的不同运行周期，单 PCU 成本、单 DAU 成本、单 MAU 成本中，若某个维度与历史数据比较，增长率为正，则说明该维度的运营成本在上涨，需要做运营成本的优化。相反，若增长率为负，则说明该维度的运营成本在下降。

2. 技术指标

SRE 对于成本优化动作主要提现在技术方案上，而技术指标则是体现技术方案是否有效的主要指标。

常见技术指标包含：

CPU 利用率：衡量服务器资源的 CPU 使用情况，通常使用 CPU 周均峰值，根据模块功能不同考核标准也不同，接入层、逻辑层、存储层的 CPU 利用率的考核标准会有差异。

内存利用率：衡量服务器资源的内存使用情况，业务进程类型不同，内存使用率也会有差异，消耗内存型的业务通常内存利用率较高。

存储使用量：衡量服务器的磁盘使用情况，如磁盘空间使用率，磁盘 I/O 吞吐量等，数据型业务的服务器较多用到这个指标。

网络吞吐量：衡量服务器网络使用情况，例如网络流量使用量、数据包收发吞吐量等，网关型服务器较多用到这个指标。

CDN 带宽使用量：涉及到用户侧客户端下载的业务多用到这个指标。

SRE 在做成本优化时，需要结合财务指标和技术指标综合评估，既要保障成本在合理的使用区间，又不能影响业务的稳定性和可靠性。

6.4.4 运营成本的统计及分析方法

1. 运营成本的统计

运营成本分类统计主要分为以下维度：

IaaS 层成本：包含 AWS、谷歌云、腾讯云、阿里云、华为云、自研云等提供底层基础设施的服务提供商，公共的 CDN 服务、计算服务、存储服务、带宽服务等的成本统计。

PaaS 层成本：公共登录组件、数据平台、安全平台等提供公共服务的平台成本统计。

SaaS 层成本：运维工具、流水线、日志管理、版本发布工具、运营工具等的成本统计。

2. 运营成本的分析

3. 单用户成本分析

业务整体单用户成本，从业务整体运营成本评估单用户成本是否处于合理使用区间。

IaaS 层单用户成本，SRE 分别评估 AWS、谷歌云、腾讯云、阿里云、华为云、自研云等不同基础设施层的单用户成本，除了在每一个独立的云服务范围内降低单用户成本外，在各种云厂商之间可以横向对比单用户成本，并可以迁移至单用户成本较低的云厂商。

PaaS 层单用户成本，SRE 需要评估业务在各 PaaS 平台的单用户成本占比，对比 PaaS 平台的总体单用户成本与单业务的单用户成本，同类型业务的 PaaS 层单用户成本等，根据综合评估的数据，进行成本优化。

SaaS 层单用户成本，SRE 需要评估业务使用各 SaaS 产品的单用户成本占比，在单 SaaS 产品的单用户成本与其他业务的差异。

2. 单用户成本增长率分析

业务会经历研发期，上线期，稳定运营期以及长尾期等阶段，同一维度在不同阶段都会有对应的单用户成本，SRE 可以对比同一业务在不同阶段的单用户成本增长率变化，对于单用户成本上涨的阶段，SRE 需结合财务指标、技术指标等综合方法降低单用户成本。

3. 多业务对比分析

针对同一类型的业务，SRE 可以综合对比同类型业务的整体 CPU 利用率，单业务模块 CPU 利用率等，与同类型业务横向对比，评估所属业务的 CPU 利用率是否在合理使用区间。

4. 资源利用率分析（以 CPU 为例说明，内存和存储类似）

针对同一款业务不同模块，可以分析 CPU 利用率是否在合理使用区间。对于消耗 CPU 的业务模块，和消耗内存的模块，CPU 利用率应该有不同的标准，SRE 需要给出合理的评估意见。

一款业务会经历研发期，上线期，稳定运营期以及长尾期等阶段。

在研发期，SRE 需要提供少量测试和开发服务器即可，对 CPU 利用率没有考核。

在业务上线期，SRE 需要保障业务充足的容量，优先满足业务用量需求。

在业务稳定运营期，SRE 需要评估业务整体 CPU 利用率、各模块 CPU 利用率等，并给出 CPU 利用率的优化建议，使业务 CPU 利用率处在合理使用区间。

在业务长尾期，SRE 除了评估单业务的 CPU 利用率，还可以采用多业务混布的方式，提高多业务的综合 CPU 利用率，达到成本优化的目标。

5. CDN 使用量分析

用户一旦涉及资源的下载和更新，便会产生 CDN 成本。CDN 成本通常位列运营成本的前三位。不同云厂商的 CDN 计费规则各有差异。国内厂商普遍采用 CDN 带宽计费模式，而海外厂商则多以 CDN 流量计费模式为主。

CDN 成本优化可围绕以下几个思路：

CDN 流量计费模式：该模式通常使用 CDN 累计使用流量来计费。尽量降低用户下载的资源量。比如可做差量更新的资源，优先使用差量更新，既提升用户体验也能降低 CDN 资源。

CDN 带宽计费模式：该模式通常使用 CDN 使用峰值带宽来计费。可通过各种技术策略降低 CDN 带宽。通过“削峰填谷”的方式，让用户下载更新资源时更分散到各个时间区间，避免所有用户集中在高峰期下载从而达到降低 CDN 带宽的目的，比如提前下载，预下载等策略。

6.4.5 运营成本的优化方法

1. 单业务优化方案

缩减业务容量：

(1) 业务周期。业务会经历上线期，稳定运营期以及长尾期等阶段。在业务上线期，SRE 需要充分满足业务容量需求。在业务进入稳定运营期之后，需要根据业务实际运营情况，结合在线规模、服务器 CPU 利用率、内存使用率、存储使用量等综合情况，对资源容量进行缩容，在不影响业务稳定运营的前提下，降低资源容量，达到成本优化的目标。在业务长尾期。

(2) 业务混部。除了降低资源容量外，在业务进入稳定期后，业务正常负载无法重复使用服务器资源，在一定程度上造成了资源浪费。此时，可以考虑多业务混部的方式降低业务运营成本，即，多个业务或服务部署在相同的服务器上，从而起到合理利用服务器资源的目的。

业务混部的方式，有一定的风险，需要做好全面的架构评估和技术方案评估，需要评估的方面有：

a. 业务等级评估

进行业务混部首先就是要进行业务等级的评估，如果业务等级很高，不容许有失败率，那么趁早放弃这个方案。适合进行混合部署的业务可能有如下特点：

失败不敏感，重试成功后不影响

不直接服务用户

无状态

b. 至少 3 个月稳定状态的性能评估

比如，连续 3 个月 CPU 利用率低于 5%，连续 3 个月内存利用率低于 10%等

c. 资源消耗互补评估

计划混部的业务在资源消耗偏好方面具有互补性，例如：A 业务属于 CPU 消耗型，B 业务属于内存消耗型，这两款业务可以考虑进行混部，即使某些特殊情况发生资源消耗增加也不至于两个业务相互争夺资源。

d. 峰值波动互补评估

计划混部的业务的业务峰值具有互补性，例如：A 业务的峰值发生在上午 10 点，B 业务峰值发生在凌晨 2 点，这两个业务考虑进行混部，避免了资源的争抢。

e. 应急方案评估

任何一种方案都不能保证万无一失，一定要准备完整的应急方案，例如：紧急扩容方案、故障隔离方案、资源隔离方案等。

(3) 动态扩缩容。业务每天或者每个阶段会有在线的波峰和波谷，不同在线规模需要的资源容量不同，可以根据业务在线规模动态扩缩容，通过容器的 HPA 等技术动态调整资源容量，对业务运营成本进行优化。

2. 平台化优化方案

空闲资源调度：对于单业务的空闲资源，SRE 平台部门可以资源整合，组成联邦集群，在业务负载低峰期，通过统一调度，处理分布式的离线任务，提高资源利用率。空闲资源被平台部门使用的同时，可以返还部分运营成本。

容量规划与评估平台：SRE 可以根据业务历史数据，准确评估未来容量需求（例如未来一年周期），通过对未来周期业务容量的准确评估和规划，可以通过批量集中采购的方式获取低价资源。同时，SRE 可以将不同类型业务的资源评估方案沉淀为容量规划与评估平台，当其他业务需要进行容量评估和规划时，可以借助容量规划与评估平台的能力，得出未来周期合理的容量，并合理地批量采购低价资源。

内部资源交易平台：不同业务体量不同，使用的资源类型不同，所获得的折扣券种类、折扣力度、数量都会不同。为充分盘活内部折扣资源，一个组织（例如同一家公司）内的 SRE 资源管理团队，可以开发一个内部折扣资源交易平台。不同业务间可以互相交易资源折扣券，使得一个组织内的折扣券，可以最大程度被利用。

6.5.5 运营成本优化持续运营

1. 运营成本的统计与分析工具

SRE 基于成本运营的统计和分析方法，可以建设运营成本统计和分析的可视化工具。功能涵盖但不局限于以下几个方面：

(1) 运营成本的组成，SRE 可以直接看到 IAAS 层、PAAS 层、SAAS 层的运营成本构成及占比等信息，并可以对比不同周期的成本变化。

(2) 运营成本的分析，针对财务指标、技术指标等运营成本数据，可以与同业务不同周期，不同运营阶段纵向对比；也可以与同类型不同业务横向对比。

(3) 运营成本的优化方案推演，通过调整不同维度、不同指标的运营成本的数据，可以直观观察成本的变化情况，预估成本优化的 KPI 目标。

2. 资源调度工具

SRE 基于空闲资源调度的方法，可以建设资源调度工具。功能涵盖但不局限于以下几个方面：

(1) 空闲资源分析，从联邦集群的角度，分析每个小集群（业务）的资源空闲情况，根据空闲 CPU、内存等资源情况，分析可被公共平台调用的空闲资源。

(2) 空闲资源调度，根据空闲资源分析得出可以调度的资源，并合理分配可被调度的分布式离线任务。

(3) 空闲资源监控，实时监控集群的空闲程度，优先满足业务的正常运行，当离线任务资源与业务进场资源冲突时，优先保障业务资源的使用。

3. 容量评估工具

SRE 基于容量评估与规划的方法，可以建设容量评估工具。功能涵盖但不局限于以下几个方面：

(1) 容量的展示，实时展示当前与历史的业务资源容量使用情况。

(2) 容量的分析，不同业务类型的容量，不同资源类型的容量，可以与同业务纵向对比；也可以与同类型不同业务横向对比。

(3) 容量的预测，基于单业务历史的容量使用数据，和同类型多业务的容量使用数据，智能推荐未来周期的容量数据。为运营成本优化提供合理的建议。

4. 运营成本返点售卖工具

SRE 基于运营成本返点售卖的方法，可以建设成本返点售卖工具。功能涵盖但不局限于以下几个方面：

(1) 可售折扣券的展示，实时展示不同业务当前可出售的折扣券的数量与类型等。

(2) 折扣券的使用推荐，SRE 输入业务的资源类型和资源容量数据，运营成本返点售卖工具可以自动推荐合理的折扣券使用方案。

(3) 折扣券的使用数据分析，基于业务使用的折扣券的历史数据，分析得出折扣券的覆盖率和使用率等数据，辅助 SRE 做出运营成本分析的决策。

6.5 持续改进

6.5.1 效率持续改进

效率的改进可以分为两类：

1. SRE 个体工作效率的改进与衡量

一般情况下来说，某项具体工作效率的改进得益于某个具体的人的技术水平或者某项新技术的诞生对工作产生了效率上积极的影响。从持续改进的角度来说，我们需要关注的是：

1. 员工个人能力的培养

具有某方面技术优势的员工重点培养，给予更多空间和尝试机会。

2. 新技术的关注与应用

鼓励用新技术解决老问题，在安全可靠的范围内测试新技术的稳定性，尝试新技术落地。

对于某项具体 SRE 工作效率改进效果的衡量主要是从时间角度，当然需要考虑“整体”时间这个概念。例如：

某项工作原来的执行时间是 10 分钟，新方法改进后为 8 分钟，效率提升 20%。但是新方法在执行前需要用 3 分钟时间来做准备工作，整体时间为 11 分钟，实际效率提升为-10%。所以衡量需要考虑整个改进对于原有工作所有的改变和影响。

2. SRE 团队整体工作效率的改进与衡量

团队整体效率的提升需要用更宏观的视角和更长的时间跨度来看待，主要以半年或者一年为周期看待某一类工作是否产生了积极的影响。从持续改进的角度来说，我们需要关注的是：

1. 某一类工作的人力投入是否持续降低

对所有的发布变更类工作进行至少半年以上的人力投入统计，评估人力成本投入是上升趋势还是下降趋势，并分析原因。呈上升趋势的，找到人力投入持续上升的原因，并及时解决。呈下降趋势的，找到是因为做了什么优化而导致的人力投入下降，继续强化去做。

2. 在团队整体配备保持不变的情况下，是否能够承接更多或更大规模的服务

评估整个团队解决目前所有的工作所使用的解决方案有哪些，是否具有通用性。团队输出通用性方案的能力越强，这个团队保持配备不变的情况下，接手更多更大服务的能力越强，团队整体服务能力越强。

6.5.2 质量持续改进

质量的持续改进，一般是指 SRE 团队所提供的服务质量，这部分工作的持续改进大致可以分成两部分：

1. 服务质量的衡量

衡量服务质量的方法很多，例如：故障分级制度、满意度调查、平均无故障时间、SLO 等。

故障分级和满意度调查，可以了解当前服务质量状况。

平均无故障时间或平均故障间隔时间（MTBF），可用于持续地改进衡量标准。因为我们可以希望这个时间尽可能地长，说明我们的服务一直处于无故障状态，即服务质量一直达标。

在平均无故障时间（MTBF）的概念下，还可以进一步细计算：平均故障恢复时间（MTTR）。

平均故障恢复时间（MTTR）又可以继续被细分为：

MTTI，平均故障发现时间

MTTK，平均故障定位时间

MTTF，平均故障解决时间

MTTV，平均故障修复验证时间

以上都可以作为非常细致的服务质量持续改进的衡量指标。

这种衡量方法，不足的地方在于，他可以衡量那种引发了故障的情况非常好用，清晰且可衡量。但是，对于没有引发故障，但是用户的实际体验却实实在在受到影响的情况就不好衡量了，这时候我们可以用到 SLO 这个指标。

SLO 可以根据服务不同等级相对稳定在某个值，我们需要关注的是 SLO 有没有下降趋势。如果有下降趋势，那么具体是哪些微小的因素影响了用户体验？对于提升用户体验方面可以参考 3.6.1。

2. 服务质量的复盘

服务质量的复盘，包含对优秀案例的复盘和故障案例的复盘。

优秀案例的复盘，重点关注：

我们具体做了哪些工作导致服务质量提升？

为什么这样做可以提升服务质量？

我们如何把这种方法复制到其他服务质量的提升？

故障案例的复盘，请参考 3.5.4。

6.5.3 安全持续改进

一次安全事件的发生很可能将其他的所有工作都归零，安全的持续改进，也可以理解为是安全预防工作的持续改进，如何持续地预防安全事件的发生。

安全的持续改进可以从如下几个方面着手：

1. 默认安全规则

默认安全规则，即要求 SRE 团队在所有服务上需要做到的最低安全规范。

例如：禁止 SSH 服务对公网开放、在服务器上开放类似 VPN 的功能、安装未经安全评估的开源软件等。

2. 安全意识

所有人员应当定期参加安全培训，提升安全意识。不仅仅是 SRE 团队，更应该包含产品运营团队和开发团队，落实是“人人都是安全责任人”的概念。

3. 研发期安全预防

践行 DevSecOps 理念，把安全能力无缝且柔和地嵌入到研发过程，在产品研发过程中加强安全能力的建设，预防安全事件发生。例如：定时向研发团队强调默认安全规则，介绍经过评审的开源软件，推送具有安全风险的开源软件，介绍安全的系统架构设计原则等等。

4. 运营期安全预防

运营期的安全预防机制，是常态化进行巡检，输出安全报告或安全工单。例如：高危端口扫描、高危服务扫描、入侵检测系统接入扫描、具有安全漏洞的开源软件扫描、密码泄露扫描、内外部代码扫描、安全补丁未更新扫描等。

5. 评审机制

定期组织安全能力评审，对新上线系统以及在运营系统进行安全评估，制定整改计划。安全评审，要确保评审团队与 SRE 团队之间的轮换，避免长期固定的评审关系。

6.5.4 人员能力持续提升

人员能力持续提升，其实就是为了让团队的技术能力能够持续健康发展，这需要有一个可持续运转的机制来保障。类似这样的方法有很多，没有什么标准做法，下面介绍一下我们内部的方式。

1. 海量运维能力机考

每年进行一次海量运维能力上机考试，题目包含：Linux 基础/Window 基础，海量运维脚本开发、故障排除、云原生基础、安全常识等，可以一定程度上帮助大家看到自己的水平变化和技能的熟练度。

2. 技术水平能力积分

SRE 岗位所需要具备的海量运维、运维开发、DevOps、AIOps 等各项能力所对应的系统、工具的使用和操作记录、贡献记录、开发记录、代码质量、维护记录、创建记录等进行积分规则制定，根据这些规则为每一个 SRE 个人和 SRE 团队计算积分。这个积分从一个侧面体现其 SRE 个人和 SRE 团队的能力，用于衡量个人和团队能力提升的参考。

这个积分规则的合理性决定了这个制度是否能够执行下去，所以，这个规则需要能够客观地展示个人和团队能力，并且起到带动 SRE 团队发展作用，这是这个规则制定的关键。

例如：

某个 SRE 开发的一款 SaaS，不仅能够帮助其他的 SRE 完成日常工作，而且产品开发、产品运营都经常使用这个 SaaS，并且允许稳定，代码质量高，BUG 少。那么这个 SaaS 的开发、维护、代码质量等各项工作的积分就算在这个 SRE 的运维开发的能力项下，可以从一个侧面反映这个 SRE 的运维开发能力。

再例如：

当我们衡量云原生能力时，那么所有在 K8S、容器平台上的贡献纪录将被用于衡量每一个 SRE 在云原生方面的能力。如果我们要鼓励大家更多地学习和应用云原生技术时，相对应的这部分积分的权重可以在一定的周期内调高，以鼓励现有团队进行转型。

有了这样的体系，整个团队的人员能力有了一个大致的数据衡量方法，可以客观地看到各个团队在不同维度上的能力强弱，从而人员能力提升有了动力。

6.5.5 流程持续改进

SRE 团队各个环节的工作其实都有专门的持续改进措施，以保证能够持续优化。例如：故障复盘、混沌工程等等。作为 SRE 团队的整体发展，其实还需要一个整体的流程以保证整个 SRE 工作是可以持续改进的。接下来就介绍一下这个整体能够保持持续改进的方法。

这套流程方法大致分为如下几个步骤：

1. 建立上线评估标准

SRE 团队需要出具一个上线评估标准，此标准用于评估一个新的服务是否满足交付给 SRE 团队持续的维护。

这个标准可能需要保护如下内容：

架构评估

合规评估

安全评估

其他评估

通过以上各项评估初步了解即将接手的服务特点，以便 SRE 团队制定维护目标和计划。

2. 确定 SLI/SLO

现在，可以根据现有的信息来初步制定维护目标即 SLI/SLO。

3. 运行分析

这个阶段对即将要接手的服务进行运行分析，记录系统运行的关键数据和指标，验证上面几个步骤是否还需要重新评估或修改目标，也需要验证 SRE 团队的维护方案是否可行。研发团队必须在这个阶段给予 SRE 后备支援与各项建议。这种关系成为团队未来工作的基础。

4. 改进与重构

上一个步骤的运行分析，在这个步骤需要输出改进和重构的建议，提交研发团队修改，SRE 团队也改善当前的维护方案。

5. 综合培训

这个步骤是确保整个团队都有所准备，不仅仅是 SRE 团队，也包含开发团队、运营团队、产品团队等，需要全面地让所有人都知道上线后，我们所需要遵守的规范和长期运行的目标。

这个培训应该包含：

项目整体概况

安全规范

该项目的 CI/CD/CO 的标准流程与协作方式（版本交付、测试、发布等）

线上系统的部署模式

故障处理与应急流程

其他需要整个项目团队知晓的内容

6. 正式服务

此时，SRE 正式开始接手这项服务，从这个时候开始各项运维职责、权限都转交 SRE 团队，包括运维操作、发布变更、访问控制等等。研发团队持续给予 SRE 后备支援与各项建议，并与 SRE 团队长期保持良好的协作关系。

7. 持续改进

活跃的服务根据运营需求的变化，不断变化系统依赖，技术升级和其他变化。例如：新增用户需求等。SRE 团队在面对这种持续改动的同时维护服务的可靠性。当面对每一个新的变化的时候，需要根据当时的具体情况，重复执行上述的 7 个步骤以保证 SRE 团队的工作能够在良性的循环中持续改进。

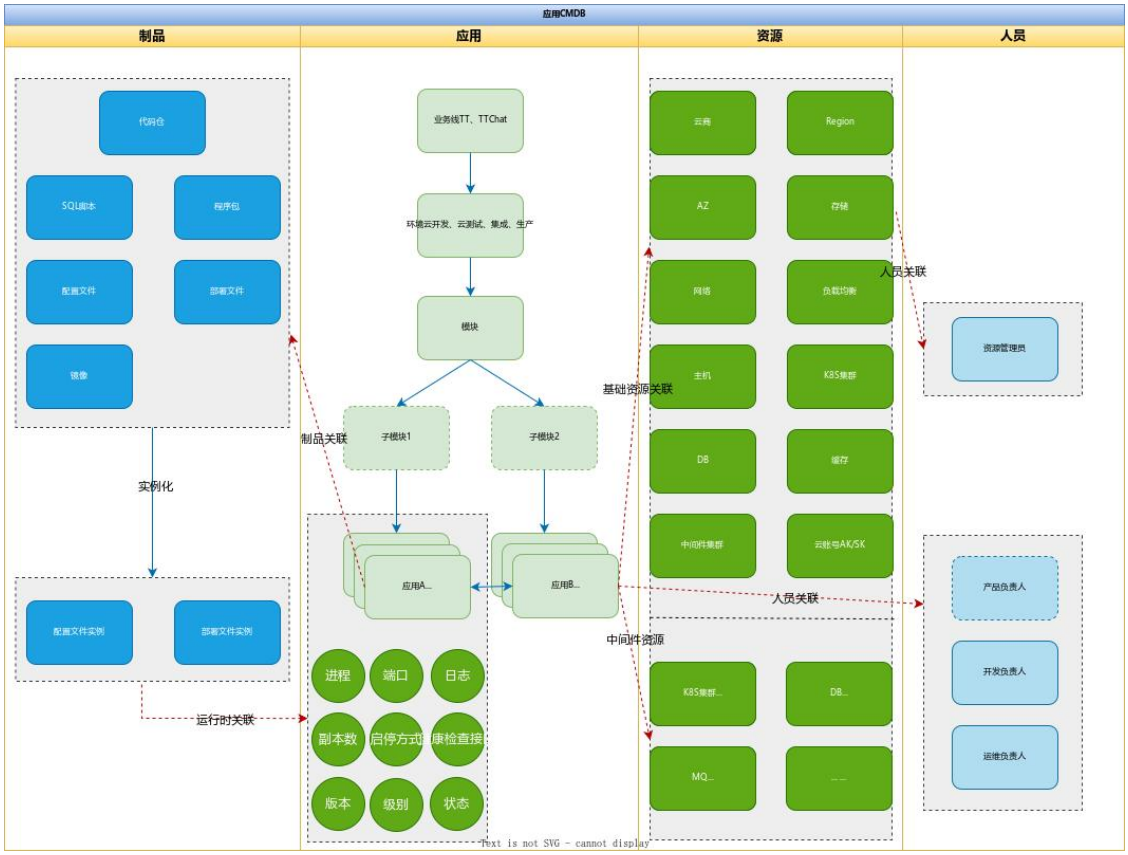
7 平台工程

7.1 标准应用平台工程建设

在当今多变的技术环境中，构建一个既能满足当前需求又能预见未来扩展的应用平台，对于任何规模的组织来说都是一个挑战。标准应用平台工程的建设不仅需要考虑到技术的先进性和可靠性，还要兼顾操作的简便性和整体成本的经济性。因此，我们的目标是通过标准化、模块化和自动化的方法，构建一个能够支持快速迭代、持续集成和高效运维的标准应用平台，以适应不断变化的业务需求和技术创新。

在本章节中，我们将详细探讨构建标准应用平台工程的各个关键组成部分，包括应用元信息平台的设计与实现，统一资源供给的策略和方法，以及扩展模块的集成和管理。下面将会讲述如何将这些组件融合成一个协同工作的有机体，确保整个平台的稳定性和可扩展性，同时提供清晰的管理界面和后端支持。

7.1.1 应用元信息平台



应用元信息平台旨在为整个标准应用平台工程的建设提供数据底座基础。一切的技术建设最后的核心关注对象都是应用本身，因此应用本身的各类信息、属性都需要被统一管理，诸如应用的唯一标识、应用归属（组织/业务/财务等）、应用属性（名称/应用类型/等级/保障机制等）、运行时信息（端口/健康检查等）、人员角色信息（负责人/研发/SRE/测试等）、应用关联资源（容器/主机/数据库/缓存等）。

1. 应用概念描述

(1) APPID：APPID 组成结构：..。各组成属性的命名，存在约束条件，仅可使用小写字母和减号-

(2) 业务域：业务域是指由一系列联系较为紧密的业务的集合，是业务对象高度概括的概念层次归类，目的是便于业务的管理和应用。

(3) 业务：一些能够为公司（或组织）达成某个目标或产生价值而进行的一系列活动。特定业务的业务能力取决于这个业务的类型。例如，保险公司业务能力通常包括承保、理赔管理、账务和合规等。在线商店的业务能力包括：订单管理、库存管理和发货

(4) 应用（服务）：在既有业务能力范围内，可以为每个能力或相关能力组定义应用。应用是企业业务组成的最小单元。

(5) 组织：一个组织整体的结构，是在企业管理要求、管控定位、管理模式及业务特征等多因素影响下，在企业内部组织资源、搭建流程、开展业务、落实管理的基本要素。

(6) 产品：财务提供的拆账粒度，大于业务，小于组织

(7) 应用场景：内部各个应用、运营系统通过 APPID 来进行应用的唯一定位，通过 APPID 进行相关资源的关联。

(8) 环境：资源所处的环境，如 PROD、PRE、UAT、FAT

(9) 资源（资源类型）：维持一个应用在各生命周期所需的资源，如容器、数据库、缓存、负载均衡、物理机等

2. 业务价值

3. 平台方

(1) 资源提供方作为租户接入，并注册需要关联到应用的资源信息；资源挂树/回收时，服务树会校验该租户是否有资源的管理权限，保证租户间资源隔离。

(2) 商品化资源可以定时向资源运营平台推送账单，资源运营平台也会检查该租户的身份（非该租户的资源不能接受此租户推送的账单）；最终会按照财务层级聚合账单推送给业务方。

(3) 租户可以基于平台来构建资源管理模型，使得自己的资源管理更贴合业务，租户可以将 Quota、预算、资源利用率、SLO 等基于平台来构建，通过这些措施，租户可以更了解业务方的资源使用情况， 实现更合理的资源调配

2. 业务方

(1) 贴合业务线：相比通过变动频繁的组织架构，贴合业务线的服务树更适合用于各团队在产品/项目工作中进行资源归属的管理。（业务资源的盘点可在应用元信息平台进行）

(2) 统一元信息：对于业务团队、计费中心、安全合规，有统一的资源 meta 信息，避免信息不对称并提升协同效率。

(3) 批量管理：资源挂树后，针对层级的单/多类资源的批量管理，包括新建、删除、配置、迁移等，就有了发挥的空间。

(4) 统计资源利用率：以帮助业务方提升资源效率，优化成本。

(5) 统计 SLO 信息：以面向用户落实稳定性保障的成果。

(6) 面向业务的权限管控：有了层级资源模型，结合服务树的鉴权模型，就可基于节点的人员信息进行权限管控，服务树基于 IAM 实现了一套灵活的认证鉴权机制。

7.1.2 统一资源供给

1. 应用概念描述

资源：资源是一个抽象的概念，由各资源平台决定资源的最小粒度。例如：容器平台中的一个应用、数据库中的数据库实例、对象存储中的 Bucket、一台物理机、负载均衡中的一个负载实例、一个业务域名，视频直播中的一路接入等，这些都是各云计算资源平台提供的一种资源。

2. 统一资源供给的价值

面向用户侧：平台工程的目标是通过对云资源进行抽象，以产品化思维，构建面向研发人员的自动化平台能力。通过向上屏蔽底层的复杂度，让研发人员能够不用太关注底层的复杂能力。进阶的平台工程期望能够实现底层资源的联动，比如操作负载均衡，向上联动域名，向下联动 RS 等能力，为开发者提供全套的云计算解决方案。

面向云平台内部：平台工程是很好的落地标准化的手段，通过间接约束研发人员的使用，逐步摒弃个性化的配置，使研发团队使用云资源更趋同和标准，大大减少云平台内部 SRE 团队的维护复杂度。

因为云资源的业务复杂性，甚至维护管理云资源的组织复杂性问题，很容易导致面向体系内用户使用的时候各产品/服务相互独立，交互体验、操作流程和概念标准不一致。

平台工程可以通过资源供给侧前端 UI 的标准化和流程的统一，确保用户在各个产品页面都能获得一致的用户使用体验，减少使用云能力的复杂度，可以拉通各个云资源标准化区域、可用区、环境和项目分级等，拉齐用户对多个产品的认知与共识，同时为未来的上层建设提供基础。

3. 统一资源供给的前提

(1) 具备一个公司通用的资源树（服务树/IAM），通过业务/项目/服务将资源进行挂载。

(2) 在业务/项目/服务这些维度维护关键角色，诸如业务负责人，成本接口人，SRE 负责人，并与云资源提供方和用户方尽量拉齐一套申请资源和审批资源的流程。

(3) 具备灵活的 BPM/任务流/工单体系

(4) 与各类资源提供方达成一致，用户对资源的使用统一收口在平台中，资源提供方尽量提供视角一致的资源增删改查接口，理想情况下平台工程团队对所有资源接口再做一轮标准化封装。

7.1.3 持续集成

CI 平台帮助开发者自动化构建-测试-发布工作流，持续、快速、高质量地交付产品，通过屏蔽掉所有研发流程中的繁琐环节，让工程师聚焦于编码。提供流水线、代码检查、代码库、凭证管理、制品库等核心服务，用于满足企业不同场景的需求。

1. 应用概念描述

流水线：CI 最核心的服务，将团队现有的研发流程以可视化方式呈现出来，编译、测试、部署，一条流水线搞定；

代码检查：提供专业的代码检查解决方案，检查缺陷、安全漏洞、规范等多种维度代码问题，为产品质量保驾护航。

代码库：支持如 GitLab 与 Gerrit 仓库代码源。每个应用下的流水线都会有默认的代码仓库，流水线运行时会拉取指定分支代码运行。开启子仓库、设置关联应用、多仓库构建时，流水线中运行的代码将会是多个仓库打包后的代码。

凭证管理：为代码库、流水线等服务提供不同类型的凭据、证书管理功能

制品库：基于分布式存储，可无限扩展，数据持久化使用对象存储。

触发机制：手工：在流水线列表后方点击执行；定时：设定时间自动执行流水线；自动：监听代码 PUSH 或者 MERGE 操作，执行流水线；代码评审触发：在代码管理评审设置中配置评审时触发。主要用于预合并验证代码。

流转方式：

未完成流转：执行了就会开始到下一任务（不关注任务结束时间、不关注任务结果成功或失败）

完成流转：任务执行完成开始下一接任务（需等待任务执行结束，不关注任务结果成功或失败）

完成且成功流转：执行完成且任务执行成功才开始下一任务（需等待任务执行结束，需要任务结果为成功），如果任务不支持配置流转方式时，则默认使用完成且成功流转

2. 核心功能

流水线主要包含【基本信息】、【流程配置】、【变量配置】、【安全管控】四个部分，本文主要介绍基本信息和流程配置部分：

3. 基本信息

一条流水线的基本信息包含以下配置项：

流水线名称：必填，可以重复。

代码仓库：根据应用设置中的代码仓库自动填充。

默认关联分支：用户手动执行流水线时，默认运行的分支。只能关联一个分支

触发方式：手工、定时、自动。详情参考 触发方式

开启集成模式：详情参考 分支集成

默认流水线：一个应用下只能标记一条默认流水线，标记后，生成版本时默认会选择该流水线执行。

4. 流程配置

按照各个项目的研发模式和业务诉求配置流水线阶段与任务。一个流水线中至少配置一个任务才允许保存。

阶段：通常我们按照开发流程将流水线配置为不同阶段。如构建阶段、并行测试阶段、集成测试阶段、发布上线阶段。阶段可以配置串行或并行。

任务：一个阶段中可以添加多个需要在这个阶段中执行的任务，一个阶段内的任务可以全部选择【串行】或者【并行】运行。目前支持的任务参考流水线任务与仓库支持

(1) 添加串行阶段。点击流水线【添加阶段】，或者流水线之间的【+】，可以添加一个串行阶段。

(2) 添加并行阶段。将鼠标移动到一个阶段下方的空白区域，会自动出现【添加阶段】按钮，这个时候可以在这个阶段下添加并行阶段。

(3) 添加阶段时，可以设置阶段内任务的串行或者并行。设置后，属于这个阶段的任务将按照串行/并行执行。

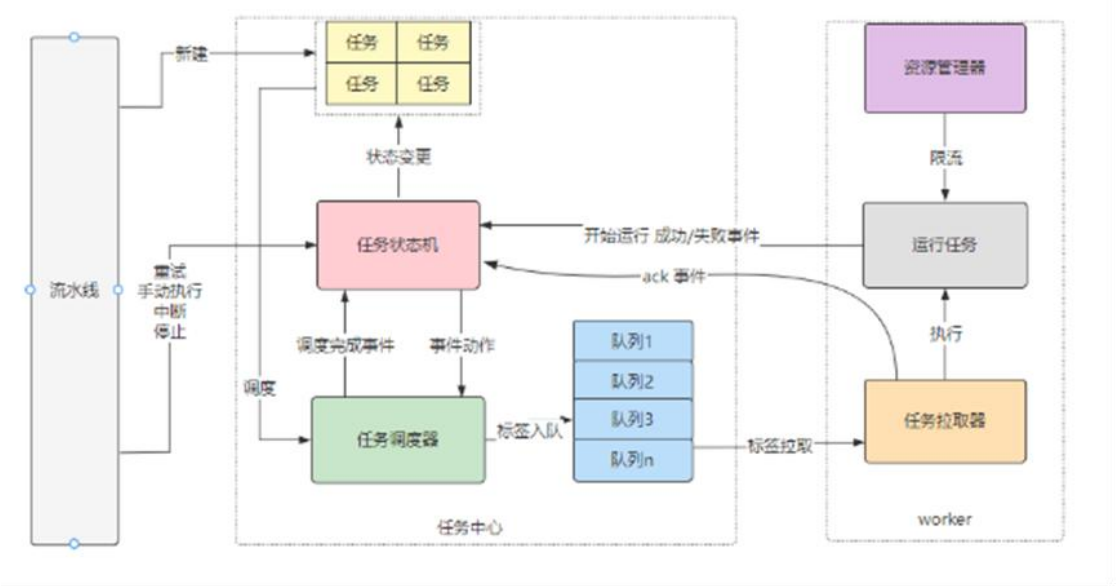
(4) 添加任务。可以点击阶段中的【+】来添加任务。

(5) 任务添加默认都是阶段内最后一个，你可以拖动任务来改变任务顺序。

5. 流水线设计简介

流水线作为持续集成的核心组件，是否健壮、易用直接影响着持续集成的建设水平。在交付高峰期服务不可用现象频出，严重影响着业务交付的顺畅度。同时业务方的构建资源没有做限制，部分业务方大量占用资源，影响平台的其他用户正常使用。

随着用户量、构建场景的增多上述的问题也会愈发严重，对于流水线的场景，资源最优解并不是强诉求，可通过任务拉取的方式，控制任务执行的频率，从而降低了资源的竞争和系统的可用性



(1) 增加任务状态机，管理任务状态流转

(2) 增加任务调度器，调度任务进入队列等待

(3) 增加任务队列，从原来的主动选择 node 节点修改为现在的 node 节点主动拉取

(4) 增加资源管理器，管理当前 worker 节点的资源信息

7.1.4 持续部署

持续部署将用户的业务实例组织起来，打通下游持续集成，并对接容器、虚拟机、物理机等物理资源，并为上游服务提供部署底座，为用户的业务提供部署的基础服务。

1. 应用概念描述

2. 实例

用户用于部署的最小部署单元，可以是单个独立运行的 K8s pod，一个 statefulsets 生成的 pod，也可以是云平台虚拟机、物理机等。

2. 分组

将多个实例组织起来的逻辑概念，用于支撑业务应用的并发或多活，同时负责为这一批实例提供统一的通用化配置选项等功能。

3. 包部署

包部署是分组的一种类型，其实例可以是单独的 K8s pod、云平台虚拟机或物理机，在使用方面通常用于提供有状态应用服务。

4. 镜像部署

镜像部署是分组的另一种类型，其实例由自研的 K8s statefulsets crd 生成的 pod 组成，通常用于提供无状态应用服务。

2. 核心功能

3. 部署设置

针对实例的定制化配置选项集合，定义了实例对外提供服务的基础能力。

基础信息，例如应用根目录、运行用户、日志目录等。

实例启停信息，例如资源池、JDK 环境（针对 Java 应用）、初始化及启停脚本等。

实例高级配置，例如实例对外服务端口、配置下载目录、分散策略等。

实例健康检查信息，例如 HTTP/TCP/脚本检查方式、实例健康检查端口/周期/延迟/失败阈值等，以及健康检查时的告警策略及自动恢复策略等。

2. 分组管理

与部署设置管理，对同样功能的实例进行统一化配置，主要包含：

基础信息，例如分组的类型（包部署/镜像部署）、关联的部署设置、是否灰度分组、注册中心及网关流量标签等。

规格信息，例如分组内的实例数，规格（CPU 核数、内存大小、GPU 型号及规格），磁盘大小等。

高级设置，例如实例是否保留固定 IP，接入配置中心的环境及版本，容器标签、注解、宿主机亲和性等。

3. 部署变更

从持续集成制品库获取应用版本后，对实例进行部署变更，可以定义当次变更的以下内容：

单个实例部署的超时时间。

分组间并发度及分组内并发度。

部署维度，定义是按分组维度来部署还是部署分组内的部分实例。

3. 业务价值

4. 部署能力

通过自研基于 dumb-int 的基础镜像，结合镜像仓库及持续集成等制品存储，持续部署实现了业务应用部署的全生命周期管理。

2. 自定义部署设置

基于大量丰富的部署设置实现了无状态及有状态两套业务容器方案，充分满足不同的业务需求。其中无状态容器通过自研 K8s statefulsets crd 实现，有状态容器通过 K8s 原生 pod 及自研 agent 实现。

3. GPU 资源利用

除了基础资源，持续部署还与 GPU 资源深度结合，允许用户创建带 GPU 资源的容器。

4. 健康检查及告警

通过 K8s 探针及 agent 建立容器的生命周期管理机制，同时实现了自动告警、自动拉起等预警保活机制。

7.1.5 部署编排

在持续部署的基础上，部署编排将跨应用的发布进一步组织起来，允许用户通过一个编排计划发布多个业务应用、降低发布复杂度，以及提升发布效率。

1. 应用概念描述

2. 编排计划

用于将跨应用的多个分组组织到一起的逻辑概念，由一个个编排任务组成，同时可以控制这些任务的顺序及串并行。

2. 编排任务

组成编排计划的最小单元，涵盖多个领域的不同任务类型，在编排计划中用于不同的分工。

3. 监控模板

通过打通云监控，实现了多个指标的对接，同时允许用户设置监控超阈时的动作用以及时止损。可直接与编排计划关联，用于监控整个编排计划的发布过程，也可在部署计划中自由添加并指定监控时长，用于单点验证。

4. 通知模板

通知模板用于在编排计划的生命周期内，对不同的时机发送不同的用户通知。

2. 核心功能

3. 编排实例部署

部署编排对实例部署的编排主要通过以下几个方面来实现：

发布顺序及串并行控制：用户可以将若干应用的多个分组编排进一个编排计划内，将其以用户自定义的串并行方式进行部署。

精细化暂停点管理：为用户提供丰富的实例暂停点控制设置：允许用户按实例、实例比例、全部实例等规则选择暂停点；允许用户自定义暂停点的手自动通过方式及卡点时间。

实例分批次发布：提供了实例分批次发布的能力，允许用户将单个分组内的实例划分为任意个数的批次，并拆分成不同的编排部署任务；允许用户在同一个分组的不同实例批次之间任意插入需要的其他任务。

2. 编排任务

编排任务主要分为三类：

核心任务：部署、卡点、监控。

其他任务：覆盖了诸如自动化测试、扩缩容、实例批量操作、静态部署、自定义脚本等与发布相关的各个方面。

定制化任务：用户可以自己定义任务的内容，以插件的形式接入编排。

3. 智能监控

通过与云监控的紧密结合，我们为用户提供了不同的指标、告警及使用方式：在指标及告警方面，当前已经对接了业务黄金指标、自定义指标监控（日志监控、自定义监控、指定时序数据库等）告警、调用链指标及量化发布指标（自定义指标及调用链接口）。

在这些监控能力的使用方面，我们提供了部署计划级监控及监控任务，既允许用户在整个发布过程中进行监控，同时也允许用户在某个指定时间段内进行监控。

4. 通知

在部署编排执行的过程中，虽然自动化程度较高，但有些情况仍需用户关注，因此我们允许编排计划及编排任务关联通知模板。在通知模板中，用户可设置以下内容：

通知时机：分为计划级别的时机和任务级别的时机，涵盖了计划及任务的生命周期。，时机包含编排计划的开始、成功、取消、超时，编排任务的开始、成功、跳过、超时、失败等；接收人可以指定角色，也可以指定具体的员工工号。

接收人：允许用户灵活配置通知接收人的服务树角色或指定的接收人工号。

通知渠道：除直接的 TT 消息外，允许用户添加告警机器人，在通知时自动发送 TT 群内消息。

3. 业务价值

4. 发布提质提效

通过编排计划将流程固化，以及将多个应用的发布流程串联，最大化地降低了发布过程中人为操作的出错以及跨应用之间的沟通延时及误差，从而降低了故障率及提升了发布质量。

同时，由于部署编排对接了测试平台及云监控，实现了发布自动验收及发布故障自动止损，进一步提升了发布质量及发布效率。

2. 精细化实例管理

除了允许用户为实例逐个设置暂停点外，部署编排还允许用户以下几个方式来管理暂停点：

按一定规则为整个分组的实例设定暂停点，例如全部实例、实例百分比、实例下标等。

为不同的实例定制暂停点手自动通过方式及卡点时间，从而便于用户更精细化地管理。

而对于实例本身的管理，部署编排还允许用户对单个分组的实例实行分批次管理，在批次之间可以任意插入编排任务，从而更好地管控实例发布。

7.1.6 可观测

为了确保我们的应用平台的高效运行，我们需要构建一个全面的可观测性框架。这一框架应包括以下几个关键组件：

(1) 日志管理：日志是可观测性的基础。我们将实现一个集中式日志管理系统，该系统能够收集、存储和查询来自平台上所有服务和应用的日志数据。此外，我们将采用标准化的日志格式和丰富的元数据，以便于搜索和分析。

(2) 监控与警报：监控系统将实时收集关于系统性能的指标，如响应时间、吞吐量和错误率。当这些指标超出预定的阈值时，警报系统将通知运维团队，以便他们能够迅速采取行动。

(3) 分布式跟踪：在微服务架构中，一个用户请求可能跨多个服务。分布式跟踪系统能够追踪请求的整个生命周期，帮助我们识别性能瓶颈和故障点。

(4) 服务健康检查：健康检查机制将定期检查服务状态，确保服务的可用性和可靠性。这包括从简单的 HTTP 检查到复杂的业务逻辑验证。

(5) 性能分析：我们将部署性能分析工具来收集关键应用和服务的性能数据。通过分析这些数据，我们可以优化系统配置，提升资源利用率和响应速度。

(6) 可视化仪表盘：所有的监控数据和分析结果都将通过一个可视化仪表盘展示，这使得任何非技术人员也能轻松理解系统的当前状态和性能。

(7) 事件管理和自动化处理：通过事件管理系统，我们可以跟踪和响应系统中发生的事件。结合自动化工具，我们可以实现对某些类型事件的自动化处理，减少对人工干预的依赖。

通过上述组件的整合，我们将能够实现对整个应用平台的深入洞察。这不仅仅是为了解决问题，更重要的是预防问题的发生。我们将能够实时监控应用的性能，预测潜在的系统瓶颈，并在问题影响用户之前主动地解决它们。

7.1.7 成本（定价、用量、出账）

1. 资源的计量计费：

资源的计量计费是企业内部平台管理中至关重要的一环。随着资源使用频率和范围的提升，业务部门渴望了解资源的使用情况，而成本作为一个统一的度量，能够为业务方提供量化的使用感知。因此，资源的计量计费成为必不可少的手段。为实现企业内部平台与公有云一样的计量、计价和出账能力，需要设计可监测且可计量的平台能力项，并确定其计价策略，监控统计各业务的服务用量，最终通过量价输出各业务的成本账单。

2. 计量

每个平台能力项对业务方提供的服务不一样，因此需要制定特有的计量指标，可参考以下通用计量模型设计流程：

(1) 定义平台能力项：在各公共平台部门中，根据服务功能的不同，制定面向使用方角度的平台能力项，作为计量的承载主体。

(2) 业务对象标签：定义平台所服务的业务方结算主体，主要目的是建立财务与技术之间统一的成本主体概念，可以直接引用应用元信息的业务概念或者以应用元信息为基础制定归集映射策略，与财务结算单元对齐。

(3) 成本驱动因素：针对平台服务功能的不同，每个平台定义相应的计量指标（根据平台的实际情况，可设计一个或多个计量指标）。计量指标应该是面向业务方角度，对业务方有感知的指标，例如任务调用量、审核量、数据存储量等。

数据采集与统计：平台能力项需搭建成本驱动因素的日常数据采集能力，为计量提供基础数据信息，至少实现 T+1 按天维度的数据采集粒度，能细化实现小时或者分钟采集粒度更优，计量应该以业务对象标签维度进行统计。

3. 定价

平台能力项的服务定价，也就是成本驱动因素的平均单价，参照以下公式：

单价=平台能力项资源 TCO/成本驱动因素最大月取值

- 平台能力项资源 TCO：包括该平台能力项所直接使用的资源成本总和；
- 成本驱动因素最大月取值：按照一定月份周期内，平台能力项统计到的每月成本驱动因素数量，取最大月份的数量；

4. 出账

确定单价之后，业务使用方对成本就有了稳定的预期。账单可以根据计量的颗粒度进行输出，为便于技术方和业务方及时了解成本变动情况，至少实现 T+1 按天维度的成本明细账单，能细化实现小时或者分钟成本明细粒度更优。

通过公式可以输出费用账单：成本=单价*用量

1. 技术方：对单价的负责，是平台能力项的成本责任方，通过对底层资源的合理选型、技术架构的优化、资源利用率的提升和资源管控的强化，推动平台能力项单价的优化，将优化成果转化为成本收益，提升平台的成本竞争力。

2. 业务方：对用量的负责，通过管理和优化业务应用实例的数量、存储量、生命周期、资源占用方式（共享/独占）、调用策略，降低对平台能力项的使用量，从而节约业务成本。

示例说明：

以 K8S 容器计算平台作为例子，按照每个集群进行定价，建立每个集群的成本驱动因素监控和统计，定期输出 K8S 算力成本账单。

平台能力项：K8S 容器计算服务，不同集群由于机型和地区不一样，采用不同单价计费

业务对象标签：业务 A，业务 B，业务 C ...（通过应用元数据一级模块归集映射，与财务结算单位对应）

成本驱动因素：考虑人力及时间投入，建立单一的综合指标-算力核时，包含 CPU 和内存用量按照一定比例系数折算的用量之和，比例系统可以参考行业或按照云厂商主机的 CPU 和内存成本比例统计。

监控和统计：T+1 天输出

平台能力项资源 TCO：以 K 集群为范围计算 TCO，K 集群 TCO=云主机成本+磁盘成本+日志服务成本+网络成本

成本驱动因素最大月取值：以近半年为周期统计，K 集群最大月取值为 5000 核时

K 集群单价=K 集群 TCO/5000 核时=xx 元/核时

账单：T+1 天输出，如业务 A 一天使用了 1000 核时，则业务 A 算力成本=1000*K 集群单价；如业务 B 一天使用了 1500 核时，则业务 B 算力成本=1500*K 集群单价

7.2 异构应用平台工程建设

异构应用（Heterogeneous Application）是指由多种不同技术、架构、编程语言或平台构建的应用程序。在异构应用中，各个组件可能运行在不同的操作系统上，使用不同的数据库系统，或者由不同的编程语言编写，由此而带来了一系列的运维复杂度，如集成和管理的复杂性、技术栈的多样性导致的维护成本等。

面向异构应用，平台工程建设需要考虑不同应用系统和平台之间的异构性，包括不同的数据格式、协议、接口等，通过技术手段实现数据的转换、集成和共享，通过管理手段，包括对异构系统的统一管理、安全管理、维护管理等。

7.2.1 总体设计

如图所示，面向异构应用的平台工程的总体结构可划分为 3 层，自下而上依次是：原子平台层、PaaS 层和 SaaS 层，其中 PaaS 层包括 aPaaS 和 iPaaS 两大核心能力，SaaS 根据场景分为一级 SaaS 和二级 SaaS 。除此之外，平台工程本身也需要提供平台服务管理和安全审计的能力。

7.2.2 aPaaS 结构设计

aPaaS，全称是 Application Platform as a Service，即应用程序平台即服务。与传统的 PaaS（Platform as a Service）平台不同，aPaaS 更加注重应用程序的开发和部署，而不是基础设施和资源的管理。aPaaS 平台提供了一系列的工具和服务，包括应用程序开发、测试、部署、托管、监控和管理等。开发人员可以使用这些工具和服务来快速构建、测试和部署应用程序，而无需关注底层的基础设施和资源管理。

aPaaS 主要提供 2 个能力，面向开发者的工具和面向应用的运行环境托管服务。

1. 开发者工具

2. 统一的后台开发框架

一套完善的前后台开发框架可以帮助开发人员更快地开发应用程序，并且可以简化代码的编写和维护。

(1) 分析并确定框架功能：需要提供统一登录、身份认证、安全防护等基础的功能，提供消息通知、日志记录、访问统计等通用的组件服务，提供多种数据库访问方案；提供基本的预处理函数或公共组件使用的代码编写样例等。

(2) 选择框架的技术语言：针对不同的编程语言，需提供相应的开发框架。目前，主流编程语言包括 Python、Golang、Java 等。应依据平台工程开发者的技能特点，提供所需之开发框架。

(3) 研发前制定编码规范：为了保证代码质量和一致性，除了最基础的命名规范、注释规范、代码格式外，还需要包括平台级的公共变量规范、系统变量规范、错误码规范、日志输出规范等，并通过代码审查等手段确保开发者遵循规范。

2. 通用的前端组件库

一组根据业务场景和产品特性预先定义好的前端组件和代码库，在进行原子平台开发/SaaS 开发的过程中，可以帮助开发人员快速地构建出高效、可维护的产品页面。

(1) 制定设计规范：由产品经理和设计师主导，根据行业标准和业务场景，制定前端组件共同遵守的设计规范，确定最基本的设计元素，包括颜色、字体、图标、网格和布局等。

(2) 交互设计：基于基本元素，设计一套常用的 UI 组件和交互模式，如按钮、表单、导航、对话框、搜索等。保证各产品间一致的视觉交互体验，延续相同的用户习惯，减少认知上的理解差距，减少无意义的探索和重复设计。

(3) 统一组件实现：组件包含了通用的场景和产品的经典交互逻辑有前端组件，如横向导航必要字段与交互方式、页脚 footer 的通用内容与交互、项目空间选择器、无权限通用页面、版本日志展示框等，更能提升研发效率及平台的一致性。

(4) 持续更新前端技术语言：一些著名的通用的前端组件库包括 React、Vue 和 Angular 等。这些组件的技术语言都在不断发展，选择了一个语言后，对应的前端组件库要不断更新。

3. 可视化开发（LessCode or LowCode）

可视化开发（LessCode or LowCode）：是一种可视化编程语言，用于创建可拖放、可配置的界面和交互式应用。

(1) 分析并确定产品功能：需要提供的基础功能有可视化代码编辑器、拖拽式的前端页面生成方案，还需要提供在线调试和预览能力，与 PaaS 的应用托管能力打通，支持一键部署并发布应用。

(2) 前后台协作模式：这种模式需要预先定义好前台和后台的数据和协议规范，开发者先以前端的身份在可视化开发平台生成需要的产品 Web 页面，再以后台开发者的身份编写业务逻辑、处理数据存储等。

(3) 可扩展的模板市场：提供常用的前端交互逻辑函数，这些函数可以统一使用 JS 语法编写代码，通过发起 Ajax 请求，获取接口数据，转发后台配置，绑定组件事件，配合页面生命周期等，实现不同的功能。

(4) 可扩展的模板市场：提供官网类、后台管理类、公告类等应用级模板，包含应用完整的功能，如：函数，变量，数据库等，可直接基于应用模板创建新应用，无需从 0 到 1 进行组装，只需要将模板对应内容进行修改即可快速完成应用的开发。

4. 增强服务（如 AIDev）

在提供了基础的开发者工具后，可以结合当下先进的技术，降低工具的使用门槛，提高工具效率。

(1) AIDev：通常指的是人工智能（AI）开发（Dev），通过这个技术可以与现有产品的资料查询、技术支持、代码生成等场景结合，进一步降低工具的使用门槛。

(2) 微前端：一种前端和后端协作开发的模式，可以在大型的、复杂的 SaaS 研发中使用该技术方案，将组件拆分成多个小型的、独立的服务，通过 API 进行通信和协作。只要提前预先定义每个组件都要遵循的一套标准和规范，就可以将这种模式应用到 Web 应用、移动应用、桌面应用等。

2. 运行时环境托管服务

3. K8s 容器托管

Kubernetes（通常简称为 K8s）是一种开源的容器编排和管理平台，它本身特性有自动化容器部署、升级、回滚、监控、扩展和负载均衡等。SRE 更侧重于建设产品化 Web 页面，让开发者无需掌握专业的 K8s 技能，无需关注基础设施细节，也能使用容器托管能力。

(1) 规范用户操作路径：将代码库、开发框架、可视化开发等开发者需要的资源通过产品化的 Web 页面呈现出来，开发者就可以按照“我要开发--创建应用--填写应用基本信息--选择开发框架--选择代码库”的路径完成 SaaS 开发前的简单准备工作；当开发者完成业务逻辑代码后，再继续按照“应用部署--部署到预发布环境--部署到生产环境”的路径，完成应用的发布上线。

(2) 提供最佳运行资源：每一个应用运行时，提供基础设施的基本要求，如运行时 Pod 的副本数、CPU、内存、存储要求等。

2. 应用部署流水线建设

制定开发的代码分支管理规范和制品晋级规范，可以进一步减少产品-研发-运维-测试各岗位间沟通成本，快速建设应用部署的流水线。

(1) 规范代码分支管理规范：用语义化版本号“主版本号.次版本号.修订号（Major.Minior.Patch）”，严格遵守版本号递增的规则。

a. 主版本号 Major：

产品侧：大功能，影响现有用户的使用习惯，新增亮点服务

研发侧：不兼容的变更，影响软件的升级与维护，影响基于该软件二次开发的服务，如架构升级，研发语言变更，数据结构重组，API 下架或 API 不兼容

运维侧：停机升级 / 数据迁移等，其他产品关联产品也需要配合升级

b. 次版本号 Minior：

产品侧：新增功能

研发侧：接口新增/接口调整（向下兼容）/性能优化等; 原则上不允许 API 下架

运维侧：升级前需要查看下升级文档，允许做一些数据迁移相关的操作

c. 修订号 Patch：

产品侧：展示/交互/描述等小调整，例如文案的修复等

研发侧：bugfix，不会新增功能

运维侧：为了解决一些 bug 或安全问题，不会新增功能

(2) 规范应用制品晋级标准：选择并使用几个关键节点作为制品晋级的必经阶段，定义制品的生命周期，并约定制品晋级的准则。如 1.0.0-alpha 表示当前应用的版本号是 1.0.0，该制品处于 alpha 阶段，仅完成了开发自测，可以发布到测试环境，提交给测试发起测试验收，禁止发布到预发布环境等其他环境。若需要晋级为 beta，则需要通过测试验收，才可以发布到预发布环境。

(3) 提供把应用发布到某个环境的 CD 流水线插件：通过提供必要的应用基本信息，密钥信息，版本号，制品等级，环境信息等，使用 YAML 或者 Json 配置的方式，就能实现应用发布。

3. 扩展服务（扩缩容、CLI 等）

(1) 可扩展的存储服务，如为应用开发者提供存储服务，默认提供关系型数据库 MySQL、非关系型数据库 Redis，可扩展支持非关系型数据库 MongoDB、时间序列数据库 InfluxDB、搜索引擎数据库 Elasticsearch 等；

(2) 扩缩容能力，在 K8s 能力的基础上，针对应用运行时需要的资源进行实时监控，确保对计算、存储、网络等资源要求是合理的，并能弹性扩展，以满足不同的业务需求；

(3) 命令行界面 CLI（Command Line Interface）功能，不需要鼠标和图形化界面，而是通过键盘输入命令来完成各种操作。需要提供应用信息查询，应用部署，部署结果/历史查询等功能，支持 Linux、Windows 等主流的操作系统。

7.2.3 iPaaS 结构设计

iPaaS 是 Integration Platform as a Service 的缩写，翻译为集成平台即服务。在整个工程平台的设计中，iPaaS 连接上层的 aPaaS 和下层的原子平台，需要对已有的原子平台进行自动化的管理，自动化地接入外部第三方系统，实现对上提供统一的标准协议 API，可以让使用 aPaaS 的开发者们，快速拥有 SaaS 的组装能力。

1. 接入端设计

(1) 原子平台提供 API：所有原子平台都通过 API 的方式，向上提供平台本身的核心能力。

设计网关的接入流程：让原子平台可以最低成本注册 API 资源。提供可视化的 Web 页面，首先需要“创建网关-完善网关基本信息”即是原子平台的基本信息，然后逐个“新建资源”，填写注册 API 的前端“请求方法、请求路径”，后端“Method、Path、Hosts、超时时间、Header 转换”，完善安全设置（如 API 使用的认证方式），实现一个 API 的有效注册。

提供 API 的管理能力：一个原子平台可能会提供数十个具体的 API，在 API 创建过程时，要提供可视化的 API 配置和文档编写页面，提供将 API 发布到不同的环境进行测试的方案，可以指定 API 进行特殊操作，如限制频率、授权使用、下架等。

(2) 开发者使用 API：用户能看到的 API 要确保文档内容准确，并可以快速验证 API 可用性。

API 文档：要提供可供搜索的 API 文档，且采用统一的格式编写 API 文档，文档内容应该包括 API 基本信息（如更新时间、是否要申请权限）、API 地址（如请求环境、请求方法、请求地址）、公共请求参数、输入参数说明、调用示例、返回结果、返回结果说明等。

通用 SDK：要提供自助发布 SDK 的能力，每个原子平台可以将自己的 API 发布成 SDK，开发者通过下载并安装对应的 SDK，即可使用原子平台的全部 API。

在线调试：提供在线的测试环境，让开发者可以通过 mock 的方式，输入必要的参数，得到文档描述的预期效果。

2. 管理端设计

(1) 确定网关服务模式：

选择流量网关：流量网关在 iPaaS 架构中的作用是全局性的，与后端服务完全无关的策略网关。不支持代码扩展。如一般使用 IAS，底层一般使用 Nginx，也可以选择 ngx_lua 。

设计业务网关：业务网关，即 API 网关， 它与后端服务（如原子平台，业务系统）有一定的策略逻辑的网关。支持代码扩展，提供公共的能力（如限流、鉴权、监控等），不涉及后端服务具体的逻辑，不涉及 API 具体功能。可以采用插件的模式提供这些公共能力，需要设计一套插件的加载流程。

插件扩展功能：插件的内部要采用统一的结构，包括初始化 init、检查外包依赖 check_schema、rewrite、access、log 等，这样可以对插件进行统一管理。依照这样的结构化模式，可以提供协议转换、路由寻址、负载均衡、服务发现、流量管理、安全防护、服务治理等插件，增强 API 网关的能力。

(2) 建设插件生态：

插件文档：文档内容遵守统一的规范，包括插件名、插件版本、功能描述、属性字段、启用插件步骤、测试插件的效果、删除插件方法。

插件市场：提供“插件模板”、“插件商城”等功能，使插件开发者和插件使用者快速分享并启用感兴趣的插件。

7.2.4 通用原子设计

平台工程需要建设多个方面的能力，而原子平台位于整个平台工程的最底层，它的能力就组成了整个平台工程的最基础的核心能力。以下是在平台工程建设中，SRE 推荐的首要的、基础的原子平台。

1. 配置管理平台

(1) 主机管理能力：提供主机信息的管理，支持主机信息的编辑，如导入导出，修改，复制等基础操作，可以扩展主机相关的字段，主机所属的运维人员，业务信息，SLA、城市等；主机信息不仅包括企业自建机房，私有云的主机，还应该可以同步各类云主机，并无差别对主机进行统一管理；主机信息变更可以通过订阅等方式，与周边系统进行实时同步与修订。

(2) 空间管理能力：可以按照“业务”“项目”等空间管理的概念，划分主机的管理空间，并在这种空间维度上，对主机进行分类，如空闲机、故障机、待回收等；建设主机的默认拓扑关系，如“业务-集群-大区-模块”，可以通过“服务模板”“集群模板”的方式，快速创建同类型的主机拓扑关系。

(3) 模型管理能力：“主机”的模型是最基础的配置管理信息，支持抽象，自定义创建更多的模型，如交换机、路由器、负载均衡、防火墙等。按照设备类型，组织架构，职能等多维度自定义模型，并使用对应的模型数据，进行配置信息的管理。

(4) API 能力：作为基础的配置管理信息，需要提供全面的 API，供消费方使用，并要封装部分场景类 API，提高 API 的性能。

(5) 查询能力：围绕主机的全文检索能力，要有基本的 IP 信息查询，还要建设动态分组查询的功能，以适应不断变化的配置基础信息。

(6) 云资源管理能力：对于高频使用的云产品，如 IaaS 层的产品，必须进行纳管，并提供针对全生命周期的操作管理功能（申请、购买、新建，操作，配置调整，回收，销毁）。对于云上 PaaS 层的产品，不同的云，有不同的形态，尽量按照统一管理模型纳管，如 CLB、COS 等。

2. 作业管理平台

(1) 脚本执行：支持通过手动编写、本地上传的方式写脚本，脚本支持 SRE 常用语言，包括 Shell、Bat、Perl、Python、Powershell、SQL 等；需要有“脚本管理”的功能，常用的脚本可以通过“脚本引用”的方式被其他脚本再次使用；选择目标执行的主机后，通过“滚动执行-滚动策略-滚动机制”的方式，能分批快速完成脚本执行。

(2) 文件分发：支持从本地或从服务器上选择源文件上传后分发的模式，满足“一对多”“多对多”“多对一”等多种分发文件场景，且能自由控制上传/下载的速率，并设置文件分发超时时长；可以通过“滚动执行-滚动策略-滚动机制”的方式设置批量文件上传的滚动方案。

(3) 任务编排：支持将脚本执行、文件上传、定时策略等多线任务进行编排，组装成一个作业，可以提前预置作业“执行方案”，在执行的时候，输入必要的参数即可。

3. 容器管理平台

(1) 集群管理：提供 K8S 原生集群创建的能力，支持自定义设置 Master 和 Node 节点，一键自动安装集群组件，在用户独占集群时，保证安全隔离性；提供集群导入的能力，支持通过集群 kubeconfig 文件导入外部集群，对外部导入集群和已有的集群统一在容器管理平台进行管理；支持通过公有云的云凭证导入云上 K8s 集群，支持 Worker 节点自动扩缩容；集群管理支持节点添加和删除，集群删除，支持节点标签、污点与资源调度等节点管理功能，支持集群和节点级别的监控告警及主要数据的视图展示等。

(2) 模板管理：提供在集群中部署资源的管理方案，支持设置容器编排的 Helm Chart。可以将同一套 Helm Chart，实例化到不同的命名空间，通过不同的 values，完成差异化的资源编排。

(3) 应用管理：提供容器视图功能，可以通过应用视图或者命名空间视图管理容器，查看应用、POD、容器等的在线状态，启停容器，重新调度容器，对应用做更新，例如扩缩容、滚动升级等。

(4) 镜像管理：提供公共镜像管理，包含了一些实用程度比较高，且开源共有的镜像资源。公共镜像对所有用户可见；提供项目私有镜像管理，是项目成员主动添加的镜像，或者是通过 CI 流程归档的私有镜像。项目私有镜像只有项目中指定的权限所有者才能访问。

(5) 网络管理：提供服务管理能力，可以查看服务的列表，以及每个服务的详细信息，对服务进行操作，例如更新服务或者停止服务；提供负载均衡器管理能力，查看线上负载均衡器列表，以及每个负载均衡器的详细信息，启动、删除或者更新负载均衡器，并提供多云负载均衡器管理的能力。

(6) Workload 场景能力：针对业务场景定制面向无状态服务的 Deployment 和有状态服务的 StatefulSet 功能，如 Deployment 支持 Operator 高可用部署、支持滚动更新 / 原地更新、支持设置 partition 灰度发布、支持分步骤自动化灰度发布、支持 HPA、指定 pod 删除、支持 pod 注入唯一序号、支持防误删功能、支持 Readiness Gates 可选功能等；StatefulSet 还

支持 Node 失联时，Pod 的自动漂移、支持并行更新、支持 maxSurge / maxUnavailable 字段等。

4. DevOps 平台

(1) 流水线：提供基本的流水线管理功能，如创建，查看、删除、设置标签等；提供可视化的 UI 界面来编排流水线，通过 Task（插件）-Job（作业）-Stage（阶段） 的结构组成一个可视化 Pipeline（流水线），提供手动触发、定时触发、Commit 触发、制品库变化时触发等方式，最终可以让流水线的产出物，如 Artifact（构建），归档到指定的仓库中。

(2) 代码分析：提供静态代码分析的自动接入功能，通过插件的模式接入到流水线中，就能检查源程序的语法、结构、过程、接口等，找出代码隐藏的错误和缺陷，如内存泄漏，空指针引用，死代码，变量未初始化，复制粘贴错误，重复代码，函数复杂度过高等，并将检查结果汇总，输出数据报表，给出修复的指导建议。提供自定义规则，可以在特定的场景对屏蔽代码分析。

(3) PreCI 功能：通过 JetBrains IDE 插件、VSCode IDE 插件 和 PreCI 命令行工具等方式，提供基于开发者常用编辑器的本地代码检查功能，在打开文件/保存文件时，秒级返回检查结果；提供本地的规则集配置与代码分析插件的线上规则集同步功能，保持规则集一致；支持在 IDE 中查看检查结果，双击某个问题即可定位到其所在编辑器代码中的位置；支持在 IDE 中忽略问题和标记问题，支持注释忽略；提供 pre-commit 和 pre-push 功能，实现不符合质量要求的不能提交到代码库。

(4) 质量红线：提供一个“制品是否准出”的质量红线检查插件，插件可以综合项目团队内的要求（如单元测试，自动化测试通过率）和代码分析检查的指标（如代码缺陷、代码安全、代码规范、重复代码、复杂度等）设置阈值来决策代码质量是否符合预期，是否需要被红线拦截。

(5) 编译加速：使用分布式编译、编译缓存等技术方案，支持 C/C++ 编译、UE4 代码编译、UE4 Shader 编译等，提供加速效果总览、任务注册、任务列表、加速记录等功能；编译构建机可以选择公共构建资源，也可以支持用户自行导入并注册构建机。

(6) 环境管理：提供构建环境管理，用于在流水线中提供编译构建环境；提供服务器环境管理，开发人员可以将流水线中构建好的包（或者是自定义仓库中包）发布到服务器环境中，完成软件的功能验证或性能测试。

(7) 凭证管理：管理的凭证类型要有密码、用户名+密码、SSH 私钥等，不同的凭证类型可以应用在不同的第三方服务当中。比如，通过 http 方式拉取代码库时，需要用到用户

名密码+私有 token 凭证类型；而推送镜像到某个镜像仓库，则可能会用到用户名+密码凭证类型。

(8) 研发商城：提供一个 DevOps 内容发布和聚合的中心，插件开发者、IDE 插件贡献者、流水线模板贡献者、容器镜像开发者都可以将自己的作品发布到研发商城，对应的用户可以在研发商店快速检索、选择并安装到项目下使用，以此将 DevOps 最佳实践得到借鉴和推广。

5. 其他各类平台

包含但不限于：

(1) 制品库平台：提供将 maven、rpm、npm、docker image、helm chart、二进制包等产物归档并分享的功能；结合流水线，提供制品库插件功能，直接归档 DevOps 阶段产品的制品；提供制品查询功能，可以根据制品名字、更新人等查询；提供制品的操作管理功能，包括重命名、移动、复制、删除、共享、下载等。

(2) 代码库：在关联 GitLab、GitHub 等代码库时，提供源代码地址和访问凭证授权关联代码库；可以对代码库进行重命名，删除等操作。

(3) AI 平台：提供贴合企业需求的“场景方案”，如高危命令识别、单指标异常检测、多指标异常检测等；提供用户可以自定义创建的“算法模型”，可以接入样本集，经过特征工程、模型训练、评估，将训练好的模型发布，并应用到企业中；提供“样本管理”功能，可以进行查看、复制、删除等操作。

7.2.5 SaaS 分级

在异构应用平台工程建设过程中，针对 SaaS（Software as a Service）生态建设，可以根据其功能和服务的复杂程度进行 SaaS 分级，通过把各种共性场景的 SaaS 抽象出来成基础 SaaS，并提供原子能力，再基于之上组合各种特定场景 SaaS，以便用户能够更低成本地快速构建覆盖不同应用场景的 SaaS，减少重复的开发工作量。通常可以分为以下两大类

1. 基础 SaaS（一级 SaaS）

基础 SaaS 是指通用的、可复用、可抽象原子能力的 SaaS，包含通用的功能模块和功能点，例如通用发布、通用监控、通用配置管理、通用流程管理、大屏表单编排管理等。基础 SaaS 可以为场景 SaaS 提供原子调用能力。示例如下

基础 SaaS	功能描述

基础 SaaS	功能描述
通用发布	提供自动化作业的编排能力，支持对接制品库、负载均衡等系统，支持复杂逻辑如分支、并行等编排方式，实现发布执行作业全流程自动化
通用监控	提供流程引擎、表单引擎、值班管理、移动端等能力，支持根据运维工作需求自定义编排 IT 服务流程，并在流程中集成监控数据、配置数据、自动化执行作业等能力，可以将工作流和执行流无缝衔接
通用配置管理	提供可扩展的配置自动采集能力，支持云资源、数据库、中间件等 IT 组件的配置自动采集；提供数据质量运营服务，通过度量促进数据准确性工作的开展；满足资产管理、安全管理、故障排查等场景的数据查询需求，提供各运维角色的配置管理视图
通用流程管理	提供可观测数据（指标、日志、调用链、事件、性能、告警）的采集、处理、存储能力，基于不同的数据类型有不同的数据处理能力，每个服务都可以独立配置和复用，满足上层可观测场景的消费需求
大屏表单编排管理	提供拖拉拽方式组装的大屏和报表开发能力，支持从关系数据库、非关系数据库、API 等数据源获取数据，支持自定义的指标计算，用于运维各场景的度量统计和可视化展示

2. 场景 SaaS（二级 SaaS）：

场景 SaaS 是指针对特定垂直领域或特定业务场景的 SaaS，包含特定的功能模块和功能点，特点是给用户提供符合其用户习惯的定制化界面，用户体验大大提升。如面向单业务的发布、变更、配置管理等 SaaS。一些较轻量化的场景 SaaS 还可以基于 LessCode 或自动生成等方式来快速实现，高效满足不同业务特定场景的需求。

SRE 流程中可靠性架构设计、研发保障、入网控制、发布管理、故障应急、上线后持续优化工作等环节均可根据工作需求进行场景 SaaS 拼装，实现线上化操作。示例如下：

SRE 流程	场景 SaaS	场景说明
发布管理	统一发布中心	在通用发布的能力基础上，增加制品管理、参数管理、流程管控等能力，对接云上云下资源，对主机类应用和容器类应用的发布进行统一管理，通过发布方案和回滚方案的管理，有效控制发布质量
故障应急	统一告警中心	建设全链路可观测能力，整合 CMDB、trace、log、metric 数据、自动排查作业、自动恢复作业形成排障决策树，缩短故障恢复时长
故障应急	故障诊断中心	建设全链路可观测能力，整合 CMDB、trace、log、metric 数据、自动排查作业、自动恢复作业形成排障决策树，并通过告警快照等方式协助故障排查，缩短故障恢复时长
故障应	应急管	对应急预案、应急组织、应急流程进行统一管理，将自动化恢复作业关联到应急预

SRE 流程	场景 SaaS	场景说明
急	理中心	案中的故障场景上，识别到对应故障即可进行应急恢复。支持应急演练线上化管理
上线后持续优化	运营成本分析	通过资源负载数据的统计分析和趋势预测，对低效无效资产及业务高峰带来扩容需求进行识别，并基于规则给出资源优化建议，提高资源运营精细化程度

除以上示例场景外，通过 iPaaS 的原子能力和一级 SaaS 的通用能力进行更多场景 SaaS 拼装，可有效减少运维琐事，如应用健康度评分、容量管理、错误预算统计等场景均可通过场景 SaaS 结合不同技术实现线上化、自动化、智能化

7.2.6 服务管理

在异构应用的平台工程建设中，涉及一些公共的、需持续投入的服务管理功能建设，包含但不限于：平台可观测性、计算资源调度，服务计费等能力。

1. 平台可观测（日志、监控、APM）

在平台工程建设中，从 aPaaS、iPaaS，到原子平台，各类 SaaS，都需要一个完整的可观测服务。

(1) 数据采集：提供监控采集插件，通过下发采集插件的方式，获取目标服务的数据，这些数据按照一定的采集格式，可以支持自定义数据上报；数据类型支持指标数据（Http 简易上报，Prometheus SDK 进行 PUSH），Trace 数据（符合 OpenTelemetry 协议），事件数据（HTTP JSON 数据，命令行工具，SNMP trap 上报）等，还可以接入告警源（如 Zabbix、Open-Falcon、Prometheus、腾讯云监控）。所有的数据都支持容器内的数据采集。

(2) 数据检查：每个采集任务都要提供一个可视化的检查视图，可以查看当前主机/实例采集的数据情况；

(3) 数据探索：提供数据检索功能，仪表盘功能，场景视图，视图报表等功能，将采集的各类数据（Metrics、Events、Logs、Traces、Alerts）可视化展示出来；

(4) 策略配置：提供针对不同数据的检查或者告警策略，并通过 iPaaS 调用原子平台的能力，如作业管理平台，触发告警的产生后的运维操作。

(5) APM：提供 APM（Application Performance Monitoring）即应用性能监控功能，通用原子平台或者 SaaS 通过 SDK 上报数据，就可以展示应用内的 Trace 调用链，对应用进行排查故障，常规巡检，设置主动告知应用问题的策略等。

2. 服务治理

(1) 服务可用性发现：提供统一的服务可用性 API，如 healthz 接口，上报服务的可用性状态；通过“观测服务”的能力，将服务注册到监控的仪表盘中；

(2) 资源调度：使用容器管理平台，通过可视化的 web 页面统一管理平台工程的所有容器资源使用情况和在线状态，根据服务的负载、可用性、性能等因素，对服务进行调度和优化。

(3) 故障自愈：这是服务优化的一个重要目标，SRE 根据平台工程架构和系统维护经验，对服务的监控和调度结果进行分析，提供当故障发生时的一个确定的恢复路径，并复用各类原子平台能力，实现工具的自动化。

3. 服务计费

(1) 确定计费模式：根据各产品服务场景，提供按时间，请求量，数据量，次数，带宽，套餐等计费方式。

(2) 采集使用数据：在产品内按照用户维度或者业务维度埋点并采集使用数据，可以通过日志记录、计时器、硬件时钟、第三方系统（如 Google Calendar、Time.is 等）、API 等方式采集数据。

(3) 建立服务账单：账单需要考虑成本、用户体验、收益等各方面的因素，按照用户或者业务的维度，统计所有平台工程内的使用数据，给出按照天/月的账单，或者打包整体账单。

7.2.7 安全与审计

异构应用的平台工程由于涉及的模块多，涵盖多种技术栈，不同原子平台之间存在相互的数据通信，访问的用户角色千差万别，必须有统一通用的安全策略和审计标准，实现一致性和标准化的管理。

按照用户的操作路径，安全与审计主要体现在如何保护平台的安全、防止数据泄露、规范资产合规，以及如何全方位地跟踪和记录平台的使用情况。需要建设以下几个方面的能力。

1. 用户身份认证

(1) 确定使用哪种认证方式：一般要提供最基础的用户名和密码认证方式，对于高敏感的平台，要提供 token，短信验证码等辅助认证方式。当认证通过后，用户可以获取平台的访问许可；当认证不通过，要给出明确的提示反馈，如密码错误，用户名不存在等。必要

情况下，提供双认证方式，当用户名和密码认证方式生效后，可以启动短信验证码的认证方式。

(2) 开放认证接口：让原子平台、SaaS 开发者可以通过接口集成统一的用户身份认证，提供 OAuth 2.0、OpenID Connect 等身份验证协议，方便让第三系统对接平台工程。

2. 权限管理

根据平台的功能和用途，设计授权策略，比如明确的授权范围（如某功能点的使用、某实例的数据访问等）、授权级别（如普通用户、管理员、高级用户等）以及授权期限等。

需提供集中统一的权限管理服务，支持基于 aPaaS 开发框架和第三方平台的权限控制接入，做到细粒度的权限管理。

(1) 权限接入：提供不区分技术栈的权限接入方案，进行统一的权限管理；给通用原子平台和一级 SaaS 提供权限接入的 SDK（Python、Go、Java、PHP 等），使用 SDK 进行鉴权逻辑校验；给二级 SaaS 提供直接鉴权的方案；提供统一的无权限页面、申请权限页面、申请返回信息页等通用的逻辑界面。

(2) 权限申请：提供 3 种主动申请权限的方式，申请加入用户组、申请自定义组合权限、从接入系统侧无权限跳转到统一的管理端申请权限；权限申请时支持拓扑实例选择、属性条件两种实例选择方式，申请人也可以组合选择所需要的权限；支持多个系统批量提交申请。

(3) 权限模板：在接入的系统超过一定数量，单个系统内，系统之间的权限组合关系较多的情况下，提供一个可复用的权限集合，通过权限模板的方式，把权限授权给用户或者用户组，实现权限模板更新后可以同步给已关联授权的用户组。

3. 数据保护与加密

(1) 确定需进行保护的数据范围：最基本的敏感信息是用户个人信息，如用户的手机、电话、邮箱；根据国家政策、行业属性、企业要求也会有敏感数据的规定，如在互联网领域敏感信息有主机账号和密码，公有云的账号、密码、token，资产数据等。这个范围可以参考《中华人民共和国国家标准-信息安全技术网络安全等级保护基本要求》；根据平台工程的特性，系统初始化或运行中的配置文件，各系统间的数据传输，也需要加密。

(2) 选择加密算法：一般的产品是默认要支持国际加密算法的，建议要支持国家商用密码，简称商密，拼音缩写是 SM。其中 SM1 和 SM4 是对称算法，对标 AES；SM2 是非对称算法，对标 RSA、ECDSA；SM3 是摘要算法，对标 MD5。可以按照具体产品的场景选择使用合适的加密算法。

(3) 加密方案实施;存储中的数据，采用对称或非对称加密技术，确保储存在数据库或文件系统中的敏感数据得到保护。传输中的数据，使用 SSL/TLS 协议加密客户端和服务器之间的通信。产品页面上提供明文、密文统一的交互和表现形式，如密码统一显示 6 个*。每个技术栈要提供统一的加密 SDK 等。

4. 操作行为审计

(1) 确定用户操作的日志格式：梳理各平台/系统的操作审计日志协议/标准，规范各平台/系统的操作行为输出内容要求，提供丰富的日志接入方式，如：日志采集、API 推送、各种技术栈的 SDK 等。

(2) 建设统一审计中心：提供数据上报的 Web 端页面，系统管理员可以把操作审计日志接入到统一的审计中心；提供数据检索的功能，支持多维度检索权限范围内的系统操作日志；提供数据存储功能，支持平台管理员切换设置审计中心的存储，提供设置默认存储功能；提供审计策略的管理功能，支持用户新建、编辑、删除、克隆、启用/停用常规策略，当有用户行为触发审计策略后，支持查看命中审计策略的审计异常事件，支持对异常事件的检索；当审计数据量变多，各系统之间的操作日志进行关联审计的时候，可以通过 AI 策略发现异常数据。

5. 风险发现、分析与处理

从最上层的 SaaS、到 aPaaS、iPaaS、原子平台，各系统本身在设计的时候，就通过日志的方式，上报风险，并借助监控等可观测的能力来发现风险。建设面向异构应用的平台工程，除了保证平台工程本身服务可用性，更重要的是在风险发生后，有明确的处理流程，快捷的处理工具，及时恢复服务。

(1) 提供风险发现能力：参考上述“服务管理-观测服务”的能力要求，各系统接入并使用观测服务的日志和监控能力，及时发现风险。

(2) 工单跟踪风险处理流程：提供统一的风险工单处理流程，制定标准的规范化的风险工单，通过工单必要字段，建立基于工单响应和处理的 SLA 机制，全链路跟踪风险工单，并在风险工单处理完后，主动分析并输出风险报告。

(3) 实现自动化响应和修复：基于已有的 SRE 风险工单处理经验，梳理不同审计场景的处理方法，沉淀出标准化的处理工具/套餐；提供自动化处理规则配置能力，满足一些明确处理方法的匹配规则，利用自动化工具实现自动处理、修复问题，可以更及时和快速地解决风险问题。

(4) 持续改进风险分析能力：提供基于安全审计风险事件角度出发的分析数据视图和工具，帮助审计人员定期总结分析审计运营数据，主动发现风险隐患和可优化点，生成定制的审计报告，帮助业务方提供安全可靠的服务。

第四章 SRE 的进阶

整理第四章内容的时候已经是 2025 年了，SRE-elite 已经并入信通院云大所稳定性保障实验室，叫 SRE 工作组，成员超过了 40 人，每周二晚上我们延续着例会，分享新技术和各公司的 SRE 趣事，也偶尔在 Qcon 和 GOPS 等行业会上办专场。

回顾前边的几个章节，最长的是第三章，详细介绍了国内各企业 SRE 团队当前的职能，对于资深的 SRE 从业者还是有所帮助的，至少看到了与其他公司的差距；第二章计划更改为面向刚入行的初学者或者在校学生，主要讲述 SRE 的基础理论和基础概念。

第四章，面向带团队的 SRE 管理者，计划整理四个模块，探讨一下 SRE 团队短期如何生存，长期怎样发展；SRE 团队手中的重资产，也就是资源与算力的变现；以及 SRE 在新技术浪潮下如何跟上甚至驾驭数据与 AI 能力；最后是实现从止损到增收，从维护到运营。

1 SRE 团队的生存与持续发展

前几年，有些 DevOps 组织宣称的“谁开发，谁运维”“去运维，去测试”等理念影响了部分企业，裁减了运维团队，这其中有些做得好的，转向或者正在转向平台工程，希望以内部一体化研运平台的方案替代传统运维职能，持续解决质量效率成本及安全问题；还有些做得不理想的，将裁减的运维职能直接闭环给了研发团队，但没有补充统一的工程化能力，本质上是放大了各业务开发团队的成本，放弃了统一的质量效率管控；其他的，介于两者之间。

本白皮书第三章中所描述的 SRE 职能，有些来源于转型成 SRE 的运维团队，还有一些来源于把自己称作 SRE 的研运工具开发团队或者中台团队，这两个群体融合在一起，构成了与 google 相似的 SRE 组织。

这样的 SRE 组织看似职能宽泛，以保障 SLA 为首要职责，但能否经得住各种新生理念、AI 新技术，甚至是经济波动的冲击，会不会重蹈传统运维组织的覆辙。

1.1 SRE 团队的生存法则

SRE 组织的生存法则，取决于设立这个组织的初衷，目前行业中有两种思路，一种思路是将 SRE 作为质量控制的决策者，组织架构上独立于业务产研团队之外，类似于监察机构，地位上与业务团队平齐甚至略高，比如可以驳回业务团队的发布变更等决策；另一种思路是组建一支辅助业务成功的 SRE 团队，业务的产研团队作为营收主体，对业务的财务结果负责，同时拥有决策权，SRE 团队可以提供建议及解决方案，可以将 SRE 人员分散于业务研发团队内，也可以统一组建 SRE 部门；

两种思路的生存法则都跟预算来源和权利来源相关，前一种思路来自于 SRE 与业务产研共同的上级领导，后一种来自于业务团队及共同的上级领导。

1.1.1 没有 SRE 岗位行不行

国内各行业对工程师岗位的命名规则不统一，按最大化原则，本章节中提到的 SRE 泛指从事第三章 1-7 节范围内职能的，对应用的可靠性及连续性有影响的工程师群体。在不同企业内可能被称为基础运维、应用运维、运维架构师、运维开发、工具平台开发、运营开发、平台工程开发、后台开发等。

任何岗位的生存与持续发展，都需要有谦虚的心态和适度的危机感，如三体中描述的，“弱小和无知不是生存的障碍，傲慢才是”。SRE 团队首先应该正视自己的存在性价值，说的直白一些，就是没有这个岗位，公司还能否运转，答案是，可以。

事实上，只有重视应用可靠性，而营收尚可的组织，才会设置 SRE 岗位，并寄予厚望；蜜月期过后（大致一年左右），如果可靠性未有明显改善，同时又出现负面评价，例如成本过高、引发矛盾、拖慢业务等，SRE 岗位将被缩减；如果可靠性有明显改善，而 SRE 岗位没有扩展出例行服务，SRE 岗位也将被缩减。

在组建初期，SRE 团队的重要性完全取决于业务研发及运维组织共同的上级管理层态度；一至两年后，SRE 团队的重要性取决于业务可靠性提升的情况，以及 SRE 团队所占据的业务核心“研发运维运营”过程中，例行服务的范围和深度。

因此，SRE 虽然重要，但并不是不可或缺，SRE 团队要明白，自己的职能是可以被业务研发等岗位替代的，只是隐性成本会升高，业务质量会下降，管理层会权衡这些损失与上述 SRE 问题的利弊。而且管理层的要求也是会变化的，可能受公司经营情况、高层轮替等因素影响，难以预测。所以不论建立 SRE 团队的初衷是质量控制决策者，还是辅助业务

成功，SRE 团队都应该具备谦虚服务于业务营收组织的心态，成为通过技术杠杆放大业务收益的工程力量。

1.1.2 SRE 团队的预算从哪里来

SRE 团队能否生存，最关键的因素是有没有固定预算，固定预算才代表这个事情要长期做，根据组织架构的不同，SRE 的预算有三种来源。

1，SRE 的预算及职能由各业务团队闭环

各业务研发团队自己承担 SRE 职能，招聘 SRE 人员，或者复用业务研发自己做，这种方式的弊端是不同业务之间的 SRE 职能难以统一，质量标准也难以统一，总成本大概率会超出妥善管理的集中式 SRE 团队，而且工具及平台建设能力弱，不利于业务长期发展。规模较小或业务线单一的企业大多选择这种管理方式

2，SRE 集中管理，统一划拨预算

规模较大，业务线较多的企业往往选择将 SRE 整合为集中的一个或多个团队管理，通常命名为“SRE”“平台工程”“技术风险”“技术运营”等，统一质量标准，集中资源建设工具平台，以中立的身份服务不同业务线，跨业务线全局统筹技术专家、算力等资源，提高 SRE 专业度，减少重复建设。

为了兼得分散管理的优势，有些 SRE 团队又在内部细分了 SRE 平台团队与 SRE 服务团队，前者集中发展 SRE 核心能力，后者紧密绑定服务各业务线，甚至可以定制个性化 SLA、SLO。

这种统一的 SRE 团队，预算来自上一级的统一拨款。

3，SRE 集中管理，预算来自各业务的货币化支付费用

对于实施了货币化改革的 SRE 团队，其人员、机器、软件等预算均来自各业务团队的实际采买，公司内部采用甲乙方结算模式控制供需平衡，由财务及 HR 团队参与管理，这种方式解决了中台团队投入资金的合理性及价值问题。

1.1.3 SRE 上一级领导的视角

SRE 团队可能误判工作内容的重要性，站在 SRE 团队+1 或+2 管理者的视角，也就是站在决定 SRE 团队预算的管理层视角，产品线的业务方向、市场营收，甚至合规安全等问题更为重要，对 SRE 团队的预期主要集中在上线后的可靠性方面，也就是说，作为幕后团队的 SRE，其主要工作内容大概率是不被理解的，甚至少有了解的意愿，更多的是看结果，而不是消耗精力参与分析过程；基于这个现实情况，SRE 团队做好可靠性只是基础工作，

而且难以持续，难以发展，因此应该逐步将工作重心左移，以保障代码或研发过程可靠性的名义，参与到业务研发过程中（详见第三章第 2 节研发保障），在研发提效获得认同之后，可以逐步扩展线上监控体系，将以止损为目标的运维监控和以提效为目标的研发观测，逐步扩展为以拉升业务指标为目的的“业务监控”（详见第四章第 4 节 SRE 的四维监控），以确立 SRE 团队在代码生命周期中的固定价值和定位。

在此之前，SRE 团队的成本应该尽可能透明，并且有明确的参照标尺，例如与业务营收考核挂钩，可以通过成本分摊或费用结算等方式来实现，并且要给予承担业务营收 KPI 责任的团队一定的自主选择权。在猥琐发育的阶段，只有以算得清的成本自证价值，才能经历风浪，抵御组织架构动荡和管理层认知偏差的影响。

1.1.4 SRE 团队的生存法则总结

总结一句话，以高于同行或业务自建的 ROI，保障基础可靠性，不断降本增效创新，伴随及辅助业务发展

一，保障业务可靠性

对 SRE 来说，不论是采用分散管理还是集中管理，对业务线上可靠性的保障，都是基础工作，也是底线，这一项做不好，就会丧失存在价值。

1. 贴近业务优化 SRE 流程

可靠性方面的 SRE 流程，主要包括上线前的风险评估及入网控制（第三章第 3 节）、变更管理（第三章第 4 节）、上线后的监控观测（第四章第 4 节），故障自愈、故障应急、灾备演练（第三章第 6 节）等，SRE 应该伴随业务各版本不断升级优化可靠性流程。

1. 建设和升级自动化智能化平台

在多变的业务发展和技术环境中，构建一套既能满足当前需求又能支持未来扩展的 SRE 平台，对于任何规模的 SRE 组织来说都是必要的，SRE 作为以工程化能力立足的支持团队，应该面向代码全生命周期设计一体化的工程平台，在这一领域，入选 Gartner2023 和 2024 十大技术趋势的平台工程体系是不错的参考（第三章第 7 节）

1. 左移参与业务研发服务

线上可靠性损失的根因主要集中在三个阶段，上线前研发阶段引入的代码 bug，上线中的变更事故，上线后的故障。前边第一点“贴近业务优化 SRE 流程”主要针对的是后两者，根据不同企业 Dev 和 Ops 水平的差异，代码 bug 导致的可靠性损失大致在 20%～40%之间，

极端的有可能超过一半，因此如果 SRE 团队完全不参与业务研发阶段的服务（包括了代码分析和扫描，详见第三章第 2 节研发保障 2.1.1 代码可靠性），对可靠性的管理是不全面的。

SRE 参与研发服务还可以提升研发连续性，为业务版本提效，SRE 处理研发环境及测试环境问题的效率远高于业务研发团队，从 ROI 视角更加合理，还能拉近团队间关系，增加 SRE 服务粘性，便于创新服务的推广。

基于跟业务开发良好的合作，还可以推动跨职能协作，例如在设计和编码阶段提前识别性能瓶颈、单点故障等隐患，避免后期重构的高成本（详见第三章第 1 节可靠性架构设计）；共同约定监控埋点、依赖项检查等，确保代码上线前符合生产标准；共享指标、日志和依赖关系视图，避免故障应急过程中的信息盲区；与业务研发配合实现限流、降级等容错设计，实现有损服务；提前嵌入非功能性需求（如安全合规），减少上线后的合规风险等等

1. 定期反馈 SRE 质量指标

SRE 团队定期向业务反馈可靠性指标的核心价值在于建立透明化协作机制，通过数据驱动实现业务目标与系统稳定性的动态平衡。

首先要对齐业务目标，量化业务需求，共担风险，通过 SLO、错误预算等指标，将业务方对可用性、性能的模糊需求转化为可衡量的工程目标（如 99.9%可用性），避免资源过度、投入不足或过度损害用户体验。例如，电商大促前 SRE 与业务方共享错误预算使用情况，若剩余预算充足，业务可加速新功能上线；反之则需优先修复稳定性问题。

其次，驱动业务主动优化与资源配置，例如 性能瓶颈可视化，定期反馈延迟、错误率等黄金信号（如支付接口延迟>500ms 占比），帮助业务团队定位高优先级优化点；再比如平衡成本与效益的决策 SRE 通过容量指标反馈资源利用率，指导业务方在扩容需求与预算间权衡。

再次，定期反馈 SRE 质量指标可以增强跨职能协作信任，打破“黑盒”认知，SRE 团队通过周报展示故障 MTTR（平均恢复时间）下降趋势，减少业务方的质疑； 同步指标，推动研发自主优化代码健壮性

最后，存档的 SRE 质量指标可以用于合规 审计，例如金融行业需要向监管证明系统符合 SLA（如年故障时间<52 分钟），定期指标报告成为合规依据； 银行 SRE 与业务连续性委员会共享 RTO（恢复时间目标）达成率，联合演练提升应急响应能力。

二，可靠性保障 ROI 高于平均

SRE 团队保障业务可靠性的同时，还要少花钱，才能体现专业度，获得更多业务的支持。尤其是对比同类团队，或者对比业务研发团队自己保障同样的可靠性指标所预估的成本，这就要求 SRE 团队可以量化自己的投资回报。值得注意的是，业务团队自己预估的保障预算，大多过于自信，趋于乐观，低于实际费用。这对于身处同一家公司内，薪酬结构相似的 SRE 团队来说是不利的，因此 SRE 团队可以利用资源杠杆，通过自身技术手段，加大对业务资源优化的投入，以冲抵 SRE 团队自身成本，优化 ROI。

1. 单业务资源优化

SRE 的单业务资源优化，最直观的就是容量管理，包括资源预算、实时监控、预测分析、自动化调度等（详见第三章第 6 节 04.，也包括了对商务团队提供技术支持，在与各家资源供应商的谈判中通过议价及组合策略实现费用节约，SRE 需要建设可视化成本工具支持该场景。

1. 跨业务算力调度

跨业务的算力调度，指的是 SRE 在按各业务预算，从各供应商采购回异构资源之后，统一转换为算力，通过调度复用，使得各业务的碎片算力可以被用于业务及中台团队的临时性需求，或预算外需求，例如运行虚拟化、微服务、训练推理等（详见第四章第 2 节）。SRE 需要建设综合算力调度及算力交易系统支持该场景。

1. 工具工程化效率

SRE 是工程化团队，相对于业务研发团队，SRE 更聚焦于自动化及效率工具领域，因此在业务的全生命周期内，成熟的 SRE 团队能以远高于业务研发团队的效率建设及维护工具（详见第三章第 6 节 03.，并以此冲抵 SRE 团队的成本。

1. 故障损失规避预估

减少故障时长，就是间接减少业务损失，SRE 可以将此类间接价值转化为可量化的指标，例如根据历史故障的业务平均损失情况，推算 SLA 提升或错误预算的节约，与减少损失之间的数学关系，但站在某些类型的业务团队视角，用户在故障时段的收入损失，其中一部分是会在故障恢复之后补充支付的，这取决于业务对用户的粘性，例如电商；相反，有些业务故障对用户造成的间接损失，是无法预估的，例如金融交易，可能导致用户对业务线索赔，因此在估算 SRE 的挽回损失金额时，需要设定双方认同的系数。

除此之外，避免业务线品牌声誉损失、通过工具为业务提升版本效率等，都可以估算间接收益，前提是算法获得业务团队的认同。

三，降本增效创新

在以相对于业务研发及同类团队，以更低的成本保障业务可靠性之后，SRE 团队需要考虑的是后续的持续发展，也就是保持甚至扩大团队的价值，表现为 SRE 团队预算的增长。SRE 应该首先扩展业务团队有预算的服务项，因为这类服务项的业务价值相对真实。同时，还应该逐年降低已有服务的成本，以体现增效。

1. 扩展有预算的新服务项
2. 升级现有服务项
3. 降低稳定服务的单价
4. 增加客户数量

四，伴随业务发展

SRE 是辅助性的职能，其价值最终要叠加到业务上，SRE 不是花钱的运维团队，是通过技术杠杆放大业务收益的工程力量，因此 SRE 团队应具备经营意识，尝试通过 ROI 管理自身团队，并将救火止损定为短期目标，而将拉动业务指标，伴随业务增长作为长期使命。

1. SRE 货币化转型
2. 始终紧跟公司战略
3. 转型为业务增长伙伴
4. 持续提升内外部产能

1.2 SRE 团队的阵地与产能

应用可靠性的提升是有上限的，

在管理层的认知中，在可靠性被拉高之后，维持当前值的预算至少不应该再持续增长，甚至还可以降低，这就给 SRE 团队带来了困扰，需要时不时的解释和汇报；**而效率的提升是没有上限的，**

自第一行代码问世至今，软件带来的社会效率提升不知道已经是多少倍了，因此很多 SRE 团队会将对业务的可靠性保障作为主阵地，对业务的效率提升作为阵地的扩展，占据更多的例行预算。

1.2.1 SRE 团队的阵地

可靠性和效率，在代码的全生命周期中都有价值，SRE 团队可以根据自身情况，选择适合自己的阵地；从技术规划层面说，拥有平台工程的 SRE 团队，选择面更宽，扩展速度更快（详见第三章第 7 节 01. 。

下图概述了在代码的全生命周期中，SRE 团队可以尝试进入的有价值服务领域。

从服务模式的角度看，有三种可以选择

1，技术服务，指的是 SRE 按业务要求，提供专业咨询、技术方案和操作运营等工作，使用业务方或第三方提供的工具平台，简单来说就是卖人头，专业话术叫人力云。

2，技术产品，指 SRE 团队根据业务需求，提供定制或者已有通用的工具软件，及相关维护，简单来说就是卖产品和维保，专业话术叫 SaaS 服务或私有化软件定制。

3，技术托管，指业务团队将 CI-CD-CO 全领域工作托管给 SRE，只提出技术及运营指标。简单来说，业务开发人员写完代码，提交之后的环节都托管给了 SRE，能做到这个程度的 SRE 团队已经充分获得了业务信任，实现了平台工程。

1.2.2 SRE 团队的产能提升

SRE 团队有了阵地，获得了某一部分的例行工作项，怎么长期坐稳并且越做越好成了下一个问题。所谓越做越好，简单理解就是要么服务的质量越来越高，要么就是服务的价格越来越低，这就需要 SRE 团队不断提升自身产能。

1，SRE 团队的内部产能提升

所谓内部产能提升，就是根据预算投入的方向，逐个优化 ROI，SRE 团队的成本投入大致有三个主要方向，雇佣工程师、租用设备、购买软件。在近些年自动化和 AI 快速发展的背景下，有些工作变成无人值守了，有些工作可以由对业务熟悉度和技能都不高的人员完成，这就给灵活用工带来了可能性，例如将工作分配给集团员工、子公司员工、外包员工的组合，以优化 ROI；租用设备部分可以通过算力优化提升（详见第四章第 2 节）；软件部分可以采用低代码组装（详见第三章第 7 节 02. 及 AI 辅助建设（详见第四章第 3 节）。

2，SRE 团队的外部产能提升

所谓外部产能提升，就是根据 SRE 团队之外的部门分工情况，甚至是公司外的行业现状，通过资源置换、开源协同等方式，在没有预算的情况下，获得 SRE 组织之外产能的补充，例如在公司内，SRE 团队算法资源的数量和质量相对不足，而算力及维护能力充足，公司内算法资源集中的团队，算力及维护能力大概率紧缺，这就形成了资源互换的条件，双方可以进行资源互换；在公司外的同行之间，可以通过开源项目补充 SRE 内部产能，不论是使用开源组件的团队，还是自主开源的团队，只要保持长期贡献的能力和意愿，都可以获得外部代码以提升内部产能；在不同行业之间，也有像信通院云大所稳定性保障实验

室之类的组织，针对 SRE 建立了各领域关键技术的工作组，共享 SRE 成功经验，减少试错成本。

1.2.3 SRE 团队的工程经济学思维

工程经济学，是一种将技术决策与商业价值量化结合的方法论，用在 SRE，其核心就是：“将 SRE 团队有限的预算（人力、算力、软件）投向产生最大业务价值的服务”。

工程经济学的四个核心原则是，量化一切，平衡取舍，动态优化，长期视角。

SRE 团队需要自证“成本与价值”的合理性，可以通过定期的价值汇报，也可以通过货币化改革，一次性解决。

1.3 SRE 团队的货币化改革

可靠性不是所有企业各个时期的刚需，SRE 团队也不是可靠性的唯一路径。运维和 SRE 都可以被业务研发替代，只是这种替代丧失了集约效应，违背了木桶原理，使得可靠性与研运效率的上限降低、长期服务创新能力降低，而管理层有可能意识不到或者并不在意，因此 SRE 团队的生存与发展不能完全寄托在管理层的认知上，SRE 团队需要从商业模式上具备独立生存和发展的能力。

在企业中符合财务模型（预算-收入）的经营模式才是稳健的，SRE 团队需要将价值具象化到业务愿意付出的成本上，也就是即使业务团队自建，也不得不付出的成本上，成功的 SRE 团队通过集约效应与木桶原理，可以使得自身成本远低于业务自建的成本，同时在可靠性上保持较高的业务满意度，从而获得业务团队的价值认可，货币化的同时也意味着业务对 SRE 的服务拥有了自主选择权

SRE 团队货币化的目的 —— 获得无需自证的价值，长期发展的条件。

1.3.1 SRE 团队的货币化方案

SRE 团队货币化方案有四个参与方，SRE、业务、财务、HR

首先 SRE 团队需要梳理服务项并定价，之后与业务方确认，双方达成一致后 SRE 每月发出账单，业务确认账单，财务将账单计入业务当月成本，并承担价格监管及调整公示，HR 团队根据财务计算的 ROI 数据协助 SRE 团队调整成本。

1，SRE 团队动态成本确认

首先需要明确 SRE 团队的年度/月度成本，比如人力成本、资源消耗成本、软件采购成本，办公场地、差旅、外部合作（会议、认证、培训）、企业分摊等其他成本。

在虚拟结算阶段、试运行阶段，或者因部分团队配合度不够无法获得准确成本数据的情况下，可以与上级管理层商定一个总包费用，作为货币化起点。进入货币化正式运行期之后，再由财务团队按固定周期（月），给出 SRE 团队的精确成本数据，计算 ROI。

2，货币化服务项的设计

SRE 服务项的设计，以业务团队能接受为准，服务粒度可大可小，小到制品库单工具卖产品收费，大到按 OS 或核心数分阶梯设定单价，也就是研运全托管的服务模式

服务项应该是业务能感受到的服务，最好对照行业已有的服务收费模式，便于业务团队理解，例如 CI 工具的软件费用、变更、灾备、工具定制等，服务项最好不是 SRE 团队内部优化的工作，例如应急管理、故障自愈等，业务团队更愿意为结果付费。

初阶 SRE 团队可以以人力成本为基数，按月收费或分次结算，也就是卖人做技术服务，高阶 SRE 团队可以尝试以 SRE 服务的价值定价，这要求 SRE 团队在维持同等质量安全指标的前提下，效率及成本远优于同行。

3，货币化服务项的单价

核算 SRE 团队投入的每个服务项的单月总成本，例如人力、资源、外采等，确定每个服务项的计量单位，例如按核心数、OS 数、调用量、并发量、计件等；总成本除以用量，算出初始单价，并根据考核方案设计冗余；也可以对标同类型商业化方案设定折扣。

4，与业务方确定服务项价格

用业务方的历史用量数据，推算过往几个月的 SRE 服务费用，与不接受某服务项账单价格的业务商谈，确定折扣或是否继续使用，再根据协商情况调整单价及各业务折扣，计算差额成本，联合 HR 团队制定降本增效计划，或与财务团队制定利润返还策略。

5，不适合货币化的服务

公司强制业务团队使用，不允许自由选择的服务，不适合货币化，例如监管审计类工作。管理层出于公司战略考虑，要求 SRE 提供难以收回成本的服务，例如对投后公司、战略伙伴的支持，应该给予单独的预算补贴。前瞻性的技术研究类投入，短期难以在生产中大规模体现价值，需要管理层拨款投资。这些不适合货币化的项目，需要定期审核，控制开销。

1.3.2 SRE 团队货币化的管理

货币化改革有可能是 SRE 团队主动发起的，也有可能是被动的。有些是管理层看不清中台预算的合理性，希望借助业务部门的评判，筛选出 SRE 团队的哪些服务是必要且成本合理的，哪些是必要但成本偏高的，哪些是 SRE 在自嗨，其实没什么用的。相反，还有些是公司对 SRE 团队的管理自由度较大，SRE 团队希望有更大的发展空间，甚至未来独立为科技公司，因此需要先行货币化来评估过渡。因此，SRE 团队货币化的管理严苛程度是不一样的，根据经营属性深度的差异，可以大致分为三类。

1，成为一支算得清成本的 SRE 团队

以算得清成本为目的，还是相对容易的，这要求 SRE 团队每月核算出准确的团队总 ROI，管理层根据情况设定 KPI，例如要求 SRE 团队总体 ROI 达到 70%即可。对总体 ROI 做要求，也意味着不同服务项之间可以相互补贴，以这种方式实施货币化改革，可以减少管理层对 SRE 团队的精力投入，明确 SRE 团队自身的价值成本，以及需要补贴给 SRE 团队的金额。

2，培养一支有经营意识的 SRE 团队

经营导向的货币化改革，需要对货币化项做精细化管理，即 SRE 团队的每个服务项 ROI 均要达到不同的固定比例，总 ROI 再设定一个比例。这种方式下，管理层可以看到 SRE 在各个方向的投入，以及业务实际认同度，给出资源调配建议，并对前瞻性方向给予宽松的 ROI 进行补贴。

以上两种方式，都是将 SRE 团队作为中台支持团队进行管理的，因此本着服务业务，不给业务增加额外负担的原则，服务项 ROI 和总 ROI 的 KPI 都不应该超过 100%，实际 ROI 如果超过，应该以调降单价的方式减少业务费用。

3，打磨一支具备独立生存能力的 SRE 团队

独立生存，意味着需要 SRE 团队经得起业务团队的波动和同行团队的竞争，在没有外部补贴的情况下持续发展。SRE 团队需要具备造血能力，也就是可以在扣除试错成本之后，不断成功地创新服务项，发掘出业务团队愿意持续付费的卖点。这需要 SRE 团队内部管理模式既要维持稳定的质量控制，又能保持较高的效率以补偿试错成本，并且可以通过技术迭代控制成本低于业务自建及同行，最后还得建设基本的安全防护能力以防止团队品牌因漏洞、合规等突发隐患猝死。

年初服务项 ROI 的目标应设置于 100%以上，年底达标后，以调降单价的方式修正回 100%，并向业务团队发布降价公示，次年再设定新的 ROI 目标，以此模拟逐年的团队盈利。

1.3.3 SRE 团队的货币化工具

货币化自身是有管理成本的，SRE 团队应以工程化能力降低长期的管理成本。需要建设四种工具能力，HR 视角的单人成本和单服务项成本核算能力；SRE 视角的单业务资源费用和单服务项资源费用核算能力；业务团队视角的单业务账单系统以及明细（每个服务项的账单）；财务视角的单服务项 ROI 和 SRE 总 ROI 考核。这四个视角的工具能力，可以分开建设，相互调用，也可以做到一起，取决于企业内原有系统的建设情况，以及对货币化改革的支持程度。

- **人力成本系统：**

主要面向 HR 团队的工作界面。HR 团队有单人成本数据，但这类数据管控非常严格，一般不会做系统级的对接。SRE 可以建设针对货币化服务项的人员填报系统，将人员与货币化服务项挂钩，再由财务人员按月给出 SRE 每个服务项的准确人力成本数据，甚至 excel 交付都可以，应包括集团员工、子公司员工、外包员工等各类用工模式的总成本。

- **资源成本系统：**

SRE 给业务提供服务，必然会搭建各类工具，占用一定的算力资源，这部分费用也包含在 SRE 团队的货币化费用中，需要以某种形式分摊到各业务账单里，例如按不同业务的操作量，或 OS 数、核数比例等。需要注意的是，这部分费用与业务模块占用的算力费用是不同的，这部分算力并不在业务的 CMDB 归属中。SRE 货币化算力成本的管理，可以建设在账单系统中，也可以统一归属到管理所有资源及算力的 FinOps 系统中。

- **SRE 账单系统：**

业务团队需要一套查看账单，确认账单的系统，要能清晰地看到每个月的，每个货币化服务项的 SRE 人力和算力费用、账单明细、历史趋势等，可以点击确认，也可以拒付。

- **SRE 货币化考核系统：**

根据业务确认账单的情况，财务人员可以核算出 SRE 每个货币化项的 ROI 和总的 ROI，用于对照目标对 SRE 团队进行考核，为 HR 和管理层提供数据。

1.4 SRE 货币化后的持续发展

货币化改革前 SRE 是公共资源，业务团队可以免费使用 SRE 团队的人力和部分算力，因此除了故障之外，对 SRE 还是比较宽容的，也不关心 SRE 搞了哪些靠不靠谱，用不用得上的创新和副业，必要的时候甚至还会配合点赞；而货币化改革之后，SRE 团队的成本

需要通过各种货币化项结算给业务，且业务团队还有自己的ROI压力，这时候SRE会感觉到业务团队开始精打细算了。因此相对应的，SRE团队的思路也应该跟着调整，转变为面向业务真实需求和满意度的发展思路，追求业务方对SRE团队的服务质量和服务价格双满意。

1.4.1 优化服务项的“商业”模式

作为企业内部的SRE团队，优势是理解业务及客户关系，劣势在于业务方和其他中台团队也同样了解SRE的能力，人员沟通和转岗的便利，使得技术优势的有效期更短。

在这种情况下，要想达到让业务对服务质量和价格双满意是不容易的，需要借助SRE团队既能深入业务，又有工程化能力的优势，设计对业务团队粘性更大的“内部商业模式”。

1，服务项的设计要避免技术服务，至少是技术产品，推荐技术托管

在服务项的设计上要避免单纯的技术服务卖人头，这很容易与其他竞争对手陷入卷人均成本的恶性竞争，最终伤害到SRE团队的工程化能力，没有SRE平台的粘性加持，竞争对手或者业务研发团队也更加容易替代。

以技术产品模式设计服务项会稍好一些，可以隐含一部分价值定价，团队需要持续提升技术产品的迭代和运营能力，但服务质量会受限于业务用户能力和第三方技术服务团队；最理想的服务项模式是技术托管，即技术平台和操作使用均由SRE团队提供，达到最佳质量效果。在有驻场SRE服务的前提下，能够更好地感知到业务团队的变化，获取一手真实业务需求，帮助业务解决一些突发的，没有预算支持的场景，这种付出可以一定程度上冲抵业务方对SRE货币化服务费合理性的质疑。

2，低成本延展服务范围以冲抵货币化费用

在货币化之前，以中台身份支持业务的SRE团队原则上应该是对业务有求必应的，但实际上SRE并没有动力扩大支持范围；在货币化之后，虽然有了明确的服务项和服务范围，但由于业务有权利质疑单价及费用的合理性，反而使得SRE团队愿意以某些额外支持，来换取维持自身货币化项的价格体系。关键在于为了减轻负担，SER会优先扩展那些事半功倍的服务，也就是在技术栈和服务资源上与原有货币化项可以重叠复用的服务。

业务生产环境的自动化方案都是SRE负责的，因此从技术能力和服务人员的角度看，很容易扩展到其他环境，例如预发布环境、各类测试环境、研发环境等，甚至可以为业务研发团队提供全套的研发环境托管服务，降低业务开发除了写代码之外的维护负担。

成本优化是SRE的职能之一，货币化之前SRE的主责是可靠性，在保障可靠性的前提下，会根据业务要求，或出于责任心在力所能及的范围内优化成本，一般是采取与业务约定资源利用率，定期优化；货币化之后，SRE会意识到资源杠杆的作用，在业务挑战货币化服务费的情况下，SRE会与业务商谈，用计划外的成本优化资金冲抵货币化费用，对于业务团队来说，这种计划外的现金收益必然是多多益善，因此双方很容易达成一致。因此SRE主观上会非常愿意通过自身经验，或调动SRE团队内的工程化技术能力进行大规模的资源成本优化，甚至不惜在错误预算范围内平衡成本优化的资金收益与技术风险。此类尝试也没有超出SRE的技术范围和服务人力，除了风险外，没有给SRE增加负担。

3，以平台工程视角加大SRE团队的技术投入

SRE团队可以通过增强粘性的服务模式和主观上积极努力的服务扩展提升业务团队的满意度，但这些都是有瓶颈的，而且门槛较低。作为工程化团队，持续投资技术平台才是SRE不断提升竞争力的关键策略，投资技术平台对SRE团队来说不仅是工具升级，更是能力转型，使SRE团队在标准化、自动化、数据化、智能化的每一轮技术迭代中占据效率高点。

从当前来看，基于PaaS理念的平台工程，是最适合货币化后的SRE团队持续经营的技术底座（详见第三章第7节），其iPaaS层的原子化切割，保证了每一轮技术迭代的敏捷性和低维护成本，aPaaS层的快组装能力，适合新服务项的低成本试错，PaaS的分层结构与SRE团队内部的平台及服务两大职能划分相似，这也为团队内部的精细化管理探索铺平了道路。

1.4.2 SRE内部的精细化管理

货币化后的SRE团队管理者，需要不断主动提升内部管理水平，尤其是以经营意识和独立生存为目的的两类货币化团队，都有持续降本增效和创新增收的动力。在预算有限的情况下，除了服务模式和商业规划的技巧，还可以通过SRE团队内部的精细化管理，持续提高产能，让每一分技术投资都精准驱动货币化收入和业务的增长。

1，了解基层SRE组织的状态和产能

SRE团队的服务范围宽，覆盖了代码全生命周期，涉及到的技术栈多，在不同阶段各技术栈需要掌握的深度也有差异。SRE管理者需要设计一套数据指标，绘制从SRE工程师到各级SRE团队的各类技术度量指标，用来有针对性地培训提升和调整重组各级SRE组织的状态和产能，以适应不断发展的业务需求。

以腾讯 IEG 对 SRE 工程师的技术度量指标为例，分为海量自动化、运维开发、DevOps、AIOps 四个大类，各类能力由所对应的工具平台的操作记录数据经计算得出数字指标，计算方法由各团队 SRE 代表共同商定，每半年商定更新一次，该指标仅用于 SRE 工程师的职级晋升。SRE 管理层通过技术度量指标，可以了解 SRE 个人及团队的技能状态和产能，也可以通过该指标直接调整 SRE 团队的技术重心，每次调整在三至六个月内就会在基层看到明显的转型效果，以适应业务团队、SRE 行业、IT 技术的变化。

2，算得清各服务团队和服务项的精确 ROI

SRE 管理层有了员工和各级团队的状态与产能之后，就可以将其转换为具体的业务产出，业务团队根据 SRE 的产出，按月支付货币化费用，作为 SRE 团队的收入，并结合 SRE 团队的成本，计算团队和具体服务项的 ROI。SRE 管理层可以根据 ROI 来调整各团队及服务项的成本投入，决策预研新的服务能力，通过技术度量指标调整 SRE 团队的技术方向，形成新的闭环。

这个闭环中除了技术度量指标外，还有一个精细化管理重点，就是 SRE 的业务产出度量，不同于业务线的全职研发工程师，他们往往是通过 DevOps 研效指标来度量产出，SRE 的服务项覆盖代码全生命周期，且技术栈宽泛。可以通过 AI 化的 ITSM 体系做 SRE 业务产出的半自动化，甚至全自动化度量。

以腾讯 IEG 的 SRE 团队为例，首先将 SRE 的时间消耗分类，例如操作类、应急类、质量优化、成本优化、自主建设、会议、学习等，再通过自动化平台、应急灾备平台、SLA 平台、Finops 平台、运维 PaaS 平台等自动计算出 SRE 工程师在各服务项消耗的工时，暂时不能准确自动统计的，通过 chatops 辅助 SRE 工程师创建工单并自动审计。以各 SRE 工时的消耗，作为业务产出，作为“单核服务费”货币化项的明细依据；以每个货币化周期单场景 SRE 工时的下降，作为 SRE 提效的依据，作为“单核服务费”货币化项降低单价的依据；以业务各个操作场景的工时为源数据，绘制各业务的画像，作为后续各类型新业务接入 SRE 服务的定价依据，通过同类业务的 SRE 方案及经验分享，对新客户体现 SRE 团队的专业度。

3，感知到客户们的真实态度

在每个月业务团队手动确认账单的系统页面，设计方便易用的满意度调研功能，例如对本月货币化价格“满意”或 “不满意”两个大尺寸按钮，作为用户表达情绪的渠道，和 SRE 管理层的告警通道，以及时联系业务用户了解基层 SRE 团队是否出现失职或误判，以适当

的措施挽回，及时避免负面情绪在客户群体中积累扩散。保障 SRE 团队的服务效果，保障 SRE 货币化收入的稳健增长。

1.4.3 持续扩展货币化收入

基于现有的预算，或者当前货币化收入，结合精细化的 SRE 团队管理，就可以不断扩展服务范围，提升货币化收入了。可以尝试以下四个方向。

1，将 SRE 的服务扩展到代码全生命周期

在本节 1.2.1“SRE 团队的阵地”中，概述了 SRE 在业务代码全生命周期内，可以尝试进入的有价值服务领域，获得扩展的收入。其中各部分更详细的介绍都在本白皮书第三章。

2，尝试扩展公司外客户

在完成内部扩展，或内部扩展受阻的情况下，SRE 可以尝试将成熟服务延伸到公司外，以 toB 服务的形式扩展收入，例如同集团下的其他子公司，或有投资关系的企业。不建议 SRE 跨行业拓展服务。

3，公司内效率软件定制

工程化能力较强的 SRE 团队可以扩展业务团队的效率工具定制服务，SRE 在工具领域的能力积累相对于业务研发团队更强，因此在低于业务研发团队自评估成本的前提下，SRE 团队依然有很高的利润，而且在项目样本越来越大的情况下，更容易沉淀出通用工具，进一步提高货币化利润。对 SRE 来说，可靠性的提升是有上限的，而效率的提升没有，也就意味着内部 SRE 团队定制化服务的扩展空间是无限的。

在商机洞察方面，由于 SRE 团队可以提供驻场服务，与业务研发运营人员协同工作，因此非常容易总结出业务团队所需要的提效需求，如果以快速试错的方式先开工再谈费用，对业务团队来说感受更好。在商业模式上，既可以采取项目制收费，也可以采取内部订阅的 SaaS 模式收费，取决于业务客户初期可以腾挪的预算。

4，综合算力调度

SRE 团队管控的资源成本，是 SRE 自身人力成本的 5 至 10 倍，SRE 投入技术资源在成本优化方面，可以形成杠杆效应，收回成倍的货币化收益。下一节，我们详细探讨 SRE 团队的综合算力调度。

2 SRE 团队的综合算力调度

综合算力调度，指的是将公司购买的私有设备，和租用的物理及虚拟设备，用统一的技术方案进行资源管控，并转化为可以随时使用的弹性算力池，根据各业务系统的需要进行自动化调度，承载业务稳定运行，平衡质量与成本，并精确计算各业务成本费用的能力。

算力调度所覆盖的职能，在 SRE 团队还处于发展中的企业内，可能分散于不同团队之间，按距业务由远至近的顺序，可能包括自建 IaaS 或外采云资源的“基础架构类”团队，负责业务发布变更、故障处理的“业务运维类”团队，甚至是各个业务内的研发自己把控等等。从公司经营的视角，业务团队是创造公司价值的核心组织，从支持所有业务，辅助各个业务成功的视角，应该有一个通用团队依托自动化调度能力，统筹不同业务线业务代码的托管运行、发布变更、故障处理、成本控制等职能，本模块为有意承接该职能的 SRE 团队主管提供参考。

从职能角度看，SRE 团队需要辅助业务管理资源，并通过技术手段用最少的资源保障业务稳定运行。具备平台工程实践能力的 SRE 团队，可以将资源转化为算力，并直接管理和调度算力。业务通过 SRE 平台消费算力，SRE 平台通过对基础能力的抽象保障业务消费的稳定性、可靠性，以及可复制性。业务方也能够通过 SRE 平台的算力调度能力，构建专属的领域级算力消费方案。

SRE 团队的算力调度能力不仅可以为业务保障可靠性、节约运营成本，还可以增大 SRE 团队的可调度算力池，为 SRE 团队自身的工程化及 AI 建设提供资源支撑；对于已经完成货币化改革的 SRE 团队，掌握算力调度能力可以灵活的将各个业务的空余算力复用，实现资源层面的二次收入，并保障 SRE 团队自身技术发展所需的算力消耗。

综合算力调度的出现是各行业 IT 技术发展到一定阶段的必然结果，其设计思路与落地效果是对中华人民共和国工业和信息化部于 2025 年 5 月 21 日正式印发的《算力互联互通行动计划》的直接响应。该计划旨在推动全国算力资源的高效整合与协同利用，提升算力基础设施的互联互通水平，基于国家数字化发展战略，结合当前业务需求，为算力调度和资源优化提供了明确指导。

计划的核心架构围绕多个关键方面展开，以确保算力体系的全面优化。首先，筑牢算力互联基础，重点加强基础资源管控、公共基础设施以及流程和自动化基础设施的建设，确保算力服务的稳定性和可靠性。其次，优化算力设施互联，通过完善算力调度基础设施和算力市场与供给管理，提高不同算力设施之间的连接效率。第三，促进算力资源互用，

推动算力资源的共享与调度，避免重复建设，提升资源利用率。第四，创新算力业务互通，包括提升通用算力消费能力和扩展专有领域算力消费，鼓励算力与各行业应用场景深度融合，拓展新型业务模式。第五，打造算力互联应用场景，针对特定领域如智能制造和智慧城市，开发示范性应用。此外，计划还强调算力能效与安全控制，以及夯实算力网络和数据安全保障，建立健全安全防护体系，保障算力服务的连续性和数据隐私。

在实施层面，综合算力调度聚焦于计划核心架构的四个方向落地：一是牢固算力基础，通过加强基础资源管控和基础设施优化，确保算力网络的稳健运行；二是促进算力互用，推动资源的高效共享；三是创新业务互通，拓展算力应用边界；四是夯实算网安全保障，强化数据和网络防护。通过这些举措，计划将有力支撑国家数字经济发展，为算力产业的高质量发展提供制度保障。

2.1 基础资源管控

基础资源管控是综合算力调度的基础。基础资源管控的核心主要体现在管理和控制上。管理就是资源的信息与流程管理，包括资源的生命周期管理、属性管理、关系管理，还有一个 SRE 团队必须关注的成本管理。除了信息和流程管理外，资源的实例需要能够批量操控，例如批量执行命令，上传下载文件，采集数据等，以支撑上层各种场景的需求。例如，业务新版本发布时，就需要确认部署到哪些服务器上，底层系统就需要拿到这些服务器的 IP 地址，并在这些服务器上部署一些基础组件来支撑业务程序运行；在发布动作执行时，也需要将对应的版本文件下发到服务器上；在业务运营期，也需要持续从这些服务器上采集监控数据和日志，以完成对业务状态的持续观测。虽然在顶层算力调度场景，业务不一定真的需要直接面对服务器，但是在平台工程实践里面，最底层平台需要处理这些工作。对资源实例的操作控制依赖三方面的建设：管控通道、通道节点管理能力，以及基于通道的作业管理能力。

以下将从资源生命周期管理，属性及关系管理，成本管理，资源管控通道，资源节点管理，作业管理六个方面来阐述。

2.1.1 生命周期管理

生命周期管理涉及很多不同类型的资源，例如服务器主机、VPC、安全组、子网、硬盘等 IaaS 资源，根据业务需求的不同，还能扩展出对负载均衡器、云数据库、云中间件等 PaaS 资源的支持。

在整个算力调度体系中，资源生命周期管理解决了资源的来和去的问题，来是资源从哪里来，去是资源用完后应该还到哪里去。这里必然涉及到很多流程，比如从公有云上购买资源，或者从公司采购团队申领资源，都要按照相应的流程来处理。从不同的公有云和专有云购买申领资源的流程细节上都有差异，生命周期管理的作用就是对用户屏蔽这个差异，让这个流程处于统一的、受控的状态之下。

从 SRE 的视角来看，资源的生命周期流转主要围绕资源供应商、SRE 资源管理服务方和业务方三者展开。资源供应层负责提供两类基础云计算资源，包括 IaaS 资源（如物理机、虚拟机、裸金属等算力资源，以及安全组、VPC 网络、弹性 IP 等虚拟组件）和 PaaS 资源（如负载均衡、云数据库等平台级服务），其核心职能是响应资源采买请求，构建基础资源供给能力。SRE 资源管理服务层则承担资源全生命周期的运营中枢职责，具备四大核心能力：资源池化治理通过建立动态资源池与额度池管理机制，依据业务需求水位线实时调节供给规模，并以分时复用模式提升分配效率；双维观测体系涵盖以业务为单位的效率观测（监控 CPU/GPU/MEM 等资源利用率）和成本观测（实时统计资源消耗并预测下一周期成本）；资源流转控制负责执行资源分配与下架流程，包括触发业务残留检查、资源状态重置、回收入池等环节，并对闲置超限资源实施安全审查与物理释放；运营调度枢纽则对接供应商采买接口与业务方操作请求，形成资源出入库的管理闭环。业务应用层作为资源的付费方，拥有资源决策权，可自主发起资源购买或销毁需求，按需使用已分配资源，并决定资源下架时机。

其中资源供应商作为资源生命周期的起点，对资源供应商的管理是非常重要的环节。特别是在混合云场景下，SRE 往往需要管理多个公有云，或公有云加私有云的混合资源。如何高效且低成本对接各种云，并管理云上资源是 SRE 平台工程实践需要重点考虑的问题。

在海外，Terraform 为解决这个问题提供了不错的思路。Terraform 通过 Provider 机制对接了全球众多云供应商，并在用户侧定义了一套通用的声明式语法来描述跨云资源，从而避免了为不同的云来开发脚本。另外 Terraform 还定义了标准化的执行流程，确保从资源计划到执行过程的一致性，并且不受底层云 API 差异的影响。Terraform 在海外拥有很大的影响力，以至于海外公有云都会主动开发适配和及时更新自己的 Provider，以确保用户能够使用最新的云功能。

在国内，也有不少企业希望继承 Terraform 的这种影响力，并围绕 Terraform 的技术生态开发了一些多云管理产品。不过综合来看，由于国内外开源生态环境的差异，Terraform 的直接使用可能面临如下问题：

一是生态成熟度相对滞后。国内云厂商 Provider 的更新维护节奏存在一定延迟。部分私有云的对接需要使用者自己完成适配，存在一定的改造成本。

二是企业版完整功能受限，开源版本合规性存疑。HashiCorp 企业版软件在国内使用、部署和安装有一定限制，Terraform 作为 HashiCorp 的核心产品，也在此限制范围。2023 年 8 月，HashiCorp 将 Terraform Core（核心引擎）的许可证从 MPL 2.0 切换为 BUSL 1.1（Business Source License 1.1），有 4 年“商业限制期”（之后自动转为 MPL 2.0），对其商业竞争对手的使用有一定影响，开源版本的使用未来是否有合规性问题存在不确定性。

三是国内运维支撑能力不足，对于商业化大规模应用的可靠性难以保障。例如 Terraform 的中文文档覆盖不全，复杂问题的排查依赖英文社区，虽然能够借助 AI 翻译等工具，但仍然可能增加排障难度和成本。另外国内使用的开源版本，无法得到官方企业级 SLA 支持，对于一些潜在的问题和缺陷可能无法得到及时修复。

虽然如此，Terraform 仍然给 SRE 平台工程实践中混合云资源管理提供了非常有价值的设计思路。

在完成多源异构资源供应商的标准化接入与管理后，SRE 资源管理服务方可通过统一的资源抽象层，实现对计算、存储、网络等全域资源的深度池化整合。基于虚拟化与资源调度引擎，资源池将形成弹性可扩展的逻辑单元，支持按业务需求动态分配资源配额，并通过自动化策略引擎实现资源的智能调度与负载均衡。此过程覆盖资源从供给、监控、扩缩容到回收的完整生命周期，最终达成资源利用率提升、SRE 响应效率提升的集约化运营目标。

其中在资源关键流程控制机制上，资源的正向流转始于供应商的资源供给，随后由管理方进行池化纳管，业务方可根据实际需求提出申请，管理方则依据资源情况进行动态分配。在此过程中，技术上的关键在于资源池水位动态调节机制需设置弹性阈值作为缓冲带，以保障资源分配的灵活与稳定。

资源回收治理流程由业务方发起下架请求，随后 SRE 资源管理服务方按步骤执行：首先进行安全审查，核验业务数据是否已彻底清除；接着对资源状态进行重置，解除与业务的绑定关系；然后将资源回收入池，转入待分配资源库；对于超出需求的资源，最终执行物理释放审查协议，完成销毁。

此外，需要构建“效率-成本”全链路监测模型。效率监控的颗粒度细化至每个业务单元，例如 CPU 利用率、GPU 利用率、MEM 利用率等等；成本观测不仅覆盖实时计算业务当前的资源成本及明细情况，还能实现基于时间周期的预测，形成从时间到空间的全面覆盖。

在混合云管理场景中，由于不同云平台的账号管理体系各异，生命周期管理需有效维护各类账号，并尽可能减少对账号的直接访问，从而降低因人员流动或业务交接导致的权限泄露风险。为实现这一目标，账号管理应遵循体系化的管理标准，将流程标准化作为基础，通过统一流程系统固化账号管理的操作规范，覆盖从申请、变更到销毁的全过程，确保所有操作可追溯。同时，在高频场景如云服务器采购中，强制推行标准化账号使用模式，避免权限操作的随意性。服务能力上应实现立体化，通过场景化服务集成对不同职能团队提供差异化接口，并将账号管理融入资源申请、分配和监控全流程，形成透明可查的服务链路。此外，需强化安全审计机制，记录关键操作行为以满足合规要求。在账号架构方面，应构建科学的多级账号体系，结合主账号、组织账号与子账号模型，明确各层级在资源分配、成本分摊及安全管控等方面的职责边界。权限管理则需实现精细化，支持实例级权限控制，贯彻最小权限原则，并通过统一的权限模型降低多云环境的管理复杂度，例如标准化权限申请流程、统一操作语义和简化策略配置界面。最后，建立灵活的授权机制，借助可组合的权限模板提升管理效率，使用户能够通过简便的勾选操作完成权限申请。

在通用流程之外，资源生命周期管理的核心能力在于提供全面的资产治理支持。具体而言，该能力通过与 ITSM 工单系统、审计系统、权限管理系统及登录控制系统等平台功能的深度集成，实现资源从需求提出、供给执行到合规检查和审计追踪的全流程闭环管理。同时，系统通过与 CMDB 的对接，整合资产元数据治理能力，统一数据来源、操作方式和权限控制策略。在多云环境下，该方案支持对接全球主流公有云，实现对不同云平台上同类资源的统一治理，并能基于统一模型轻量扩展至其他公有云或私有云。此外，在业务上线部署前，资源评估功能可通过大数据分析，主动推荐合适的部署区域和机型，辅助业务在满足容量需求的同时控制运营成本，并提供用户覆盖率、网络延迟和综合成本等预测数据。系统还能根据用户策略、业务属性及应用敏感程度，智能推荐资源类型和计费方式，例如帮助选择固定资源或弹性资源，包年包月或按量计费等，从而提升资源供给的智能化水平。

2.1.2 属性及关系管理

解决了来去问题，资源就可以被分配使用了。SRE 必须提供一种数据一致性能力，确保所有人看到同种资源实例都有相同的属性和关系描述，这就是 CMDB。

在企业中，通常使用 CMDB 来管理资源属性与关系，它是 IT 管理领域的核心组件，用于实现对各类资源的标准化管理和可视化控制。CMDB 通过一系列核心模式、基础能力及集成机制，支撑起现代企业的自动化运营体系。

CMDB 的核心模式包括资源抽象建模和动态模型实例化。资源抽象建模通过配置项（CI）来定义标准化的元数据模型，每个配置项包含唯一标识符、属性集以及关系图谱，从而精确描述资源的特性和关联。在此基础上，CMDB 构建分层模型体系，从硬件层、虚拟化层到应用服务层，实现全栈资源的映射与集成。同时，采用基模型与模型继承机制，允许通过模型演化派生出适应不同场景的子模型，提升系统的灵活性和扩展性。动态模型实例化则强调实时性与自动化，通过自动化发现技术（如网络扫描或 API 集成）实时采集资源配置数据，确保信息的准确性和时效性。此外，建立版本控制机制，用于追踪配置项的变更历史，支持审计和回滚操作，保障管理过程的可靠性与一致性。

CMDB 提供多项基础能力，涵盖资源管理的全生命周期、关系依赖及多维度管控。全生命周期管理能力负责覆盖资源从采购部署、运行监控到退役回收的完整流程，实现资源状态的可视化，包括使用率、健康度和成本分布等关键指标，辅助管理者进行优化决策。关系依赖管理能力则通过构建服务拓扑图谱，揭示资源间的复杂关系，如物理连接、数据流或服务调用，并支持变更事件管理及影响分析，预测变更可能带来的范围影响，降低运营风险。多维度管控支撑能力进一步细化为资产管理、合规管理及成本优化等方面；资产管理跟踪软硬件资产库存及许可证状态，合规管理记录审计轨迹以满足监管要求，成本优化则识别低效资源，指导回收或迁移决策，提升资源利用效率。

CMDB 作为自动化运营的基座，需要具备强大的集成能力。接口集成能力提供标准化 API，用于对接 ITSM、监控系统及自动化运维工具，并构建事件驱动架构，实现配置变更的实时同步，确保各系统间的协同工作。资源元数据治理体系则需要建立数据质量管控机制，包括自动校验、冲突消解和版本控制，保障数据的准确性与一致性；同时应实施细粒度权限控制，基于角色的访问策略管理数据访问权限，确保安全性与合规性。这些能力共同使 CMDB 成为企业 IT 治理的基石，支持高效、智能的资源管理实践。

在 CMDB 中，资源属性与关系管理是核心功能之一，其基础是模型树及其关系演变机制。模型树是一种层次化结构，以基模型（BaseModel）作为所有模型的祖先模型，位于树状拓扑的顶层。基模型定义了最基础的原始属性信息，例如标识符、名称或创建时间等通用元数据，这些属性为后续子模型提供了标准化模板。子模型通过继承机制自动获取父

模型的属性和关系，同时可以根据业务需求添加或修改自身的属性与关联，从而实现资源管理的灵活性和一致性。

模型树的具体结构通常包括基模型下的多个分支，例如设备模型和应用模型。设备模型可以进一步细分为主机模型、交换机模型等，代表物理或逻辑基础设施组件；应用模型则涵盖云原生应用模型、游戏服务端模型、数据库应用模型等，表示软件服务层面。这种分层设计允许 CMDB 高效地管理异构 IT 环境，例如在混合云场景中，基模型确保所有资源具备统一标识，而子模型通过添加特定属性（如设备的 IP 地址或应用的版本号）来适应不同资源类型。属性继承机制是模型树的关键特性，子模型默认继承父模型的所有属性，无需重复定义，这减少了数据冗余并提升了维护效率。同时，子模型可以扩展新属性，例如主机模型添加 CPU 核心数，或修改继承属性的默认值，但需注意避免属性冲突，通常建议通过命名空间或版本控制来管理变更。关系继承机制同样重要，子模型默认继承父模型的关联关系，如“部署于”或“依赖”关系，并可新增或调整关系约束。例如，应用模型可能从基模型继承“归属于”关系，并添加“调用”关系以描述微服务间的依赖，这有助于在复杂系统中可视化资源交互。

实践方案方面，设计 CMDB 模型树需遵循系统化流程。首先，进行需求分析，识别业务实体和通用属性，确定基模型的最小属性集。其次，构建树状结构，以基模型为根，逐步衍生设备模型和应用模型等分支，确保层次清晰。在继承策略上，应强制核心属性（如 ID）的继承，同时允许子模型灵活扩展；关系继承需考虑业务逻辑，避免循环依赖。实施时，可结合 CMDB 平台工具自动化部署，并定期审计继承链的完整性。例如，在云资源管理中，基模型定义区域和租户属性，设备模型继承后添加云厂商特定细节，应用模型则扩展部署关系，从而统一管理多云环境。维护阶段需监控模型演化，使用血缘分析工具检测变更影响，并预留扩展点以支持未来新增模型类型，如 IoT 设备。总之，模型树与继承机制通过标准化和可扩展性，为 CMDB 提供了强大的资源管理能力，是 IT 治理现代化的基石。

构建子模型及模型实例的过程是 CMDB 实现资源管理的基础，这一过程从资源类型的划分开始，逐步细化到具体实例的创建。首先，需要对性质差异较大的资源类型进行类别划分，例如将主机和业务区分为不同的类型，这有助于在管理时隔离不同资源的特性和行为。资源类型的划分为后续模型的构建提供了顶层框架，确保各类资源能够独立且一致地被处理。

接下来，在资源类型的基础上，构建模型作为对资源管理对象的抽象。模型用于定义某一类资源的公共模板，包括模型名称、模型 ID、分组、图标、状态等元数据。例如，主机模型可以设定模型名称为“主机”，模型 ID 为“host”，并归属于“主机管理”分组，同时启用模型状态以允许实例化。模型的定义过程实现了资源类型的标准化，为管理提供了统一的描述基础。在算力调度等场景中，模型还可以嵌入调度相关的属性，如资源权重或亲和性规则，以优化资源分配。

模型构建完成后，需要定义模型属性，以使模型具备描述具体资源特征的能力。模型属性是描述资源的关键字段集合，例如对于主机模型，属性可能包括固资编号、内网 IP、外网 IP、操作系统类型、内存容量等。这些属性覆盖了资源的基本信息、运维细节和物理特征，确保模型能够全面反映资源的实际状态。属性定义需考虑管理的粒度，避免过度复杂化，同时兼顾扩展性，以适配多云或云原生环境中的新需求。

在属性定义的基础上，还需对每个属性进行详细描述，规定其数值校验类型、是否必填、可编辑性等特性。例如，固资编号属性可能被设置为短字符类型，非必填，但需通过正则表达式（如+）进行校验，而内网 IP 则需符合 IPv4 规范。属性描述确保了数据的完整性和准确性，在自动化调度中，这种校验机制可以预防资源配置错误，如调度器依据属性值拒绝无效的资源分配。

最后，通过对模型属性进行赋值，构建模型实例，使其代表具体的资源对象。例如，为主机模型创建一个实例时，需为固资编号、内网 IP、操作系统类型等属性赋予实际值，如固资编号设为“A1001”，内网 IP 设为“192.168.1.10”，操作系统类型设为“Linux”。实例化过程可以通过人工输入、API 接口或自动发现工具完成，实例数据需严格遵循属性描述规则，以确保资源信息的一致性和可靠性。在算力调度中，模型实例的实时更新能够支持动态资源调整，例如根据 CPU 使用率或地域信息优化任务分配。

除了属性，关系也是现代企业 IT 资源的重要特征，模型与实例间关系的构建与维护也是实现 IT 资源精细化管理的基础。在 CMDB 中构建模型是第一步，接来下就可以基于模型构建关联关系。关联类型定义了模型间关系的规则，包括方向性、约束和语义。以常见的“上联”关联为例，其源模型为主机，目标模型为交换机，并设置模型关联 ID（如 host_connect_switch）以便唯一标识。关联规则通常包括基数约束，例如 N:1 关系，表示一台主机只能上联一台交换机，而一台交换机可连接多台主机。关联描述则明确关系的语义，如“主机上联交换机”表示主机通过网络端口连接到交换机。这一步骤的关键在于确立关系的逻辑框架，避免实例化时的混乱。

然后进入配置模型的关联关系阶段。在模型级别，基于已定义的关联类型，建立允许的关系网络。例如，允许主机模型通过“上联”关系关联交换机模型，同时允许交换机模型之间或交换机与路由器模型之间建立类似的“上联”关系。这形成了类型级的拓扑结构，如主机-交换机-路由器链路的雏形。配置时需考虑关系的传递性和约束，例如交换机间的级联可能支持 1:N 关系，以适配实际网络设计。这一步骤确保了关系规则的推广性，为实例化提供蓝图。

最后是关系实例化，即为具体资源实例建立关联。例如，主机 1 实例通过“上联”关系连接到交换机 1 实例，交换机 1 实例又连接到交换机 3 实例，而交换机 3 实例最终连接到路由器 1 实例。实例化过程必须遵循模型级约束，如基数限制，并通过自动化工具维护一致性。例如，当主机迁移时，系统自动更新其上联关系，防止数据过期。这一步骤将抽象关系转化为可操作的数据，直接支持运维活动。

模型与实例关系的应用场景广泛，涵盖资源管理到故障分析等多个方面。资源关系维护场景利用关系数据动态跟踪资源连接状态，例如监控服务器与交换机的物理链路变化。拓扑视图查询场景生成可视化网络图，如展示从用户端到后端服务器的完整路径，辅助架构理解。资源依赖分析场景识别关键资源的依赖对象，例如分析某交换机故障时受影响的服务列表，提升容灾能力。关联链路追踪场景支持端到端路径追溯，如安全审计中还原异常访问链。故障根因分析场景通过关系网络快速定位问题源，例如基于交换机下线推断主机不可用原因。变更影响分析场景评估配置修改的潜在风险，如交换机升级前模拟业务影响，确保变更安全。

在云原生技术日益普及的背景下，CMDB 需进行扩展以支持动态、微服务化的资源环境。CMDB 作为 ITIL 框架的核心组件，原本用于管理静态 IT 资产配置项及其关系，但云原生架构引入容器、Pod 等短暂生命周期资源，要求 CMDB 模型能够处理资源的弹性伸缩和实时变化。例如，Cluster 作为 K8s 集群管理单元，Namespace 提供逻辑隔离，Workload 控制部署，Pod 是最小调度单元，而 Container 是应用运行时。这些资源生命周期短、关系复杂。对 CMDB 模型及关系类型进行扩展的目标包括构建层次化资源模型、增强动态关系管理，并集成多种拓扑视角。

支持云原生资源扩展后的 CMDB 能够支持对云原生应用进行拓扑可视化和容量规划，例如通过 Pod 与 Node 关系计算负载均衡度，触发自动扩缩容。任务执行场景实现运维自动化，如蓝绿部署或故障恢复，并可结合算力调度系统优化资源分配（如根据 GPU 需求动态调度 Node）。合理场景扩展还包括监控告警（关联资源指标设置阈值）、安全合规（检

查 Namespace 隔离）以及 DevOps 集成（跟踪部署历史），这些应用赋能企业实现高效运维和成本优化。

CMDB 作为现代 IT 运维管理的核心组件，用于存储和管理 IT 基础设施的配置项及其关系，其重要性在云原生和动态化 IT 环境中日益凸显。随着企业 IT 架构的复杂化，传统手动管理资源的方式已无法满足效率与准确性需求，自动化发现和采集导入方案成为确保 CMDB 数据实时性、一致性的关键。

自动发现与采集导入方案主要包括两种核心方法：API 自动导入和 Agent 自动采集。API 自动导入方案允许业务平台通过 CMDB 提供的原子化及场景化 RESTful 接口，实现对第三方系统（如云平台或 DevOps 工具）资源数据的实时同步。例如，当云资源发生变更时，第三方系统触发 API 事件，经 API 网关鉴权后写入 CMDB，确保数据及时更新。这种方案的优势在于实时性强且无需部署额外组件，特别适合云原生环境，但挑战在于依赖第三方 API 的稳定性，并需处理数据格式转换。Agent 自动采集方案则通过在目标资产上部署轻量级采集器或插件，自动收集资源属性（如 CPU 使用率）和关系（如服务依赖拓扑），经数据处理后上报至 CMDB。该方案覆盖度高，能采集细粒度数据，适用于容器环境或网络隔离场景，但需注意维护 Agent 版本一致性和避免性能开销。两种方案可协同使用，形成混合模式，例如 API 用于云资源同步，Agent 用于物理设备，通过版本控制和冲突解决机制保障数据一致性。

自动化发现导入的重要性体现在多个层面。首先，它能显著提升运维效率，减少人工干预，据行业统计可降低 70%的数据维护成本，并加速故障定位，例如通过实时关系拓扑快速识别服务瓶颈。其次，它保障业务连续性，防止资源漏管和数据错误，在动态环境如自动伸缩组中尤为重要，避免计费或安全风险。此外，自动化发现支撑高级应用，如算力调度优化，在 AI 集群中基于精准资源数据实现弹性分配，或通过识别闲置资源控制成本。未来，随着行业发展，自动化方案将融合算法和 AI 相关技术，实现预测性管理，企业应结合自身架构设计混合发现策略，强化数据一致性治理。

2.1.3 成本管理

对 SRE 而言，资源成本管理的本质是将财务约束转化为工程实践。据调查，行业内 85%的 SRE 都已经承担了成本管理的职责，其中 31%的 SRE 将成本管理纳入到绩效考核中。

对资源成本的管理工作，使 SRE 岗位得到了战略升级，SRE 不再仅仅是技术保障者，现在 SRE 需要在业务、财务和技术三者中寻求平衡。由此 SRE 的战略价值也得到了加强：

从业务层面来说，由于SRE的工作，大多数业务都应该变成预算驱动的成本管理模式，而不再是一直在稳定运行和节省成本中寻求平衡点；从财务层面来说，有了SRE的支持，成本不仅仅是账面数字，而是具备精细化度量和洞察能力的大数据图表。成本优化也能够更主动挖掘降本增效潜力。从工程技术层面来看，成本管理相关工作帮助SRE实现了从业务到资源的自治系统闭环。

在资源成本管理实践中，成熟的SRE平台工程实践往往参考FinOps的成本治理理念。FinOps将成本治理分为三个层次：成本洞察、成本优化和成本运营。成本洞察是指对成本的构成能够精细化展示，对成本的变化能够有效洞察；成本优化是指基于成本数据分析潜在的成本优化空间，给出成本优化建议，并且通过系统流程支持成本优化工作的开展。成本运营更偏向长期举措，调动跨职能团队间的协作，建立成本管理文化，来确保成本洞察和优化能够持续进行。

成本洞察作为成本管理基础，需要SRE重点关注成本透视、成本分析以及资源用量分析等几个方面的建设。

1. 成本透视：此部分聚焦于成本数据的可视化呈现与实时监控。通过成本分摊透视，基于不同科目（如部门、项目、产品等）清晰展示成本核算数据、用量数据以及成本占比，提供结构化的成本视图。同时，成本数据趋势支持按时间周期或多个维度（如同比、环比）展示成本变化趋势，助力用户及时掌握成本动态。为了优化资源效率，监控看板集中展示包括CPU、内存(MEM)、GPU等关键资源的分配与使用趋势大盘数据，辅助识别资源利用低效环节。此外，成本异常检测运用先进的AI算法，能自动侦测费用中的异常波动并主动发出预警，提升成本风险的应对速度。

2. 成本分析：在成本透视的基础上，该模块提供深层次的分析能力以诊断成本动因并预见未来变化。其核心能力之一是下钻及关联分析，支持从多个维度对具体成本子项进行深度钻取和关联分析，将纯成本数据解读延伸至具体资源消耗和业务场景层面，从而迅速锁定驱动成本变化的业务因素或关键事件。通过对比分析功能，用户能对不同的成本项进行横向（不同对象间）或纵向（历史同期）的对比，以更直观地衡量成本表现。动因分析针对波动显著的成本项，通过自动化匹配运营活动、节假日等内外部因素，直观揭示成本变化的潜在原因。成本预测也是对SRE极为有帮助的功能，基于AI算法，综合考虑业务增长计划、预期运营活动、主要影响事件等多维度信息，对下一阶段的成本趋势做出预测，能够为预算制定和决策提供重要依据。

3. 资源用量分析：此模块将视角延伸至支撑成本的底层物理资源。其中，设备用量分析聚焦于硬件基础设施（如服务器主机），能够针对单个业务线或整个组织分析其硬件设备的区域分布情况、实际用量以及平均成本单价等关键指标。而设备负载分析则从业务或应用维度出发，深入统计 CPU、内存(MEM)、GPU 等核心资源的利用率，并支持按机型、环境、逻辑区域等多样化维度进行聚合展示，精准定位性能瓶颈与优化空间。在连接性资源方面，网络带宽分析专注于解析业务所消耗的网络资源，涵盖运营商带宽、专线带宽以及 CDN 带宽等不同类型的使用情况。最后，为确保灵活性，特殊资源分析预留了扩展接口，以便根据特定需求（如高性能计算场景下的超算资源）进行定制化的资源监控与分析支持。

总结来说，成本洞察旨在构建从成本数据可视化监控（成本透视）-> 成本变化深度诊断与预测（成本分析）-> 资源消耗详细解析（资源用量分析）的完整闭环，为企业高效的成本管控与资源优化决策提供强有力的支持。

成本优化是在成本洞察基础上提供成本优化方法论支持和流程支持。

为构建系统化的成本优化机制，SRE 平台工程实践应通过闭环流程整合三大核心模块，确保成本管控的科学性与执行力：

1. AI+优化策略

通过人工智能与算法分析技术，系统可为单一业务线或组织整体生成精准的成本优化建议。该模块深度解析资源使用模式，识别潜在优化空间，并结合算力调度能力动态输出可行的资源调度策略，为后续行动提供数据驱动的决策依据。

2. 策略执行中枢（流程枢纽）

作为承上启下的关键环节，此模块将 AI 生成的优化策略转化为可落地方案。通过跨部门协作机制，统筹调度资源与任务分配，确保优化策略与业务场景深度适配，并为专项执行提供标准化操作框架。

3. 成本优化专项

由专业 SRE 成本顾问主导，为业务单元提供定制化优化服务。基于 AI 策略与业务特征，量身设计阶段性优化方案，覆盖技术架构调整、资源配置重构等核心领域，并全程监督执行过程，保障方案实效。

此外为固化优化成果并形成长效管理机制，还需建立以下三类标准化流程：

1、计划提报与复盘机制

在SRE系统中搭建周期性成本优化计划提报端口，支持业务团队或SRE主动提交优化目标。通过定期复盘会议（如季度评审），追踪计划进展并迭代优化方向。

2、操作审批与统计规范

对涉及资源配置调整、服务降级等具体优化操作，须通过SRE系统申报并履行分级审批流程。所有操作均纳入年度成本优化统计库，形成可追溯的执行档案。

3、成效度量与验证

设定科学的成本优化成效评估指标（如单位业务资源降本率、ROI提升值），通过成本监控工具量化验证优化效果，避免主观评估偏差，持续完善优化模型。

综合来看，成本优化应通过“AI策略生成→中枢调度→专项执行”的闭环运作，将技术驱动与流程管控深度融合。同时借助三大支持机制（计划、审批、度量），既保障优化动作的规范性，又实现成效的可视化管理，为组织构建可持续的成本竞争力提供方法论支撑。

最后成本运营通过建立成本评价体系、管理体系，以及通过文化手段建立良性循环，持续推动成本管理工作。

1、建立成本评价基准体系

成本管理的首要环节是构建多维度的评价标准。成本健康度综合评价需整合业务视角（资源投入产出比）、财务视角（成本利润率）及SRE视角（资源可用性成本）建立复合评价模型，并将该模型纳入周期性经营评审体系，形成常态化的成本健康监测机制。

资源作为成本的载体，还需要建立资源健康度量化指标，针对计算资源、存储资源等基础设施的使用效能，设定可量化的利用率、闲置率、碎片率等核心指标。通过可视化看板实时呈现指标动态，为资源优化决策提供数据支撑。

此外，SRE还需要关注业务及平台情报的关联分析。通过系统级深度关联业务运营指标（如用户增长率、交易量波动）与平台运营数据（如扩容频率、故障率），通过归因模型动态识别业务发展策略或平台架构缺陷对成本结构的深层影响。

2、设定成本管控执行机制

建立预算闭环管理机制，实行预算编制-执行监控-偏差分析的闭环管理。将预算额度按业务单元分解，通过预算达成率（实际消耗/预算额度）量化考核执行效能，强化预算刚性约束。

建立资源效率目标管理机制，根据业务周期特性（如促销期/平峰期），按季度设定CPU使用率、存储压缩率等资源效率目标值，并将其纳入技术团队绩效考核指标。

实施成本异常监测预警，建立月度成本复盘制度，运用阈值检测算法自动识别超支、资源闲置等异常情况，触发分级预警流程（邮件/企微/钉钉/飞书/电话逐级上报）。

3、培育职能协同支撑能力

生成战略级成本分析报告，例如按季度生成三维度专项报告：成本全景分析（总量/趋势/结构）、重大异常根因解析、优化方案建议书等，并定向推送至组织管理层级。

配置管理者全景视图。通过配置差异化数据看板：例如 CEO 视图聚焦战略投入产出比，CTO 视图透视技术资源效能，CFO 视图监控现金流健康度，实现基于管理角色的数据穿透。

注重成本文化长效培育机制，例如开展"成本优化标兵"评选，树立典型标杆；建立成本节约专项奖励基金与超标问责制度；定期组织跨部门成本管理工作坊等，推动成本意识融入组织基因。

2.1.4 资源管控通道

基础资源管控包含资源信息管理和操作管控两部分，它们相辅相承，缺一不可，这两部分是实现 IaC（ Infrastructure as Code ）的底层依赖，并且共同构成 SRE 平台工程实践的基础。资源管控通道是一种基于软件定义网络和代理技术的集中式管理框架，在互联网和云计算领域中扮演关键角色，用于实现对分布式计算设备的远程控制、数据传输和任务调度。随着云计算和分布式系统的快速发展，设备数量激增且环境复杂度提高，例如混合云和多架构平台的普及，传统手动管理方式已无法满足高效运维需求。管控通道通过标准化协议如 HTTP/2 和 gRPC，以及加密通信如 TLS，提供安全可靠的连接，实现自动化管控，从而提升算力资源的利用率和弹性。其核心价值在于降低运维成本、提高任务执行效率，并支持实时监控和动态扩展，成为算力调度体系中的基础设施支柱。

在实际落地中，管控通道分为两种实现方式：一类是无 Agent 模式，一类是有 Agent 模式。

无 Agent 模式的典型代表为 Ansible。Ansible 是一款开源的自动化运维工具，专注于简化配置管理、应用部署、任务编排和基础设施自动化。其核心设计基于无代理架构和声明式语言（YAML），通过 SSH 或 WinRM 协议实现远程主机管理。

以 Ansible 为代表的无 Agent 类管控通道实践方案，一般具备架构简单，初期实施成本低，SRE 或运维人员能够快速上手的特点。不过使用上也存在一些局限性，例如：在大规模资源集群管理时，SSH 连接可能存在延迟导致任务调度效率低下；由于 SSH 协议的限制，

无 Agent 模式无法承载复杂的资源实例及任务状态跟踪机制，对资源实例的操作也局限于资源实例本身能够通过 SSH 唤醒的功能。在权限控制上完全依赖 OS 本身的账户，控制力度较粗，不一定能够满足应对复杂场景的 SRE 平台一体化设计的功能要求。

有 Agent 模式一般需要在资源实例（例如 VM、物理机、裸金属）上部署 Agent 常驻进程。因为要部署 Agent，所以这种模式初期安装和升级的成本较高。不过也因为有独立进程，在安全性和功能上相较于无 Agent 模式要强大很多。下文中主要以有 Agent 模式来探讨具体实践方案。

SRE 对资源的操作大体可以抽象为脚本执行、文件传输、数据采集三大类，管控通道是这三大类操作的上下行管道。在应用场景方面，管控通道广泛适用于设备批量作业执行，允许用户向大量设备如云服务器或边缘节点下发批量命令，实现并行任务处理，这依赖于作业调度算法如 FIFO 或优先级调度，确保任务有序执行，常用于自动化运维或大规模软件部署。此外，它还支持文件投递、收集和传输，通过断点续传和压缩技术优化带宽使用，并内置校验机制确保数据完整性，适用于日志收集或模型参数传输场景，在算力调度中促进计算节点间的数据共享。数据采集和汇聚功能则从分散设备实时收集指标或日志，汇聚到中心平台处理，支持多种协议和数据格式，用于监控资源利用率并为调度决策提供数据支撑，可结合流处理框架实现实时分析。管控通道还具备扩展性，可定制化适应新兴需求如安全审计或自动化故障恢复等。

功能特性上，管控通道支持云服务器和办公设备的多场景类型管控，采用基于角色的访问控制和策略引擎实现细粒度权限管理，借助代理程序和心跳机制保持连接活性。它兼容公有云、私有云、办公网和混合云环境，通过 VPN 或专线连接实现跨网络管控，并利用软件定义广域网技术优化流量，减少延迟，在算力调度中实现负载均衡和成本优化。网络方面，它支持 IPv4 和 IPv6 双栈网络以及虚拟网络穿透，使用 NAT traversal 技术解决私网访问问题，增强分布式算力调度的灵活性，尤其在边缘计算场景。跨平台兼容性覆盖 Linux、Windows、MacOS 和 Unix 系统，代理程序采用跨平台框架开发，提供统一管理接口，简化异构环境运维，并支持 x86 和 ARM 等多种架构类型。

以腾讯蓝鲸管控平台为例，Agent 基于私有安全协议与服务层通讯，相比于 SSH 协议更难破解。此外 Agent 还原生支持双向证书验证、动态密钥注入，在敏感数据保护上更安全。在用户权限控制层面，也不依赖 OS 的账户限制，可以基于业务的 RBAC 模型，将权限控制到主机或任务级别。

在功能层面，有 Agent 模式相比无 Agent 模式，具有更高的并发性能和更大扩展能力。在任务并发能力上，无 Agent 模式的 Ansible 在默认并发数时，千台主机并发命令执行耗时通常在 10s 以上，并且容易受到 SSH 连接池限制影响。蓝鲸有 Agent 模式，在支持万级并发的同时，千台主机并发命令执行耗时在 2s 以内。在文件传输场景，Ansible 支持通过 SCP 或 SFTP 方式，千台主机分发 500MB 文件通常需要 40-60 分钟，并且带宽消耗基本与并发数量成线性关系。蓝鲸 Agent 支持采用智能 P2P 方式，千台主机分发 1GB 文件完成时间可控制在 200s 以内，并且由于采用 P2P 传输，带宽节省率超过 80%。

除命令执行和文件传输功能，有 Agent 模式因为具备实时数据上下行能力，能够更方便扩展数据采集能力。蓝鲸 Agent 支持插件化部署采集能力，目前能够支持主机监控数据、应用观测数据、日志数据等的采集上传。并且通过预留的通讯接口能够低成本扩展满足其他数据采集场景的插件。

有 Agent 的模式，安装和维护 Agent 都是成本很高的工作，因此通过一个 Agent 来尽可能完成更多的操作支持更多的场景，能极大提升这种模式的收益，这就是 One Agent 模式。One Agent 模式是运维与 SRE 领域的一种创新机制，指通过统一的技术框架整合多种运维能力，解决传统多 Agent 部署的资源浪费与运维复杂性问题，并提升系统可观测性与运维效率的架构方案。在 One Agent 模式下，有 Agent 模式具备很多想象力，只要技术架构设计合理，这种扩展性几乎是无限的。例如在混沌场景下，基于同一个 Agent，可以快速衍生出混沌控制能力，而不用安装专用的混沌控制端；在主机安全场景，基于同一 Agent，可以衍生出安全类插件来完成相关工作；在云原生场景下，Agent 还能作为镜像加速传输和部分特殊操作的安全通道。

在较为完善的 SRE 平台化工程实践中，可以通过设计管控通道的自治能力来降低 Agent 的升级维护成本。例如通过 OTA 来解决 Agent 自升级的问题，从而避免繁琐的人工操作。Agent 应该能够在接收到 OTA 升级信息后，自行选择合适的时机完成升级，以保证正在执行的任务不会中断，正在传输的文件能够断点续传，正在采集的数据不会丢失。

在中大型企业，资源所在区域往往涉及到多个网络逻辑区域，彼此之间通过防火墙等网络设备隔离通讯。管控通道需要支持这种场景，通过开启有限的网络策略，能够跨网络区域支持对所有区域内资源实例统一管控。

2.1.5 资源节点管理

有 Agent 模式的管控通道本身也是一个庞大的系统，对于大型企业，可能涉及多个逻辑网络分区下的数万甚至数十万服务器中部署的 Agent 的管理。SRE 能否高效管理这些 Agent 及其配套组件，包括安装、卸载、升级、改配、扩缩容等操作，很大程度上决定了整个 SRE 平台工程实践的可靠性。

在较为成熟的 SRE 平台工程实践中，一般存在资源节点管理服务，用来管理管控通道，负责管控通道 Agent 相关组件的安装、升级、维护等。结合管控通道的复杂性、扩展性和可维护可观测性要求，资源节点管理需要具备以下能力：

2.1.5.1 管控通道 Agent 安装维护能力

当需要在数千、数万、甚至数十万资源节点上安装 Agent 时，通过 SSH 登录，SRE 执行安装脚本的方式效率太低，安装工作可能需要耗费大量人力和时间成本。卸载 Agent 也是常规工作之一，例如某些服务器实例过保了，被业务替换下来之后，就需要执行清理动作。当业务数据、业务组件全部清理完后，就需要卸载 Agent，然后重装操作系统，服务器进入供应链管理系统继续流转。为了提高效率，SRE 需要一种自动化且能够批量安装、卸载 Agent，并能自动完成状态检查的操作管理服务，这就是资源节点管理解决的主要问题之一。

除安装卸载外，Agent 升级也是一项繁琐的工作。特别是在一些安全控制流程比较复杂的企业，要操作现网服务器上的程序，需要层层审批，并且需要和各个关联单位协调操作时间。如果 Agent 升级不能自动化，那么这项工作将成为整个 SRE 平台工程实践中维护成本最高的。

OTA（Over-the-Air）升级是比较符合 Agent 工作场景的方式，此外为了尽可能降低升级过程对上层服务的影响，Agent 升级动作应该尽量做到无损，也就是无损 OTA。整个无损 OTA 的流程大约如下：

第一步，资源节点管理侧描述 Agent 更新计划，设定分阶段预期目标和过程控制策略。例如待升级目标 Agent 范围，可能是一个 ID 集合，这是一个灰度升级策略约束条件；待升级目标 Agent 版本范围，这是另一个灰度策略约束条件，用来进一步限制只对当前版本号符合约束范围的 Agent 进行 OTA 升级；OTA 升级的目标版本号，这是一个升级计划的目标描述，说明本次升级计划是要将符合灰度条件的 Agent 升级到这个目标版本。由于 Agent 在收到 OTA 升级信号时，可能正在执行作业，那么需要设定一个柔性等待时间，在

这个时间范围内，Agent需要挑选一个空闲时间窗口启动自升级动作，这个就是柔性升级模式，柔性升级的目的是为了确保用户体验无损；当然在某些紧急情况下，例如紧急漏洞修复，需要强制更新Agent，升级计划中也可以选择强制模式，强制模式下，Agent会主动打断正在执行的作业，并立即执行自升级动作。

一个OTA升级计划，会分多个阶段来实施，并逐步放宽约束条件，直至达成计划目标。更新计划中需要明确定义每一阶段的灰度约束条件、目标、和进入下一阶段的条件。例如第一阶段，通过200个ID的集合和当前版本为beta1，这两个约束圈定这一阶段的灰度范围。并设定目标为将符合条件的Agent全部升级为beta2。进入下一阶段的条件为90%符合条件的Agent升级成功，并且验证功能正常。第二阶段则可以将ID集合扩大到1000个，当前版本号范围扩大到beta1.1，beta1.2，beta1.3都可以升级。依此类推，直到达成本次整体升级目标。

更新计划中还需要描述异常情况下的处理逻辑，例如如果OTA失败的处理方式，柔性等待时间超时的处理方式等。极端情况下可能需要回退版本，更新计划中同样要有匹配的回退策略。

第二步，启动升级执行计划，然后观测阶段性结果与计划目标是否一致，并依据结果确定是否需要人工接入。大批量的Agent升级是个持续性过程，不管是柔性升级还是强制升级，都存在一些偶然因素不能确保每个更新计划都能一次完成全部Agent升级。因此实时跟踪更新计划的执行情况尤为重要。资源节点管理必须能够实时准确监测每个更新计划的目标Agent状态，并统计每个阶段的达成情况，对于没有在计划范围内完成升级的Agent能够追查原因，并给出下一步建议。当需要人工接入时，资源节点管理需要提供人工接入通道。

2.1.5.2 管控通道 Agent扩展插件管理能力

插件生态是Agent重要的能力扩展基础，Agent定义好与插件的标准交互接口和管理方案，第三方团队可以据此开发不同类型的插件，例如数据采集插件、安全类插件、数据库操作插件等。

Agent对插件的管理主要包含以下三个方面：

一是插件的版本管理，每个插件都应该有版本划分，以便于控制插件功能与运行环境的兼容性，插件的升级和回退都以插件的版本信息为操作基准。除插件版本信息本身的管理外，还需要支持插件的灰度发布能力，因为不同的插件作用域不一样，例如日志采集插件只需要部署在日志服务器上；数据库类插件只需要部署在存储模块的服务器上。在不同

的环境，插件的版本可能存储在不同的存储介质中，资源节点管理服务需要有足够的扩展能力支持不同存储方案。

二是插件的配置管理，插件也是独立的运行实例，通过不同的配置来确定插件的行为。例如日志采集插件中，需要明确日志存储的路径，日志文件名等；数据库插件中需要明确不同数据库的访问地址，访问权限信息等。

三是插件生命周期管理，插件从安装到卸载的全生命周期过程中，也会有升级、重启、重载等动作，插件的状态也会变化，生命周期管理就是提供对插件全生命周期过程中状态等数据和操作维护流程的管理。上层服务可以通过资源节点管理服务的接口来操作指定插件的安装、卸载、重载、升级，并通过同步或异步接口查询操作结果。

2.1.5.3 管控通道拓扑管理能力

管控通道具有相对复杂的部署架构，在一个大型企业中，资源节点可能分布在多个网络区域，为了统一管理所有的资源节点，除了部署管控通道的服务端外，还需要若干代理节点，来完成跨区域网络穿透，部署在资源节点上的 Agent 将通过这些代理节点组建管控集群。

管控通道拓扑管理，就是要将现网部署的 Agent、代理节点、管控通道服务端通过可视化的方式展示出来。并通过视图中的节点来代表资源节点上部署的服务进程，节点之间的连线来代表通讯关系。对于成熟的 SRE 平台工程实践，还可以通过视图中节点的颜色、形状、大小、明暗程度等变化来展示节点上管控通道服务的状态；通连线颜色、粗细、虚实等来展示通讯的状态。

相对于传统 CMDB 中的静态拓扑关系，资源节点管理中的管控通道拓扑视图更贴近于现网实时情况，当用户通过资源节点管理安装新的代理节点或 Agent，或者卸载某个 Agent 后，拓扑视图会在很小的刷新周期内展示现网变化。

管控拓扑视图的核心价值在于将复杂的管控通道部署关系通过信息化手段浓缩在一个 web 视图中，极大增强了多节点信息获取能力，降低了管控通道自身的可维护可观测成本。

2.1.6 作业管理

作业管理和管控通道的关系，类似于长途汽车站和高速公路。长途汽车站实现了对大巴车的管理和调度，并通过高速公路将乘客安全高效地运送到各地。如果没有高速公路，大批量运送乘客的难度会显著增加，效率和安全也得不到保障。如果缺乏长途汽车站，乘客需要自己买车，使用高速公路的成本就会很高，单车运载效率低也会一定程度浪费高速

公路带宽资源，同时高速公路上的车辆也可能因为缺乏管理而混乱。对于作业管理服务和管控通道服务来说，如果没有作业管理，所有上层系统都需要直接对接管控通道，那么作业管理逻辑、权限控制逻辑、用户体验相关功能等需要所有对接的系统都开发一遍，不仅浪费工程师人力，而且在场景抽象和标准化上留下技术债。

作业管理在管控通道的能力基础上提供了初级场景封装，负责抽象面向用户的作业模板，管理作业形态，提供作业执行流程控制，完成作业下发和执行结果回溯，并提供权限控制和安全审计相关能力。对以上场景进行拆分和抽象，可以发现作业管理需要面向用户提供如下三类操作界面和接口：作业分发管理、流程化作业编排、作业权限与安全控制，来实现对这个基础场景的完整覆盖。

一、作业分发管理：构建高效可控的作业执行中枢。

作业分发管理作为作业流水线的核心调度层，通过模板化作业管理机制实现作业的标准化和复用性。用户可创建通用作业模板，快速生成实例并进行克隆共享，显著提升复杂作业的设计效率。

在作业执行层面，作业管理需要支持多维度分发策略，例如：单次分发模式，这种模式适用于即时任务，实现秒级触达，快速返回执行结果；计划性分发模式：支持秒级精度的定时作业，定点触发作业，或周期性作业的设置；智能分批滚动模式：对大规模目标节点动态分批次执行（如千台服务器按故障域分批），并结合执行半径控制技术 自动规避网络隔离域。

对执行过程的监控，作业管理需要提供全链路状态追踪，通过可视化面板实时展示作业状态，并自动触发异常结果通知。管理者可通过精细化控制体系干预作业生命周期，例如：作业超时熔断、强制终止等防灾手段，确保失控作业不会造成任务风暴。

二、流程化作业编排：原子任务组合的智能化引擎。

在成熟的 SRE 平台工程实践中，流程化作业编排通常采用乐高式原子任务架构，将脚本执行、文件分发等基础能力解耦为可编排单元。用户通过表单化、图形化流程编辑器自由组合原子任务，构建跨系统的复杂作业方案，例如：停服-备份-升级-验证-启服等全流程性作业。

为确保作业编排能力的可复用性、可扩展性，流程化作业编排应支持如下核心特性：

- 流程韧性设计：支持单步骤异常重试、人工审批跳过关键节点；
- 动态参数体系：支持自定义环境变量注入或传入自定义作业输入参数；
- 执行方案模板：支持将验证通过的流程沉淀为可复用作业模板库；

在定时作业管理方面，应支持秒级定时时间配置，支持作业执行时间窗口配置，支持周期性作业的循环模式设置，以贴近 SRE 实际的工作场景。对于定时类作业，应提供执行质量看板，查看历史作业成功率情况，作业流程耗时情况，为后续作业的持续优化提供数据支撑。

三、权限与安全控制：企业级 SRE 的合规性基石。

在权限控制层面，基于多租户隔离模型构建纵深防御体系，实现基于 CMDB 的精细化授权模式。例如通过 CMDB 构建动态资源视图，将资源实例按业务拓扑分组，按业务实现权限隔离。这样用户仅能操作其被授权业务下的资源实例。另外作业执行、文件分发等具体操作也需要精细化划分权限，实现“用户-资源-操作”层面的隔离。

作业执行时，尽量避免使用管理员账户，避免出现权限过大导致安全风险。作业管理服务需要支持管理员预置资源实例 OS 账户，并绑定到平台账号，作业执行时自动使用托管账户鉴权，避免用户直接持有账户和密码等高危信息。

此外有些大型企业有多级分权的场景，例如企业内每个部门需要一个专属的管理空间，在这个空间内可以配置独立的人员权限边界和资源授权策略。作业管理的权限设计上也需要支持这类复杂场景。

除账号权限外，作业执行的安全控制策略也是重要的实时防御和风险阻断内容。这里主要有三个方面需要考虑：一是作业自身的安全控制。在成千上万台资源实例上执行的脚本，如果存在高危命令，则可能造成难以估计的损失。因此对作业内容的实时监测对于作业执行安全非常重要，常规的安全检测方式为扫描作业中脚本内容并拦截违规命令，并记录拦截日志，日志内容包括但不限于触发规则、执行人、目标资源实例等信息。对于成熟度较高的 SRE 平台工程实践，采用 AI 模式识别的方式，更有利于沉淀不同场景的策略模式。二是 API 的访问控制。第三方系统通过 API 访问作业管理服务时，需要申请唯一的权限票据，并限定权限范围和过期时间。当 API 调用出现异常时，应及时熔断处理，并返回预设响应，防止故障或攻击扩散。三是作业内容的加密保护。作业内容包括分发文件的内容，需要支持加密传输，并增加完整性校验措施，防止作业内容在传输过程中发生变化，或者被恶意修改。

最后是作业执行记录的全链路操作留痕处理，以便于需要时进行安全审计。作业执行记录需要包含如下内容：执行参数，包括脚本内容、全局变量、目标资源实例列表等；环境快照，包括 Agent 状态、资源拓扑信息等；执行结果详情，包括标准输出、错误输出、作业退出码等。

作为 SRE 平台工程实践中的重要平台，作业管理承担了用户态服务底层入口的作用，因此其生态集成与场景化扩展能力尤为重要。通过标准化接口中枢实现与其他系统的无缝协同，实现开箱即用的生态集成能力，从行业优秀实践经验来看，其核心设计遵循三层集成范式：基础协议层——支持 RESTful API 和 WebSocket 双通道通信协议，基本能覆盖 99%第三方系统对接需求；内置 JSON/XML 格等式解析引擎，能够自动适配不同平台的响应数据结构；深度集成 CMDB 配置数据库，实现资源实例属性（如 IP/所属业务组）自动同步至作业管理系统。

在场景扩展能力上，作业管理建议采用微内核加插件化设计模式，开发者可通过轻量级扩展满足定制化需求。例如通过开放式插件接口扩展支持自定义文件源，满足多元文件分发需求。在金融机构，敏感文件传输合规要求较高，就可以通过这种插件对接企业内部的加密存储系统。通过 SaaS 封装作业管理接口解决某些具体场景下的问题，是另一种扩展方式。例如基金公司通过封装作业管理原子接口，实现日常巡检管理能力；游戏公司通过封装作业管理原子接口，组合脚本执行、配置修改、服务启动等作业模板，实现新区服快速开服的能力。

2.2 公共基础设施

基础资源管控完成了对资源元数据的管理和资源实例操作控制的管理，这是 SRE 综合算力系统的基础。但是光有基础还不够，还需要一些公共基础设施来共同组合，以便于围绕资源搭建更多更深层的服务，这些设施主要包括：网关系统、登录管理、用户管理、权限管理和审计中心。如果拿建设酒店作为比喻，基础资源管控只是完成了酒店的房屋主体建设。房屋是酒店的基础，决定了酒店有多少房间能够容纳多少客人。但光有房屋还无法接待客人，还需要酒店前厅服务、客房服务、安保服务、客服服务等才能确保酒店正常运作。前厅服务用来接待客人，并登记客人信息发放房卡；客房服务确保客人入住的是打扫清洁过的房间，并为住宿客人提供权益内的服务；安保服务维持酒店秩序，保护正常住宿客人权益，防止越权等破坏性事件发生；客服服务则对酒店服务进行回访统计，接受客户投诉，帮助酒店服务查漏补缺改善提升。

对于 SRE 综合算力调度这座“酒店”也是一样，也需要这些公共基础设施来搭建基础服务框架。网关、用户管理、登录管理就像前厅服务，让授权用户访问流量能够进入 SRE 综合算力系统；权限管理则像安保服务，确保合法授权用户的行为得到保护，拦截非法用户

越权行为；审计中心则类似客服服务，对用户行为进行记录，并定期审计，合规检查，风险预警。

2.2.1 网关

API 网关是一个重要的网关形态。在复杂架构的软件系统中，上下层通讯关系错综复杂，如果通讯消费方和提供方直连，最终会形成无法维护的蜘蛛网架构。此外，直连的消费方都需要实现独立的认证、限流逻辑，功能开发存在重复。服务 API 提供方也因为消费方直连，难以处理兼容性问题。

API 网关的价值就在于破解复杂系统的链接困局。API 网关统一管理南北向 API 流量，并提供标准化入口，用来收敛调用路径，简化调用逻辑；通过内置功能插件来统一实现认证、限流、熔断等通用逻辑，降低 API 消费门槛；通过网关的标准化操作，也更容易实现不同协议间的通讯转换，从而避免后端私有协议暴漏带来安全风险；通过 API 网关能对 API 进行更精细化治理，例如 API 的版本灰度发布控制、API 文档自动生成、API 自助上下架等。组合型 API 也是 API 网关带来的一大好处，通过将 API 提供方的原子 API 自定义组合为新的 API，并对外暴露新 API 来降低消费者调用逻辑的复杂度。

除 API 网关外，流量网关是另一种网关形态。流量网关一般工作在网络协议栈四层，直接代理了业务的南北向流量。相比于 API 网关，流量网关更关注转发效率、高可用和可靠性。

微网关是一种新的分布式网关工作模式。微网关是将网关能力拆分为轻量化，可以独立部署的组建，并由统一的控制面集中管理的数据面代理集合。同一套分布式网关下，不同的微网关可以是 API 网关形态，也可以是流量网关形态。

通过微网关这种模式，不同业务领域可以定制专属的微网关，即保持业务特性的个性化支持，又能避免实际工作时相互影响。例如支付业务系统使用 HTTP 协议的微网关，而交易系统使用 gRPC 协议的微网关。微网关实例之间相互隔离，单一微网关的变更就不会造成全局影响，对微网关因业务容量变化而带来的扩缩容操作也更容易控制。

网关的统一控制面需要能够一站式管理所有微网关实例，简化网关的整理架构。不同的微网关实例可能具备不同的版本，网关控制面能够根据版本信息进行统一的发布管理。控制面还应该提供网关访问的权限控制，提供对网关操作日志的审计能力。针对每个微网关实例，提供可观测视图，展示微网关的工作情况，例如微网关的 QPS、通讯延迟、错误率统计等。

2.2.1.1 网关设计

按照上述要求，设计一个支持既支持 API 网关又支持流量网关的分布式网关系统的思路如下：

作为网关系统中最重要的两大组件，控制面与数据面需在底层设计上采用模块化思想，在各自保持一定独立性的前提下相互协作。

数据面以高性能开源网关软件为基础，通过插件扩展能力，以支持丰富的认证模式、流量控制等功能。作为独立组件，数据面支持单独部署并可以方便地进行水平扩展。为了适配云原生、二进制等多样化的部署环境，数据面的控制器组件，可同时兼容 Kubernetes CRD、etcd 等多种数据源，无差别地驱动数据面正常工作。

如前所述，一个控制面负责管控一个或多个数据面，二者间遵循严格的数据交互规范。作为直接面向用户的组件，控制面针对不同场景，提供多种网关管理模式：

- 网关 Dashboard：基于 Web 技术的管理站点，面向所有用户提供直观的网关和 API 资源的管理功能；
- 网关管理 SDK：支持各类编程语言的 SDK 工具，开发者调用这些 SDK 能便捷完成声明式注册、更新网关版本等操作；
- 网关管理 API：开放式后台 API，开发者通过编码调用这些 API，实现更为精细、复杂的网关管理操作。

2.2.1.2 网关控制面

一个网关，从开始接入到投入使用，要经历许多个不同阶段。控制面的功能需要完整覆盖网关的整个生命周期。

从接入阶段开始，除了通过 Dashboard 自助接入，控制面还需要支持网关的导入/导出功能。在 API 定义和描述领域，OpenAPI 规范目前是行业中的事实标准，用户编写符合 OpenAPI 规范的网关描述文件，在控制面完成网关实例的导入。

接入网关的后端服务，常拥有测试、预发布、生产等多套不同的部署环境。因此，网关系统的控制面也需要配套提供多环境管理功能。不同环境间，利用环境变量、后端服务等概念进行区分管理。

网关成功接入后，需要发布才能被其他人调用。在发布方面，控制面利用“版本”机制来管理网关每一次的资源变化。每次发布前，展示待发布版本与当前线上版本间的资源差异，以方便网关管理者直观地了解到发布可能带来的影响，避免错误操作导致线上异常。

发布网关后，网关管理者还需要解决另一个重要问题："如何方便用户使用我的网关？"。API 资源文档，作为用户了解一个网关的重要资料，需要每位管理者的悉心维护。网关系统控制面支持中文、英文等多语种的资源文档录入功能，并在统一的 API 文档/帮助站点，展示各网关的资源文档与调用说明、示例，最大程度给予用户便利。

为了更进一步方便用户使用，控制面还提供了自动生成 SDK 的功能。录入资源后，网关管理者可一键生成 Python/Go 等多种语言的 SDK，并发布到仓库与帮助中心。使用对应编程语言的用户，可以下载使用 SDK 来快捷调用网关资源。

控制面通过"资源权限审批"来保障 API 的调用安全，避免敏感 API 资源出现未授权调用的情况。任何想要调用网关的用户，需先发起资源调用申请，网关管理员审批通过后，相关的请求才会被放行；资源授权并非永久有效，过期前需手动（或自动触发）完成续期。这套完善的审批机制可以很好地保障业务安全。

使用流水日志功能，网关管理者可以查看用户发向网关的每一个请求细节。Dashboard 支持按时间维度查看每个 API 资源的每次调用详情，所展示的字段包括请求路径、请求参数（脱敏后）、响应时间、响应状态码等。除了流水日志外，控制面还提供了调用数据统计看板，其中用图表展示了请求量变化趋势等信息。

基于以上这些数据，每当网关表现异常（如突然出现大量错误响应状态码）时，控制面会及时发送告警给网关管理者。

2.2.1.3 插件机制

网关是管控 API 和流量的核心服务，不同类型的用户常常会对网关有各种差异化需求。若是仅仅依靠网关自身去进行功能迭代，很难满足所有用户。因此，成熟的网关必须提供完善的"插件"功能。

插件，是指在不修改网关本身的前提下，通过接入外部插件来为网关扩展出新的功能。一个完善的插件体系需要支持以下特性：

插件管理：提供基础的插件管理功能，如安装与卸载、启用与停用插件，等等；

自定义插件：具备插件开发能力的开发者，可以使用网关支持的编程语言编写自定义插件，实现各种强大的功能，自定义插件可以选择上线到市场。

插件市场：提供一个统一的场所来分门别类地展示各类插件，供用户了解与使用；

2.2.2 权限管理

权限管理是SRE平台工程实践中的重要一环，它直接决定了某个人在某系统上能执行哪些操作。SRE体系中的各个系统与业务稳定运行息息相关，权限管理如果出了问题，可能会导致安全漏洞，进而酿成灾难性的后果。

权限管理系统为体系中其他系统提供统一的权限管理服务。相比放任各系统各自实现权限管理功能，引入权限管理系统来统一管理权限，拥有以下优势：降低各系统各自开发权限管理功能的成本；能更方便地完成跨系统的复杂权限配置；面向用户提供统一的权限申请、审批体验；规避各系统权限回收不及时导致的安全风险。

各外部系统使用统一权限管理的第一步，是将自身产品中的权限模型接入权限管理系统中。因此，要设计一个能服务好其他系统的统一权限管理系统，“鉴权模型”是关键。

为了在保证接入便利性的同时，满足少部分系统的复杂鉴权需求，权限管理系统采用了综合“角色鉴权”与“属性鉴权”的混合模型设计。其中：**角色鉴权（RBAC）：**实现基于用户角色的权限设计，用于顶层权限管理策略设计落地；**属性鉴权（ABAC）：**对于部分复杂的鉴权需求，允许系统在RBAC顶层权限策略下，增量实现基于属性的鉴权方案。

角色鉴权（RBAC）模型符合直觉，接入方便，适用于大多数系统的权限管理场景。属性鉴权（ABAC）更复杂，支持权限策略（“资源+操作+对象”）的灵活组合，是对角色鉴权的一种有效补充，适用于少数系统的特殊鉴权需求。

优秀的统一权限管理系统，需要尽量降低其他系统的接入成本。前面提到，权限管理系统采用了混合模型设计来降低模型适配成本，除此之外，系统还需要在其他方面进一步提升接入便利度，这些措施包括：提供各语言的SDK来支持各系统便捷接入；提供权限模型注册与变更脚本；提供权限管理类OpenAPI，便于各系统二次开发权限管理功能。

除了混合鉴权模型，接入统一权限管理的外部系统还需要了解系统中的另一个重要概念：“分级权限管理”。分级权限管理指通过下放授权权限，设置分级管理员，来提升权限管理效率。它能支持企业内更灵活的权限管理需求。

举例来说，某公司总部的一级授权人A，可以下放授权权限到产品部的二级授权人B，B之后便可代为管理三级部门的其他员工的权限。

用户发起权限申请后，下一步是等待审批。权限审批是整个权限管理中的又一个重要环节。对于审批流程，不同企业常常有着强烈的自定义需求。为满足这些需求，权限管理系统提供了自定义审批流程功能，以适应各种不同企业的多样化审批需求。

在大型组织中，不同分公司/业务线常常需要独立管理权限，权限管理系统也要支持租户隔离功能，从数据到功能使用层面，实现不同租户之间的强隔离。

前面提到，统一管理权限的一大好处是可以统一对权限进行安全治理。对于员工离职交接、转岗清理等情况，系统可以及时进行权限回收，必要时也可以对权限的使用情况进行审计，来保障安全。

在权限审计与追溯方面，系统需要对权限的分配、修改、撤销的全流程记录操作日志，日志内容包含（Who-What-When）关键信息。通过“权限地图”功能，后台可视化展示用户/角色的权限覆盖范围，帮助管理员快速识别越权风险。

在权限的整体管理策略方面，权限管理系统严格遵循“最小权限原则”，默认拒绝所有权限，按需逐级开放；在特殊场景中，系统也支持短期有效的临时权限（如 24 小时内访问权限），到期后自动回收，最大程度规避安全风险。

2.2.3 用户管理

企业内的用户与组织架构信息是最为核心的数据之一，SRE 体系内的大部分系统，都需要这份数据来实现各自的功能。举个例子，告警平台每次发送告警邮件时，就必须用到告警接收人的邮箱地址。

用户管理是负责管理用户与组织架构信息的系统，它统一对接了每个企业内部的用户数据，面向其他系统提供数据消费能力。这份用户与组织架构信息，也是实现权限管理的基础。

设计一个成熟的用户管理系统，需要解决的第一个问题，是对接不同企业的内部用户（及组织架构）数据源。各企业的内部用户体系，往往各不相同，比如 A 企业可能采用 LDAP，B 企业使用企业微信，C 企业甚至可能使用完全自研的用户体系。

为了对接这些各不相同的用户数据源，用户管理系统引入了灵活的“同步插件”机制。每类同步插件负责对接一种特定的数据源类型，完成用户和组织架构数据拉取。系统内置了 LDAP、MAD、企业微信等同步插件，当它们不满足需求时，企业还可以编写自定义插件来完成同步。

企业内的用户与组织架构数据，经由同步插件拉取后，还需要进行清理、合并和标准化，才能存放到用户管理的数据库中。假如企业需要为用户添加某些自定义字段，也可以在用户管理中配置。

数据准备就绪后，便可以提供给其他系统消费。用户管理使用统一的 HTTP API 来对外提供用户和组织架构数据。各 SaaS 和系统，按需在 API 网关中申请 API 权限后，通过调用这些 API 来获取和操作用户数据，实现业务功能。

2.2.4 登录管理

登录管理系统扮演了桥梁的角色，其核心任务是通过插件化架构连接并整合各企业千差万别的内部用户认证与授权系统（如 LDAP、SAML 2.0 IdP、OAuth 2.0 授权服务器等），并最终对外提供一套统一的登录协议（通常是 Session-Cookie）。

这种设计的背景源于企业实际需求：随着企业内部系统和外部 SaaS 应用的增多，必然面临多种不同的认证协议和系统共存。直接让每个应用对接所有可能的认证系统不仅开发工作量巨大，维护成本高，也难以实现统一的安全策略管控和登录行为审计。因此，登录管理需要解决的核心问题就是统一对接这些异构的企业内部登录体系。

针对这个问题的技术实现方案为引入“登录插件”机制。每个登录插件专门负责对接一种特定的登录协议或内部体系，承担了关键的技术适配工作。这包括处理底层通信细节，以及身份令牌的格式转换。例如，OIDC 插件专注于处理基于 HTTPS 的 JWT 令牌交互；SAML 插件则负责解析复杂的 XML 断言；LDAP 插件需要处理二进制协议并与目录树结构交互。

为了最大程度简化企业的接入过程，登录管理系统预置了多种业界广泛采用的标准协议插件。这些“开箱即用”的内置插件覆盖了常见的场景，包括用于现代 Web/移动应用 API 授权的 OAuth 2.0/OIDC（典型案例如微信登录）、用于企业间单点登录集成的 SAML 2.0（常集成 Active Directory 域）、以及用于企业内部目录服务认证的 LDAP。对于非标准或非常特殊的内部登录体系，系统提供了强大的扩展性，允许企业自行编写“自定义插件”，以确保灵活对接任何需求。

统一由登录管理来负责用户登录为 SRE 平台工程实践带来了显著的技术价值：

1.集中化的登录状态与凭证管理：用户的登录状态由系统统一生成、管理和撤销。这使得系统能够清晰地追踪每个用户的登录会话生命周期。

2.统一的安全策略与审计：集中管理身份认证入口点后，更容易集中实施强身份验证、密码策略、会话超时、设备绑定等安全管控策略。同时，所有用户的认证行为都可以在此处进行统一的记录和分析，满足合规审计要求。

3.对接标准化与系统解耦：得益于登录管理系统内部对各种协议的差异化处理，并向上输出统一的 Session-Cookie 协议，上层的应用系统只需对接这一套标准协议接口，彻底屏蔽了底层企业认证系统的复杂性。这极大降低了应用开发和集成的整体成本与维护负担。

2.2.5 审计中心

在金融、央企等高度监管行业，审计中心的建设已成为满足《数据安全法》合规要求的核心基础设施。通过部署轻量级日志采集工具从容器、虚拟机等数据源头实时捕获操作日志，配合 TLS 加密传输机制，实现审计数据的完整性与保密性保障。审计中心对原始日志进行清洗、标准化处理，建立数据血缘图谱以消除噪声，显著提升分析准确性。例如中国石化通过构建跨 42 个业务系统的"审计数据宽表"，关联操作日志、数据库变更和网络流量，完整还原 SRE 操作全貌。分布式存储系统（如 HDFS）与区块链存证技术的结合，则确保审计数据十年内不可篡改，满足监管存档要求。

传统人工审计存在抽样风险与效率瓶颈，审计中心依托大数据技术实现精准溯源。基于 Elasticsearch 等数据库构建的多维度索引系统，支持秒级检索异常操作（如非工作时间访问敏感数据）。通过全局唯一事务 ID 串联操作链路（用户登录→数据导出→文件传输），可完整追溯金融转账等关键业务的全过程行为。结合聚类算法识别批量删除日志等异常模式，以及图数据库挖掘内部舞弊关联线索，使审计分析从被动响应转向主动预测。

面对供应链金融欺诈等新型风险，审计策略需从静态规则升级为动态智能体系。预置的审计规则基于 NIST、ISO27001 等框架，通过大数据处理引擎实时检测权限越界等违规行为。人工智能技术的深度应用进一步强化决策能力：例如 LSTM 模型分析登录时序特征预测暴力破解风险，决策树模型识别虚假贸易模式（如中国石化曾发现数百例"空转走单"违规案例）。通过与 CMDB 系统集成，审计策略自动关联资产变更事件（如服务器下线未注销权限），实现资产全生命周期风险监控。

为规避风险处置延迟导致的数据泄露扩大化，审计中心构建分钟级闭环响应机制。例如基于 CVSS 框架的风险分级系统，对高危事件（如数据泄露）自动触发企业微信/邮件的多渠道通知。通过 Splunk 等 SIEM 系统对接防火墙、IDS 等安全组件，形成全网威胁态势统一视图。风险处置采用工单系统追踪解决进度，将 90%中低风险事件的闭环时间压缩至 72 小时内，并将处置结果反馈至审计策略库持续优化模型。

2.3 流程及自动化基础设施

在云计算和分布式系统高速发展的背景下，IT 资源的高效调度日益依赖流程自动化的抽象能力。流程的本质是将资源操作序列标准化为可复用的控制流，而自动化则通过预设触发条件实现流程的自主执行。当前主流流程管理基础设施包含 ITSM（IT 服务管理）、标准运维及应用配置管理三大核心服务，它们通过对资源管控操作和公共基础设施操作的组合打包，支撑了基础 IT 流程的自定义组装，最终实现资源向算力的高效转化。

资源转化为算力的过程存在显著瓶颈：人工操作易出错且难以响应动态负载。例如云服务器资源池若采用手动调度，流量峰值时易出现资源闲置或过载；而虚拟机启动、负载均衡配置等复杂依赖关系的处理更会引发 CPU 争抢等资源分配不均问题。云计算平台（如 AWS、阿里云）的 Spot 实例调度机制即依赖自动化流程消除此类瓶颈，确保低延迟、高吞吐的算力供给。

自动化系统通过流程组装实现场景无缝适配。典型实践包括：通过 API 网关集成即时通讯或邮件系统触发算力扩容，应对电商大促流量洪峰；在 AI 训练场景中快速调度 GPU 资源支持模型推理；结合灰度发布与 SRE 审批机制实现多云环境下的可靠资源调度。这种自定义组装能力使系统可覆盖 DevOps 流水线、微服务多租户隔离（如 Kubernetes 命名空间）等复杂场景，充分释放算力潜力。

流程自动化的核心价值体现在三方面：一是操作效率提升，平均恢复时间降低 50%以上，资源利用率优化 20%-30%；二是通过 IaC（基础设施即代码）校验机制规避配置漂移导致的算力衰减；三是与算力调度深度协同。当前技术演进方向已形成"编排即服务"框架（如 Kubernetes Operators、AWS Step Functions），并结合 AIOPs 实现负载预测驱动的流程自动组装，构建出完整的云原生算力调度生态。

2.3.1 ITSM

在当今数字化浪潮中，IT 服务管理（ITSM）作为企业核心运营支撑，已深度融合云计算、算力调度等先进技术，实现从传统运维向智能化转型。尤其在互联网行业，标准化的流程和自动化工具能大幅提升服务可靠性与资源利用率，为高并发应用如云平台、数据中心提供持续优化方案。

ITSM 的核心能力首先体现在标准化与自动化上。通过遵循 ITIL、ISO20000 等国际标准，设计覆盖服务请求、变更审批等全环节的流程，确保操作一致性和合规性，减少人为失误。自动化则借助平台工具实现工单自动分派、配置变更执行以及 SLA 实时监控，显著

提升响应速度并降低成本。例如，在云算力调度中，AI 引擎驱动的智能路由可优先处理高优先级任务，结合 Kubernetes 动态资源编排工具，实时触发 CPU 或内存的弹性伸缩，优化资源分配。同时，自动化还可嵌入安全策略如 GDPR，保障数据隐私，并通过 AIOps 预测负载峰值，在电商大促期自动扩容云服务器集群，处理效率提升 30%。这不仅降低了人工干预率 50%，还通过历史数据分析改进调度策略，体现出标准化的严谨与自动化的灵活融合。

其次，流程化与平台化支撑为 ITSM 提供了端到端的协同框架。流程化覆盖服务全生命周期，从策略制定到服务改进，以服务台为统一入口整合事件管理、变更管理等模块化流程，确保服务连续性。例如，在算力资源管理中，实现从申请到退役的闭环控制，如 AI 训练任务自动分配 GPU 资源并在完成后释放，避免浪费。平台化则整合知识库、CMDB 和 AI 分析技术，打破信息孤岛，支持跨部门数据驱动决策。CMDB 可精准映射虚拟资源如容器或 VM，结合微服务架构如 Istio 自动编排跨节点服务链路，降低延迟 40%。知识库升级为 AI 知识图谱后，推荐基于历史数据的优化方案如 GPU 分配策略，平台如 ServiceNow 提供混合云统一视图，协调 AWS 或 Azure 负载，提升协同效率。实际案例中，Netflix 利用此框架实现毫秒级算力调整，应对视频流高峰，SLA 达标率达 99.99%，彰显了平台化的统一管理优势。

最后，管理效益聚焦于高效协同和精细化的优化机制。高效协同通过服务台调度和协作工具实现跨部门无缝沟通，例如开发团队通过 API 实时获取运维数据加速故障修复，同时融入 FinOps 理念将算力成本纳入 SLA 指标，自动推送报告驱动资源回收。精细化管理则依托 SLA 量化指标扩展至算力效率如 PUE 监控，并结合配置项追踪预判风险，如磁盘故障预警触发自动迁移虚拟机。这不仅优化了资源使用率，还支持持续改进——分析工单数据识别瓶颈，例如某金融企业通过数据库查询优化将响应时间缩短 50%，提升整体产出效率 40%以上。行业趋势显示，ITSM 正结合 DevOps 和 AIOps 推动业务创新，如在制造业中自动解答重复问题以降低运维成本 30%，其管理框架已演变为数字化转型基石，通过智能化驱动服务可靠性与成本节约的平衡，确保业务敏捷性和市场竞争力。

在 SRE 平台工程实践领域，IT 服务管理（ITSM）已成为提升系统稳定性和工程效率的核心框架。其标准化流程与自动化机制在实际工程中显著优化了故障响应时效性——当监控系统触发告警时，自动工单引擎会即时生成故障事件单，通过值班系统精准推送给责任团队，处理过程全程留痕可追溯，大幅压缩了从告警到介入的时间周期。这种闭环管理

不仅加快 MTTR（平均修复时间），还通过联动 SRE 的故障根因分析模型形成持续改进机制。

在业务需求管理场景中，ITSM 提供了全生命周期的管理能力。无论是业务部门提出的技术运营需求，还是内部优化方案，系统均会完整记录需求来源、实施内容及责任人信息，并通过数据分析模块量化需求响应效果。例如借助卡诺模型对需求类型分级，将影响业务连续性的关键优化需求优先落地，同时生成周期性的效能报告辅助资源调配决策，显著提升工程团队的交付效率。

针对变更发布的高风险场景，ITSM 与 SRE 的协同实现了精细化管控。发布流程设置多级审批关卡，技术负责人和安全团队的电子签批完成后，系统自动触发 CI/CD 流水线执行部署，规避人工操作风险。过程中严格遵循 ITIL 变更管理规范，要求每次发布必须包含明确的变更窗口预案和回滚方案，在通过集成测试的自动化验证后才准许投产，有效平衡了迭代速度与系统稳定性。

对于日常服务请求场景，自助化服务台大幅降低了人工介入成本。员工通过服务目录提交密码重置、资源申请等标准化请求时，后端流程引擎自动拆解任务并调用 API 执行操作，形成工单自动流转的处理闭环。该机制配合智能知识库的使用，使得超过 80%的常规服务需求可由用户自主完成，既释放了运维团队的生产力，又通过标准化路径减少了人为操作失误的可能。

2.3.2 标准运维

在云原生架构普及和业务连续性要求（例如金融行业 99.99%可用性）日益严苛的背景下，现代 IT 运维与站点可靠性工程（SRE）领域正面临巨大挑战：系统复杂度与变更频率激增，传统运维模式因其依赖人工操作和经验积累（如手动脚本巡检、应急处理）、工具碎片化（独立运行的 Zabbix 监控、CMDB 配置管理、Jenkins 发布平台等系统间数据格式与 API 互不联通）、跨系统协同困难（权限体系冲突、告警联动监控与资源配置扩容等任务触发逻辑不一致造成高度耦合）以及显著的人力成本压力（Gartner 数据显示运维团队 70%时间消耗在重复性任务上）而捉襟见肘，导致操作随意性大、风险高、效率低下。

为解决这些问题，标准运维（Standard Operations）作为一种创新性的系统工程化解决方案应运而生，其核心目标是通过规范化、自动化和可视化的手段重塑运维操作流程。它依托预定义的标准化流程模板，将原本分散的发布、变更、扩缩容等操作固化为统一、可复用的工作流，并借助流程引擎（或称调度引擎）驱动任务编排和跨系统的一键自动执行，

确保了操作的可靠性与一致性。本质上，标准运维实现了运维模式从经验依赖型的“人治”向流程保障型的“流程治”转变，显著降低了人为失误和操作偏差，提升了整体规范程度。

标准运维的技术支撑包括基于流程引擎的集中任务调度编排、通过服务总线（如 ESB 或 API Gateway）对接内部 API 整合离散系统、以及将各类系统功能封装为可复用插件的标准化体系。这不仅解决了工具碎片化（避免运维人员手动跨平台操作），还通过整合异构系统大幅降低了集成复杂度。此外，结合职能化工具的自助服务能力，使得非专业人员也能执行如业务发布等标准化操作，扩展了运维的适用范围。

在解决跨系统协同挑战方面，标准运维利用流程引擎配合事件驱动架构解耦告警联动、资源配置扩容等复杂逻辑，实现端到端发布流程从数天级缩短至分钟级的显著优化。面对高业务连续性要求（如全年故障时间≤52 分钟的目标），其融入的自动化熔断机制和预案自执行能力（如 Kubernetes 的自愈特性）提供了坚实保障。同时，通过将重复任务自动化（例如某银行通过自动化改造年节省 300 人/天工作量），结合低代码流程编排工具与合规性校验，大幅释放了人力资源，降低了成本压力。

标准运维的核心价值在于持续提升运维效率、规范性并降低风险与成本，例如通过自动化调度缩短平均修复时间（MTTR），利用 AI 进行故障预测、自动迁移并配合资源优化策略控制费用。展望未来，它将进一步驱动运维数字化转型，沿着标准化流程、数据沉淀到智能决策的路径演进，例如通过统一的算力调度底座管理混合云环境，利用 AIOps 平台训练随机森林模型提升故障根因分析准确性，为企业构建可复用框架，向综合算力调度和更深入的智能化（AIOps）等方向拓展。

在成熟的 SRE 平台工程实践中，标准运维的具体实现往往遵循以下经验：

2.3.2.1 流程编排引擎实现机制

在自动化运维领域，可视化流程编排引擎采用 BPMN 2.0 国际标准构建拖拽式设计界面。该引擎支持串行/并行等基础结构，具备分支决策与循环控制能力，通过子流程嵌套实现流程模板复用。在数据交互层实现全局变量与局部变量联动机制，基于 JSONPath 表达式进行动态参数解析传递。为满足异构系统对接需求，提供 SDK 开发套件供用户扩展自定义插件，例如开发 Kubernetes 任务调度插件或公有云 API 接入模块。

2.3.2.2 分布式任务调度体系

面对企业级任务调度场景，系统同时支持 CRON 表达式定时调度与 Webhook 事件驱动触发模式。任务执行过程配备多层容错策略：采用指数退避算法实现智能重试机制，设

定超时熔断阈值防止资源阻塞，并设置人工审批节点保障关键操作安全。通过植入全链路 TraceID 实现任务追踪，结合火焰图进行性能瓶颈分析，显著提升系统可观测性。

2.3.2.3 安全控制防护模型

针对运维操作风险，安全沙箱通过正则表达式匹配与语义分析双引擎，实时拦截 rm -rf、DROP DATABASE 等高危命令。构建基于 RBAC 的权限分级体系，实施动态细粒度访问控制策略，例如限制特定 IP 段执行敏感操作。该机制在保持操作灵活性的前提下，有效防范误操作与恶意攻击风险。

2.3.2.4 开放平台集成方案

系统通过 API 网关提供 OAuth2.0 标准化鉴权服务，自动生成 Swagger 接口文档降低集成门槛。在生态兼容性方面，预制资源管控、CMDB、持续集成、可观测等系统的官方插件，形成开箱即用的自动化生态链。这种开放式架构设计使平台能快速融入现有技术栈，满足企业渐进式自动化改造需求。

2.3.3 应用配置管理

在现代企业多云架构和混合部署环境中，统一的配置管理变得至关重要。云原生应用、传统进程服务以及跨不同云平台的运行，迫切需要一套标准化、统一的管理方案，以实现配置与代码分离、屏蔽底层差异、减少配置漂移导致的故障，并满足资源调度优化和安全合规的需求。

应用配置管理产品定位聚焦于提供企业级配置统一管理能力，强调标准化管理与配置代码分离的原则，避免配置散落在各处。同时，核心目标包括实现对容器环境和传统进程的混合纳管，以及对主流多云环境的兼容支持，目标是屏蔽不同云厂商的技术差异。在算力调度场景下，这种配置统一管理直接关系到资源优化效能。配置与代码分离允许动态调整诸如 CPU、内存限制等资源配额，减少因配置错误导致的调度失败，例如任务分配不均等问题。多云支持的关键在于提供一个抽象层，能够兼容像 AWS SSM、Azure App Configuration 这样的主流云服务，并通过 API 适配不同配置格式。这个抽象层在跨云资源调度中至关重要，它可以作为调度器接口，自动适配 AWS Fargate、GCP Compute Engine 等不同云平台的 API，确保配置的一致性并有效提升资源利用率。实现容器与传统进程混合纳管则依靠标准运维插件管理进程，并通过云原生技术如 CSI 临时卷、ConfigMap 和 Secret 实现容器配置的同步与热更新。这种能力在资源调度场景下尤为宝贵，它能无缝处

理异构资源，例如利用 Sidecar 模式动态注入配置，避免关键任务的中断。在资源隔离方面，常用如 cgroups 和命名空间等技术。实施标准化管理能显著减少配置漂移（据统计，20%~30%的故障源于配置不一致），可以通过模板引擎如 Helm Charts 实现自动化部署。当然，在混合环境中纳管不同操作系统类型（如 Windows 与 Linux 进程）可能带来兼容性挑战，通常建议通过构建插件生态来解决。在性能层面，多云 API 适配层可集成调度策略，例如优先选择低延迟的云区域加载配置，目标响应时间需控制在 100ms 以内。

产品在管理方式上提供文件和键值型配置的区分。文件配置支持文本、二进制等多种类型并可设置自定义权限，键值型配置则支持 String、JSON、YAML 等多种数据格式并具备类型校验功能。在版本和发布管理方面，提供了可视化版本树、变更内容差异（Diff）比对、多策略审批流程（或签/会签）、定时发布以及便捷的回滚操作。对于配置的消费管理，则涵盖了服务密钥轮转、细粒度的访问权限控制（可精确到单个文件或 Key）以及客户端拉取配置的统计分析功能。从算力调度视角审视这些特性，键值配置的灵活性和版本回滚控制对提升系统弹性与可靠性意义重大。版本树和 Diff 功能使得在调度系统（如 Kubernetes 的 Job 配置出错时）能够实时恢复，有效缩短平均故障恢复时间（MTTR）。密钥轮转和细粒度权限控制则在需要共享资源的集群调度环境里，为防止未授权访问导致资源滥用提供了安全保障，确保良好的隔离性。发布控制机制中的灰度发布若与调度器结合，能够实现更为智能的动态金丝雀发布策略：当调度器检测到资源负载过高时，可自动对灰度组进行扩容，避免系统过载，这在大数据或 AI 模型训练等资源密集型任务中尤为实用。支持多样化的数据格式，如 JSON/YAML 等，可以极大地简化配置在算力工作流或资源参数定义中的注入过程；同时敏感数据可结合 TLS 加密保证传输安全。版本历史记录则为调度系统提供了审计追踪能力，有助于分析由配置变更引发的节点故障等根因问题。客户端管理的统计功能，尤其是拉取质量数据，可作用于调度决策优化——例如，当探测到配置拉取延迟过高时，可以自动切换到备用配置源，保证资源分配的响应速度（如将 P99 延迟控制在 50ms 以下）。

应用场景层面，配置热更新能力是核心技术之一，它通过 Sidecar 或 SDK 的 Watch 机制实现配置的实时更新，配合前后置脚本对运行业务进行动态调整。在云原生支持方面，借助 CSI 临时卷挂载，可以突破原生 ConfigMap 的大小限制（如超过 1MB 的情况），并支持基于规则的按需配置过滤和拉取。灰度发布支持按应用标签进行分组管理，实现从灰度测试到全量发布的全流程控制。安全方面，提供对敏感数据的底层存储和传输加密保障。从算力调度视角看，热更新机制能显著减少资源浪费：调度器可通过实时 Watch 机制动态

更新诸如容器内存限制等资源配置，避免了任务重启的开销。热更新通过 Sidecar 代理也能有效减轻调度主进程负载。云原生能力的增强，特别是突破 ConfigMap 限制，对于大模型训练场景至关重要。CSI 卷能够挂载大型配置文件（如超过 10GB 的 AI 模型参数），优化 GPU 集群中的调度效率，降低配置加载延迟。灰度发布能力结合资源标签，使调度器能够对资源进行精细化分配控制，例如可以优先在特定区域（如 region=us-east）的实例上灰度部署新配置，先行测试资源密集型任务，验证通过后再全量部署，提升了整体的容错能力。在资源密集型共享集群的调度环境中，敏感配置的加密能有效防止配置泄露导致的计算资源被劫持，集成 KMS 管理调度密钥是保障敏感调度策略安全的常用手段。针对云原生优化，CSI 卷的按需加载机制可避免网络 I/O 成为瓶颈，甚至可以结合 CDN 进行加速。需要注意的是，灰度发布需仔细处理不同环境（如开发与生产）的兼容性问题，采用蓝绿部署进行配置验证是常见的应对策略。

配置管理产品的核心价值主要体现在显著降低风险、确保多环境一致性以及增强合规审计能力三个方面。通过严格的版本控制和完善的变更管理流程，能大幅减少因配置错误导致的系统故障。统一的配置模板和变量替换机制保证了从开发、测试到生产等不同运行环境的高度一致性。完整的变更记录和详尽版本历史则为满足安全合规要求和审计追踪提供了坚实基础。从算力调度视角分析，配置一致性和风险降低能力直接优化了调度系统的可靠性。在弹性调度环境中，统一配置模板能保障所有节点资源（包括竞价实例）使用相同的设置，据估计可减少 15%~20%由配置差异引发的任务故障率。合规性审计则能在调度日志中整合配置变更历史，便于生成符合规范（如 PCI DSS）的资源利用率报告。整体而言，强大的应用配置管理系统如同算力调度的触手，能够驱动自动化配置变更来修改调度策略。在典型的“云爆发”（cloud bursting）场景中，系统可动态加载多云配置，实现计算资源在云间的无缝扩展与收缩，以应对负载峰值。

2.4 算力调度基础设施

算力调度基础设施是 SRE 综合算力调度体系的核心模块。作为底层能力平台，其核心职能是将物理或虚拟化资源（如服务器、存储、网络设备）高效转化为可用的计算能力（算力），为上层应用提供按需、弹性的服务支撑。在数字化转型加速的背景下，算力已成为继土地、劳动力之后的新型关键生产要素。该基础设施通过标准化资源组织、算力转化和管理模式，有效解决了异构资源整合困难与算力需求弹性多变的核心挑战。据统计，

全球算力需求年增长率超过30%，而通过算力综合调度可显著提升资源利用率，并降低运营成本。

算力调度基础设施本质上是资源向算力转化的底层能力平台，其技术框架依托三大关键机制：一是集群管理，主要解决资源组织形式，通过抽象层屏蔽底层硬件（如CPU、GPU、内存）的异构性，构建逻辑统一的算力调度资源池。二是运行时管理，运行时是资源向算力转化的引擎，实现计算能力的动态生成与供给。三是应用治理，应用是算力调度的服务对象，是算力的消费侧打包形态，也是算力的基础管理模式，应用治理提供对算力打包形态的全生命周期管理、持续交付管理和提效管理。

SRE综合算力调度技术演进源于计算资源虚拟化与自动化管理的持续发展：传统数据中心在资源分配时面临资源碎片化、调度延迟等瓶颈问题。本方案则通过前述抽象层克服硬件差异，以容器为载体，并支持如Kubernetes等通用编排模型，实现资源的精细化管控与高效利用。综合算力调度在全球市场驱动力显著，尤其是AI大模型训练对海量弹性算力的需求激增（例如，ChatGPT的单次训练能耗约相当于10万个CPU的算力消耗）。算力调度基础设施能够动态调配资源，精准满足此类高强度的弹性算力需求。

2.4.1 集群管理

集群管理是综合算力调度基础设施层的核心组件，通过优化资源分配与调度策略，保障系统高效运行、实现弹性扩展（秒级响应）并支持异构兼容（跨云/混合环境）。集群管理位于SRE综合算力调度三层架构的中间层。三层架构作为软件系统设计的经典范式，其核心价值在于通过分层实现“高内聚、低耦合”的目标。该架构通常划分为表示层（UI层）、业务逻辑层和数据访问层。表示层专注于用户交互与数据展示；业务逻辑层封装核心业务规则与处理流程；数据访问层处理与数据库等持久化存储的交互。这种分层设计显著提升系统模块化程度，使开发人员能够聚焦单一层次的功能实现，并降低层间依赖，从而提升系统的可维护性与扩展性。业务逻辑层作为架构的核心枢纽，需高效响应表示层请求并协调数据访问层操作，其设计质量直接影响系统的灵活性。

在SRE综合算力调度体系下，经典三层架构的理念进一步演化为更细粒度的层次模型。其技术栈可划分为资源管理层、集群管理层和应用层。资源管理层承担整合IaaS基础设施（如OpenStack或裸金属资源池）的职责，提供计算、存储及网络资源的抽象化接口。集群管理层则基于Kubernetes等技术实现资源的标准化逻辑组织，提供节点生命周期管理、服务发现、跨节点调度等核心能力，并集成网络策略引擎保障安全通信。应用层则负责承

载微服务化业务组件，通过声明式 API（如 K8s YAML）实现自动化部署与 CI/CD 流水线集成。这种分层设计不仅继承了经典模式的解耦优势，更依托云原生相关技术强化了环境一致性及弹性伸缩能力，例如通过多云管理系统统一纳管虚拟机、裸金属等异构资源，将这些异构资源加入逻辑集群中统一管理和转化，最终向应用层交付标准化算力。

在集群扩展能力方面，系统需依据业务场景选择适配的模式。独立集群专为高性能敏感场景（如 GPU 密集型 AI 推理或高频交易）设计，通过物理资源独占实现强隔离，是该领域的通用基础能力。共享集群则利用 Namespace 与 QoS 策略实现资源池化，支持百万级微服务的混合部署及高效处理突发流量（例如应对 AIGC 推理峰值）。虚拟集群提供秒级创建/销毁的轻量化环境，非常适用于开发测试沙箱及初创业务试运行。联邦集群则突破了地域限制，通过跨域服务发现与统一调度技术（如 Karmada、Clusternet 等）实现异地容灾（RTO < 30s），典型应用于多地区分服游戏或混合云灾备场景。

2.4.1.1 独立集群

独立集群管理作为集群管理通用能力，为企业提供了一套统一、全面的解决方案，旨在解决多环境和异构 Kubernetes 集群的管理复杂性。其核心能力覆盖了集群生命周期管理的各个方面。

在当前混合云环节下，强大的公有云集群管理能够整合管理混合云环境下的 Kubernetes 集群，无缝支持国内外主流公有云平台，如腾讯云、华为云、AWS 和 Azure 等。更重要的是，它有效地屏蔽了不同云厂商在集群管理接口和操作上的差异，为用户提供了一个统一的控制面和操作能力，大大简化了多云环境的运维工作。

其次，对于自建集群的支持也是重要的集群管理能力。自建集群支持基于虚拟机、物理机甚至裸金属服务器来搭建高可用的 Kubernetes 集群。它不仅负责这类集群的初始部署，还承担后续的版本升级和日常维护任务，并且广泛兼容包括 Linux 和 Windows 在内的多种操作系统集群类型，也具备良好的扩展性以支持未来可能出现的其他操作系统。

在集群节点管理层面，需提供灵活的节点模板管理。用户可以通过自定义模板，定义节点在上架部署时需要执行的特定业务脚本或流程，以满足个性化的初始化配置需求。同时，还需要支持对不同节点上部署的组件参数进行差异化的精细配置，以适应不同节点或应用的独特要求。

在云原生模式下，集群具备节点弹性伸缩管理能力是重要的特性。该功能能够依据业务负载情况，自动对托管在各大公有云平台以及自建资源池中的工作节点进行横向扩展或收缩（扩缩容），动态调整集群计算资源规模，有效助力业务实现降本增效的目标。

对于已经存在的 Kubernetes 环境，需要提供了便捷的存量集群纳管功能。企业可以将已有的、非该系统初始创建的集群（存量集群）轻松纳入统一管理平台。纳管之后，用户既能查看这些集群上既有应用的状态和数据，也能通过平台能力对这些集群进行新应用的发布和管理变更操作。纳管过程需支持多种接入方式，例如既可以通过标准的 KubeConfig 文件进行对接，也可以直接利用云厂商账号权限进行集成。

在集群内部架构管理上，还应包含了对集群配置面的管理。这主要聚焦于控制面的核心组件（如 API Server、Scheduler、Controller Manager 等）的稳定性与配置优化。用户可以调整控制面组件的运行参数，管理集群的 Kubernetes 版本升级流程。此外，它还需要提供控制面稳定性相关的巡检能力，能够主动发现潜在问题并给出加固和优化的专业建议。

在网络层面，集群网络管理能力支撑着关键的网络模式设定。它可以自动化管理采用 Overlay（如 Calico，通常用于跨主机网络）或 Underlay（通常用于高性能、低延迟场景，如 SR-IOV）网络模式的集群网络。同时，它也负责自动化管理相关的云上网络资源，如 VPC（专有网络）和 CIDR（IP 地址段）的分配与配置，确保网络层面的连通性和隔离性。

集群组件管理功能负责管理部署在 Kubernetes 集群内部的关键基础设施组件，这些组件涵盖编排、调度、网络、数据同步、安全控制等多个核心领域。系统支持对这些内置组件进行全生命周期的管理，包括其部署安装、版本更新以及运行时状态的可观测性（监控、日志、追踪等）维护，保障集群内部生态系统的健康运行。

独立集群管理通用能力通过整合以上八个方面的关键功能，构建了一个强大的、统一的管理平台，使得企业能够高效、可靠地管理跨公有云、私有云以及异构环境的庞大 Kubernetes 集群基础设施。

2.4.1.2 共享集群

共享集群是现代云计算和算力调度的核心实现方式之一，也是集群管理的重要能力之一。共享集群专门用以解决传统独立集群模式中的资源孤岛问题。其技术根基在于容器化技术和云原生理念，以命名空间作为核心的资源隔离单元，支持多租户高效、安全地共享底层物理资源。这种架构通过标准化 API 与云平台深度集成，提供统一的资源管理和调度接口。管理工具（如 KubeSphere、OpenShift 等）动态调配计算资源，优化节点负载并显著降低运维负担。例如，在 Kubernetes 生态中，共享集群通过特定的资源配额控制组件，确保了各租户间的资源分配公平性与有效性。

这种架构的核心价值在于巧妙地平衡了资源集中化管理与应用独立化部署的矛盾。由统一平台负责底层集群的基础运维，而上层的各个业务项目则能够专注于自身应用（如

Pod）的部署与运行，真正实现了“一套硬件集群，支撑多种业务”的集约化目标。其适用场景广泛，涵盖了公有云服务商的多租户服务、企业内部的 DevOps 开发环境，乃至资源受限的边缘计算场景，并能顺畅支持从私有云向公有云的弹性扩展。

共享集群带来的优势环环相扣，形成一个高效的技术闭环。通常来说，在成熟的 SRE 平台工程实践中，共享集群具备如下优势：

首先，它显著提升了资源利用率并降低了运维负担。多个项目共享同一个资源池大大减少了资源的闲置与碎片化问题，相比传统独立集群，硬件利用率得到大幅度提升。集群管理员统一管控所有节点，结合智能的节点自动伸缩技术，自动化处理资源调度，极大地简化了运维操作并节省了运营成本。例如大型电商在节日大促期间，多个业务线可共享同一集群应对瞬时高峰流量，避免了临时部署多个集群的繁杂与浪费，显著节省了运维人力投入和能源消耗。

其次，共享集群提供了强健的隔离性和安全保障。它构建了多层次、多维度的隔离机制，包括容器层面的 CPU/内存资源限制、网络流量和磁盘 IO 的精确限速，以及基于命名空间的逻辑隔离。权限控制系统严格遵循最小权限原则，确保一个项目中的应用无法越界访问其他项目的资源。这种设计深度契合了云计算的安全规范，有效解决了困扰多租户环境的“嘈邻干扰”问题，尤其满足了金融、医疗等高合规性要求的场景。通过实施默认的网络隔离策略和镜像安全检查等手段，共享集群能够显著降低潜在的安全风险。

第三，共享集群赋予资源以秒级弹性伸缩的能力。它能根据应用的实时负载，在极短时间内动态调整工作负载实例（如 Pod）的数量，几乎无感地实现大规模实例的快速扩增或缩减。这种弹性依赖高效的调度算法和优化的容器启动速度，同时还能无缝支持高性能计算需求（如 GPU 资源）的灵活伸缩。无论是在视频直播遇到的突发流量洪峰，还是在 AI 模型训练任务中需要的算力瞬间提升，共享集群都能敏捷响应。这种机制不仅确保了应用的高性能体验，更能在业务低谷期自动回收闲置资源，带来显著的成本节约效果。

第四，它支持精细化的成本管理。共享集群采用按实际使用量付费的模式，用户只为容器工作负载运行期间消耗的计算资源付费，闲置期间不发生费用。这使得预算控制更为精确透明，尤其优化了初始投入成本。通过细粒度的计量机制，集群能精准追踪各个项目或租户的资源使用情况。这种模式对初创公司快速进行业务验证（如 A/B 测试）和科研团队按需使用算力资源特别有利，避免了前期大规模硬件采购的浪费。

第五，共享集群提供了开箱即用的服务集成体验。它深度集成了云原生的主流服务组件，包括网络服务自动配置、日志实时收集与分析、以及服务性能监控等关键运维能力。

用户部署应用程序时，无需额外配置即可自动启用这些功能。自动化工具进一步简化了服务维护流程，将应用从部署到上线运行的时间周期大大缩短。随着技术发展，智能化分析能力（如自动故障诊断）的引入也正不断提升用户的运维效率和体验。

共享集群的实际应用价值在多类场景中得到验证：云服务商能够利用共享集群高效服务众多客户，每个客户在独立的命名空间中运行其项目，既保障了安全隔离也实现了计费精准；在企业内部 DevOps 体系中，开发测试与生产环境共享同一集群资源，弹性扩展能力和开箱即用的运维支持加速了软件迭代与故障响应；在分布式广泛的边缘计算节点和工业物联网设备接入点，共享集群提供了统一的管理界面、快速的算力响应和增强的安全防护；对于初创企业和学术研究机构等资源敏感场景，这种模型显著降低了基础设施的门槛和运维成本，使资源效益最大化。此外，内置的预检规则能有效阻止用户的误配置或潜在风险操作，为整个共享环境的稳定性提供了重要保障。

2.4.1.3 虚拟集群

在容器化和云原生技术成为主流的背景下，Kubernetes 作为标准编排平台在多租户场景下面临资源争用和安全隔离的挑战。传统部署独立集群的方案成本高昂且资源利用率低下，共享集群因多业务共享集群控制面而存在负载瓶颈。而虚拟集群技术通过轻量级虚拟化有效解决了这一难题。其设计原理借鉴了操作系统级虚拟化（如容器和命名空间），并融合了先进的调度算法优化，能够显著减轻宿主集群 APIServer 的负载，缓解资源瓶颈。

虚拟集群（Virtual Cluster）是在宿主集群（Host Cluster）之上构建的轻量级、高隔离虚拟化 Kubernetes 环境。这项技术通过虚拟化 API Server、Controller Manager 等控制平面组件，赋予每个虚拟集群独立的 Kubernetes 控制面，同时共享底层计算、存储和网络资源。其核心依托 Kubernetes 的命名空间隔离与资源抽象能力，结合双向资源同步机制，有效实现了多租户管理和物理资源的高效利用。

虚拟集群的核心功能集围绕资源优化与高隔离性展开。首先，它能够完全模拟独立 Kubernetes 集群的功能。通过支持自定义 Operator、Calico 或 Flannel 网络策略、CSI 存储驱动以及细粒度的 RBAC 授权，每个虚拟集群都能提供完整的行为特性。该功能依赖 Kubernetes 的 CRDs 扩展机制，利用名称重写和资源代理技术，允许用户进行专属配置（如在开发环境部署私有 NFS CSI 存储类）而无需触碰宿主集群底层，有效避免了“噪音邻居”效应引发的资源争用，并支持集成 Istio 等插件化服务网格。这不仅显著降低了运维复杂性，也增强了跨云平台（如 AWS、华为云）的兼容性和集群的可移植性。其次，虚拟集群的控制面组件（API Server 等）采用容器化部署，支持根据集群实际规模进行资源的

动态伸缩，配置可从小规模测试环境的 1 核 2GB 弹性提升到大规模生产环境的 4 核 8GB。此功能基于容器运行时（如 Containerd）和 Kubernetes 的 HPA（Horizontal Pod Autoscaler）实现，通常选用 k3s 或 MicroK8s 等轻量级组件，相比虚拟机方案能减少 50% 以上的资源开销。在算力调度场景中，结合实时监控可触发自动扩缩容，高效应对负载变化，避免宿主集群资源浪费，在保障性能的同时优化 20-30%的物理资源占用，尤其适应电商大促等突发流量场景和资源受限的边缘设备部署。第三是其秒级创建与销毁能力。通过与容器管理平台集成，用户可以一键式完成集群的生命周期管理。该能力可以基于声明式 API 或 GitOps 工作流，用户通过 YAML 文件定义 Helm Charts 等集群模板，平台则通过 Operator 或 CI/CD 管道自动编排资源。这极大地扩展了 DevOps 自动化场景：例如，结合 CI 系统，可为每个代码分支按需创建独立的临时虚拟集群用于测试，任务完成后自动销毁回收资源。这种模式显著减少了“僵尸集群”问题和环境残留，提升了算力敏捷性（支持每秒创建多个集群），从而加速了开发测试的迭代周期，降低了测试环境成本，非常适合 A/B 测试等高动态需求。第四点是资源双向同步机制。该机制对高级资源（如 CRDs、ConfigMaps）和低级资源（如 Pods、Services）进行区分处理：高级资源仅在虚拟集群层面存在，不写入宿主集群；低级资源则通过名称前缀映射等技术同步到宿主集群的对应命名空间。这种设计充分利用了 Kubernetes 的 Admission Controllers 和 Webhooks 来实现同步策略和冲突检测，在大规模场景下可将宿主集群的 API 请求压力降低 30-50%，提升集群整体性能。在混合云部署中，虚拟集群仅同步必要资源至云服务商（如腾讯云），宿主集群专注于底层资源调度，既保障了租户间的高安全隔离性，也支撑了多租户资源视图的统一化管理（例如集成统一的监控平台）。

虚拟集群的应用场景广泛且收益显著。在多租户安全管控方面，它为共享集群环境提供了基于 RBAC 和 Network Policies 的逻辑隔离能力。例如，在金融行业可分别为支付、风控等核心业务线部署独立虚拟集群，各团队拥有专属控制面，结合宿主集群的 QoS 机制有效规避数据泄露和性能干扰。该方案相比部署多个物理集群能大幅降低隔离成本，并提升资源调度的公平性和安全性。在 CI/CD 优化领域，虚拟集群的快速启动销毁特性与 Argo CD 等工具链结合，完美实践了“环境即代码”（Infrastructure as Code）的理念。典型场景是为每次代码提交快速启动独立虚拟集群执行自动化集成测试，甚至并行运行上百条流水线；测试结束自动销毁集群确保资源零残留。它同样能应用于生产环境的蓝绿部署，实现快速版本切换和回滚。该方案将环境准备时间从分钟级降至秒级，达成 100%的空闲资源回收率，为持续交付提供了强大支撑。在边缘计算与资源受限环境，受益于其轻量化设计，虚

拟集群可运行在低配置的 IoT 边缘节点上。智能工厂可在边缘部署独立的设备监控和 AI 推理模块作为虚拟集群；在车联网中，甚至可以在车载计算机上实现“一车一集群”，通过双向同步与中心联动管理。这使得它能够在低内存设备上运行，提供故障隔离能力，在降低硬件投入成本的同时，有效提升了边缘场景的可靠性和可管理性。

总结而言，虚拟集群是一项创新的算力调度技术，通过轻量级虚拟化实现了 Kubernetes 环境的弹性扩展、安全隔离与高效管理。其核心价值在于显著优化资源利用，并通过多租户安全管控、CI/CD 流程深度优化以及边缘计算适用性等关键场景，大幅提升了运维效率和业务敏捷性。

2.4.1.4 联邦集群

联邦集群技术源于应对云原生环境下单 Kubernetes 集群的固有局限。随着企业应用规模的扩张，单集群面临 5000 节点的官方支持上限瓶颈，难以支撑超大规模应用需求；同时，混合云（如 AWS、Azure 与本地 IDC 组合）及边缘计算场景下异构集群的差异化配置管理、跨 VPC 网络隔离导致运维复杂度激增，地理分散的集群间服务调用延迟也显著影响分布式应用性能。

联邦集群技术以 Kubernetes API 扩展机制（如聚合层）为基础，通过 CRD（Custom Resource Definition）定义联邦资源模板，并依托开源生态项目（如 Clusternet、Karmada 等）构建统一控制平面，将多个独立集群虚拟化为逻辑单元。其意义在于能够支持中大型企业的高可用架构（如金融系统异地多活）、混合云资源弹性调度以及百万级边缘节点（如 IoT 设备集群）的轻量化纳管，从而突破单集群规模限制，提升资源利用率与容灾能力。

在功能实现层面，联邦集群首先通过非侵入式架构实现混合云异构集群的统一纳管。例如，利用 Clusternet CRD 注册集群元数据，支持 Push/Pull 双模同步：Push 模式由控制面主动下发配置至成员集群，Pull 模式则依赖成员集群的 Agent 拉取指令，动态适配跨云厂商（如 AWS、GCP）或本地数据中心的网络环境。其次，联邦层完全兼容原生 Kubernetes 资源及 Helm Chart，用户可直接提交标准 Kubernetes 资源（如 Deployment、StatefulSet）、自定义 CRD 或 Helm Chart，无需修改原有定义，并通过 Subscription 和 Description 定义策略驱动资源分发，避免应用 YAML 重写。差异化配置则通过 Globalization 和 Localization 来实现全局覆盖和本地覆盖，例如为不同集群覆盖镜像地址或环境变量，确保 AI 训练任务自动调度至高算力 GPU 集群。全局服务发现能力依赖联邦 DNS 或网关层（如 Istio、Apache APISIX）构建跨集群流量路由，例如电商服务在亚洲与北美集群部署时，基于服务发现策略优先调度互访服务至邻近集群，显著降低网络延迟。灵活的调度策略进一步支持

多维规则：例如按权重分配副本（如集群 A:B=1:2），结合集群资源水位监控实现成本优化调度。动态故障迁移能力则通过联邦监控集群健康状态，故障时触发备份数据同步与 Pod 再调度至备用集群，保障业务的连续性。

联邦集群的实际应用收益体现在一下几个方面：其一，容灾能力提升，跨集群冗余部署将单点故障影响范围缩小 90%以上，实现机房级 SLA 99.99%保障，边缘场景下支持百万级节点接入（如智慧城市传感器集群）；其二，资源弹性扩展，突破单集群节点限制，支持水平扩展至数万节点（如电商大促资源池），并通过联邦 HPA（Horizontal Pod Autoscaler）实时采集子集群资源使用数据，动态调整副本分布，避免固定资源配置导致的浪费；其三，灵活的算力资源成本控制，结合不同地域不同时间成本因素，按需分配工作负载至不同集群，利用混合云特性降低业务开销。

2.4.2 运行时管理

在 SRE 综合算力调度实践中，算力调度需将物理资源（CPU/GPU/存储/网络）高效转化为应用可用的标准化算力。运行时管理（Runtime Management）作为调度体系的核心引擎，通过容器化技术（CRI/CNI/CSI）实现资源抽象、隔离与动态供给，解决异构硬件适配、多租户隔离、弹性伸缩等挑战。

综合算力调度基础设施依赖云原生技术，其运行时管理方案在多个关键领域提供了强大的扩展能力，以满足不同的业务场景需求。

在运行时扩展方面，核心目标是将异构硬件转化为可统一调度的容器算力。通过遵循 CRI（容器运行时接口）标准，系统能将 CPU、GPU、NPU、内存等多种硬件资源抽象化为池化的算力单元，实现动态分配。这种架构天然兼容多种工作负载，无论是标准的 Kubernetes Pod、旨在增强安全性的轻量级安全容器（如 Kata 或 gVisor），还是借助 KubeVirt 运行的虚拟机化负载，都能通过统一接口进行资源调度。针对 AI 训练等场景，设备插件（Device Plugin）将 GPU 显存映射为容器可声明的资源，并借助显存虚拟化调度技术，使得多个分布式训练任务能够高效地共享单块显卡算力，显著降低训练集群成本。对于实时性要求高的场景，则可以通过 NUMA 亲和性绑核配合 CPU Manager 策略，减少跨核访问延迟，确保任务满足严格的实时性 SLA。

存储支持的优化关键在于提供高可用、高性能的数据访问。通过 CSI（容器存储接口）标准，可以动态地为容器挂载云盘、NAS、OSS 等多种云存储，支持跨节点的多节点读写（ReadWriteMany）模式，满足共享数据需求。CSI 还能感知存储卷的拓扑结构，确保 Pod

被调度到最优的存储节点上。为了提升数据访问速度，EmptyDir 临时卷可以实现容器间的高效、近零拷贝数据交换，而结合内存盘（tmpfs）则能大幅优化需要高频 IO 操作（如日志记录或缓存）的应用性能。数据库容器化场景下，利用 Local PV 直接挂载本地 SSD 存储能将访问延迟降低至微秒级，同时卷快照（VolumeSnapshot）功能基于增量备份机制，可提供秒级的数据恢复能力，有力保障业务连续性。在大数据处理场景，使用类似 Ceph 的分布式存储并采用纠删码（Erasure Coding）冗余策略，可以在保持数据可靠性的同时显著减少存储副本数量，提升存储效率。

强大的网络支持是算力调度的基石。根据网络部署环境的不同，可选取高性能方案：Overlay 网络（如 Calico 的 IPIP 或 VXLAN 模式）通过在 IP 包基础上封装来解决跨可用区或跨网段的通信隔离问题；而对于需要极致低延迟的场景（如高频金融交易），Underlay 网络方案（如 MACVLAN 或 SR-IOV）则能绕过虚拟化层直接通物理网卡，通常能将网络延迟降低 40%以上。精细化流量治理由 NetworkPolicy 实现，用于控制微服务之间的东西向流量访问权限，支持基于命名空间或 Pod 标签的细粒度策略，并可结合 eBPF 技术实现策略的动态下发更新。在 5G 边缘计算部署中，Multus CNI 插件允许多块网卡绑定，将关键的业务流量与常规的管理流量进行物理隔离，并通过 QoS 策略为关键业务预留必要带宽。

安全与隔离能力在算力调度能力中至关重要。在内核层面，Seccomp 用于限制容器内危险系统调用的范围，AppArmor 或 SELinux 则提供强制访问控制（MAC）机制，精细化控制文件系统访问路径，有效阻断容器逃逸攻击风险。在多种负载（如在线服务和离线批处理任务）混合部署（混部）的场景下，利用 cgroup v2 按权重（cpu.weight）分配 CPU 资源配额，同时内存 QoS 采用水位线分级回收策略，优先保障在线服务的稳定性。硬件隔离层面，通过静态绑核策略（CPU Manager）和 NUMA 亲和性调度，将任务固定到特定核心和 NUMA 节点运行，显著减少跨 NUMA 访问带来的延迟，提升内存密集型应用性能。针对金融等需要高级别审计的场景，eBPF 技术能实时监控容器内部的网络流量和行为，动态拦截诸如端口扫描或异常连接等可疑活动，并辅以证书自动轮替机制来保障服务间相互认证（mTLS）通信的长效安全。

2.4.2.1 运行时扩展

在现代云计算和边缘计算环境中，高效调度和利用多样化的硬件资源至关重要。运行时扩展技术通过创新的运行时管理机制，解决了这一核心问题。其核心思想是将各类底层资源——包括不同的容器运行时（如 Docker、Containerd）、虚拟化环境（如 KVM）、以及专用硬件加速设备（如 GPU/NPU）——统一抽象并转化为标准化的“算力”单元。这种转

化基于云原生平台的核心接口标准（如 Kubernetes 的 CRI、CNI、Device Plugin），显著提升了资源利用率，并为混合云部署、边缘 AI 推理等复杂场景提供了强大的支撑。

在容器编排生态中，运行时是容器创建和管理的底层引擎。传统的容器运行时如 Docker 因其完善的工具链常用于开发测试，而 Containerd 则因其轻量化和高效性成为生产环境的首选。对安全性要求更高的场景则可采用 Kata Containers 这类基于轻量级虚拟机的运行时，提供更强隔离以防御容器逃逸风险。运行时扩展技术的多容器运行时支持能力，允许在同一集群的 Linux 和 Windows 节点上无缝使用这些引擎。例如，Linux 节点可通过原生运行时（如 runc 或 Containerd）高效运行 Linux 容器，而 Windows 节点则利用特定运行时（如 Docker Engine for Windows）支持 Windows 容器。其架构关键在于 Kubelet 通过标准化的 CRI 接口与一个通用的运行时管理器交互，管理器再通过适配层（CRI Shim）将请求转换为具体运行时引擎的命令，如启动容器或处理生命周期事件。这种设计解耦了上层调度与底层实现，使得企业可以在混合云环境中根据应用需求灵活选择运行时。

对于需要超越容器隔离性的场景，运行时扩展技术提供了对虚拟化运行时的支持。它能够管理“容器化的”KVM，产生出比普通容器具备更强安全边界的虚拟机。这些虚拟机（支持 Linux 或 Windows Guest OS）可以通过扩展后的通用运行时管理器，经由 CRI 接口被 Kubernetes 像调度容器一样统一编排，并利用 CNI 实现网络连接。其技术基础是利用硬件辅助虚拟化技术（如 Intel VT-x/AMD-V）在容器内创建隔离的 Hypervisor 环境。相较于纯容器方案，这种容器化 KVM 虚拟机在 CPU 和内存资源隔离方面表现更优，能大幅减少资源争用和降低内存泄露风险。这使得该技术在多租户公有云环境中极具价值，能有效隔离不同租户的应用负载。

为了充分利用日益普及的硬件加速能力，运行时扩展技术深度集成了对 GPU 和 NPU 等专用设备的支持。其目标是将物理的 GPU 或 NPU 设备识别出来，并通过虚拟化技术（如 NVIDIA 的 vGPU 或 MIG 分片）转化为可被容器或虚拟机按需调度和共享的标准化算力单元。架构上，Kubelet 借助 Device Plugin 框架来发现、声明和管理节点上的加速设备资源，同时通过集成 RDMA 等高性能网络接口（CNI）满足分布式 AI 训练的通信需求。GPU 更擅长通用的并行计算，尤其是深度学习模型训练，而 NPU 则是为边缘端 AI 推理场景高度优化的专用硬件。相较于传统独占式的设备直通（Pass-through）模式，虚拟化技术允许将一块高性能物理卡分割成多个逻辑单元供给不同工作负载使用，极大提升了硬件利用率和成本效益。在大型 AI 训练集群中，多容器可以共享 GPU 资源运行 TensorFlow 或 PyTorch 模型，显著提升集群利用率。在工业制造现场，集成 NPU 的容器可加速实时图像

处理，将缺陷检测的延迟降至毫秒级别，提高生产效率。在智慧零售的边缘节点，轻量化的 NPU 算力单元能够本地化处理摄像头数据，实现快速分析并减少对云端带宽的依赖。这一能力的实现依赖成熟的工具链，如 NVIDIA GPU Operator 自动化管理驱动、监控与调度，或针对华为昇腾 NPU 的等效方案。通过设备虚拟化（如 vGPU 插件），物理 GPU 被灵活地映射到多个容器或虚拟机实例，带来了显著的硬件成本节约。

运行时扩展的实现有以下四种典型形态：Linux 容器形态基于 runc 或 Containerd 等轻量引擎，结合镜像缓存技术，非常适合快速迭代的微服务开发和自动化 CI/CD 流水线。Windows 容器形态借助 Docker 引擎和 CRI Shim 适配层，无缝支持企业遗留应用（如旧版 SQL Server）的容器化迁移，在混合操作系统环境中运行自如。容器化 KVM 形态创新性地将虚拟机视为特殊容器进行管理，通过 KVM-in-Container 技术和 CNI 网络，为金融安全沙箱等需要强隔离的环境提供了高效安全的解决方案。GPU/NPU 加速单元形态则通过精细化的设备虚拟化技术和高速网络（如 RDMA），成为 AI 模型训练和边缘端实时推理的理想载体。所有这些形态的核心在于统一的编排层（基于 CRI 等接口扩展），允许在一个平台上混合调度容器、虚拟机和加速硬件资源，同时兼顾了安全隔离和资源密度之间的平衡，并日益增强对国产芯片（如华为昇腾 NPU）和操作系统生态的兼容支持。

本文将以容器化 KVM 场景运行时扩展及 GPU 消费场景运行时扩展为例，阐述具体的扩展形态与实施方案。

基于 Kubernetes 生态的容器化 KVM 方案，为游戏等需 Windows 环境的领域提供了灵活的虚拟化解决方案。传统容器化技术在支持 Windows 应用时存在局限性，而通过 KubeVirt 扩展方案，可实现 Windows 虚拟机的高效容器化部署。该方案不仅适用于游戏产业，还可扩展至云游戏、企业应用及边缘计算等多种场景。

方案的核心设计在于整合 KubeVirt、Libvirt 与 QEMU-KVM 等技术。KubeVirt 作为 Kubernetes 的自定义资源定义，负责调用 Libvirt 接口以管理虚拟机的生命周期；Libvirt 则封装 QEMU-KVM，提供异步操作的守护进程支持；QEMU-KVM 利用 KVM 的硬件虚拟化能力，支持声卡、USB 等外设。整个架构涉及 GameVMIRS、virt-controller、virt-launcher 等多个组件协同工作，共同实现虚拟机在 Pod 中的运行与管理。

在技术层面，本方案重点优化了扩展性、性能与资源管理。具体包括：支持联邦批量发布以应对大规模部署；扩展网络方案以兼容云下多种模式；扩展存储方案以同时支持本地盘与云盘；通过 vGPU 实践优化图形性能以提升游戏体验；支持虚拟机与容器的混合部署以复用集群资源；并提供针对不同存储场景的冷热迁移能力。例如，本地盘结合 ceph 和

vGPU 可支持热迁移，而云盘及普通本地盘则支持冷迁移，从而增强系统的灵活性与可用性。

此方案适用场景广泛。除游戏领域外，在云游戏和边缘计算中结合 vGPU 可优化图形渲染；在企业环境中，混合部署与资源复用有助于降低基础设施成本。通过对网络、存储等功能的扩展，方案能良好适配私有云或混合云环境。总体而言，容器化 KVM 方案通过融合 Kubernetes 编排与虚拟化技术，为 Windows 应用提供了可扩展、高性能的运行时环境，未来在 AI 训练或实时渲染等领域亦具备进一步应用潜力。

承接上述潜力，在人工智能模型训练和推理场景中，GPU 算力转化被广泛用于高效调度和利用硬件资源。随着 AI 工作负载复杂化，GPU 运行时扩展技术成为云原生环境中管理算力的重要方案，其通过标准化组件和虚拟化手段实现 GPU 资源的灵活分配与隔离，并支持多样化芯片架构。基于容器编排平台如 Kubernetes，GPU 算力调度涉及多层组件协作，包括 API 服务层、调度器、节点代理、设备插件、容器运行时及扩展运行时机制，这些组件协同实现从资源注册、调度到容器内 GPU 设备的动态注入与隔离，确保任务高效执行。

关键组件方面，Device Plugin 实现单 GPU 分时执行多个任务，例如每 100 毫秒切换，但任务切换存在性能损耗且无显存隔离，适用于任务优先级调度；动态注册通过将物理 GPU 切分为独立 vGPU 实例，各实例隔离算力与显存，需硬件支持且划分后资源固定，属于 GPU 虚拟化方案，适合多租户环境；GPU Runtime 提供精确设备分配，如动态注入设备文件描述符并自动挂载同版本驱动库以确保容器内驱动兼容性，同时通过安全隔离机制（如基于 cgroup 的显存用量限制）防止资源溢出；用户态虚拟化通过劫持 CUDA 调用（如 vcuda 库）实现显存与算力隔离，支持按百分比请求资源且算力分割粒度灵活，属于软件级虚拟化，无需硬件依赖。异构芯片支持包括英伟达 GPU、华为昇腾 GPU 和寒武纪 MLU 等，通过相应插件和运行时实现资源转换与调度。

基于这些组件，GPU 运行时扩展能力主要围绕三个典型场景展开，以提升资源利用率和运维效率。GPU 池化扩展旨在将分散的 GPU 资源集中管理，形成可弹性伸缩的资源池，其设计核心是构建自动化管理的 GPU 节点池，并通过异构资源纳管机制统一识别不同厂商的 GPU 卡，适用于大型数据中心或混合云环境。GPU 虚拟化扩展通过对物理 GPU 进行细粒度划分与隔离，实现资源高效复用，聚焦于算力和显存的精细切分以及多层次安全隔离，尤其适合云平台等多租户场景。算力调度扩展则优化 GPU 资源分配策略，适应异构硬件和配额管理，依赖于调度器感知性能差异并进行标准化匹配，常见于异构集群或科研共享平台。

为实现这些扩展，多种 GPU 虚拟化技术被开发，旨在解决多容器环境中高效共享和复用 GPU 资源的问题。技术方案主要分为以下几类：内核级虚拟化方案（如腾讯云 qGPU）在驱动层进行资源调度，具备强隔离性；硬件辅助虚拟化方案（如 NVIDIA 安培 GPU）依托硬件能力提供接近原生性能；中介直通方案（如 NVIDIA vGPU）在虚拟机与物理 GPU 间协调请求，平衡隔离与性能；API 转发方案（如 vCUDA）通过软件层虚拟化，部署灵活；时分复用方案（如 NVIDIA MPS）共享 GPU 上下文，提高吞吐量；硬件分区方案（如 NVIDIA MIG）将 GPU 划分为独立实例，提供硬件级隔离。总体而言，这些方案在隔离性、性能、成本与灵活性间取得平衡，适用于不同场景。

作为实践案例，OPPO 设计了 sGPU（共享 GPU）运行时扩展，以优化 AI 训练、推理等 GPU 密集型业务的算力成本。该方案支持多租户资源隔离与动态共享，适用于大规模 GPU 集群管理。其核心设计采用分层架构：用户态集成 Kubernetes 平台和容器运行时，负责资源编排；内核态包含内核模块与驱动，实现资源切分与隔离。关键技术由四个组件支撑：内核模块处理 GPU 系统调用，容器运行时注入资源配置，Device Plugin 暴露虚拟化资源，Scheduler Extender 优化调度策略。

sGPU 的核心设计围绕完整调度流程：用户提交 Pod 后，Kubernetes 调度器初步过滤，GPU Share Extender 查询节点资源并决策，由 kube-apiserver 协调，Kubelet 执行创建。Device Plugin 上报资源状态，使 Pod 与 sGPU 实例绑定。关键机制包括 GPU 反亲和性、显存超分、细粒度分配和动态扩缩，带来高效利用和云原生集成优势。

该技术的应用场景可进一步扩展。在多租户云平台中，提供隔离的共享 GPU 资源；在边缘计算场景，支持物联网、自动驾驶等实时 AI 推理；在数据中心内统一调度混合工作负载，并服务于科学计算、图形渲染等领域，推动算力调度向智能化和普适化发展。

2.4.2.2 网络管理

网络管理主要提供容器网络解决方案。容器网络是实现容器间通讯和外部网络连接的关键技术，其核心目的是通过网络虚拟化及控制技术解决容器网络连通性、隔离性和性能问题，为综合算力调度提供网络基础设施。

容器网络方案主要分为两类，一类是指容器集群内部的通讯，即同一宿主机或不同宿主机上的容器、服务间交互的流量，通讯主体是容器、POD 以及宿主机。这类通讯在行业内通常被称为东西向网络通讯，东西向网络管理主要提供东西向网络通讯的技术方案。第二类是指容器集群与外部网络的通讯，即外部客户端访问容器服务，或容器访问外部资源（如公网 API）的通讯，通讯主体是外部用户或外网资源与容器之间。这类在行业内通常

被称为南北向网络通讯，南北向网络管理主要是提供这类通讯的技术方案。下文将对这两类网络管理实现方案做详细说明。

容器网络接口（CNI）是 Kubernetes 等容器编排平台中用于管理东西向网络通信的标准插件接口，它允许不同的网络解决方案无缝集成，提供灵活的网络配置。CNI 网络主要分为 Underlay 和 Overlay 两种类型，它们基于不同的架构原理，适用于多样化的部署场景。

Underlay 网络直接利用物理网络设备的能力，如路由协议或网卡虚拟化技术，使容器网络与宿主机网络处于同一平面，这意味着容器 IP 地址可以直接路由到物理网络，无需额外封装。这种方式的优势在于提供高性能、稳定性好以及物理隔离，非常适合延迟敏感的应用，如数据库或高频交易系统。然而，它的缺点是对底层网络支持有较高依赖，需要网络设备配置协作，部署复杂度较高，因此在没有可控物理网络的环境中使用受限。

Overlay 网络则在现有物理网络之上创建一个虚拟层，通过封装技术如 VXLAN 或 IPIP 将容器流量隧道化，使容器网络独立于底层网络。这种方式灵活性高，易于在跨子网或多云环境中部署，简化了管理，但会引入性能开销，如增加 CPU 负载和网络延迟，可能影响吞吐量。Overlay 网络适用于中小型集群、开发测试环境或需要快速原型部署的场景。

在常见的 Overlay CNI 方案中，Flannel VXLAN 使用 VXLAN 封装技术，适合中小集群和混合云环境；Calico IPIP 基于 IP-in-IP 封装，结合 BGP 路由，适用于需要网络策略的中型集群；Weave Net 采用自定义封装协议，支持动态环境和边缘计算。另一方面，常见的 Underlay CNI 方案包括 Calico BGP，它利用 BGP 路由协议实现高性能，适合大规模容器部署；MACVLAN 通过网卡虚拟化提供低延迟，适用于数据库或高频交易；IPVLAN 共享 MAC 地址，优化资源使用，适合高密度容器环境；Flannel host-gw 使用主机网关路由，要求节点间二层网络互通，适用于同子网的中小集群。

总体而言，选择 Underlay 或 Overlay 网络取决于具体需求：如果追求极致性能、有可控物理网络，或运行延迟敏感应用，Underlay 是优选；如果需要在灵活环境中快速部署或集群规模较小，Overlay 更合适。在实际应用中，企业可能结合两者，以平衡性能和灵活性，在一些更高要求的环境 CNI 技术还可以与服务网格和 eBPF 等集成，实现更智能的网络管理。

容器网络接口（CNI）是实现 Pod 间东西向通信的核心组件，其选型、部署与运维对网络性能、安全性与稳定性有直接影响，需要站点可靠性工程师（SRE）进行全生命周期管理。从 SRE 视角，该工作涵盖四个关键环节。首先是 CNI 评估选型，核心工作是评估业务对网络性能、安全性及多租户隔离的需求，验证插件与 Kubernetes 版本、操作系统及底

层网络的兼容性，并权衡不同架构（如 Overlay 与 Underlay）的资源开销与运维复杂度；关键注意事项包括避免选择社区不活跃的插件，以及优先支持标准 API 以保证可移植性。其次是 CNI 生产部署，核心工作包括使用基础设施即代码工具实现标准化部署，采用金丝雀发布等渐进式策略，并遵循最小权限原则进行安全加固；部署后需立即验证网络基础功能，并确保所有配置变更被详细记录和版本管理。然后是 CNI 日常运维，致力于通过可观测性（指标、日志、链路追踪）和自动化来保障稳定运行，核心工作包括建立监控体系、自动化网络策略与资源管理，以及基于服务等级目标（SLO）和错误预算进行数据驱动的运维决策；关键点在于定期巡检网络配置与资源，并构建易用的运维平台赋能开发团队。最后是 CNI 故障处理，聚焦于快速响应与系统性改进，核心工作包括制定常见故障的应急预案、利用可观测性数据协同排查定位根因，以及组织复盘以落实改进措施；必须遵循“止损第一”原则优先恢复业务，并通过混沌工程主动测试系统容错能力。

南北向网络管理通常通过端口映射、负载均衡器、网关代理等方式，实现容器内服务的对外暴露。其中，端口映射主要通过 iptables 配置，完成宿主机端口与容器端口间的消息转发绑定；负载均衡器通过 Ingress 或外部 LB（如公有云厂商提供的云负载均衡器）劫持流量，并转发至容器内部服务；网关代理则借助 Istio Gateway 等流量代理方案，通过管理 TLS 配置及七层路由策略实现服务暴露。南北向流量治理对保障服务高可用性、安全性及可扩展性至关重要，尤其在多云与混合云环境中表现突出，以下将详细解析常见南北向流量治理技术实践的核心要点。

CLB 直通 Pod 的核心技术要点是通过配置 Ingress 规则，实现云负载均衡器（CLB）与 Kubernetes Pod 的直接动态绑定，同时 Ingress 规则也支持直接绑定 Service 的 NodePort。从技术背景来看，CLB（Cloud Load Balancer）是公有云厂商提供的负载均衡服务，核心作用是将外部流量分发至后端实例；在 Kubernetes 生态中，Ingress 作为管理外部访问的 API 对象，通常由 Ingress 控制器负责落地实现，而 CLB 直通 Pod 即绕过传统 Service 层，直接将外部流量路由至 Pod，可有效降低网络延迟与架构复杂度。该方案在微服务架构中应用广泛，例如实现金丝雀发布或 A/B 测试时，可直接将流量引导至特定版本的 Pod，无需经过 Service 代理，具备响应速度更快、流量控制更精细的优势，不过该方案可能需要依赖云厂商的特定配置，且在高动态集群环境中，Pod IP 的动态变化可能需要系统自动更新绑定关系，实际应用中，可结合 Kubernetes Ingress-NGINX、云原生 Ingress 控制器等工具，实现 Pod 的自动发现与绑定，例如腾讯云 TKE 中，可通过注解（annotations）配置 CLB 直连 Pod。

流量灰度/跨域绑定支持通过动态调整流量权重的方式，实现应用流量的灰度分发，同时也支持跨域绑定，例如南京、上海地域的 CLB 可直接绑定至广州的 Kubernetes 集群。流量灰度（又称金丝雀发布）是一种渐进式应用部署策略，通过逐步将流量切换至新版本服务，降低版本迭代的风险；跨域绑定则涉及多地理区域的负载均衡器与集群交互，是全球化部署中实现异地多活、灾难恢复的关键技术。以电商应用为例，发布新功能时可先将10%的流量路由至新版本 Pod，待监控确认性能稳定后，再逐步提升流量权重，最大限度降低故障影响，而跨域绑定可让用户从就近地域接入服务，显著提升访问体验，同时异地备份集群可保障服务高可用，适用于多云部署场景，可实现不同云厂商 CLB 的流量统一管理。

随机端口映射（HostPort）提供随机 HostPort 分配能力，使 Pod 在无云网络（自建机房）场景下，可实现跨集群互访。HostPort 是 Kubernetes 的一种网络模式，允许 Pod 直接绑定到节点的特定端口，实现外部直接访问；随机端口映射可有效避免端口冲突，尤其适用于自建机房、混合云等云网络服务不可用的场景。该方案适用于企业自建 Kubernetes 集群、边缘计算等场景，例如 IoT 设备可通过随机端口直接访问 Pod，无需依赖云负载均衡器，大幅简化网络配置，不过需重点关注安全风险，端口暴露后需配合防火墙规则，防范未授权访问。

StatefulSet 端口段支持为 StatefulSet 管理的有状态应用，提供端口段绑定能力。StatefulSet 是 Kubernetes 中专门用于管理有状态应用（如数据库、分布式中间件）的工作负载资源，核心优势是为 Pod 提供稳定的网络标识与持久化存储，而端口段绑定允许为每个 Pod 实例分配连续的端口范围，而非单个端口，可满足需要多端口运行的应用需求（如数据库复制、服务监控等）。以 MySQL StatefulSet 为例，每个 Pod 需开放 3306 端口用于数据库连接，同时开放 9104 端口用于 Prometheus 监控，端口段支持可简化此类多端口配置，确保端口资源有序分配，该方案广泛应用于 Kafka、Elasticsearch 等分布式系统，此类系统的每个节点通常需要多个服务端口协同工作。

PortPool 端口池支持通过端口池机制，预先创建一批端口资源，与应用 Pod 实例的端口段实现一一映射。端口池是一种高效的端口资源管理机制，通过预先分配一组端口供应用使用，类似 IP 地址池的管理模式，在 Kubernetes 中，可通过自定义资源定义（CRD）或 Operator 模式实现端口池管理，有效避免端口碎片与端口冲突问题。该方案适用于大规模部署场景，例如游戏服务器、实时通信应用等，此类场景中每个 Pod 可能需要多个动态端

口；端口池可实现端口的快速分配与回收，大幅提升资源利用率，此外，在 CI/CD 管道中，临时 Pod 需占用临时端口进行测试，端口池也能提供高效的端口支撑。

多云能力可兼容腾讯云、华为云、阿里云、GCP、AWS、Azure 等主流公有云平台，实现跨云环境的统一流量治理。多云策略是企业数字化转型中的常见选择，核心目的是规避单一供应商锁定风险、提升服务冗余能力、优化成本投入；南北向流量治理需适配不同公有云的负载均衡器、网络服务，实现跨云环境的一致性管理。例如，企业可将生产环境部署在 AWS，备份环境部署在 Azure，通过统一的流量入口管理跨云流量，既满足灾难恢复需求，也能适配数据本地化等合规要求，该能力适用于全球业务场景，可根据用户地域将流量路由至对应云平台，提升访问体验。

在云原生技术广泛应用于高性能计算与人工智能领域的背景下，传统网络已成为算力提升的关键瓶颈。为充分发挥 GPU 等异构硬件的性能，容器环境需具备高速网络设备调度能力，而 Kubernetes 的 DevicePlugin 机制提供了标准接口，可使特殊网络设备像 GPU 一样被集群纳管，并挂载至业务容器中。当前主流的核心网络方案主要分为三类：第一类是高速私有网络，以 InfiniBand 为代表，作为专为高性能计算设计的互连技术，InfiniBand 具备极高带宽、超低延迟及基于硬件的可靠传输能力，通过专用交换机与主机通道适配器构建，可为 MPI 等并行计算框架提供支撑，主要应用于对网络性能有极致要求的线下 AI 训练集群、科学计算等私有化环境；第二类是高速公有网络，典型代表是基于 RDMA 技术的云服务，RDMA 允许计算机通过网络适配器直接访问另一台计算机的内存，无需操作系统内核介入，彻底消除 CPU 开销与内存拷贝延迟，使云端 HPC 和 AI 训练能获得接近本地的性能，适用于云上大规模机器学习训练、分布式存储等场景；第三类是 GPU 高速网络，以 NVIDIA NVLink 技术为代表，该技术实现了 GPU 间高带宽、低延迟的直接点对点互连，突破了 PCIe 总线的性能限制，结合 NVSwitch 可构建大规模无阻塞的 GPU 互连拓扑，是万亿参数大模型并行训练的关键基础设施。

这些高速网络设备通过 DevicePlugin 机制集成到 Kubernetes 集群的具体流程如下：首先，设备的驱动或管理组件需实现对应的 DevicePlugin，并向 Kubelet 注册设备类型、数量及健康状态；随后，用户在 Pod 资源申请中声明所需的高速网络设备资源，集群调度器据此将 Pod 调度至满足条件的节点；接着，Kubelet 在创建容器时调用 DevicePlugin 的 Allocate 接口，获取设备挂载信息并完成容器内配置；最终，容器内的应用程序可直接使用挂载的高速网络设备，获得极致网络性能，无需感知底层复杂的管理细节。通过这一系

列方案与机制，容器化应用得以无缝利用高性能网络硬件，为计算密集型任务构建了高速数据通道。

在面向高性能计算与人工智能的应用场景中，容器化工作负载对网络性能有着极为苛刻的要求，传统容器网络方案往往难以充分调度和利用 InfiniBand 等专用硬件，进而形成资源瓶颈，因此，设计定制化的容器运行时网络方案，实现极致的传输性能与资源效率，显得至关重要。该方案的核心设计始于硬件资源上报机制：通过定制 DevicePlugin 组件，将 InfiniBand 设备作为可调度资源上报给 Kubernetes 集群，使系统能够感知并抽象化管理物理 IB 网卡。在业务部署阶段，Pod 通过资源声明请求特定的 IB 设备，调度器据此将其绑定到对应节点，并在容器启动后，直接将 IB 设备文件挂载入容器内部，使应用能够绕过虚拟化层、直读写硬件，从而显著降低性能开销。

为最大化资源利用率，同一物理节点上的多个 Pod 可共享所有可用的 IB 网卡带宽，依托 cgroups 等隔离技术实现带宽多路复用与精准资源分配，避免资源争用。在网络架构设计上，每个 Pod 通常配备多张 IB 网卡和一张以太网卡，形成分层网络架构：多张 IB 网卡通过 bond0 接口实现链路聚合，并连接到专用 IB 交换机阵列，专门承载高性能数据平面流量；以太网卡则接入管理网络，负责传输控制流量。Pod 内部通过 veth 设备对连接到主机的 bond0 聚合接口，实现网络命名空间的隔离与流量的高效转发。这一系列设计共同聚焦于性能优化与高可用性提升：容器对 IB 网卡的直接访问，避免了传统虚拟网络栈的性能开销，结合 RDMA 协议与内核参数调优，可实现接近线速的传输性能；同时，bond0 接口与多网卡配置提供了链路级的负载均衡与故障切换能力，确保网络服务的稳定性。该方案的设计思路可进一步扩展至 GPU、FPGA 等其他专用硬件的统一调度与管理，适用于对算力与网络有极端要求的机器学习训练、科学模拟等场景，并为混合云或多集群环境下的跨云高性能容器网络，提供了可行的基础架构范式。总而言之，该方案通过将专用网络设备深度集成至容器生态，实现了硬件资源的抽象、直通与共享，为高性能工作负载提供了坚实且高效的网络支撑。

在云计算和容器化技术快速发展的背景下，高性能网络成为支撑 AI 训练、大数据分析和科学计算等应用的关键需求。传统网络方案在延迟和吞吐量上存在明显瓶颈，而 RDMA（远程直接内存访问）技术通过绕过操作系统内核、实现直接内存访问，可显著提升网络性能。公有云环境中的容器运行时，需具备灵活的网络方案来整合 RDMA 等高速网络技术，从而优化算力调度效率与资源利用率，针对不同部署环境，目前主要有两种主流的高性能公有云容器运行时网络方案。

第一种是公有云集群对接云厂商 RDMA 方案，该方案专为公有云环境设计，直接利用云厂商提供的 RDMA 基础设施，通过混合云统一架构将 RDMA 网络集成到容器平台中，为业务提供一致的网络性能体验。其核心技术要点包括：集群节点通过 VPC 网络连接公有云标准网络服务，同时通过 RDMA 网络访问高性能硬件资源；Pod 可配置多张网卡，例如 eth0 用于常规 VPC 通信，rdma1、rdma2 等专用网卡用于 RDMA 高速数据传输；云厂商 RDMA 服务通常基于硬件虚拟化或 SR-IOV 技术，可实现低延迟与带宽保障，并通过统一管理平面支持混合云场景的部署与管理。

第二种是通用 RDMA 方案（使用 Multus CNI），该方案适用于通用或私有云环境，采用 Multus CNI（容器网络接口）实现 Pod 多网卡支持，用户可根据需求为 Pod 附加物理 RDMA 网卡，通过 host-device 插件直接管理硬件资源。其核心技术要点包括：Multus CNI 允许单个 Pod 拥有多张网卡，例如 eth0 连接标准网络用于管理通信，rdma1 和 rdma2 连接 RDMA 网络用于高性能数据传输，实现网络隔离与性能优化；host-device 插件将主机上的 RDMA 物理网卡直接映射到 Pod 中，大幅减少虚拟化开销，确保网络性能接近原生硬件水平；该方案支持灵活的网络配置，可结合 VPC 网络传输管理流量，RDMA 网络专门承载数据面的高性能通信。

这些方案可广泛扩展到多种高性能计算场景：在混合云环境中，跨公有云和私有云的部署可通过统一网络架构，实现 RDMA 资源的无缝集成，支持集群弹性伸缩与流量负载均衡；在机器学习训练、实时流处理、高频交易等对网络延迟敏感的高性能应用中，可通过 RDMA 网络大幅加速数据传输，提升业务处理效率；在资源调度优化方面，结合算力调度系统，可根据网络性能动态分配容器任务，提升集群整体运行效率，降低业务延迟。总之，高性能公有云容器运行时网络方案通过整合 RDMA 技术与多网卡支持，有效解决了容器化环境中的网络性能瓶颈。方案设计兼顾云厂商集成与通用部署场景，技术要点聚焦于网卡配置、网络隔离与硬件管理，可广泛应用于需要低延迟、高吞吐量的算力密集型场景，为公有云环境下的高性能容器应用提供了可靠的网络支撑。

2.4.2.3 存储管理

存储管理是综合算力调度方案中支撑应用的核心基础能力，其主要实现方案是通过抽象化机制完成数据持久化与存储算力动态供应。

在 Kubernetes 中，存储管理是容器化应用数据持久化和可靠性的核心组成部分，它通过一系列概念来确保数据在容器生命周期中的有效处理。首先，Volume（卷）作为存储的基本单元，允许容器在 Pod 内部或跨容器共享数据。Volume 分为临时存储和持久化存储两

种类型：临时存储的生命周期与 Pod 绑定，例如 emptyDir 用于 Pod 内容器之间的临时数据共享（如缓存文件），而 hostPath 则挂载宿主机目录，主要用于测试和开发场景，但由于依赖特定节点，不适合生产环境；持久化存储则通过 PersistentVolume（PV）和 PersistentVolumeClaim（PVC）机制实现，确保数据在 Pod 重启或迁移时得以保留，适用于有状态应用如数据库或文件存储。

PersistentVolume（PV）是集群级别的存储资源，由集群管理员预先配置或动态供应，它抽象了物理存储（如云存储卷或网络文件系统），并定义了关键属性，包括容量（例如 10GiB）、访问模式（如 ReadWriteOnce 允许单个节点读写，ReadOnlyMany 允许多个节点只读，以及 ReadWriteMany 支持多节点读写）和回收策略（如 Retain 保留数据需手动清理，或 Delete 自动删除存储资源）。PV 的创建方式包括静态创建（管理员手动设置）和动态供应（通过 StorageClass 自动生成），这使得存储资源能够灵活适配不同应用需求，提升集群的弹性和效率。

PersistentVolumeClaim（PVC）则是用户或应用对存储资源的请求，它声明所需的存储特性，如容量（例如 5GiB）和访问模式。Kubernetes 系统会根据 PVC 的请求自动匹配可用的 PV 或触发动态供应来创建新 PV，从而实现存储资源的自动分配和管理。这一机制抽象了底层存储细节，允许开发者专注于应用逻辑，而无需关心存储基础设施的具体实现，大大简化了运维工作。这些概念协同工作，为 Kubernetes 应用提供了可靠、灵活的存储解决方案，支撑从简单 Web 服务到复杂分布式系统的各种场景，确保数据的高可用性和一致性。在实际应用中，理解这些核心概念有助于优化存储策略，例如优先使用动态供应以简化管理，并根据业务需求选择合适的访问模式和回收策略来避免数据风险。

在 Kubernetes 环境中，存储方案的选择和配置方式是确保应用可靠性和性能的关键环节。存储管理主要涉及两种配置方式：静态配置和动态配置。静态配置是指管理员手动创建持久卷（PV），然后用户通过持久卷声明（PVC）来绑定到这些预先存在的 PV 上。这种方式适用于资源预分配的场景，但缺乏灵活性，需要人工干预来管理存储资源。相比之下，动态配置则更加自动化：用户只需创建 PVC，系统便会通过 StorageClass 资源触发容器存储接口（CSI）插件，自动创建 PV 并完成绑定过程。这大大简化了存储供应，适合云原生应用和快速变化的部署需求。

在存储类型方面，有多种方案可供选型，每种都有其特定的应用场景和特点。本地存储，例如使用 hostPath 方案，通常用于单节点测试或开发环境，因为它能提供较高的性能，但由于数据存储在节点本地，它不具备跨节点能力，且缺乏隔离性，可能导致数据冲突，

因此不适合生产环境。网络存储，如 NFS 或 CephFS，支持多节点读写，适用于共享文件系统场景，比如团队协作工具；但 NFS 存在单点故障风险，而 CephFS 通过分布式设计提供了更好的可用性。

对于云环境中的数据库应用，如 MySQL 或 PostgreSQL，块存储方案如 AWS EBS 或 GCP PD 是理想选择，它们支持动态创建、快照备份，并具有低延迟特性，但需要注意将存储与计算资源放置在相同可用区以最小化网络延迟。分布式存储，包括 Ceph RBD 或 GlusterFS，适用于私有化集群中需要高可用性的场景，例如金融或医疗系统，它们通过冗余备份确保数据可靠性，但部署和维护相对复杂，需要专业运维知识。

对象存储也是常用的选择，如 AWS S3 或腾讯云 COS，专用于处理非结构化数据，如图片、视频或日志文件。这类存储通常需要通过 Sidecar 容器或 CSI 驱动接入 Kubernetes，并非原生挂载，而是基于 API 访问，这使得它在经济性和可扩展性方面表现优异，但集成过程可能需要额外配置。总体而言，存储选型应基于具体应用需求、环境约束和运维成本来综合考虑，以确保最佳的性能和可靠性。

CSI（Container Storage Interface）是 Kubernetes 提供的容器存储管理标准化接口，其意义在于通过解耦存储系统与编排平台，实现存储资源的动态管理，进而推动算力的灵活调度。

容器存储接口（CSI）的核心架构设计采用了分层模块化的方式，以支持容器编排系统如 Kubernetes 与存储提供商之间的高效交互。首先，CSI 的分层设计包括三个关键插件：Node Plugin 运行于工作节点上，负责处理卷的挂载、卸载以及设备格式化等直接与存储设备交互的操作；Controller Plugin 则专注于管理卷的生命周期，包括创建、删除、扩容、快照和克隆等控制面功能；Identity Plugin 提供插件的元信息，如名称和版本，并执行健康检查（Probe），用于向 Kubernetes 注册驱动并确保其可用性。这些插件共同协作，实现了存储资源的动态管理和自动化。

在系统集成方面，CSI Driver 通过 Kubernetes 原生资源进行部署，其中 Node Plugin 以 DaemonSet 形式运行于每个工作节点，确保所有节点都能处理存储操作，而 Controller Plugin 则以 Deployment 形式部署在控制平面节点上，处理集群范围的存储管理。此外，Sidecar 容器 plays a crucial role：external-provisioner 监听 PersistentVolumeClaim（PVC）事件，自动触发卷的创建过程；external-attacher 则管理卷的挂接和分离操作，协调存储设备与节点之间的连接。这种集成方式简化了存储驱动的部署和维护，提升了云原生环境中的

可靠性和弹性，适用于各种算力调度场景，如大规模数据处理或 AI 工作负载。整体而言，CSI 架构通过标准化接口和模块化设计，为容器化应用提供了灵活、高效的存储解决方案。

CSI 相关功能扩展与实践应用于多种场景，在提升存储管理的灵活性、性能、安全性和实际应用能力方面有较多实践方案。在卷管理扩展方面，CSI 支持动态扩容功能，允许在线调整 Persistent Volume Claim（PVC）的容量，但这需要 StorageClass 启用 allowVolumeExpansion 参数，从而确保应用在数据增长时无需中断即可扩展存储资源。此外，通过 VolumeSnapshot API，CSI 实现了数据备份和快速克隆，这对于数据库迁移等场景尤为重要，因为它能维护数据一致性并缩短操作时间。同时，CSI 还提供原始块设备支持，使 Pod 能够直接以块设备形式访问存储，绕过文件系统层，适用于像 Oracle RAC 这样的高性能应用，以减少延迟并提升吞吐量。

在性能与可靠性增强方面，CSI 引入了拓扑感知调度机制，确保 Pod 与存储卷位于同一可用区或区域，从而显著降低网络延迟和故障风险，这对于分布式云环境中的应用性能优化至关重要。另一方面，多卷挂载功能支持单个 Pod 同时挂载多个存储卷，例如实现日志文件与业务数据的分离，这不仅提高了资源利用率，还增强了应用的可维护性和扩展性。

安全扩展是 CSI 的另一个重要维度，它包括传输加密和存储级加密。传输加密通过 TLS（Transport Layer Security）协议保护 CSI 驱动与存储系统之间的通信，防止数据在传输过程中被窃听或篡改。存储级加密则集成 Key Management System（KMS），如 AWS KMS 或 HashiCorp Vault，实现对存储数据的静态加密，确保数据在休息时的安全，满足合规性要求较高的行业标准。

在实际扩展实践方面，CSI 支持多租户隔离，通过 Kubernetes 的 RBAC（Role-Based Access Control）机制限制不同命名空间对 StorageClass 的访问，并结合存储系统配额实现资源隔离，这适用于云平台或多团队共享环境，以确保公平性和安全性。对于灾备与迁移，CSI 可以与工具如 Rook（用于 Ceph 存储）或 Velero 集成，实现卷数据的备份和跨集群恢复，支持无缝的数据迁移和灾难恢复流程。边缘存储场景中，CSI 与 Rook EdgeFS 等解决方案结合，允许边缘节点缓存热数据并异步同步到中心集群，确保在网络中断时本地读写操作仍能进行，这特别适用于 IoT 或零售行业的分布式应用，以提升响应速度和可靠性。总体而言，这些功能扩展使 CSI 成为一个全面而强大的存储管理工具，能够适应各种云原生环境的需求，推动数字化转型中的存储创新。

在云原生架构下，容器存储接口（CSI）是 Kubernetes 编排持久化存储的核心标准，其稳定性直接关系到数据库、AI 训练等数据密集型应用的业务连续性。因此，站点可靠性

工程（SRE）团队必须将 CSI 的全生命周期管理纳入稳定性保障体系。这项工作始于严谨的驱动选型。SRE 需要深入分析业务对性能、数据功能（如快照、克隆）、扩展性及安全性的具体需求，评估驱动与 Kubernetes 版本、操作系统及底层存储设施的兼容性，并审视其社区活跃度与架构资源开销。应优先选择经过广泛验证、由主流厂商维护的驱动，避免引入因社区不活跃或测试不足导致的长期运维风险。

选型确定后，标准化与自动化的部署配置是确保环境一致与安全的基础。SRE 应利用 Helm、Operator 等工具实现一键部署，并严格遵循最小权限原则配置 RBAC。同时，需通过定义清晰的 StorageClass，将不同性能与成本的存储供给、卷回收策略模板化，以满足多样化的应用需求。部署后需立即验证核心功能，并完善配置文档。

进入日常运维阶段，构建可观测性体系至关重要。SRE 需监控 CSI 组件状态、卷操作延迟、存储使用率等关键指标，并设置有效告警。在此基础上，推动卷扩容、快照等高频操作的自动化，并持续进行容量规划与性能分析，以主动预防容量耗尽与性能瓶颈。此外，定期审计配置、清理闲置资源及规划驱动升级也是日常工作的一部分。

当面临故障时，高效规范的处理机制是快速恢复服务的关键。SRE 需要预先制定针对挂载失败、性能劣化等场景的应急预案。故障发生时，应作为协调中心，综合利用事件、日志与监控数据，协同应用与存储方快速定位根因。事后，必须进行系统性复盘，落实改进措施，并可通过混沌工程提前验证系统的容错能力。

总之，CSI 的稳定性管理是一个综合性课题。除上述环节外，在实际中还会涉及多租户隔离、跨可用区容灾、混合云统一管理等更复杂挑战。通过将 SRE 的稳定性实践深度融入选型、部署、运维与故障处理的全流程，才能为上层应用构建坚实可靠的存储基石。

2.4.2.4 安全与隔离

在多租户环境中消费算力时，算力抢占、越权操作和数据泄露是常见的安全挑战。为了应对这些风险，操作系统内核提供了一系列核心安全机制，这些机制在容器化和云原生计算场景中发挥着关键作用，确保算力调度和运行的可靠性、隔离性及合规性。

以 Linux 为例，在 Linux 内核中，一系列核心安全机制被设计用于支持容器化环境和云原生计算，以确保算力管理、隔离和网络控制的可靠性。这些机制包括 Cgroups，它通过控制资源分配来限制进程组的 CPU、内存和 I/O 使用，防止单个容器过度消耗主机资源，从而维护系统稳定性。Namespaces 则实现视图隔离，为每个容器提供独立的进程 ID、网络和文件系统视图，确保环境之间的逻辑分离，减少潜在的攻击面。此外，CNI（Container

Network Interface）规范网络栈的管理，通过标准化接口允许插件处理容器的网络连接、IP地址分配和安全策略，从而支持灵活且安全的网络配置。

这些内核安全机制共同满足如等保2.0等安全合规要求，重点关注算力隔离以防止未授权访问和数据泄露。通过Cgroups和Namespaces的协同作用，系统能够有效防止容器逃逸，即容器进程突破隔离访问主机资源，以及横向渗透，即攻击在容器间扩散。这种设计不仅提升了多租户环境的安全性，还确保了计算任务在调度和执行过程中的合规性，为综合算力调度提供了坚实的基础保障。

基于内核的安全机制，综合算力调度系统需要提供一些安全隔离措施来确保算力安全和隔离性。这些措施通过多种技术实现综合隔离效果，包括但不限于以下方案。

1、Cgroups视图隔离，通过在挂载/proc和/sys文件系统时绑定容器私有命名空间，有效过滤宿主节点信息，防止容器内进程访问宿主机敏感数据，从而增强环境隔离性。

2、磁盘空间隔离，采用XFS配额技术，为多用户SaaS服务分配和限制存储空间，确保每个租户的磁盘使用不超过预定配额，避免资源竞争问题。

3、带宽隔离，通过CNI插件如Bandwidth，基于Linux Traffic Control框架实现网络带宽的限制和优先级调度，优先保障高优先级业务流量，防止网络拥塞。

4、磁盘IO隔离支持按权重分配磁盘IO带宽，并可进一步实现BufferIO隔离，优化共享存储环境中的输入输出操作，确保公平性和性能。

5、输入校验利用Kyverno准入控制器进行正则表达式校验镜像来源，并禁止高危权限如CAP_SYS_ADMIN，从源头提升容器安全性，防止恶意部署。

6、CPU绑核通过CpuSet将进程绑定到特定CPU核心，减少计算任务跨NUMA节点的访问，提升缓存亲和性和整体性能。

算力隔离性在离在线混部场景要求更高，用途更广。离在线混部系统专门用来解决异构应用消费异构算力冲突的问题，其核心目标有两个：一是寻求在共享算力模式下异构应用安全高效共存的模式；二是在不同资源组织模式下更多更安全地挖掘异构算力。这两者本质上都是为了提高算力利用率，降低应用部署成本。

离在线混部系统在综合算力调度中至关重要，它通过一系列核心功能来优化资源管理，确保在线和离线应用之间的和谐共存。该系统首先利用系统级安全隔离技术实现离在线应用的隔离机制，通过为离线容器提供独立的Cgroup算力限制，支持对离线Pod的CPU压制和视图隔离，从而有效区分算力使用。此外，系统还允许设置内存OOM优先级，并对

网络 IO、磁盘 IO 及空间大小等指标进行隔离与限制，这有助于防止算力竞争导致的性能下降，提升整体稳定性。

此外，系统具备多维算力干扰检测与动态压制能力。它能够实时感知节点上在线程序的 CPU、内存、磁盘和网络等资源指标状况，并使用时序模型（如 LSTM）来预测在线服务的 SLO 波动风险。一旦识别到潜在干扰，系统会自动触发压制策略，并在秒级时间内下发新配额至离线容器的 Cgroup，实现快速响应，避免需要人工干预，从而维持服务级别协议的稳定性。

最后，系统通过用量画像与预测功能来增强算力规划的智能性。它实时收集在线服务的使用率数据，结合多维度信息预测服务的潮汐规律，例如识别算力需求的周期性波动。借助机器学习算法，系统能够实时计算节点可提供的冗余算力容量，从而动态调整算力分配，优化利用率并支持自动化扩缩容。这些功能集成在一起，构成了一个全面的算力调度解决方案，适用于各种互联网场景，如大规模 Web 服务或数据分析平台，以提升效率、可靠性和成本效益。

2.4.3 应用治理

在算力调度与运行管理中，应用被视为最小的管理单元，这意味着整个调度和编排过程都围绕应用的生命周期和需求展开，以确保算力的高效利用和系统的稳定运行。

应用编排是这一过程的基础，它定义了应用从创建到销毁的完整生命周期，包括算力需求的描述。原生应用编排提供了一系列基础类型，如 Deployment 用于无状态应用的滚动更新、StatefulSet 用于有状态应用如数据库的持久化、DaemonSet 确保每个节点运行特定 Pod、以及 Job 处理批处理任务，这些类型简单易用，适合大多数标准场景。为了提升精细化管理能力，应用编排还可以进行功能增强，例如增加优雅退出功能以避免数据丢失；增加原地更新功能减少因 POD 整体重启而浪费的停机时间；指定顺序缩容功能可以确保关键服务处理顺序与业务逻辑保持一致。这些增强措施特别适用于高可用性环境如金融或电商平台。此外，针对特定场景如中间件、数据库或游戏战斗服，可以通过自定义资源定义（CRD）方式扩展编排能力，实现更灵活的控制。

应用部署涉及将应用交付到算力平台的具体形式和方法，直接影响部署的效率和可靠性。原生资源定义部署使用 YAML 或 JSON 文件直接配置，方式简易快捷、学习门槛低，非常适合测试环境或小型应用快速上线。然而，对于复杂应用，HelmChart 部署通过模板化配置组织应用，支持版本管理和依赖处理，但需要将配置拆解并统一标准化以避免混乱，

例如在微服务架构中部署多个相互依赖的服务。GitOps 部署则基于版本控制系统如 Git，自动化部署流程并通过回退机制保障可靠性，当配置变更提交后自动同步到集群，并在出现问题时快速回滚，这适用于生产环境的高可用性要求，减少人为错误并提升运维效率。

应用弹性扩缩是实现自动化算力调整的核心，旨在根据负载动态缩放算力资源，优化成本和性能。原生横向扩缩容（HPA）基于默认系统指标如 CPU 或内存使用率进行被动触发，但可能无法及时响应突发流量，导致性能问题。为了改善这一点，增强型横向扩缩容支持多种触发方式，包括定时触发，基于流量、业务指标如 QPS 或用户会话数等条件触发、以及通过接口自定义触发方式等，策略丰富且对业务影响可控。例如在视频流媒体服务中根据并发用户数动态调整资源，确保流畅体验。这种自动化扩缩容需要结合业务指标，避免后知后觉，提升整体系统的弹性和响应能力。

应用 SRE 提效方案是对应用管理配套措施的升级，以应对海量、动态变化的容器应用环境。镜像加速是治理中的一个关键方面，因为在应用发布或变更时，大型镜像的拉取可能因网络延迟或高并发而失败，影响变更成功率；解决方案包括使用镜像缓存或优化分层来减少体积，例如在 AI 模型部署中加速镜像拉取以缩短延迟。远程命令执行则允许 SRE 通过工具如 kubectl exec 在部分或全量容器中执行临时命令，用于调试、日志收集或紧急修复，这结合可观测性工具可以快速定位和解决线上问题，提高临时事项处置的效率。

2.4.3.1 应用编排

在现代算力调度基础设施中，应用编排能力是确保容器化应用高效、可靠运行的核心组件。Kubernetes（k8s）作为主流的容器编排平台，提供了丰富的原生编排方案以及编排扩展能力，这些原生编排方案和增强性扩展方案，共同支撑起应用部署、管理和扩展的全生命周期。原生方案是 k8s 内置的工作负载资源，旨在满足常见场景的基础需求，而扩展方案则通过自定义机制弥补原生能力的不足，为企业级应用提供精细化控制。

Kubernetes 原生编排方案为应用管理提供了基础框架。Deployment 是其中关键组件，专为无状态应用设计，它通过确保指定数量的 Pod 副本持续运行，支持滚动更新和回滚机制，从而保障服务高可用性。例如，在 Web 服务器或 RESTful 服务场景中，Deployment 可自动化处理副本扩缩容，减少人工干预，其价值在于提升应用的可扩展性和稳定性。技术上，Deployment 基于 ReplicaSet 实现声明式管理，k8s 控制器自动驱赶实际状态向期望状态收敛。场景扩展上，结合 Horizontal Pod Autoscaler（HPA），它可应对流量峰值，优化资源调度。StatefulSet 则针对有状态应用，如数据库或缓存服务，它为每个 Pod 提供唯一标识、稳定网络标识和持久化存储，确保数据一致性。其作用在于避免 Pod 重建导致的数据

丢失，价值体现在支持分布式系统的可靠运行。技术背景上，StatefulSet 依赖 Headless Service 和 PersistentVolume 实现存储持久化。在算力基础设施中，它适用于 MySQL 集群或 Kafka 等场景，实现跨可用区的高可用部署。DaemonSet 适用于节点级任务，如日志收集或监控代理，确保每个节点运行一个 Pod 副本，其作用在于实现集群范围的均匀部署，价值是提升可观测性和运维效率。技术上，DaemonSet 控制器监控节点变化，自动适配新节点。场景扩展中，它可用于部署 Fluentd 或 Prometheus 代理，支撑大规模集群的监控需求。Job 和 CronJob 则管理批处理任务，Job 确保一次性任务运行至完成，适用于数据处理或构建脚本，而 CronJob 基于时间表定期执行任务，如数据备份或清理。它们的作用是自动化短时作业，价值在于提高资源利用率，避免长期占用。技术背景上，Job 支持并行度和重试，CronJob 使用 Cron 表达式调度。在 AI 算力调度中，Job 可分配 GPU 资源运行训练任务，CronJob 则实现定时流水线，优化存储效率。

增强性扩展方案相比于原生基础编排方案引入了更多丰富特性，以应对复杂需求。这些方案通常通过 Custom Resource Definition（CRD）实现，例如原地更新特性，它在应用更新时只重启容器实例，而不删除重建 Pod，从而保持 IP、共享内存等资源不变。其作用在于减少服务中断，价值是提升更新效率和网络稳定性。技术背景上，原地更新通过修改容器镜像触发，kubelet 负责重启，避免 Pod 重建带来的开销。场景扩展中，适用于金融或实时通信应用，可实现零停机部署，结合服务网格如 Istio，可精细化控制流量。优雅更新与删除特性则保障应用进程在完成工作后再进行更新或删除，例如确保服务下线后再终止 Pod，其作用在于避免数据丢失或请求中断，价值是提高应用可靠性。技术上，它利用 Pod 生命周期钩子和终止宽限期实现。在微服务架构中，这可配合服务注册中心如 Consul，确保服务发现一致性。按步骤灰度发布特性支持精细化控制发布过程，包括暂停、消息通知和审批机制，逐步切换流量到新版本。其作用在于降低发布风险，价值是实现渐进式验证和快速回滚。技术背景上，它常结合 Argo Rollouts 等工具，通过金丝雀或蓝绿部署实现。场景扩展中，适用于电商或 SaaS 平台，可先向小部分用户开放新功能，结合监控告警优化风险控制。指定顺序缩容特性在缩容时按优先级或标签顺序终止 Pod，其作用在于提升集群装箱率，价值是优化资源利用和成本。技术上，它可通过自定义调度器或 Pod 中断预算实现，与 HPA 或 Cluster Autoscaler 协同。在混合云环境中，这可优先保留生产环境 Pod，确保关键任务资源。

原生方案与扩展方案相辅相成，构建了完整的应用编排能力。原生方案覆盖了无状态、有状态、批处理等基础场景，提供稳定可靠的运维基础；扩展方案则填补了企业在更新、

发布、扩缩容等方面的精细化需求，推动编排向智能化发展。在算力调度基础设施中，这些能力共同优化资源分配、提升服务可用性，并支持从云原生到边缘计算的多样化场景。

在实际使用中，原生编排方案和扩展编排方案往往是组合使用的。通过组合Deployment、StatefulSet、灰度发布、优雅退出、原地更新等策略，实现部署流程的精细化和可控性，以满足企业对高可用性和敏捷发布的需求，例如在电商或金融领域，直接全量更新可能导致服务中断，因此通过扩展编排引入了可定制流程，以降低风险并提升可靠性。

以用户通过扩展特性编排组合来完成部分应用实例灰度发布为例，例如选择10%的实例作为测试目标，这可以通过标签或地域筛选实现隔离风险。触发机制随后启动更新，通常由CI/CD流水线中的事件如代码推送或Webhook引发，确保流程可追溯。控制器如Kubernetes的Deployment Controller接管协调工作，监控实例健康状态，并分步骤执行灰度发布：首先更新少量实例，并行运行自动化测试，验证通过后逐步扩大灰度范围，例如从10%到50%，每一步都可能包括人工审批或监控检查，以允许中途调整。在实例更新前，容器会执行优雅退出流程，通过信号处理确保完成当前请求后再终止，避免数据丢失，接着进行原地更新，仅替换镜像或配置而不重新调度实例，从而节省资源。同时，流程集成即时消息通知，向团队报告状态变化，并结合测试转测环节，从测试环境验证到生产环境，确保质量。最终，当所有灰度实例通过验证后，控制器执行全量更新，将新版本应用到所有剩余实例，采用滚动策略以减少停机时间，并支持快速回滚机制以备不时之需。整个流程强调用户可定制性，允许根据业务需求调整步骤顺序或条件，例如在高风险场景中增加人工干预。

为了具体说明这种组合方案的价值，可以考虑电商平台微服务更新的案例。在一个处理高并发交易的订单服务中，直接全量更新新支付功能可能导致故障，因此采用金丝雀发布策略：先指定10个实例作为灰度组，仅向北京用户开放，由CI/CD流水线触发更新；控制器分步骤扩大灰度范围，每步进行支付接口测试和性能监控，并通过IM工具通知SRE团队；实例更新前优雅退出确保无请求丢失，原地更新优化资源使用；测试转测成功后，全量更新剩余实例。这种组合不仅能够降低更新是支付错误率，还提升了部署效率，类似方案可扩展至金融或游戏领域，例如在金融系统中增加合规审批，或在游戏服务器中实现玩家无感知的热更新。

2.4.3.2 应用部署

在现代云原生和DevOps实践中，应用部署作为软件交付的直接环节，需要提供多样化的方式来适应不同技术偏好人员的需求，以实现高效且稳定的部署流程。业界常见的部

署方案包括常规资源视图、模板集、Helm Chart 和 GitOps，每种方案都基于特定的技术背景和发展需求，旨在平衡易用性、标准化与自动化。这些方案不仅覆盖了从简单可视化操作到复杂自动化交付的全场景，还体现了云原生技术的演进趋势。

资源视图方案源于 Kubernetes 生态对降低操作门槛的需求，它通过可视化界面支持用户灵活配置复杂的资源结构，例如基于原生 Kubernetes 资源声明进行定义，允许通过表单式填写或拖拽方式完成部署。这种方案提供了核心资源的表单与 YAML 双向转换、统一环境变量渲染以及发布版本管理等功能，特别适合开发测试环境或缺乏深度 K8s 经验的团队，因为它能减少人为错误并加速部署过程。例如，在快速原型开发中，资源视图可以简化多容器 Pod 的配置，而无需用户手动编写易错的 YAML 文件。

模板集方案是对原生 YAML 管理方案的升级，它引入了资源模板、发布版本管控等能力，支持标准化部署和流水线集成。通过将 Kubernetes 资源定义为可复用的模板，并允许参数化注入，模板集提升了部署的可靠性和可审计性，同时便于与现有运维系统对接。这种方案适用于企业级环境，例如在需要严格合规的大型组织中，模板集可以确保所有微服务部署符合安全策略，并通过版本历史实现回滚和跟踪。

Helm Chart 方案作为 Kubernetes 的包管理工具，专注于简化多环境部署和 DevOps 集成。它通过 Chart 封装应用及其依赖，支持基于值文件适配不同环境（如开发、测试和生产），从而便于业务团队一键部署复杂应用套件。Helm Chart 的优势在于提高部署的可重复性和生命周期管理，例如在电商平台中，它可以快速部署包含前端、后端和数据库的全栈应用，并通过社区 Chart 库共享最佳实践。

GitOps 方案符合现代交付理念，以 Git 仓库为唯一可信源，适用于复杂版本管理、大规模发布和自动化交付场景。该方案强调通过代码编写、审查和持续同步流程，确保环境状态与声明一致，并支持环境的回滚与审计。GitOps 提升了部署的可靠性和安全性，特别适合高合规行业或大规模微服务集群，例如跨国公司可以利用 GitOps 管理全球集群，实现零手动干预的持续交付。

以上四种方案各具特色：资源视图以低门槛见长，模板集注重标准化，Helm Chart 简化多环境适配，而 GitOps 则强调审计与自动化。随着应用复杂度的提升和环境规模的扩大，对部署流程的可靠性、审计能力和自动化程度提出了更高要求。在这个背景下，GitOps 作为一种以 Git 为核心可信源的声明式部署范式，逐渐成为企业级云原生部署的重要演进方向。它通过将基础设施和应用状态全面代码化，实现了版本可控、变更可溯、自动化同步

的现代交付模式，特别适用于跨集群、多环境的大规模场景。接下来，我们将重点展开介绍 GitOps 的基本原理、实践路径及其在 SRE 中的独特价值。

GitOps 是一种基于声明式配置、不可变基础设施和持续交付理念的现代应用部署范式，其核心思想是将 Git 仓库作为应用配置和基础设施的单一事实来源，通过自动化流程实现部署的版本化、可审计和可回滚。这一方案起源于云原生领域，特别适用于 Kubernetes 环境，旨在解决传统部署中手动操作多、环境漂移风险高的问题。技术背景上，GitOps 借鉴了 DevOps 最佳实践，强调“一切即代码”，利用 Git 的版本控制能力确保部署过程的一致性和可靠性，从而为应用交付提供安全可回滚和效率提升的基础。

在具体实现上，GitOps 方案采用分层架构，用户通过 SaaS 层的交互界面或命令行工具发起部署请求，这些界面提供可视化操作，降低使用门槛，同时支持 REST API 调用以实现标准化集成。SaaS 服务负责认证、授权和路由，并具备扩展能力，如部署前检查、应用市场模板、状态通知和 Webhook 触发，这些能力通过仓库提交自动驱动流程，减少人工干预。核心的 GitOps 引擎以 Argo 为核心组件，包括 GitOps-Manager 作为 API 服务器协调整个流程，以及多个专用控制器，如 Terraform Controller 用于基础设施即代码管理、Vault Controller 用于安全密钥注入、Workflow Controller 支持复杂部署模式，以及 BKMonitor As Code 实现监控配置代码化。引擎层通过监听 Git 仓库变化，自动同步配置到目标 Kubernetes 集群，并与代码托管平台（如 GitHub、Gitee）集成，通过 webhook 实现事件驱动。平台集成方面，方案支持多集群管理，连接权限中心、流水线工具和监控平台，形成闭环运维，工作流程从提交、同步到验证和通知，确保部署高效可靠。这种实现可扩展到混合云或边缘计算场景，例如在金融行业中结合合规性检查，提升适应性。

GitOps 对站点可靠性工程（SRE）具有显著价值，它通过自动化运维减少手动错误，使 SRE 从重复任务中解放，专注于高价值工作如容量规划。部署效率大幅提升，声明式配置支持快速迭代和多环境一键推广，缩短交付周期。安全方面，版本化存储和密钥管理增强系统韧性，实现快速回滚；可观测性通过状态通知和监控 As Code 提供实时洞察，便于故障排查和合规审计。此外，不可变基础设施降低环境漂移，结合算力复用能力优化成本，推动运维文化向声明式和自愈式转变。

2.4.3.3 应用弹性管理

应用弹性管理是综合算力调度中的重要组件，主要作用是通过动态调整算力来应对业务负载波动，从而保证服务质量和算力效率。这一方案以 Kubernetes 的水平 Pod 自动伸缩器（HPA）为基础机制，实现 Pod 副本数的自动化扩缩，并与集群自动扩缩容协同，形成

从应用到基础设施的全栈弹性。HPA 作为 Kubernetes 的原生组件，其设计背景源于容器化技术的普及，使得微服务和无服务器架构能够高效应对可变负载。弹性管理不仅适用于高并发互联网服务，还扩展至批处理、混合云等场景，成为站点可靠性工程（SRE）实践的关键工具。

应用弹性管理框架是算力调度的重要方案之一。它通过动态调度算力资源实现自动化伸缩，广泛应用于云计算和微服务场景。框架采用分层架构确保全流程可控：业务应用层承载具体应用，需暴露监控指标并支持容器化伸缩；弹性管理层负责策略配置，通过策略引擎定义规则；弹性控制层包含触发机制和计算单元，基于实时数据决策资源调整；运行时层以 Kubernetes 引擎管理应用生命周期；弹性资源层提供底层算力如虚拟机；观测数据贯穿各层，集成监控系统收集指标。此外，框架可扩展至微服务架构实现精细伸缩，在边缘计算中响应本地负载，或在混合云环境中协调跨云资源，以提升业务连续性和成本效益。

基于应用弹性管理框架之上，弹性管理机制的作用在于通过自动化资源调整实现应用伸缩性，支持突发流量和业务波动，提升资源利用率。该机制采用分层架构，从观测数据层采集多源数据如性能指标和日志，到弹性控制层包含触发单元识别需求、计算单元生成策略和伸缩单元执行调整，再通过运行时层管理应用状态，弹性资源池层提供虚机、节点池和外部可伸缩资源以支持节点扩缩调整，最后弹性管理层涵盖配置、策略、调度、驱逐和诊断模块确保机制可配置和可维护。技术要点包括数据驱动决策通过算法实时分析触发伸缩，模块化设计解耦各层便于扩展，资源抽象封装异构资源实现无缝伸缩，以及闭环控制结合诊断模块优化决策准确性。扩展场景应用于大规模微服务如电商或金融系统应对促销或交易高峰，自动扩展节点池并调度外部资源确保低延迟；混合云环境集成外部可伸缩资源提升灾难恢复能力；成本优化通过策略模块配置成本感知策略在负载低谷时缩容降低算力支出。本设计强调自动化、可观测性和可管理性，为算力调度提供标准化框架，适用于高动态性互联网应用。

HPA 的基础能力遵循 Kubernetes 官方规范，是基于弹性管理框架和弹性管理机制设计上的简单实践。HPA 的工作流程包括输入层、决策层和输出层。输入层通过监控 Pod 的算力使用情况（如 CPU 或内存利用率）采集指标，HPA 控制器定期比较这些指标与预设阈值；决策层则在指标超出阈值时触发 Scale 操作，调整 Deployment 或 ReplicaSet 的副本数，例如当 CPU 使用率持续高于 80%时自动扩容 Pod；输出层则将扩缩结果反映到 Pod 集群，实现算力动态平衡。在技术细节上，HPA 依赖 Metrics Server 进行指标采集，采样间隔可配置为 15-30 秒，并采用保守的延迟算法避免频繁振荡，例如设置稳定窗口来防止过度伸

缩。实际应用中，视频流媒体服务可在晚间高峰时段基于CPU指标自动扩容，确保流畅播放，而流量回落后自动缩容以节约成本，这体现了基础能力的实用性和自动化优势。

为了满足更广泛的应用需求，提升应用弹性管理的灵活性，需要在HPA基础能力之上进行扩展。常见有如下几种扩展模式：内置指标、自定义Metric、WebHook模式和定时模式。内置指标兼容Kubernetes原生指标，如算力指标（CPU、内存），也会按需增加其他指标如组件指标（网络IO）和应用编排指标（Pod就绪数），这些指标开箱即用，适合基础监控场景，例如在数据库服务中通过内存指标预防OOM问题。自定义Metric通过对接监控系统，支持业务级或应用级自定义指标（如QPS、错误率或订单量），HPA利用Custom Metrics API实现指标注入，使弹性决策更贴近业务需求，例如电商大促期间基于订单量扩容结算服务，避免仅依赖算力指标的误判。WebHook模式允许业务方通过HTTP接口参与扩缩决策，HPA将指标数据发送至自定义服务并接收建议，实现交互式弹性，特别适用于混合云环境，其中业务服务可结合成本策略优先扩容低成本区域Pod。定时模式则基于时间计划（如cron表达式）预设扩缩目标，适用于周期性潮汐负载，例如在线教育平台在课表时间段固定扩容，避免指标采集延迟。这些扩展能力使HPA从算力驱动升级为业务驱动，支持多指标加权和灰度发布等高级策略，提升了弹性的精度和稳定性。

应用弹性管理是SRE综合算力调度的重要能力，很多基于应用弹性管理扩展出来的能力都对SRE降本增效相关工作帮助巨大。在实际应用中，弹性管理方案可适配多样化场景。例如，互联网高并发服务如社交平台，在突发热点事件中基于QPS自定义指标秒级扩容，并结合定时模式预加载算力以避免冷启动延迟；批处理作业如大数据分析，则通过WebHook模式与作业调度器交互，按队列长度弹性调整算力。最佳实践包括设置最小/最大Pod数边界防止无限扩缩，使用多指标加权提升决策准确性，以及结合HPA冷却窗口避免抖动。这些实践确保了方案的可靠性和可扩展性。

在SRE平台工程实践中，应用弹性管理与集群弹性扩缩容往往形成联动方案，通过协同工作实现算力全局优化。具体而言，应用层弹性通过Horizontal Pod Autoscaler（HPA）监控指标如CPU使用率来调整Pod副本数，而集群层弹性通过Cluster Autoscaler（CA）根据节点资源状态动态添加或移除节点。这种联动解决了传统静态资源分配的痛点，突破了集群内算力限制，为跨集群算力调度提供了操作空间。

联动方案的核心是应用层与集群层的闭环控制，形成多种扩缩模式。常规扩缩模式是最基本的形式，当Pod因资源不足无法调度时，CA触发扩容；当节点空闲或集群装箱率低于阈值时，CA触发缩容，这依赖于Kubernetes调度器的反馈机制，适用于通用工作负

载如 Web 服务。容量 Buffer 扩缩模式则通过预设资源水位线实现预防性扩缩，例如设置低水位和高水位，确保集群保留缓冲资源，避免性能抖动，常用于对稳定性要求高的场景如金融系统。Webhook 扩缩模式允许高度自定义，CA 通过调用用户定义的 Webhook Server 获取期望节点数，实现基于业务逻辑的决策，适用于复杂需求如 AI 训练任务。定时扩容模式针对周期性负载，CA 根据时间规则在指定时间段维持最低节点数，适合潮汐效应明显的业务如在线教育，这可以减少响应延迟并优化资源使用。

这种联动方案为 SRE 带来了显著价值，包括成本优化、可靠性提升、算力利用率最大化等都具备了更多可操作性。

2.4.3.4 应用 SRE 提效

应用 SRE 提效方案是对应用管理配套措施的升级，以应对海量、动态变化的容器应用环境。通过 SRE 综合算力调度来部署发布应用，或者执行批量计算操作，或者完成 AI 模型训练，其效率都会受到多种因素的影响。例如动辄几十上百 GB 的 AI 模型镜像，如果仅仅通过 Pull 的方式拉取，不仅效率低而且容易出错；一些大型复杂应用，经常通过大型二进制配置文件来触发特定逻辑，如果无法通过类似信号这样的处理机制动态加载配置，业务服务会受到较大影响。在企业内实际落地的过程中，业务形态和部署环境千差万别，类似这样的场景还有很多，应用 SRE 提效则是通过提供一些可插拔的增值方案来解决这些问题，并避免对 SRE 平台工程主体能力构成影响。下文将以镜像加速和远程命令执行为例来说明这部分的作用。

在云计算和容器化技术广泛应用的背景下，容器镜像的拉取效率已成为影响算力分配与使用效率的关键因素。随着企业应用规模的不断扩大，镜像拉取过程中常出现规模大、效率差、失败率高等问题，这不仅拖慢部署速度，还可能引发系统不稳定。因此，构建一套高效的镜像加速工具套件势在必行，以应对这些挑战。容器镜像拉取成功率直接关系到算力调度的有效性，例如在拉取规模方面，镜像数量、镜像大小和节点数三个维度的线性增长会加剧中心仓库的负载；在拉取效率方面，中心仓库针对仓库或用户实施的限速限频策略可能导致 100 个以上节点拉取 1GB 镜像需要 5 分钟以上；而在拉取失败方面，中心仓库带宽上限易引发 I/O 超时，增加部署风险。这些问题的根源在于传统中心化仓库的单点瓶颈，无法满足云原生时代的高并发需求，从而推动了去中心化加速方案的发展。

镜像加速实践方案的核心在于采用多种技术组合来优化传输过程。首先，P2P 文件分发技术利用点对点网络对镜像 Layer 进行分发，通过节点间直接共享数据提高传输效率，并在跨云或跨网络环境下保持良好性能。这种技术源于文件共享领域，其优势在于形成网

状传输网络，分摊负载，并可通过智能路由（如基于地理位置选择 peer）优化路径。在实际场景中，例如多云混合部署或 IoT 边缘计算，P2P 分发能显著减少延迟，同时结合 Layer 去重和缓存策略（如使用 SHA256 校验和）避免重复传输。其次，节点级限速允许用户精细控制镜像 Layer 的分发速度、回源速度以及解压时的 I/O 占用，适用于对带宽或 I/O 敏感的业务场景。这一机制基于网络 QoS 理念，通过 cgroups 或令牌桶算法实现资源隔离，在算力受限环境（如开发测试集群或 AI 训练集群）中能防止算力争抢。第三，方案容灾通过设置多级回源策略（如优先 P2P，失败后回退至中心仓库）处理异常状态，如传输中断、速度过慢或服务宕机，从而增强系统鲁棒性。这种容灾设计借鉴了微服务中的熔断机制，可结合健康检查和监控告警实现自动化切换，适用于灾备或合规要求严格的场景。最后，性能提升方面，实测显示该方案能缩减超过一半的拉取时间，并有效规避中心仓库的限频限流问题，通过分布式架构提升整体吞吐量，例如在大规模集群中支持快速扩容和敏捷发布。

这些实践方案对站点可靠性工程（SRE）工作具有重要价值。SRE 的核心目标是保障系统稳定性、可用性和性能优化，而镜像加速方案直接支持这些目标。例如，通过容灾机制减少服务宕机风险，SRE 可以降低平均修复时间，符合误差预算管理原则；节点级限速和 P2P 分发优化了资源效率，帮助 SRE 更精准地规划容量；加速工具集成的监控指标增强了可观测性，支持自动化运维和持续改进；同时，该方案赋能大规模运维，确保在微服务和云原生架构下实现快速、可靠的部署，从而提升 CI/CD 流水线的效率。总体而言，镜像加速方案不仅提升了算力调度效率，还为 SRE 提供了关键工具，以应对日益分布式的运维挑战，未来可结合智能调度算法进一步扩展至边缘计算等复杂场景。

远程命令执行是另一种典型的应用 SRE 提效方案，该方案旨在解决容器化环境下半微服务化应用的运维挑战。与传统二进制应用相比，容器化应用的环境要素，如 IP 地址、内存分配和存储挂载，会随着 Pod 的创建、销毁或扩缩容而频繁变化，这种动态性使得运维工具需要更高的适应性。原生 Kubernetes 提供的 exec 命令虽然可用于单个 Pod 的交互式操作，但难以高效处理海量 Pod 的大规模运维场景，例如批量脚本下发或命令执行，因此需要集成作业管理、CMDB 和容器管理等平台能力，实现标准化和自动化的远程操作。

远程命令执行的实践方案主要涵盖批量与随机两种场景。在批量应用的脚本下发中，运维人员可以通过作业管理针对一组基于标签选择的 Pod 集群，批量分发预定义脚本，以完成特定任务，例如在数据迁移过程中运行数据清洗脚本处理用户数据。这种方案支持脚本版本管理和错误重试机制，并可扩展到 CI/CD 管道中的蓝绿部署初始化或安全合规检查。

批量应用的命令执行则涉及向一批 Pod 同步发送简单指令，如发送系统信号以实现原地更新，促使用新版本镜像，而无需重建 Pod。这种方法提升了操作效率，并支持异步监控，适用于集群维护或性能调优。在随机场景中，随机应用的脚本下发允许 SRE 通过 CMDB 定位特定 Pod，运行复杂脚本进行深度验证，例如应用发布后随机抽样分析用户数据，确保符合预期，这有助于故障诊断或 A/B 测试。随机应用的命令执行则针对单个 Pod 执行临时命令，如查看端口统计信息，用于快速诊断网络瓶颈或响应监控告警，同时集成权限管理和审计功能，保障安全性。

类似这种应用 SRE 提效方案对 SRE 的工作具有重要价值。它能显著提升运维效率、保障系统稳定性、可用性，使 SRE 能够通过标准化工具平台处理差异化应用治理场景，并减少手动干预和人为错误。

2.5 算力市场与供给管理

算力资源的高效配置是释放企业数字化潜能的核心引擎。在企业内部，通过市场化机制将算力资源汇聚并进行统一管理，能够最大化发挥其效益，显著提升资源利用率、降低用算成本，并系统性解决供需错配问题。这一过程主要依托于算力市场的构建和算力供给的管理两大方面。

算力市场旨在从概念和技术层面实现算力的互联互通，构建算力交易模式，以解决算力供需侧矛盾。其特点包括：算力资源可按需整合和调度，不严格追求公有算力模式，而是采用公有与私有相结合的方式，通过市场化机制进行调节；算力资源能够自由流通与共享，借助算力识别、整体调配和自由租借等措施，缓解因地域、服务商或风火水电等基础资源配套限制导致的分布不均衡问题；此外，算力资源可分时复用，利用算力消费的潮汐效应（如日间在线交易高峰和夜间 AI 训练高峰），通过削峰填谷的手段提高整体使用效率，从而优化资源分配。

算力供给管理则侧重于供给侧的管理宽度和灵活程度，这直接决定了算力市场的规模和资源利用效率。其特点涵盖：建立统一的算力识别与画像体系，通过标记算力的来源、类型和服务商等信息，便于系统识别和筛选，并优先匹配亲和度高的消费任务；通过技术手段降低供给方门槛，除整机算力和集群算力调度管理外，还纳入零碎算力的动态调度，利用算法和 AI 能力保障空闲零散算力的识别与调度；实施灵活的上下架方案，借助流程与自动化体系搭建完善的运维方案，自动处理算力的上架和下架，确保算力的顺畅流转和高效利用。

2.5.1 算力市场

算力市场是基于综合算力调度技术构建的一个可信算力交易服务体系，它作为连接算力供给方和消费方的中枢平台，通过高效的资源协调和交易机制，确保算力资源的可靠分配和利用。这一体系以技术为基座，整合了多方资源，旨在解决算力供需的结构性矛盾，提升整体算力生态的效率和透明度。

算力市场的核心能力主要体现在精准匹配、无差异体验和可信交易三个方面。精准匹配能力通过智能算法聚合分散的 GPU、CPU 及边缘算力资源，动态对接供需双方，例如利用 AI 撮合引擎将企业闲置算力与高需求任务匹配，从而优化资源利用率。无差异体验能力抽象了底层硬件差异，提供统一的算力服务接口，使开发者无需关注算力来源，仅需提交计算任务即可获得结果，这显著降低了使用门槛，加速了创新应用的落地。可信交易能力则构建了完整的信用体系，确保交易全程可追溯和可审计，通过信用评级和 SLA 监控，推动算力市场的规模化和规范化发展。

在算力市场架构中，算力消费侧管理模块负责全生命周期的用户交互。平台接入支持企业和个人用户的实名认证、多因素安全登录及权限分级，并通过多端接入方式如 Web、API 和移动 App 集成单点登录功能。算力订购提供灵活的采购模式，包括按需秒级计费、包年包月订阅和竞价实例，用户可批量发布订单并设置弹性扩缩容策略，同时通过可视化界面实时跟踪资源交付状态。算力使用环节提供容器化开发环境和智能运维辅助，例如 Jupyter 交互式编程和资源监控看板，以优化任务执行。算力结算则采用多维计费体系，支持混合计费模式，并自动化生成消费账单和开票流程，关联项目成本中心。

算力市场平台模块作为中间层，专注于供需匹配、分发调度和交易运营。供需匹配通过智能撮合引擎和信用评级机制，动态连接算力供给与需求，降低采购风险。分发调度支持异构资源协同和跨地域调度，利用算力路由技术优化传输路径，并实现弹性容灾，确保任务连续性。交易运营则扩展至生态化服务市场，集成模型市场（MaaS）和数据集（DaaS），并提供工具集成如资源大盘数据展示和多维度报表生成，以增强平台的可管理性和服务多样性。

算力供给侧管理模块负责算力资源的注册、接入、池化和运维。算力注册要求供应商按平台规范声明算力类型、服务等级和安全隔离能力。算力接入通过统一接口层兼容异构设备，如海光 DCU 芯片和寒武纪 MLU 加速卡，并自动转换算力颗粒标准。算力池化将物理资源虚拟化为可调度单元，并动态分级匹配高实时性需求。算力运维则涉及故障预测和多维度异常检测，构建硬件层到应用层的依赖关系图，并实施分级自愈策略，确保资源隔

离和容灾能力，提升整体系统的可靠性和稳定性。这一架构共同支撑了算力市场的高效运行和持续演进。

算力市场的供需匹配流程是一个系统化的运作机制，旨在高效连接算力供给与需求方。该流程起始于算力资源注册阶段，通过实时获取全球算力资源的部署位置、状态信息和负载情况，为后续调度提供基础数据。随后，进行需求采集，以捕捉用户的算力采购需求，包括类型、规模和周期等参数。算力资源信息同步环节将这些实时数据更新至调度引擎的核心数据库，确保信息的准确性和时效性。基于这些输入，调度引擎执行算力整合编排，综合考虑供给方的可用算力量、地理分布、供给周期以及价格策略，计算出最优的资源协同方案。接下来，算力调度执行阶段按编排策略动态分配算力资源和应用实例，实现资源的精准匹配。最终，算力资源供给环节按调度结果向用户交付算力服务，完成整个供需闭环。

算力整合编排作为核心环节，展现出多个关键特点。首先，它具备多类型兼容能力，支持通算、超算、智算等多种算力类型的无缝集成，适应多样化的计算需求。其次，适配多消费模式，涵盖整机消费、容器消费和 Serverless 消费等形式，为用户提供灵活的算力使用方式。此外，编排过程支持灵活周期配置，包括一次性任务和周期性消费如包年包月，以满足不同用户的业务节奏。最后，实施多层级调度策略，允许进行全球调度、定向多云调度、定向单公有云调度或定向私有云调度，增强系统的适应性和精细化控制。

在算力调度匹配策略方面，常见策略主要包括稳定性策略和价格策略。稳定性策略旨在提升系统可靠性，其中容灾性调度通过将任务均摊分发至多个供给方，以降低单点故障风险；堆叠式调度则尽量将任务集中到单一供给方，简化管理并提高效率。价格策略聚焦成本优化，包年包月套餐调度将任务定向至指定供给方以利用长期合约优势，而最优价格匹配则通过比较多个供给方的报价，将任务分发至成本最低的选项，实现经济效益最大化。

调度分发的核心能力构建在两大支柱上。一是基于联邦架构的应用分发调度，它实现多云异构集群的统一纳管，集中管理差异化环境；支持动态应用分发，包括灵活的任务分发模型和灰度发布策略；提供跨集群服务治理，确保服务发现和安全互访；采用策略化调度引擎，支持优先级、亲和性等多样化调度策略；通过标准化操作接口开放 API，实现应用生命周期的便捷管理；并具备高可用保障机制，在集群故障时自动迁移应用实例以维持服务连续性。二是基于集群自动扩缩容的分发调度，它适配潮汐资源，根据实时负载动态伸缩集群资源；结合 AI 预测技术，进行弹性伸缩，预判需求趋势并主动调整资源规模，提升资源利用效率和响应速度。这些能力共同支撑算力市场的高效、稳定运行。

在算力市场的交易运营中，容量与额度管理是优化资源分配、保障供需平衡的核心环节。通过分层资源池设计、动态额度调控及智能化监控体系，实现对算力资源的精细化管控，从而提升应用效率、降低成本，并确保算力供给的公平性和稳定性。这一管理机制基于多元化的算力供给池，匹配差异化的业务需求，并通过实时监测和弹性调整来应对动态变化。

容量管理的核心在于构建多元化的算力供给池，以匹配差异化的业务需求。分层资源池设计按照资源的不同属性维度，将多个云厂商（如私有云、微软 Azure、华为云等）的算力资源聚合成不同类型的资源池。例如，高优业务独立资源池为稳定性要求高的核心业务提供物理隔离的专属资源，确保算力零争抢；而长尾业务联合资源池则聚合碎片化算力，支持弹性任务（如数据分析、模型微调），通过超卖机制提升资源复用率。此外，混合资源类型标注明确标识智算 GPU、超算 NPU 等异构算力特性，便于需求方按任务类型匹配资源，如 AI 训练需高浮点算力，科学计算需高精度 NPU。

容量监测维度涵盖多个方面，包括全局总量监控（算力资源池总容量）、地域分布监测（区域级资源供给状态）、云厂商供给分析（供应商资源占比）、资源类型统计（CPU/GPU/NPU 分类）、稳定性等级追踪（SLA 达标率）以及价格区间扫描（单位算力成本波动）。这些监测手段确保了对算力资源的全面可视化和动态跟踪，为后续的额度管理提供数据基础。

动态额度管理机制是实现算力资源精细管控的关键，其核心功能包括整机资源分配、项目级可视化、弹性调整机制和网络资源管控。整机资源分配采用硬隔离策略，按项目分配 GPU/NPU 专属配额，同时允许共享集群超售，但单项目占用不超过 30%，以平衡资源利用和公平性。项目级可视化提供实时展示功能，包括算力使用率、成本占比和剩余额度，并集成风险预警机制，如当 GPU 利用率超过 85%持续 10 分钟时触发阈值告警。弹性调整机制支持人工干预（按业务优先级手动调整）和自动扩缩容（基于 AI 预测动态优化），确保额度能够灵活响应业务变化。网络资源管控则涉及 Underlay 层的物理带宽 QoS 保障和 Overlay 层的虚拟通道隔离，并配备防护机制如 DDoS 攻击实时熔断，以维护网络稳定性。

高级管控能力进一步强化了动态额度管理，包括成本追踪系统和风险预测引擎。成本追踪系统自动关联算力消耗与财务数据，生成项目级 ROI 分析，帮助用户进行整体的额度管理和成本管理，提升业务效率。风险预测引擎结合历史负载数据预测资源缺口，并提前 72 小时触发扩容建议，从而实现 proactive 的资源调整。

智能供给策略通过潮汐资源调度、动态扩缩容机制和额度共享等方式，优化算力利用。潮汐资源调度支持跨项目共享低峰期闲置算力，充分利用业务低峰时算力，为业务降本；动态扩缩容机制基于AI预测算法预判负载趋势，自动伸缩资源规模；额度共享则通过资源标签（Label）实现项目间算力逻辑隔离，保障关键任务SLA，并支持跨项目额度借贷，提升算力的应用效率。这些策略共同确保了算力资源的高效、公平和弹性分配，最终实现供需平衡和成本优化。

成本与结算管理也是算力市场交易运营的重要组成部分，这一部分旨在通过系统化的架构和能力体系实现对算力资源的精细化管理和价值度量。

完善的算力成本结算系统需要依赖算力交易过程中数据的精准采集，和数据分析系统来构建端到端的算力成本控制能力。其功能设计主要分为算力结算管理和用量与监控计算两部分。算力结算管理模块负责整体算力资源的用量与成本观测，包括对算力供给收益的量化分析、算力利用效率的持续监测以及算力消费与供给的优化建议。这一模块通过集成实时数据采集和处理，支持从服务器和第三方指标采集系统中动态获取用量数据，为后续分析提供基础。用量与监控计算模块则专注于成本精算和监控功能，其中成本精算中心实现了弹性计费模式，如按量计费和套餐计费，并支持分钟级细粒度用量打点、周期性用量统计以及全量明细数据的存储；监控中心则负责用量明细指标数据的持久化存储和计费模块的运作，确保数据的可靠性和可审计性。

在审理利用率观测能力实现上，应该涵盖用量与额度观测以及利用率观测两个方面。用量与额度观测包括对容器、节点池的持续性低装箱率告警、主机、容器和节点池的实时供给展示、历史供给趋势回溯分析以及供给明细指标的自定义视图构建与告警配置。利用率观测则通过容器级CPU、内存、GPU的实时利用率展示和历史波动分析，主机级实时负载监控和历史趋势比对，以及业务模块级利用率对比分析，实现资源利用的深度洞察。

算力供给是算力市场的核心，算力市场的成本核算更多专注于算力供给的收益管理和优化，包括供给收益月度透视、算力收益趋势分析、算力收益账单查询和算力供给优化建议。供给收益月度透视支持多维度（如资源类型、项目组或地域）的收益分布解析和趋势追踪；算力收益趋势分析提供动态时间颗粒度（如日、周、季）的预测和子类目下钻分析；算力收益账单查询允许按时间维度（天、月、年）进行聚合查询、按收益子类汇总以及明细数据导出与合规审计；算力供给优化建议则通过智能扫描闲置或低效资源、优先级排序、资源调度策略推荐和预期收益量化模拟，为用户提供数据驱动的优化措施，从而实现成本节约和效能提升。

2.5.2 供给管理

算力供给管理是SRE综合算力调度系统的重要组成部分，算力供给的数量和质量直接影响算力调度能力和消费场景。算力供给管理旨在通过标准化流程和智能策略实现多来源异构算力的高效接入与动态整合，奠定了资源供给与业务需求精准匹配的中枢调度基础。其体系结构涵盖了从算力供应商管理、算力接入、到算力整合的完整流程，并围绕四大核心要点——标准化、模式化、一体化和自动化——展开，以提升算力接入和运维效率。

在算力供给管理的流程中，算力供应商管理支持多种来源，如私有云、Microsoft、aws、华为云、腾讯云及其他第三方平台，实现跨厂商资源的无缝集成。算力接入是指能够支持用户发起算力上下架流程，并将算力资源上架到系统或从系统下架，主要能力涉及算力接入标准管理、流程管理和运维管理，以确保各类算力资源（如通算CPU、智算GPU和超算NPU）能够顺利融入系统。算力整合则通过聚合碎片化资源，形成逻辑资源池，包括高优业务独立资源池、长尾业务联合资源池、共享CA资源池、共享CVM资源池、BG周转资源池和外部弹性资源池，以适配不同业务需求。

四大核心要求可以进一步细化管理策略如下：标准化强调互操作性、资源可视化和标准化管理，例如通过全局标识符体系实时映射算力属性，并提供统一调度接口，打破异构算力孤岛；模式化聚焦高效接入、供给类型丰富和风险可控，提供一站式工具链（如SDK/API/镜像模板），支持按需分配、预留实例和竞价实例等多种模式，并通过健康度评估模型实现异常节点的自动熔断；一体化致力于全局算力利用率最大化、并保持场景适配和弹性扩展方案的一致性，例如按业务特征生成定制算力模板（如AI训练或实时渲染），并同时支持秒级资源池伸缩；自动化则确保高可用性、安全性和成本优化，通过自动化流程管控减少故障时间，保护用户数据隐私，并降低人工干预带来的隐形成本。

随着异构算力和跨平台算力的广泛供给和接入，算力管理的复杂度显著提升。由于各厂商的技术架构和服务接口存在显著差异，导致算力资源难以实现高效协同与互联互通。为了解决这一问题，统一算力标识体系应运而生，它通过标准化资源描述和唯一性编码，使算力资源能够全局可发现、可调度和可管理，从而成为算力互联互通的核心基础。

算力标识的构建首先依赖于算力资源的注册与上报过程。算力提供方，例如云服务商或边缘节点运营商，需要向统一平台注册资源的静态属性，这些属性包括算力类型（如CPU、GPU或FPGA）、具体架构（例如英伟达H100或华为昇腾）、算力规格（以FLOPS、内存容量等指标衡量）、资源位置（涵盖地理区域和数据中心ID）以及提供方身份（如组织代码）。此外，提供方还需实时上报动态可用量，例如空闲GPU卡数或剩余内

存容量，并支持秒级更新以确保数据的时效性。平台则通过主动探针（如网络时延检测和API心跳包）与被动监控（如资源负载率和会话数统计）来验证算力资源的可用性，及时剔除不可用节点，从而维持资源池的健康状态。

接下来是算力抽象与编码阶段。算力抽象涉及从注册信息中提取多维度特征，以构建标识字段，这些字段通常包括提供方、资源类型、规格、位置以及服务等级等关键维度。例如，通过抽象化处理，可以将复杂的硬件细节转化为标准化的描述符。算力编码则采用统一规则将这些多维特征组合成一个唯一的、兼容的、适用的、可扩展的且科学的标识编码。这种编码确保了在不同环境和平台中的一致性，便于后续的解析和应用。

算力标识的应用覆盖多个核心功能场景，显著提升资源管理效率。在资源寻址方面，标识解析系统（如基于DNS树状模型）能够将编码映射到物理资源地址，并支持反向查询，从而实现快速识别和定位可用算力资源。对于算力调度，调度引擎可以依据标识中的规格、位置和负载动态信息，精准匹配任务需求，提高调度的准确性和效率。容量评估功能通过聚合标识的规格与位置数据，生成全局资源热力图和缺口分析报表，有助于对海量异构算力资源进行统一管理。在成本结算中，利用算力标识可以追溯资源使用量（如GPU小时数），并结合服务等级生成精确的计费账单。性能监控则通过绑定标识的实时指标（如计算负载、带宽和时延）触发自动告警，实现对各算力资源状态和性能的实时跟踪。最后，权限控制基于提供方字段的访问控制列表（ACL），有效限制未授权用户对敏感资源的调用，确保安全性。整体而言，算力标识体系通过标准化和自动化，解决了异构环境下的协同挑战，为算力生态的健康发展提供了坚实基础。

模式化是指用户在操作算力供给接入时，系统需要具备一定的配置和策略管理能力，以实现供给算力画像，从而能够在算力和需求间实现精准匹配，在具体的场景下，这些匹配策略可以通过固定为特定的模式来加速处理过程。这些配置和策略，主要涵盖接入类型管理、供给策略配置、售卖及分配机制与安全与隔离保障四个关键维度。

在接入类型管理方面，系统支持多元化的底层算力资源集成模式。具体包括物理服务器整机集成，即将完整物理设备纳入资源池；计算节点闲置算力碎片化共享，以聚合单个节点的部分计算能力；容器集群资源纳管，实现对Kubernetes等编排平台的Pod/Node资源无缝对接；节点池批量接入，用于集成预配置的计算节点资源池；以及第三方弹性资源动态扩展，允许接入公有云或IDC的弹性算力。此外，依据资源来源实施差异化策略，例如规范公有云服务商算力的接入与协同，以及管理私有云本地化资源的供给，从而确保资源来源的多样性和可控性。

供给策略配置涉及资源投放的时间模式和范围控制。时间模式包括短期一次性供给，即设置固定使用期限并到期自动回收；周期性供给，按日、周、月等间隔自动化执行供给与回收；以及长期持续供给，资源长期可用且需手动触发释放。范围控制则限定资源的可见性与可用性，例如定向授权供给，仅限特定授权用户访问；或全局共享供给，将资源开放至公共池并按规则调度，以平衡资源利用与访问权限。

售卖与分配策略聚焦于资源调度优化和保障机制。核心在于资源调度匹配策略，如收益最大化策略，优先分配给出价更高的任务以最大化收益；或利用率优化策略，优先提升资源装箱率与整体效率。同时，资源保障与预留机制确保关键需求的资源可用性，包括按比例预留，依据资源总量百分比进行预留；以及按绝对量预留，预留具体的 vCPU、GPU、内存等资源量，从而支持多样化的业务场景。

安全与隔离机制是算力供给的基础保障，涵盖资源全生命周期隔离和多租户权限管理。资源隔离包括接入前预检，进行完整性、兼容性与安全扫描；释放时清理，执行状态清零与安全核查；运行时分域隔离，严格区分在线服务与离线计算资源，并实施网络、内核等多级隔离；离线任务资源压制，对非实时任务实施资源配额限制；异常干扰监测，实时检测并处理性能降级或资源争抢；以及不可靠实例驱逐，自动隔离或驱逐运行异常的资源实例。权限管理则通过操作权限严格管控，基于角色校验算力资源操作权限，以及业务数据视图隔离，限制不同租户仅能访问其授权范围内的资源与数据，确保多租户环境下的安全与隐私。

将不同的供给方算力池化处理，是实现供给算力一体化管理的主要方式。池化算力能够最大化算力利用效率并降低总运营成本。其核心机制依赖于成本运营和资源运营的双循环优化，通过持续的数据分析和自动化策略，实现高效、弹性的算力供给。综合算力调度基于混合云环境，整合了物理机、虚拟机、GPU 等多种计算资源构建基础资源池，并在此之上打造场景算力池。最后通过算力市场连接消费方与场景算力池，从而实现算力的高效复用。

算力供给自动化是实现算力资源高效、可靠协同运营的关键机制。它通过自动化技术集成算力运维操作管控、算力安全管控和算力稳定性监测与管控等措施，确保算力供给侧运营的规范性和可持续性。自动化流程覆盖资源管理、安全防御和性能监控等多个维度，提升整体运维效率，减少人为干预，保障服务级别协议（SLA）的达成。

在算力运维操作管控方面，自动化系统严格管控算力资源的关键操作，确保每一步都规范、安全且可追溯。算力交易平台操作包括上架和下架流程的空闲检测，执行操作前强

制检测资源空闲状态，如无运行或排队任务、非活跃关键服务或进程、以及无未完成运维任务，确认稳定后方可执行；风险操作管控则要求高权限操作进行多级审批，如集群负责人和平台管理员审批，执行前进行影响分析和二次确认，关键操作如下架大量节点可能触发平台级熔断阈值限制，并强制记录完整、不可篡改的操作日志供审计。资源池与集群操作通过统一管理界面处理规格、地域、扩容、缩容、迁移和标签修改等，操作需授权认证并触发 CMDB 配置更新。供给侧节点操作涉及主机登录账号管控，实施最小权限访问控制，禁用共享账户，强制使用平台集成统一认证与堡垒机进行授权、登录审计和会话记录，运维凭证严格管理并自动过期，仅限指定网络路径访问管理接口；主机风险操作管控定义并执行黑名单操作，如 rm -rf /、fdisk、halt 等，高危操作如系统更新或核心配置修改必须通过平台标准化运维通道如 Ansible 或 SaltStack 模板作业执行，禁止随意手动操作，并实时监控告警或阻断命令行为。消费侧应用操作包括容器特权模式管控，严格禁用或限制消费侧应用启动特权容器，确需时需申请审核后按需授权；内核参数和主机时间修改管控严格隔离限制消费侧应用，禁止修改内核参数或主机时间，此类操作仅限平台管理代理或严格审批的运维流程执行，保障节点全局配置与时间一致。

算力安全管控构建全面的自动化防御体系，涵盖基础安全、算力基础设施安全和算力消费侧防护。基础安全方面，主机内核安全扫描定期使用轻量级 Agent 如 eBPF 深度扫描节点内核状态，检测 Rootkit 和内核钩子等恶意篡改；漏洞检测与修复集成标准扫描器如 OpenVAS 或 Nessus，定期扫描节点系统与核心组件，结果联动 CMDB 自动生成修复工单，支持优先级分批自动修补或重启，高危漏洞即时告警强制修复；安全基线与审计建立并强制执行强安全基线配置，如 SSH、防火墙和服务设置，定期进行基线合规检查与漂移修复，审计日志集中至 SIEM 或 SOC 实时分析告警。算力基础设施安全包括运行时安全检测与防护，部署主机 Agent 或 eBPF 进行运行时威胁检测，实时监控进程行为、文件访问和网络连接，对入侵行为如挖矿或勒索软件告警自动阻断；编排与部署安全检测与防护集成 CI/CD 的 IaC 安全扫描，运行时通过 K8s API 审计日志和策略引擎强制安全策略，检测阻止异常部署和违规配置；网络安全检测与防护实施最小化网络策略，部署 NIDS 或 HIDS 监控南北向和东西向流量，联动平台网络管理快速隔离异常节点。算力消费侧防护防范消费侧风险影响供给，应用镜像安全扫描与防护强制深度扫描容器镜像，检测漏洞和恶意软件，仅允许通过扫描或可信仓库的镜像运行，并提供运行时镜像完整性校验；应用异常操作检测与防护部署 APM 或专用工具监控消费侧应用运行时行为，检测越权操作、异常资源消耗和已知攻击模式，触发告警、限制或驱逐。

算力稳定性监测与管控通过自动化工具精细化监测资源与容量，保障节点稳定性和服务质量。整机供给监测针对提供整台服务器资源，节点状态实时监控硬件健康指标如 CPU 温度、风扇、电压、磁盘 SMART 和软件状态如负载和服务进程，设置预故障告警和健康状态大盘；节点网络容量持续监控网卡流量、包速率、带宽利用率和错误包率，建立网络基线并设置超限告警，关联负载信息拥塞预警；节点存储容量和性能监控磁盘使用率、I/O 吞吐量、IOPS 和读写延迟，设置水位告警并诊断性能瓶颈。混部供给监测针对单节点混合部署多负载，节点状态同整机供给，重点监控混部总负载、CPU 压力、内存压力和检测“吵闹邻居”；节点算力容量监控 CPU 实际使用算力和隔离机制保障的 CPU 份额，跟踪%Steal、上下文切换和调度延迟，加速器如 GPU 监控使用率、显存和温度，内存监控总量使用率、Cache/Buffer 和 Page Faults，网络容量同整机供给但细化容器网络消耗，节点存储容量监控容器层本地卷或共享存储使用及性能；统一视图与策略综合指标形成节点资源饱和度评分，设置弹性策略如调度和驱逐，关键资源逼近瓶颈时限制新负载调度或驱逐低优任务，确保核心负载稳定。

此外，算力供给运维管理通用能力支撑自动化流程，包括 ITSM 审批流程能力实现标准化工单处理，安全扫描与防护能力集成漏洞管理和威胁响应，分布式任务管理能力协调多节点运维操作，这些能力通过自动化平台无缝集成，提升整体运维的可靠性和效率。通过上述措施，算力供给自动化实现了资源运营的智能化和协同化，为算力生态提供坚实基础。

2.6 通用算力消费能力

通用算力消费能力是 SRE 综合算力调度体系中的核心概念，它本质上是对各种算力使用场景中公共的、可复用的能力进行抽象和沉淀，并以服务治理的形式呈现给使用者。这种能力的设计灵感来源于公用事业模型，例如水电供应，用户无需了解发电或净水的具体技术细节，只需通过标准化接口即可获得稳定、可靠的服务。通过这种方式，算力消费变得简单、高效，且易于管理，从而支持企业在多云和混合云环境中实现资源的最优利用。

服务治理作为通用算力消费能力的具体体现，包含八个主要方面，这些方面共同构成了一个完整的治理框架。接下来，我们将详细阐述这些能力，首先聚焦于四个核心能力，这些能力在运维和开发实践中具有极高的优先级和实用性。

环境治理能力依赖于平台化手段，屏蔽多云资源差异，实现研发、测试、运营等运行环境的统一管理。这项能力的核心价值在于消除云厂商间的异构性，使得 SRE（Site

Reliability Engineering）人员无需为不同平台编写和维护多套管理脚本，从而大幅提升运维效率的一致性。通过环境治理，企业可以确保各类环境的标准化和可控性，减少因环境差异导致的部署错误和运维复杂度。

容量管理能力依据业务运营状况，动态平衡算力需求与供给之间的矛盾，以达成成本最优。这项能力关注资源使用的实时调整，SRE 可以利用它来确保业务高峰期的稳定性，同时避免资源闲置造成的浪费。通过智能调度和预测算法，容量管理能够自动缩放资源，优化成本支出，并支持企业实现绿色计算和可持续发展目标。

全链路灰度发布能力通过不同泳道方式，支持模块级全链路灰度发布，以及流量切换等配套功能。这是现代服务治理的关键组成部分，SRE 可以通过它实现平滑、无损的业务变更，极大降低发布风险。这项能力允许逐步 rollout 新功能或修复，确保在全面部署前进行充分测试，从而提升系统的可靠性和用户体验。

容灾规划能力从基础架构层面支持容灾，并提供高效、灵活的应用级容灾接入方案。对 SRE 而言，这项能力使得快速构建符合业务需求的灾备体系成为可能，确保业务连续性和高可用性。它通过标准化容灾流程和自动化工具，减少人工干预，提高灾难恢复的效率和可靠性。

除了上述四个核心能力，服务治理还涵盖其他四项重要能力，这些能力在整体架构中扮演支持性角色，进一步增强算力消费的通用性和效率。

组件管理能力支持对接中间件、数据库管理系统等组件，并在服务治理体系中实现无缝调用。这项能力简化了第三方服务的集成，SRE 可以通过统一接口管理这些组件，减少配置复杂度和维护负担，提升系统的可扩展性和互操作性。

对接持续集成能力支持对接 CI（持续集成）体系，实现从研发环境到测试环境，以及从代码到容器的全链路打通。这项能力自动化了软件交付流程，SRE 可以利用它来加速部署周期，确保代码质量和环境一致性，从而支持 DevOps 实践和快速迭代。

持续观测能力支持对接可观测体系，对服务的性能、状态、日志实现实时监控、跟踪与告警。这项能力提供了全面的 visibility into system health，SRE 可以通过它快速识别和响应问题，减少平均修复时间（MTTR），并提升服务的稳定性和用户体验。

架构托管能力支持将应用架构和模块依赖通过代码化、配置化的方式描述，实现应用架构的标准化和可视化观测。这项能力使得架构管理更加透明和可控，SRE 可以通过它来优化系统设计，支持架构演进和审计，并降低技术债务。

通过构建以场景聚焦、高度集成和简单易用为核心原则的服务治理体系，能够有效抽象通用算力消费能力，从而为 SRE（站点可靠性工程）提供坚实的基础支撑。场景聚焦原则确保治理体系紧密围绕开发者实际使用场景展开，通过全流程保障服务的高效管理与稳定运行，使得算力资源能够被精准匹配到具体业务需求中，减少不必要的资源浪费。高度集成原则依托平台化能力，将分散的治理功能整合为统一体系，基于稳定的平台基础实现服务治理功能的快速迭代，这有助于将算力消费抽象为可复用的模块，提升整体工程效率。简单易用原则则通过屏蔽云原生、容器、Kubernetes 等底层复杂概念，让开发者能够直观地进行服务管理，降低了技术门槛，使 SRE 无需陷入琐碎的操作细节。

基于这些原则抽象出的通用算力消费能力，使 SRE 得以从日常运维负担中解脱，将更多精力投入到高附加值的工程实践领域。例如，在研发领域治理工程实践中，SRE 可以专注于代码质量、持续集成/持续部署（CI/CD）流程的优化；在数据治理及 AI 相关工程实践中，能够深入参与数据管道设计、模型部署与监控；在安全领域工程实践中，可强化威胁检测、合规性审计等关键环节；在行业应用治理工程实践中，则能推动定制化解决方案的落地，满足不同行业的特定需求。

这种抽象化与集成化的服务治理体系，使得 SRE 能够以高效低成本的方式，扩展覆盖多领域、多场景的软件工程方案。通过将通用算力消费能力标准化和模块化，不仅提升了工程实践的敏捷性与可扩展性，还为互联网技术向实体行业的渗透奠定了坚实基础，最终助力产业升级与数字化转型。

2.6.1 环境治理

环境治理体系旨在为用户在不同研发和测试场景下提供统一、快速、稳定且低成本的运行环境支持。该体系通过分层架构设计，确保算力的高效管理和灵活适配，从而提升 SRE 工作效率与 SRE 系统可靠性。

环境治理也需要具备分层结构，整体上主要分为三个关键层次。用户场景层提供用户消费场景封装，覆盖了个人测试环境、团队测试环境、云研发环境、游戏私服环境、远程容器环境以及预发布环境等多种类型，这些环境能够灵活适配应用生命周期过程中的不同阶段需求，为用户提供针对性的支持。平台功能层整合了 Console 登录、IDE 接入、BF 登录、流程管理、版本管理、权限管理、实例接入、中间件接入和集群接入等标准化功能，形成一个统一的管理接口，保障操作的一致性和可控性。能力集成层用于集成第三方服务，来完善环境治理能力，通过垂直领域的定制化扩展，形成完整的领域适配体系，增强系统

的整体适应性，例如对接数据库管理、用户配置管理、文件传输管理和中间件管理等专业化支持服务。

环境治理能力的构建需要具备三大特性，确保其在实践中的高效性与可靠性。一是灵活快速的特性，通过算力自动化接入机制实现，支持实例、集群、中间件、网络、存储、可观测性等的快速部署与配置，同时框架的可扩展性允许用户自定义组件接入，从而提升环境的适应性和部署效率。二是稳定高效特性，依赖于标准化基础设施即代码（IaC）定义环境，并结合标准化流程与版本管控框架，对各类实体资源的生命周期进行系统化管理，确保环境的稳定运行和一致性维护。三是安全低成本特性，通过虚拟集群和资源池能力的联动，提高算力复用效率，并与全链路灰度发布机制结合，实现组件能力的复用，显著降低算力消耗和运营成本，最终达成经济高效的环境治理目标。

环境生命周期管理是环境治理的核心能力，其设计旨在为 SRE 团队提供一套标准化、自动化的消费实践方案，以实现环境从创建到销毁的全流程精细化管控。本方案基于三大核心组件构建：基础设施即代码定义、标准流程管控和插件管理，确保环境管理的一致性、可扩展性和高效性。以下将从组件设计、流程集成和实施指导等方面展开说明。

在基础设施即代码定义组件中，应采用声明式配置方法，将环境抽象为代码化模板。环境定义模块应明确算力规格、网络拓扑和依赖关系，避免配置漂移；依赖定义模块需规范中间件版本、系统组件和工具链要求，确保环境间一致性；区域定义模块应支持多可用区和多地域部署，满足高可用需求；配置定义模块则统一管理环境变量、密钥和业务参数，并通过版本控制实现可追溯性。设计时，建议将 IaC 配置存储在版本管理系统中，以便审计和回滚。

标准流程管控组件应建立规范化的操作体系，覆盖环境生命周期的关键环节。申请流程需实现环境的自动化创建和初始化，缩短准备时间；销毁流程应确保算力的彻底清理，防止泄漏；升配流程需支持弹性扩缩容，根据业务负载动态调整算力；迁移流程应实现环境在不同基础设施间的平滑切换，保障业务连续性；自定义流程机制允许团队根据特定场景扩展流程类型，增强灵活性。实施时，建议将流程与 CI/CD 流水线集成，实现端到端自动化。

插件管理组件通过 gRPC 协议提供标准化扩展接口，将各类云平台和工具抽象为统一操作插件。gRPC 调用机制应确保通信的高效和原子化操作；生命周期管理功能需涵盖插件的注册、发现、版本控制和状态监控，保证稳定性。设计时，应优先集成主流云服务（如 AWS、腾讯云），并通过插件体系实现算力管理的统一视图。

在整体流程设计中，业务用户或 CI/CD 系统应通过产品界面提交环境需求，平台自动将需求转换为 IaC 配置并纳入版本管理。随后，触发自动化申请流程创建环境，运行期间持续监控和维护，最终通过销毁流程完成算力回收，形成闭环管理。实施团队应确保数据流从版本管理开始，经环境生命周期管理核心，通过 gRPC 连接插件体系，实现连贯的自动化操作。

在环境治理体系的设计中，环境隔离与算力复用是保障系统安全性与成本效益的关键支柱。通过标准化的技术架构和智能化的管理流程，平台能够实现风险可控与算力高效利用，为多租户环境提供稳定可靠的运行基础。本方案旨在系统阐述相关机制的实施要点与优化策略。

在安全控制方面，平台构建了多层次防护体系。数据安全机制依托快照技术实现环境数据的增量与全量备份，确保数据丢失或损坏时可快速恢复；同时，基于应用配置管理的敏感数据脱敏流程，有效防止非生产环境下的信息泄露风险。环境隔离采用分级技术方案：通过 vcluster 提供独立的控制平面，实现业务配置的逻辑隔离与访问阻断；利用 lxcfs 技术隔离容器进程与母机视图，避免跨环境信息渗透；结合 Kubevirt Vm Pod 的虚拟化隔离能力，进一步提升内核级算力访问的安全性。访问控制与审计体系通过权限中心实施最小化权限原则，严格约束环境操作范围；全链路操作日志记录功能，辅以终端登入操作的 asciinema 录屏与回放支持，确保所有行为可追溯、可审计，从而强化安全治理的透明度与合规性。

在成本控制层面，平台通过算力复用与回收策略实现精细化运营。算力复用基于共享集群与算力池架构，支持算力的分时复用与弹性伸缩，降低闲置浪费；基线环境构建能力与灰度发布系统联动，可快速创建自定义特性环境，避免重复算力占用。算力回收机制集成活跃环境检测功能，支持定期销毁、策略自定义销毁及算力即时释放；针对非生产环境（如云桌面客服环境），实施按需启停策略，在非工作时间自动关闭以削减无效能耗。算力单价控制依托多云管理与容量规划策略，智能筛选低成本算力选项；通过多云负载均衡技术，根据实时价格波动自动迁移跨云工作负载，持续优化运营成本，提升整体经济性。

通过环境隔离与算力复用的协同设计，平台不仅确保了多租户环境的安全隔离与稳定运行，还实现了成本结构的动态优化。这种集成化治理方法显著降低了运维复杂度，为业务敏捷迭代与创新探索提供了可持续的技术支撑。在实际部署中，环境治理通常用于开发环境、联调环境、测试环境、云研发环境等具体场景，需要结合具体场景特性细化策略参数，以最大化治理效能。

测试环境管理是环境管理能力的一个典型应用案例。在互联网领域，尤其是涉及复杂应用开发和算力调度时，传统模式的测试环境管理面临着显著挑战，主要表现为环境配置分散、版本不一致以及资源利用率低，进而影响开发迭代速度和系统稳定性。为应对这些挑战，需要引入结构化的环境治理方案。该方案的核心设计围绕标准化、隔离性和自动化构建，其架构包括一个核心的环境治理模块，用于规划环境蓝图和管理组件版本，确保配置的一致性；同时提供共享的基础组件服务，如数据库和缓存，以供多个环境复用。方案还设立了集成了所有组件的稳定测试环境作为验证基准，并为每位开发人员提供逻辑隔离的独立环境实例，通过派生配置避免相互干扰，最终将测试流程集成以形成自动化闭环。从技术实现层面，主要依托版本控制工具确保环境配置的可追溯性，利用容器化等技术实现配置派生与资源逻辑隔离，并将基础组件封装为标准化服务以降低复杂度，同时结合算力调度系统实现环境的自动化构建与资源动态分配。这一设计可进一步扩展应用于算力密集场景，例如通过动态资源调度提高利用率，支持多版本并行测试以加速迭代，或在混合云架构中协调资源实现高可用。通过与持续集成和部署流程深度结合，该治理方案能够有效支撑互联网应用的高效开发与稳定运维。

在 AI 时代，AI 数字人环境已成为环境治理的另一个重要应用场景。AI 数字人环境旨在通过一个能够持续不间断运行的 AI 代理平台，为个体用户提供高度可用的智能化助理服务。

AI 数字人环境存在两种主要构建思路。一种是在个人终端设备上部署数字人方案，但该方式通常面临显著局限：其安全防护能力相对薄弱，容易受到威胁；受限于单点设备的计算与存储资源，处理效率存在瓶颈；同时，该方案对用户的技术运维能力有一定要求，普及门槛较高。与之形成对比的是基于集中式算力调度的数字人环境。这种方案通过云端资源池进行统一的调度与管理，能够提供更坚固的安全保障、更优的资源利用效率和更强大的服务性能。此外，它极大地降低了对用户的技术能力要求，使更广泛的群体能够便捷地享受高级别的 AI 助理服务。

集中式算力调度的数字人环境基于综合算力调度系统实现。用户通过聊天工具界面与系统交互，驱动多个云桌面助理协同执行任务。这些助理能够处理多样化的日常工作，例如编写代码、进行软件测试（包含 GUI 自动化软件测试）、撰写文档以及整理邮件；在一些企业中甚至能够覆盖 RPA（企业流程自动化）相关工作，例如财务对账、考勤统计、订单处理等。云桌面助理后端部署了多个云电脑实例，运行 OpenClaw 等程序以处理复杂计

算与任务分配，同时依托弹性可扩展的算力资源池提供稳定支持，确保整个系统的高效运行。

AI 数字人环境案例突出了智能技术在环境治理中的应用潜力。通过整合聊天驱动、云桌面助理和分布式算力资源，平台实现了人力资源与计算资源的精准匹配，减少操作冗余，显著提升环境治理项目的执行效率。这不仅为个人工作带来效率飞跃，也为组织层面的环境治理工作提供了可扩展、低成本的数字化解决方案，契合当前绿色科技与自动化智能化的前沿趋势。

2.6.2 容量管理

容量管理作为服务治理重要能力之一，其本质是通过科学方法平衡算力需求与供给之间的矛盾，最终实现成本最优的目标。随着互联网业务规模的扩大和架构复杂度的提升，算力的管理已从传统的静态配置转向动态化、数据驱动的模式。在这一过程中，需要综合考虑不同角色的关注点：业务或产品团队更注重用户体验的流畅度、服务稳定性以及成本效益的可控性；架构师和开发者则聚焦于系统性能瓶颈的识别与关键服务的容量规划合理性；而 SRE（站点可靠性工程师）则致力于在算力利用率与成本之间寻求平衡，同时建立及时的容量预警与风险管控机制。

容量管理的实施围绕三个关键功能需求展开，形成一个持续优化的闭环流程。容量观测是起点，它通过监控工具对系统运行状态进行实时采集与分析，及时发现算力使用异常或瓶颈。容量预估则基于历史数据、业务趋势或机器学习模型对未来算力需求进行预测，为算力分配提供依据。容量调整作为闭环的最后一环，根据观测及预估结果动态进行算力调配，例如通过弹性扩缩容机制应对流量波动。这三者相互关联，构成一个完整的反馈循环，确保系统既能满足业务需求，又避免算力浪费。

通过有效的容量管理，企业能够实现多方面的收益。首先，在性能与稳定性方面，系统能够保持快速、流畅且可靠的服务状态，显著减少卡顿、延迟或服务不可用的情况。其次，在成本优化上，通过精细化的算力调度与利用率提升，业务运营成本得以降低，服务性价比显著提高。最后，在风险管理层面，通过对容量相关数据的持续监控与分析，潜在问题能够提前暴露并实现预先干预，使系统运行风险处于可控状态。这种管理方式不仅适用于高并发的电商场景、流量波动大的媒体平台，也在 AI 训练、边缘计算等新兴领域发挥着重要作用，体现出其广泛的技术适应性与实践价值。

容量观测是容量管理的起点和基础，它主要包括对系统实时状态的监测以及基于历史数据的规律挖掘。实时状态监测涉及通过可观测系统持续收集关键指标，例如CPU使用率、内存占用、网络流量和存储I/O，从而及时发现算力瓶颈或异常情况，避免系统过载导致的性能下降。同时，基于历史数据挖掘规律则依赖于数据分析技术，如时间序列分析和机器学习算法，从过往运行数据中识别趋势、周期性和关联模式，这有助于理解业务负载的长期行为，为容量规划提供数据支撑。例如，在电子商务平台中，通过观测“双11”大促期间的历史流量模式，可以识别出峰值访问的规律，为后续预估和调整奠定基础。技术背景上，容量观测源于运维监控（DevOps）文化的普及，它强调可观测性（Observability），即通过日志、指标和追踪等手段实现系统内部状态的透明化。

容量预估是在观测基础上，通过数据分析和建模来预测未来算力需求的过程。其核心包括依据数据分析建立容量评估模型，以及基于当前状态利用模型进行预测。建立评估模型通常采用统计方法（如回归分析）或机器学习技术（如时间序列预测模型ARIMA或深度学习模型LSTM），这些模型能够结合历史数据和实时输入，量化算力需求与业务指标（如用户并发数、交易量）之间的关系。然后，利用模型对未来的容量进行预测，例如预测下个季度的服务器需求或带宽峰值，从而支持前瞻性的算力采购或调度决策。场景扩展上，在视频流媒体服务（如Netflix）中，容量预估可用于预测新剧集上线时的流量激增，提前部署边缘节点以降低延迟。技术背景方面，容量预估得益于大数据分析和人工智能的发展，它体现了从被动运维向主动预测的转型，有助于降低算力闲置成本并提升系统韧性。

容量调整是容量管理的执行环节，旨在通过动态调控实现算力的优化分配，主要包括精准调控冗余算力以提升利用率，以及打造弹性算力以应对突发需求。精准调控冗余算力涉及算力调度算法（如Kubernetes中的自动扩缩容机制），通过对闲置或低效算力进行重新分配或休眠，减少浪费并提高整体能效。例如，在云计算环境中，根据负载波动自动调整虚拟机实例数量，实现“按需付费”。另一方面，打造弹性算力则强调系统的可扩展性，通过云原生技术（如容器化和无服务器计算）快速响应业务峰值，如社交网络在热门事件爆发时自动扩容服务器集群。场景上，在线教育平台在疫情期间需应对用户量暴增，弹性算力可确保服务不中断。技术背景上，容量调整与自动化运维和绿色计算理念紧密相关，它推动了算力池化和动态调度的实践，提升了企业的敏捷性和可持续性。

容量观测与预估，在具体实践上往往是一体的，它们之间是相辅相成的关系。其中容量观测的内容通常包含以下三个方面：一是系统监测数据，例如OS相关监测数据，平台服务的可观测数据，APM、NPM等数据；二是应用级观测数据，例如应用metric，调用链，

日志，以及业务用量数据等；三是算力消费相关数据，例如算力利用率，装箱率，容量水位等。在数据的收集维度上，既要有历史数据，也需要考虑增长趋势数据和特殊运营活动数据。历史数据主要指上述观测内容的周期性数据，峰值数据，节假日及特殊事件时期的数据；增长趋势数据包括业务自身的趋势以及行业趋势等。

得到足够的观测数据之后，就可以进行算力容量预估，预估内容涉及 CPU 内存配额预估、IO 性能预估以及实例个数预估，这些要素直接关系到计算资源的分配效率。流量预估则涵盖出入口流量预估、峰值流量预估、流量分布预估以及东西向流量预估，这对于网络资源的规划尤为关键。此外，存储容量预估包括缓存容量预估和数据容量预估，确保存储系统能够满足业务增长的需求。预估过程中需要借助科学的预估方法，例如时间序列法中的移动平均法、加权移动平均法和指数平滑法，这些方法适用于处理具有时间依赖性的数据；回归分析法包括先行回归预测和非线性回归预测，能够捕捉变量间的关联；机器学习法则如决策树回归和神经网络，可处理复杂非线性模式，提升预测的准确性。

在真实场景实践中，容量调整通常分为分析和调整两个阶段。容量分析阶段侧重于从多维度数据中评估当前算力使用情况，识别优化空间，并制定科学调整策略。这一阶段依赖于大数据平台和 AIOPS 技术，其数据输入涵盖业务指标（如在线用户数、请求量）、算力用量（如 CPU、内存使用率）以及算力供给与消费趋势。这些原始数据通过计算平台进行清洗、聚合，形成标准化容量数据集，确保数据质量。AIOPS 模块则利用机器学习算法，如时间序列预测和回归模型，进行深度分析，输出实例规格优化、副本数调整、装箱率提升等量化策略，并推送至 SRE 团队审核。从技术背景看，AIOPS 源于运维自动化的发展，结合了监控、大数据和 AI 技术，例如 Facebook 的 Prophet 模型用于负载预测，Kubernetes 生态中的 Prometheus 提供实时数据采集，通过持续学习适应业务波动。在实际场景中，如电商大促期间，容量分析可提前预测流量峰值，自动建议扩容方案，例如通过历史数据分析识别支付服务需增加副本数，同时降低非核心任务算力配额，实现成本节约。

容量调整阶段则依据分析结果，通过平台化工具动态执行算力优化，并进行灰度验证以确保稳定性。这一阶段强调自动化和可控性，覆盖从应用层到基础设施层的多维调整。具体调整方式包括自动扩缩容机制，如 GPA 基于 AI 预测动态调整 Pod 副本数，VPA 优化单个 Pod 的算力限制，以及 CA 根据集群压力自动增删节点。节点与调度优化则涉及重装箱调度，通过重新分配 Pod 提高算力密度，而算力生命周期管理则与云平台 API 集成，实现容量申请与退回，以及算力复用，如通过容器技术共享 GPU 算力。所有调整均采用灰度发布策略，先在小范围节点验证效果，监控延迟、错误率等指标，确认无误后全量推广，

形成闭环反馈，将结果重新输入容量分析模块。从技术背景看，GPA、VPA 等技术常见于 Kubernetes 生态，依赖 Custom Resource Definitions 和控制器模式，而 AI 增强的机制引入了 LSTM 等预测模型，提前触发动作以提升响应速度。在实际场景中，如在线视频平台的晚间高峰，容量调整可通过 VPA 提升转码服务算力限制，同时利用 GPA 增加网关副本，CA 则在高峰后自动缩容节点，节省成本，重装箱调度则优化低优先级任务算力分配。

容量管理作为综合算力调度的重要组成部分，通过动态调整算力来优化利用率、降低成本并确保业务稳定性，在各类应用场景中发挥着关键作用。典型应用场景涵盖在线服务、离线计算、AI 与大数据以及研发领域，每个领域均具有独特的特点和相应收益。

在线服务领域的特点包括业务流量呈现周期性波动、对高可用性有严格要求以及实时敏感性高，例如电商或社交平台在促销期间流量激增，需保证毫秒级响应。其收益主要体现在潮汐容量管理方案上，该方案能提升算力利用率并降低业务持有成本，同时通过精准预测峰值容量，提前进行扩容操作，避免高峰时段服务受损，从而保障用户体验。

离线计算领域的特点是非实时性、算力消耗大且任务可中断，这类任务如数据批处理或日志分析允许延迟执行，但算力需求密集。收益方面，通过利用在线服务闲时算力，可有效降低算力成本，并结合错峰动态调控容量策略，充分利用电价或算力低价时段，实现降本增效。

AI 与大数据领域以算力密集型、长周期占用和突发性推理需求为特点，例如机器学习训练任务需持续占用 GPU 算力，而推理服务可能因事件突发增长。收益包括结合历史数据精准预测容量需求，避免算力过度冗余造成的浪费，以及按需自动扩缩容机制，在任务高峰期快速扩容、空闲期缩容，从而节省整体成本。

研发领域的特点在于算力需求碎片化多样，且对稳定性和灵活性要求高，如开发测试环境需快速供给且避免干扰。收益表现为通过稳定性容量保障核心业务与功能稳定运行，防止算力竞争导致的中断，同时弹性容量机制能满足效率与成本的平衡，按需分配算力以支持敏捷开发。

总体而言，容量管理通过这些场景化应用，实现了算力优化与业务需求的紧密结合，为 SRE 数字化运营体系提供坚实基础。

2.6.3 流量治理

流量治理是综合算力调度体系中服务治理体系的“灵魂中枢”，它远非可选项，而是通过精细化的流量调度、防护机制和容错策略，为应用提供高可用性、高绩效和高安全性的

基础设施支撑。在现代分布式计算环境中，随着应用架构日益复杂，流量治理需要应对多方面的挑战。首先，架构复杂性要求流量治理能够处理微服务、容器化平台带来的服务依赖和通信问题，例如通过服务网格实现动态路由和分布式追踪，以确保在庞大服务网络中快速定位并解决故障。同时，动态性与规模化要求流量治理具备弹性伸缩能力，能够根据流量潮汐特征自动调整负载均衡策略，并利用限流算法防止系统过载，从而在突发流量场景下维持稳定性能。此外，业务连续性要求流量治理集成容错和灾难恢复机制，如熔断降级和多活架构，以保障在故障或攻击事件中服务的无缝切换，并结合安全措施满足合规需求。

在SRE领域，流量治理的核心能力涵盖多个关键方面，共同支撑系统的可靠性与效率。首先，业务稳定性保障通过限流、熔断和隔离等机制，有效防止局部故障扩散引发的系统雪崩效应，从而确保核心业务链路的稳定运行。其次，提高算力效率依赖于基于实时流量的智能调度和弹性伸缩策略，避免资源闲置或过载，实现成本优化和应用的高效运作。此外，降低业务维护复杂度通过无侵入方式提供治理能力，减少代码复杂性，并借助可视化工具简化故障排查流程，显著提升运维效率。最后，支撑业务持续部署能力包括支持灰度发布、蓝绿部署等方法，保障业务平滑升级与快速迭代，同时为人工智能等高并发应用提供稳定入口。这些能力相互协同，体现了流量治理在SRE综合算力调度系统中的综合价值。

流量治理体系的建设涵盖了基础能力和场景能力两部分，基础能力包括服务发现、负载均衡、流量控制和流量导引。服务发现是自动注册和发现服务实例的机制，允许服务动态感知其他实例的位置，如IP地址和端口。在微服务架构中，由于实例频繁弹性伸缩或故障，手动配置不可行，服务发现通过注册中心（如Consul或Etcd）实现实时更新，避免硬编码依赖。例如，在电商平台中，订单服务调用库存服务时，服务发现能自动感知新扩容的实例，无需重启服务。技术实现上，它基于DNS或客户端负载均衡，关键组件包括注册中心和健康检查机制，优势在于提高系统弹性和减少运维成本。负载均衡则将传入请求分发到多个服务实例，以优化资源使用、最大化吞吐量并避免单点故障。在高并发场景下，单一实例易成为瓶颈，负载均衡通过算法（如轮询或最少连接）分散压力，提升性能。例如，在线视频流媒体服务使用负载均衡将用户请求分配到最近的CDN节点，减少延迟。算法类型包括静态和动态方式，部署模式可以是集中式或分布式，最佳实践是结合健康检查自动剔除故障实例。流量控制通过限制请求速率、并发数或带宽来防止系统过载，确保稳定性。突发流量可能导致服务雪崩，因此流量控制使用令牌桶或漏桶算法平滑流量，保护后端资源。在秒杀活动中，API网关可实施流量控制，限制用户请求频率以防刷单。控

制维度包括 QPS 和并发连接数，实现工具如 Nginx 限流模块，支持细粒度策略。流量导引则将请求引导到特定服务版本或路径，用于灰度发布或 A/B 测试。在持续交付中，需安全验证新功能，流量导引基于规则（如 HTTP 头）动态路由流量。例如，金融 App 发布新功能时，仅将 10%流量导引到新版本监控错误率。路由策略包括权重或金丝雀发布，工具如 Istio 的 VirtualService，价值在于降低发布风险。

场景能力构建在基础能力之上，解决分布式系统中的具体问题，包括全链路灰度、服务限流、服务熔断、智能路由、故障替换、认证加密、过载保护和超时重试。全链路灰度在整条服务调用链中实现灰度发布，确保特定用户仅访问新版本服务。由于微服务调用链复杂，局部灰度可能因依赖不一致而失败，全链路灰度通过传播上下文（如追踪 ID）确保一致性。例如，社交平台推出新算法时，仅对内部员工流量开启全链路灰度测试兼容性。实现机制涉及 OpenTracing 传递标签，需所有中间件支持上下文传播，优势是减少全网影响。服务限流针对特定服务或接口限制请求速率，防止资源枯竭。DoS 攻击或业务高峰可能拖垮服务，限流作为熔断的补充提供细粒度保护。例如，支付 API 对非 VIP 用户设置低速率限流，优先保障高价值客户。限流类型包括本地和全局限流，算法如固定窗口，常与认证结合实现基于身份的限流。服务熔断在检测到目标服务失败率过高时自动切断请求，避免级联故障。分布式系统中，单个服务延迟可能导致雪崩，熔断模式（如断路器）快速失败并降级响应。例如，电商网站在下单服务超时时熔断，返回错误页面。熔断状态机包括关闭、打开和半开状态，实现库如 Netflix Hystrix，配置参数基于错误率动态调整。智能路由基于动态条件（如负载或地理位置）将请求路由到最优实例。传统路由静态低效，智能路由结合实时 metrics（如延迟）实现自适应优化。例如，全球游戏服务将玩家流量路由到延迟最低的数据中心。策略包括基于内容的路由或地域亲和性，数据源来自监控系统如 Prometheus，适用于多活部署。故障替换在检测到主服务不可用时自动切换到备用服务，确保业务连续性。单点故障是分布式系统大忌，故障替换通过健康检查和冗余设计实现高可用。例如，数据库主节点宕机时，从节点自动提升为主节点。切换机制基于投票协议如 Raft，挑战包括数据一致性，需定期故障演练验证。认证加密确保服务间通信的安全性和完整性，包括身份验证和数据加密。微服务间通信可能被窃听，TLS/mTLS 提供传输层安全，JWT/OAuth 处理应用层认证。例如，医疗云平台使用 mTLS 加密通信以满足合规要求。标准协议包括 TLS 1.3 和 OAuth 2.0，工具如 Istio 自动 mTLS，管理需证书轮换。过载保护通过自适应机制（如自动扩缩容）防止系统因过载崩溃。突发流量可能超出容量，过载保护结合监控和反馈循环动态调整资源。例如，云函数服务在流量峰值时自动扩容实例。策

略基于队列长度或 CPU 使用率，实现如 Kubernetes HPA，优化需避免频繁伸缩抖动。超时重试为服务调用设置超时时间并在失败时重试，提升请求成功率。网络抖动常见，合理超时和重试避免无限等待。例如，移动 App 调用 API 超时后重试 2 次，间隔指数递增。重试策略包括指数退避，非幂等操作需禁用重试，工具支持如 gRPC 自动重试。

上述这些能力的具体落地，依赖一些共性实现，分别是流量控制方案和流量染色方案。流量控制技术尤为重要，它通过智能管理服务间通信来保障系统的稳定性、可用性和性能，是流量治理诸多能力的实践基础。例如，流量控制可以防止系统过载、支持平滑的服务发布，并实现复杂的路由策略。下文将基于常见实践，详细阐述两种主流的流量控制实现方案：一种是通用方案，基于服务网格技术（如 Istio），适用于多语言和云原生环境；另一种是特殊方案，针对 Java 应用程序通过字节码注入方式实现，适用于需要深度定制的场景。

通用流量控制方案是基于 Istio 的服务网格实现。这种方案的技术背景源于服务网格的兴起，它是一种专门处理服务间通信的基础设施层，通过解耦应用逻辑与网络治理来提升灵活性。Istio 作为开源服务网格的典型代表，提供了丰富的流量管理功能，如负载均衡、故障注入和熔断机制，特别适合 Kubernetes 等云原生平台。其核心架构包括网络控制面和数据面 Sidecar 代理；控制面负责配置管理，例如通过 Pilot 等组件定义 VirtualService 和 DestinationRule 资源，并将规则下发到数据面。数据面则由轻量级代理（如 Envoy）组成，这些代理以 Sidecar 形式部署在每个应用实例旁，拦截所有流量并执行动态路由、限流等操作。此外，该方案基于声明式 API 实现自动化，用户可通过 YAML 文件定义规则，无需修改应用代码。在场景扩展方面，这种方案支持金丝雀发布，例如将少量流量（如 10%）导向新服务版本以逐步验证稳定性，也适用于故障恢复，通过超时和重试机制提升系统韧性。它还能扩展至混合云环境，利用 Istio 的多集群功能统一管理跨云流量，展现出高度的适应性和可维护性。

特殊流量控制方案，主要是针对 Java 程序的字节码注入方式。这种方案的技术背景与 Java 平台的特性相关，字节码注入是一种在运行时动态修改类字节码的技术，常用于实现非侵入式的功能增强，如应用性能管理或自定义流量控制。它通过 Java Agent 机制实现，在 JVM 启动时加载代理 JAR 包，并利用 Instrumentation API 在类加载过程中注入控制逻辑，适用于对性能有高要求或需要精细控制的 Java 应用场景。架构上，该方案同样包含网络控制面，但规则可能通过配置中心（如 Nacos）下发，而非标准服务网格组件。应用运行在容器（如 Docker Pod）中，通过挂载方式连接 Java Agent，代理在关键点（如 Servlet 过滤器或 RPC 入口）拦截流量，添加限流或路由逻辑。在场景扩展中，这种方案适用于细粒度

流量控制，比如对高并发 API 实施每秒请求数限制，防止雪崩效应；它也常用于遗留系统现代化，在不修改业务代码的前提下为传统 Java 应用添加流量治理功能。此外，它可以与服务网格结合，形成分层控制，例如用 Agent 处理内部逻辑，而 Istio 管理外部流量，这在异构环境中尤为实用。

以上两种流量控制方案也可以混合使用，并通过统一的控制面来解决不同类型应用的混合流量治理问题。统一控制面作为策略决策中心，集中管理限流、熔断等规则，并支持与监控系统以及一些开源组件集成，实现兼容性扩展。例如，通过 xDS Adapter 实现协议转换，将通用策略适配为云原生标准；通过集成 Nacos 服务发现机制，将流量策略注入传统应用运行时，实现无缝兼容。流量治理控制面通过集成这些组件支撑了很多常见的流量控制策略，例如：熔断与降级在服务错误率超标时自动切断流量并返回降级响应，防止故障扩散；限流与过载保护通过算法限制请求量，避免系统崩溃；流量路由与染色基于标签定向分配流量，支持灰度发布；超时与重试控制设置合理时限和重试机制，提升可靠性；负载均衡动态分配流量到健康实例；故障注入与演练则通过模拟异常检验系统韧性。这些策略在混合环境中需灵活适配，例如在电商大促场景下，熔断降级可保障核心链路，而金融系统中则需严格限流以符合合规要求。

流量染色是流量治理能力体系中另一个重要的共性实现，主要用于对入口流量进行标识和分类，从而实现精细化的路由控制。随着云原生和微服务架构的普及，流量染色技术已成为服务网格和 API 网关的标准功能，支持 A/B 测试、金丝雀发布、故障隔离等场景。其核心思想是通过为流量添加特定标签（如版本号或环境标识），动态路由请求到目标服务实例，提升系统的可观测性和可靠性。在成熟的 SRE 平台工程实践方案中，一般支持两种流量染色方法：基于条件路由的流量染色和基于插件的流量染色。

基于条件路由的流量染色是一种静态染色机制，它依赖于预定义的固定规则，如域名或 HTTP 头部信息，对流量进行标识。在技术原理上，网关在接收用户请求时，会根据规则匹配结果添加染色标签，例如通过域名染色，访问“www.base.com”的请求被标记为基线版本（如 v1），而访问“www.gray.com”的请求则被标记为灰度版本（如 v2）。类似地，基于 Header 关键字染色则通过解析请求头中的字段（如“Env=gray”）来实现动态路由。工作流程通常包括用户请求到达网关、规则匹配、标签添加和路由转发等步骤。这种方法优势在于规则配置简单、性能开销低，适用于流量划分明确的场景，例如 A/B 测试或环境隔离。在实际应用中，它可以扩展用于电商平台的 UI 版本比较，或通过不同域名实现开发与生产环境的隔离，避免误操作。

基于插件的流量染色是一种动态、无侵入的染色机制，它通过网关控制面下发灵活规则，实现细粒度流量控制。技术原理上，这种方法依赖于动态规则匹配，例如基于请求属性如 Cookie 中的“cookie_uid”值进行条件判断：当“cookie_uid”小于等于 100 时，染色为 v1；在 1000 到 2000 之间时，染色为 v2；大于 2000 时，则染色为基线版本。匹配成功后，网关自动添加相应 Header（如“x-lane-label”），而业务服务无需感知染色逻辑，体现了无侵入设计。工作流程涉及用户请求经网关转发至控制面、规则匹配、标签添加和路由执行。这种方法的灵活性高，支持实时调整，适用于复杂场景如金丝雀发布或个性化路由。例如，在金丝雀发布中，可以根据用户 ID 范围逐步灰度新功能；在社交平台中，则可基于用户地域实现就近路由，提升用户体验。

以上为流量治理能力的具体建设思路，通过对流量治理体系的分层阐述，逐级剖析了每一层的职责和主要功能特性。流量控制和流量染色作为基础实现，对其不同的封装组合及不同的场景化控制策略构成了流量治理的基础能力。场景化能力则是在基础能力之上的进一步组合，其中灰度发布能力是覆盖场景最复杂，应用最广泛的能力之一。

AB 测试（A/B Testing）作为数据驱动决策的核心工具，是流量治理的重要应用场景之一。

AB 测试通过对比不同版本或策略的效果，帮助企业在数据支持下做出优化，降低风险，提升用户体验和业务指标。随着互联网业务复杂度的增加，AB 测试已成为迭代优化、精准运营和算法调优的关键手段，尤其在高并发场景下，结合流量治理技术，能确保测试的可靠性和效率。

其核心概念是一种实验方法，通过将用户流量分配到不同版本（如 A、B、C），收集并分析各版本对关键指标的影响，从而确定最优方案。这涉及实验设计、流量分配、数据收集和统计分析等技术要点，以确保结果的统计显著性。流量治理是实现可靠 AB 测试的基础，关键技术包括流量调度与路由、流量分组与分层，以及流量染色与透传。流量调度与路由控制用户请求的分配路径，基于策略将流量定向到不同测试组，设计上需结合负载均衡和服务发现，支持动态调整规则以应对高并发。流量分组与分层将用户按属性精细化划分，依赖用户画像系统和标签管理，支持多维度测试。流量染色与透传则通过在流量中添加标识信息并在系统间传递，实现全链路跟踪，技术实现涉及在 HTTP 头、RPC 调用或消息队列中嵌入数据。

AB 测试的应用场景多样。在功能上线与改版验证中，通过小流量测试评估效果后逐步发布，需设定流量百分比并监控核心指标。在运营策略对比确认中，用于比较不同促销

或推荐策略的效果，需确保策略间隔离。在算法与策略分析中，测试推荐算法、排序模型等对业务指标的影响，需设置对照组并使用统计方法分析。在产品改版与兼容性测试中，通过分流测试评估用户接受度，需关注用户反馈和性能指标。典型的 A/B 测试流程中，用户群体被导向测试节点后分流至各版本，每个版本对应特定的效果百分比，通过数据分析确定胜出方案，技术要点包括流量分割算法和实时监控。

扩展场景中，AB 测试可结合机器学习实现自动化优化，如在多臂赌博机问题中动态调整流量分配。在微服务架构下，流量治理技术可用于灰度发布和故障隔离，并可扩展至多云和边缘计算环境以支持分布式测试。技术层面的丰富体现在：流量调度可基于 API 网关或服务网格实现细粒度控制；流量分组可集成大数据平台进行实时分群；流量染色可借助分布式追踪系统增强可观测性。这些扩展提升了 AB 测试的扩展性和智能化水平，为互联网业务提供了从实验到部署的完整闭环，支持持续迭代和创新。

灰度发布能力是流量治理能力的另一个重要应用场景，在 SRE 综合算力调度中扮演着不可或缺的角色，它通过渐进式部署新版本服务，直接关系到服务可用率与用户体验的提升。这种能力允许 SRE 团队在控制风险的同时，实现平滑的业务迭代。下面将依次阐述三种常见且实用的灰度发布策略：滚动升级、蓝绿发布和全链路灰度发布，包括它们的适用场景和核心特点。

滚动升级通常适用于单一服务的变更场景，其核心过程是分批替换服务实例，逐步将旧版本升级为新版本。这种策略的资源利用率较高，因为它无需额外部署完整环境，但回滚过程通常需要分钟级时间，可能无法实现即时恢复。然而，滚动升级也存在一定风险，例如在升级期间，新旧版本可能同时处理数据，导致数据混用问题，进而引发一致性问题。此外，由于升级是分阶段进行的，部分用户可能在滚动过程中感知到服务异常，例如功能不兼容或性能波动，这要求团队在部署时加强监控和告警机制。

蓝绿发布则适用于单一服务的全量替换场景，其特点是同时维护两个完整的应用版本，即蓝环境（旧版本）和绿环境（新版本），通过流量切换实现版本更新。这种策略需要双倍资源支持，成本较高，但优势在于回滚速度极快，可达秒级，只需将流量重新指向蓝环境即可。不过，蓝绿发布也面临数据迁移的复杂性，在切换过程中容易因配置错误或数据同步问题导致出错，且全量切换可能引发短暂的服务中断，影响用户体验，因此通常建议在业务低峰期操作。

全链路灰度发布是微服务架构下应对多服务联动变更复杂场景的关键策略，特别适合复杂架构的混合业务形态。该策略通过标签化流量路由实现灰度用户流量的精准隔离，资

源利用率高，并支持微服务、消息队列等组件的灰度隔离。其回滚机制可通过调整标签实现秒级切换，效率显著。由于能确保灰度流量仅在特定链路内流转，有效避免了对主流用户的影响，数据污染风险较低，从而保障了业务连续性——仅灰度用户感知新版本，大多数用户无感。

作为流量治理的重要场景化能力实践，全链路灰度发布通过逐步引导流量的方式，实现服务新版本的平滑安全部署。流程始于前置准备：团队需完成环境搭建、代码审查、测试用例设计和监控告警配置，确保发布可控。随后部署灰度服务至独立环境，隔离风险。灰度流量验证阶段引入少量流量（如 5%）进行测试，借助 APM 工具监控链路指标。验证通过后，逐步扩大灰度流量比例（如 5%至 50%），持续监控。若运行稳定则升级基线版本，全量切换并下线旧服务；若发现问题则通过流量回切快速恢复，最小化影响。后续进行环境回收、文档更新和复盘优化。

该流程在微服务场景下优势明显：支持复杂链路的精细控制，防止单点故障扩散；资源成本低，仅需部分灰度环境，较蓝绿部署更经济；回滚效率高，快速响应问题；数据风险小，通过小流量试错避免全量数据污染；用户体验平滑，支持无感知升级和 A/B 测试。这一实践体现了 DevOps 的稳健性原则，适用于电商、金融等高可用系统，未来可结合 AIOps 实现智能化。

2.6.4 容灾规划

容灾规划是确保企业在面临自然灾害、硬件故障或网络攻击等突发事件时能够快速恢复业务运营的关键手段。随着数字化转型的深入，容灾系统已从传统的“数据保险箱”模式演进为保障业务韧性的“智能中枢”。这种演进的核心在于将容灾方案前置到算力调度层面，并结合业务级需求，推动传统容灾向智能化容灾方案转型。传统容灾方案主要侧重于灾后数据恢复，其设计基于静态、预定义的架构。在核心目标上，传统容灾旨在灾难发生后尽可能恢复数据和业务，重点在于确保数据完整性和可恢复性，而非业务连续性。这源于 20 世纪 90 年代的数据中心时代，当时业务中断容忍度较高，恢复时间目标（RTO）和恢复点目标（RPO）通常以小时或天为单位，企业主要关注数据备份以避免永久性丢失。例如，在银行系统中，传统容灾可能通过定期备份交易数据库到异地磁带库，灾后依赖人工恢复，导致业务中断数小时，这种方案适用于对实时性要求不高的场景，如归档系统。在技术架构方面，传统容灾依赖硬件冷备或定时备份，通过事件触发切换，架构通常基于物理冗余，如备用服务器或存储阵列，切换过程需手动或半自动完成。硬件冷备指备用设备处于关机

状态，灾后需启动并加载数据；定时备份则按固定周期复制数据，这种架构受限于单点故障和长恢复时间，无法适应云时代动态负载。例如，在制造业 ERP 系统中，企业可能部署冗余服务器作为冷备，灾后 IT 团队手动切换，导致生产延迟，若结合虚拟化技术可稍提升灵活性，但整体仍依赖预配置。决策方式上，传统容灾依赖人工判断或文档预案，决策过程基于预定义的应急预案手册，由运维人员根据事件严重性手动触发响应，这易受主观因素影响，响应延迟高，且预案往往静态、更新不及时，难以应对复杂故障场景。例如，电商平台在流量突增导致服务器宕机时，运维团队需查阅预案手册，人工决定是否切换到备用数据中心，过程可能耗时 30 分钟以上，虽可结合监控工具辅助决策，但本质仍非智能化。在成本效率方面，传统容灾算力闲置率高，持有成本高，备用资源常处于空闲状态，导致资源浪费；硬件采购和维护成本高昂，这源于“1:1”或“N:1”的冗余配置，算力利用率不足 20%，在资本支出模型中，成本随业务规模线性增长。

相比之下，智能化容灾方案以业务韧性为核心，通过云原生、AI 驱动等技术实现故障预测、自愈和弹性调度。在核心目标上，智能化容灾旨在确保业务持续不断，实现故障预测与自愈，重点从“恢复”转向“预防”，确保业务在故障发生时无缝运行。这受云计算和 DevOps 文化影响，企业要求 RTO/RPO 接近零，智能容灾借鉴 AIOps 理念，通过实时监控和机器学习预测风险。例如，在在线教育平台中，智能容灾可实时检测网络拥堵，自动将用户流量调度至健康节点，避免课程中断，并可扩展至物联网场景，实现设备故障预测与自愈。技术架构方面，智能化容灾采用云原生、软件定义的方式，支持跨云跨平台弹性算力调度，架构基于微服务、容器化和无服务器计算，实现资源动态分配。云原生技术如 Kubernetes 支持应用弹性扩缩容，软件定义存储和网络解耦硬件依赖，跨云调度避免供应商锁定，提升韧性。例如，金融科技公司使用 Kubernetes 集群结合多云策略，故障时自动切换算力至备用云，恢复时间达秒级，并可扩展至边缘计算处理本地化故障。决策方式上，智能化容灾由算法或 AI 驱动智能决策，利用大数据分析和机器学习模型自动评估风险、触发切换或修复动作，减少人为错误，这符合 AIOps 趋势，AI 算法可分析历史故障数据预测硬件老化或攻击模式。例如，电商平台通过 AI 监控用户行为，预测 DDoS 攻击并自动调用云安全服务调整算力，可扩展至自动驾驶领域实现实时决策。成本效率方面，智能化容灾按需分配算力，显著降低成本，采用按使用量付费模式，减少闲置，自动化降低运维和 SRE 人力成本。云计算的运营支出模型和弹性计费使成本与业务负载匹配，智能调度优化资源利用率，例如初创公司使用 Serverless 容灾方案，仅在故障时激活算力，日常成本降低 70%，并可结合绿色计算优化能耗。

容灾从传统向智能化的演进受多重技术因素驱动，包括云计算普及提供按需、弹性基础设施，使跨云容灾成为可能；AI 与大数据发展赋能智能预测和决策；业务需求变化推动高可用性要求；算力调度创新如容器编排工具使容灾前置到调度层。未来，智能化容灾可应用于混合云环境，如企业结合私有云和公有云实现灾备均衡；在行业特定应用如医疗领域确保电子健康记录系统高可用性。总之，容灾规划的演进本质是算力调度与业务容灾的深度融合，企业应逐步迁移至智能化方案以提升韧性。

容灾规划作为确保业务连续性和系统可靠性的重要能力，对于应用服务治理至关重要，任何中断都可能引发严重的经济损失和用户信任危机。SRE 团队作为保障系统稳定性的关键角色，通过容灾规划获得能力赋能，能够显著提升故障应对效率、降低运营成本，并增强整体韧性。容灾规划源于传统 IT 系统的业务连续性需求，但随着云计算、微服务架构和分布式系统的普及，其内涵已扩展到自动化、弹性和智能化领域。SRE 团队的核心职责之一是平衡系统可靠性与实施成本，而容灾规划集算力调度、自动化与监控以及成本效益等众多能力的结合。例如，算力调度利用冗余资源实现动态分配，避免单点故障，这与 SRE 的故障预防理念高度契合；自动化流程减少了人为错误，全链路监控则支撑了 SRE 的“可观测性”原则；同时，通过算力复用和精细化策略，容灾规划帮助 SRE 团队在保证可靠性的同时控制 IT 成本，符合 DevOps 中的“效率文化”。

在部署架构级容灾方面，容灾规划不受现有业务架构限制，在应用部署时即融入容灾体系，从而大幅节约人力投入成本。这基于现代云原生架构如 Kubernetes 和服务网格提倡的“设计即容灾”理念，通过多区域或多可用区设计内置容灾能力，依赖于基础设施即代码工具和 GitOps 流程实现部署自动化。例如，在电商平台的微服务架构下，SRE 团队可利用容器编排系统自动配置跨地域副本，当某个区域发生网络中断时，流量可无缝切换至备用区域，无需手动干预。这种能力提升了 SRE 团队的“预防性运维”水平，将容灾从“救火式”响应转变为“内置式”保障，尤其适合高并发场景如双十一大促，避免了因架构耦合导致的重构成本。

高效算力利用是容灾规划的另一个关键赋能，它通过冗余算力支持弹性伸缩与算力复用技术，有效降低 IT 运营成本。这涉及资源池化、弹性伸缩如 AWS Auto Scaling 和算力复用如利用空闲资源运行测试任务，SRE 团队可借助云平台的 Spot 实例或混合云调度器实现成本优化，背景包括负载预测算法和资源隔离技术。以视频流媒体服务为例，平日算力需求平稳，但突发活动如明星直播可能导致资源瓶颈，SRE 团队通过容灾规划预留冗余算力，并在非高峰时段复用这些资源进行压测或数据分析；当灾难如数据中心故障发生时，

弹性伸缩自动激活备用算力，确保服务不中断，同时日常成本降低30%以上。这种能力赋能SRE团队实现“绿色IT”，通过动态调度避免资源浪费，支持算力“削峰填谷”，在边缘计算场景中尤为关键。

灾备切换全流程监控能力确保全程监控灾备切换流程，提高操作透明度，使每一环节都可追溯且高效执行。这依赖于可观测性原则，SRE团队使用工具链如Prometheus用于指标收集和Jaeger用于链路追踪，实现全流程可视化，容灾切换涉及状态同步和数据一致性检查等。在金融系统中，灾备切换可能涉及数据库主从切换，SRE团队通过监控面板实时跟踪切换进度，从触发命令到数据验证完成，每个步骤均有时间戳和状态记录；如果超时，告警系统立即通知SRE介入，避免数据丢失。这种透明度强化了SRE的“诊断效率”，通过历史数据回溯优化切换策略，减少平均修复时间，适用于高合规要求的行业如医疗或政务。

自动化灾备响应能力支持灾备系统自动切换，能迅速应对突发事件，极大缩短业务恢复周期。这基于事件驱动架构和编排引擎如Ansible或Kubernetes Operators，SRE文化强调“自动化一切”，容灾自动切换减少人为延迟，背景包括故障检测算法和决策树模型。以在线教育平台为例，当CDN节点遭受DDoS攻击时，自动化系统在秒级内检测异常并触发切换至备用节点，SRE团队通过混沌工程演练事前验证流程，确保切换无需人工审批，恢复周期从小时级缩短至分钟级。这种能力体现了SRE的“韧性工程”，通过预设剧本处理常见故障，释放人力用于创新任务，在IoT或自动驾驶等安全关键场景中尤为重要。

精细化保护策略通过精细化的应用分级策略，在资源紧张时优先保护关键业务的稳定运行。这基于业务影响分析和资源调度策略如优先级队列或权重分配，SRE团队需定义服务等级协议和依赖图谱，背景包括策略即代码和AI驱动的资源分配。例如，在社交媒体平台中，SRE将服务分为关键、重要和普通三级，当资源不足如云计算配额耗尽时，系统自动降级非关键服务，确保核心功能如登录认证可用；在流量洪峰时，优先保障交易链路，而延迟数据处理任务。这种能力赋能SRE团队实现成本与可靠性的平衡，通过动态策略适应多变环境，特别适合多云或混合云架构，避免“一刀切”的容灾失败。

容灾规划对SRE团队的赋能不仅限于技术层面，还扩展到组织文化和业务价值。在全球化电商等综合场景中，SRE团队利用上述能力构建跨洲容灾体系，通过架构级容灾实现多活部署，结合自动化响应处理区域故障，同时监控系统提供实时洞察，确保大促活动的稳定性。容灾规划将SRE从被动运维提升为主动赋能者，通过架构集成、算力优化、监控透明、自动响应和策略精细化，大幅提升系统韧性。未来，随着AIOps和算力调度算法的进化，SRE团队可进一步实现预测性容灾，最终达成“永远在线”的业务目标。

在应用发布过程中，选择合适的容灾方案是确保业务连续性和可靠性的关键环节。容灾规划不仅涉及技术实现，还需紧密结合业务需求，以应对潜在的中断风险。典型使用场景是容灾方案设计的基础，需根据业务特性进行分类。首先，关键业务保障场景适用于对连续性和可靠性要求极高的领域。例如，金融行业需确保操作连续性，防止交易中断；急救系统则强调数据连续性，保障患者信息实时可用；政府与公共事业要求服务不间断，以维持社会稳定；而互联网应用常通过跨区域分流，优化用户体验并降低延迟。其次，就近服务与快速恢复场景关注容灾效率，包括跨可用区容灾以增强高可用性，跨地域容灾用于防范地震或洪水等区域性灾害，以及混合云容灾基于成本或用户需求实现灵活切换。此外，其他特殊场景如数据密集型容灾需保障强一致性数据，而合规要求则确保备份方案符合政策法规，例如数据保护标准。典型使用场景需要通过实践不断积累完善，并沉淀到系统方案中，以加强容灾规划能力的适应性。

常见容灾策略涉及层级和规划两个维度，帮助在实际应用中平衡成本与效果。从容灾层级看，数据集容灾适用于非实时业务，仅备份数据，实施成本及复杂度较低，但服务恢复较慢；应用级容灾通过冗余环境保障高效切换，需付出额外部署和维护成本；业务级容灾则适用于高并发场景，一般通过多活架构实现操作连续性和灾备切换。从容灾规划看，需设定合适的 RTO 和 RPO 指标来匹配方案，例如根据业务中断容忍度选择恢复时间目标；技术路径应选择适合服务场景的方案，如同步复制或负载均衡；同时，成本与风险控制是核心考量，需评估实时开销和实际风险，避免过度投入或防护不足。

容灾方案的选择应基于具体场景和策略进行综合评估，确保在应用发布时无缝集成，以提升系统韧性。以两地三中心方案为例，两地三中心是一种广泛应用的容灾模式。该模式通常涉及两个地理区域，称为“两地”，并在其中部署三个数据中心，称为“三中心”。具体来说，地域 A 作为业务生产环境，包含多个可用区（如可用区 1 和可用区 2），每个可用区都配备流量网关、算力集群以及应用实例，且可用区之间通过数据同步机制保持一致性；地域 B 则作为灾备环境，提供备份资源以应对突发故障。整个架构的上层由南北向流量网关统一管理入口流量，确保请求的合理路由。

在容灾规划中两地三中心方案支持多项核心能力，以确保在故障场景下业务的快速恢复。首先，在应用部署方面，该方案允许将应用实例打散部署到多个可用区，避免单点故障，同时灾备环境的地域可以配置最小化资源，以优化成本效率。其次，在应用迁移能力上，方案能够应对不同级别的故障：对于单可用区故障，业务可以自动迁移至同地域的其他可用区，实现秒级恢复；对于地域级故障，则根据灾备环境的资源容量，支持部分或全

量应用的平滑迁移，确保业务连续性。第三，流量切换能力提供了灵活的应对机制：单可用区故障时，系统自动屏蔽故障点并将流量调度至健康可用区；地域级故障时，则提供手工或自动两种切换方式，自动切换基于预设规则（如故障检测阈值），手工切换则适用于计划内维护或特定场景。最后，元数据切换能力确保在可用区或地域级故障发生时，支持元数据（如配置和状态信息）的手工或自动切换，维护数据一致性和系统可靠性。

在业务接入方面，两地三中心方案要求通过配置策略实现无缝集成。生产环境中，用户可以通过注解（Annotation）设置应用部署策略，例如指定应用打散部署到多个可用区，或按权重分配资源（如将 70%的流量导向可用区 1，30%导向可用区 2）。同时，针对可用区或地域级故障，需预先定义应用迁移策略，可选择指定关键应用迁移或尽力全量迁移，以平衡资源利用和业务优先级。流量切换方式也需配置为自动或手工模式，自动切换依赖规则引擎触发，手工切换则通过控制台或 API 手动执行。此外，生产环境需指定多可用区应用元数据的主备关系（如设置可用区 1 为主节点，可用区 2 为备节点），而备份环境则需明确应用备份范围，例如仅备份核心数据而非全部内容，以提升效率。

总体而言，容灾规划应提供包括两地三中心在内的各种可选容灾套餐，并结合业务特性生成个性化容灾策略，为高可用性业务提供智能化容灾服务。

2.7 专有领域算力消费扩展

SRE 综合算力调度旨在通过软件工程方法构建可扩展、高可用的系统，完成对计算资源的动态分配与调度，以满足业务的广泛需求。随着云计算和边缘计算的普及，算力调度需适应多领域、多场景的复杂需求。这种演进的背景包括数字化转型的加速、算力需求的多样化、成本优化压力以及 SRE 实践向业务场景的渗透。综合算力调度的基础是通用性和统一性，它通过标准化接口和策略屏蔽底层基础设施差异，形成统一算力视图，并支持低成本扩展，使得新增领域或场景时无需重构核心系统，仅需配置或轻量开发即可实现。在此过程中，系统需遵循信通院《可信算力调度服务商》标准进行能力建设，确保算力标识筛选字段的一致性、规格配置的完备性、调度策略的丰富性，以及状态同步、性能、计量准确定及安全性等关键维度，从而体现 SRE 从“运维保障”向“业务赋能”的转型。SRE 综合算力调度正是这种转型的平台工程化实践。

在专有领域算力消费扩展实践中，综合算力调度能力基于领域专属方案进行落地，通过选择并设定合适的基础能力实现场景化适配。例如，在 AI 场景，调度系统支持弹性 GPU/CPU 算力分配，用于模型训练和推理任务，通过动态扩容 GPU 集群应对高峰需求，

并集成框架如 TensorFlow 以优化流水线管理；在研发场景，它提供按需测试环境，支持 CI/CD 流水线的自动化构建和测试；在运营场景，则保障高可用算力用于监控和数据分析，通过多地域部署避免单点故障；在游戏领域，调度系统应对峰值并发，实现全局负载均衡和低延迟处理；在金融领域，它确保高可靠和低延迟，通过算力隔离和合规策略满足监管要求。扩展时，可通过编写专属 Workload 定义应用特点，封装专属 Operator 串联算力调度基础能力以完成个性化调度，并设定环境配置、容量策略、发布策略、容灾方案，配置闲时算力供给策略，选择 CRI、CSI、CNI 扩展，以及扩展配套脚本作业等内容，从而低成本适配医疗、物联网等多样领域，如医疗影像处理中的 GPU 加速调度。SRE 综合算力调度基于通用算力调度能力，通过模块化设计提升灵活性和效率，实现了算力消费的场景化落地，为各领域数字化提供坚实基础。

下文将通过一些成熟的多领域扩展实践方案来说明 SRE 综合算力调度专有领域算力消费扩展的实践细节和效果。

2.7.1 研发领域算力消费场景扩展

研发服务是算力消费的重要领域，基于综合算力调度来搭建研发服务治理方案，能够通过动态分配计算资源，将研发流程中的各个环节，如编码、代码扫描、测试和编译构建等，无缝串联起来，从而实现算力的优化利用。这一方案能够解决传统研发环境中常见的资源孤岛、算力浪费和成本高昂等问题。

在企业 IT 架构中，传统模式往往将研发环境与生产环境分离，导致资源割裂和运维复杂化。综合算力调度方案通过统一调度，使软件研发环境和应用部署环境共享同一套算力资源。例如，借助虚拟化或容器编排技术（如 Kubernetes），调度器可以实时监控资源利用率，动态调整配额，如在业务高峰期优先保障生产需求，而在低峰期将闲置算力分配给研发任务。这种架构升级不仅提升了环境一致性，还支持 DevOps 实践的落地。此外，该方案还可扩展至多云混合部署，企业能灵活调用公有云资源应对研发峰值，增强整体架构弹性。

降低研发资产成本是另一重要优势。传统模式下，研发团队需预购大量专用 IT 资产，如高性能服务器和存储设备，但这些算力在非工作时间利用率低下，导致固定资产折旧和运维成本攀升。综合算力调度通过建立算力共享池，实现按需使用模式，无需大量采购专用资产。调度算法基于历史数据预测需求，在研发低峰期自动释放算力至其他业务，从而降低总体拥有成本。

提升算力使用效率方面，企业常面临“潮汐效应”，即生产业务高峰与研发需求存在时间差。综合调度通过合理分配时间窗口，如将研发任务集中在凌晨等空闲时段，优化算力利用率。调度策略采用优先级队列和智能算法（如强化学习），动态调整任务顺序，确保低优先级的研发任务（如自动化测试或代码扫描）在生产任务完成后自动触发。

基于综合算力调度方案的研发服务治理能力从算力静态分配升级为动态优化，核心价值在于提升算力弹性、降低成本和加速创新。企业实施时只需关注调度算法精度、安全隔离性及研发工具链集成完整度，随着算力网络的发展，该方案还可扩展至跨地域协同研发场景，成为企业 IT 进化的重要基础。

在软件研发过程中，云研发、代码扫描和编译构建等环节对算力的弹性需求最为明显。通过综合算力调度带来的解决这些场景中短期算力高消耗的问题是 SRE 业务赋能的直接体现。

传统开发模式中，每位开发者通常需在本地物理机或虚拟机上配置复杂的开发环境，不仅耗时费力，而且极易产生环境不一致问题，导致“开发时正常、测试时异常”的困境。通过算力调度，可以利用云计算资源实现环境的即开即用和动态管理。具体而言，综合算力调度系统能够将开发环境及其所需的操作系统、依赖库、中间件等整体封装为一个可调度的算力单元，开发者提交代码后即可自动获取一个独立的、环境一致的云研发空间。例如，为一个大型新功能开发任务，综合算力调度系统可以快速从共享资源池中划分出包含特定 CPU、内存和 GPU 资源的完整环境，开发完成后环境可自动释放，资源回池，极大地简化了流程，提升了协作效率并降低了成本。

随着代码规模扩大和安全左移的实践推广，静态代码分析已成为研发流程中的必备环节。然而，对大型单体仓库进行全量扫描往往耗时数小时，严重拖慢开发节奏。综合算力调度在此环节引入了分布式扫描技术，将扫描任务智能拆分为多个子任务，由调度系统并行分发到多台计算节点上同时执行。这不仅突破了单机性能瓶颈，还将扫描时间从小时级缩短至分钟级，使开发人员能够快速获得反馈，及时修复缺陷与漏洞。例如，在持续集成流水线中，一旦有代码入库，综合算力调度器便会自动触发分布式扫描任务，根据代码变更范围智能分配扫描资源，优先扫描关键模块，从而在保障软件质量与安全的同时，最大化提升了研发效率。

编译构建环节同样可以受益于综合算力调度。大型项目，尤其是像操作系统或大型微服务套件这样的项目，编译过程极其耗费计算资源。通过分布式编译加速技术，综合算力调度器将整个构建任务根据模块依赖关系进行分析，将可并行处理的编译单元分发到编译

集群中的多个节点上同时进行。同时，调度系统会协同缓存机制，对于未发生变动的模块直接使用缓存结果，避免重复编译。在这种模式下，综合算力调度不仅考虑了任务分发，还综合了资源成本与效率。例如，它可以在团队集中提交代码的晚高峰时段，自动弹性扩容编译集群，快速消化构建队列；而在闲时则自动缩容，将资源释放给其他任务（如夜间批量测试），从而实现跨集群的闲时算力利用，显著降低整体研发成本。

为支撑上述研发场景，需要对综合算力调度进行适当的扩展。在运行时能力方面，通过集成 Kubevirt 等虚拟化套件，调度系统可以统一管理虚拟机和容器等异构工作负载，满足一些特定开发场景（如需要特定内核版本或图形化界面的开发测试）对传统虚拟机的需求。同时，对 Windows 容器的生命周期管理支持，使得基于.NET Framework 等技术的应用也能无缝融入云原生研发体系。

网络能力扩展确保了研发环境的灵活接入与通信。裸金属 CNI 插件解决了容器与物理网络直连的需求，特别在混合云场景中，保障了开发环境与线下数据库或内部服务之间的低延迟、高带宽通信。固定 IP 与 Mac 地址分配功能则为每个研发环境提供了稳定的网络身份，便于网络策略管理、安全审计和服务发现，满足了企业级研发对网络可预测性和安全性的要求。

存储能力扩展针对研发活动中的多样化数据需求提供了解决方案。对于游戏开发、多媒体处理等 I/O 密集型任务，调度系统可优先调度至配备高性能本地 SSD 的节点，以满足美术资源处理对磁盘读写速度的极高要求。而对于代码仓库、依赖包缓存等容量敏感型数据，则可采用高容量云盘存储。磁盘备份与恢复功能则为研发环境提供了“时光机”式的数据保护能力，一旦节点故障或误操作发生，可快速恢复至之前的状态，保障了研发工作的连续性与数据安全。

在应用分发与部署层面，综合算力调度通过联邦集群管理能力，将多个物理分散的 Kubernetes 集群资源统一纳管，形成一个逻辑上的“资源超级池”。调度系统可以根据预设策略（如成本优先、资源利用率最大化等），智能地将研发任务（如自动化测试、代码编译）分发到不同集群的闲时算力上运行，从而打破单个集群的资源限制，实现全局资源的优化调度与成本节约。

下文中将对上述扩展方案和技术细节展开详细说明。

随着企业数字化转型的深入，很多传统基于 Windows 架构的应用（例如依赖.NET 框架或 IIS 服务的系统）需要融入云原生环境，但由于其对操作系统内核的强依赖，直接迁移至 Linux 容器平台面临诸多兼容性挑战。与此同时，容器与虚拟机并存的混合负载场景

日益普遍，亟需一种能够统一调度异构工作负载的架构方案。在此背景下，基于Kubernetes的Windows容器与容器化虚拟机（如KubeVirt）运行时扩展技术应运而生，它通过扩展Kubernetes原生调度与资源管理能力，实现对Windows容器及传统虚拟机的云原生式管理。

在Windows容器方案中，控制平面仍然部署于Linux系统，保障Kubernetes核心组件的稳定运行，而工作节点则采用Windows Server系统，能够承载基于Windows镜像的Pod。每个Windows节点通过特制版本的kubelet与容器运行时（如containerd）接入集群，使得IIS服务、SQL Server等典型Windows应用可以以容器化方式交付和运行。调度器借助节点标签识别系统类型，实现Linux与Windows工作负载的分区调度与资源复用，有效避免为两类系统单独维护资源池，显著降低运维复杂度。

另一方面，针对需要完整可交互式操作系统环境或更强隔离性的场景，KubeVirt提供了容器化虚拟机的运行时扩展能力。该方案控制平面与工作节点均构建于Linux系统之上，通过自定义资源定义（CRD）扩展Kubernetes API，定义虚拟机、虚拟磁盘等资源类型，并配套开发相应控制器与Operator，通过libvirt接口实现对虚拟机生命周期的精细管理。这意味着用户可以使用kubectl工具创建、启动或迁移虚拟机，就像管理普通Pod一样便捷。KubeVirt将每个虚拟机实例封装为一个Pod，使其能够继承Kubernetes的网络、存储和调度能力，实现虚拟机和容器在同一个集群中的混合部署与统一管理。

在实际落地上，该架构支持多种灵活机制以满足复杂生产需求。用户可通过设定节点模式，将集群节点划分为不同的资源池，分别承载Windows虚拟机、Windows容器或Linux容器等不同负载类型。借助CRD和控制器扩展，能够实现个性化编排策略，例如设置优雅退出机制，确保有状态虚拟机在重新调度前自动执行状态保存。网络方面，支持Underlay网络模式，使虚拟机或容器可直接使用物理网络接口，适用于高性能、低延迟场景。存储方面则通过CSI插件扩展，整合本地存储与云存储资源，并配合卷快照功能实现虚拟磁盘的备份与恢复。此外，结合集群联邦能力，可实现跨集群的应用分发与资源复用，支持混合云场景下的动态调度与容灾切换。

从应用场景来看，该方案特别适用于传统企业应用现代化改造、混合云资源调度以及高性能计算环境。例如，在混合云中，可将Windows业务负载调度至本地集群，而将Linux容器与虚拟机扩展到公有云资源，实现跨云弹性伸缩。在AI训练与科学计算场景中，可借助统一调度器协调运行于Windows环境的数据预处理任务和运行于Linux的模型训练

任务，通过 Underlay 网络保障通信效率。对于遗留系统，可先以容器化虚拟机形式纳入 Kubernetes 管理，再逐步进行容器化拆分，实现平滑迁移。

研发活动通常构建在混合云、边缘节点或异构物理网络之上，例如自研云中的裸金属服务器、PCDN 物理机等构成的云研发专区。这些环境往往存在网络平面复杂、基础设施不统一的问题。研发流程对网络提出了特殊需求：云研发桌面通常需要与母机处于同一物理网络平面，即使用 Underlay IP，以保证与远程管控服务、安全组件等实现无缝通信；同时，研发实例往往要求固定 IP 地址与 Mac 地址，以维持基于 IP 或 Mac 的访问控制、监控策略的长期稳定。此外，老旧 PCDN 机房、云上裸金属或非标准网卡等异构物理网络的广泛存在，进一步增加了网络管理的复杂度，也对综合算力调度中的资源分配与网络适配能力提出了扩展需求。

为应对上述挑战，需对网络管理能力进行系统化扩展，并将其深度整合至综合算力调度体系中。在容器网络层面，可通过扩展 CNI 插件来支持多样化的网络平面。例如，为云研发离岸场景扩展能够直接分配 Underlay IP 的 CNI 插件，避免 Overlay 网络带来的性能损耗；同时针对 PCDN 环境扩展 macvlan 类插件，使容器能够桥接至物理网络，实现与宿主网络同平面的通信能力。在 IP 地址管理层面，可通过扩展 IPAMD 服务，将 Underlay IP 资源抽象为 Kubernetes 自定义资源，实现 IP 资源的精细化与自动化生命周期管理。节点上线或下线时，IP 资源能够自动分配与回收；该服务还能为 Deployment、StatefulSet 等多种工作负载的 Pod 分配固定的 IP 与 Mac 地址，确保实例重启后网络标识不变，极大提升了管控策略的可靠性。在调度层面，则需对 Kubernetes 调度器进行扩展，增加网络感知能力，使 Pod 的调度决策能够参考目标节点的 IP 资源余量、网络拓扑状况等因素，从而实现网络资源与计算资源的协同优化，避免因 IP 资源不足或网络路径不佳导致的调度失败或性能下降。

这些网络能力的扩展，在功能上带来了显著收益。系统能够支持 Underlay IP、MacVlan 及多种 Overlay 网络平面，满足不同研发场景的隔离与性能要求；可为 Pod、虚拟机等多种资源类型分配固定 IP 与 Mac 地址，保障了网络策略的持久性；同时，通过对 PCDN 机房网络、云上裸金属网络及非标网卡网络等异构物理环境的兼容，实现了跨基础设施的统一网络管理。从场景扩展来看，该体系能有效支撑跨地域的研发协同，例如全球团队可共享一套网络策略稳定的研发环境，综合算力调度系统可结合网络延迟与资源状况，智能分配任务至最优节点。在混合云与边缘计算场景中，网络能力的统一抽象使得算力可

在中心云与边缘节点之间弹性伸缩，为 DevOps 流水线提供敏捷、可靠的底层网络支撑，从而整体提升研发效率与资源利用率。

对存储管理能力的扩展是另一个关键技术。研发活动涉及代码仓库、依赖库、容器镜像、流水线日志等海量非结构化数据，传统单一存储方案已难以应对性能、弹性与可靠性的综合挑战。综合算力调度系统需通过混合存储架构、动态编排机制和数据保护策略，构建智能化的存储管理能力，以支撑高效的研发服务治理。

从技术背景来看，研发场景中存储需求具有鲜明的二元性。一方面，代码编译、容器镜像构建等 I/O 密集型任务对延迟极其敏感，需要本地固态硬盘提供毫秒级响应；另一方面，测试日志、版本归档等冷数据又需云存储的大容量和弹性扩展能力。同时，微服务实例的弹性伸缩要求存储资源能够随 Pod 动态绑定与释放，而多租户环境下的数据隔离与容灾需求进一步增加了存储管理的复杂性。CSI 插件链、VELERO 备份工具以及混合存储调度框架，正是针对这些挑战的系统化解决方案。

在混合存储调度方面，技术架构通过本地盘与云盘的协同使用平衡了性能与容量矛盾。对于游戏美术、客户端开发等高 I/O 需求场景，调度器会优先将工作负载分配至配备本地 SSD 的节点，利用其低延迟特性加速数据处理流程；而对于数据备份、测试报告存储等容量敏感型任务，则自动切换至弹性云盘。这种智能调度不仅体现在初始分配阶段，还通过资源预留机制为关键服务（如 Git 仓库）保障稳定的 I/O 性能。特别值得注意的是，系统通过标签识别机制动态匹配 Pod 需求与存储特性，例如 API 网关类服务会自动绑定高性能存储，而批处理任务则使用经济型云盘，实现资源的精细化运营。

镜像备份机制为研发数据持久化提供了坚实保障。VELERO 工具与对象存储（COS）的集成构建了端到端的备份恢复流程。当开发环境进行重要变更时，备份控制器会监听自定义资源（Backup CR）的变更，自动生成磁盘快照并上传至云端存储为版本化的 disk.img 文件。这种机制不仅能够防止因节点故障导致的数据丢失，还支持快速环境重建——当开发人员需要复现特定测试场景时，可通过镜像秒级恢复完整工作环境。更重要的是，备份数据为研发治理提供了审计追踪能力，满足合规性要求，特别是在金融、医疗等监管严格行业的开发场景中具有重要价值。

CSI 插件体系的扩展实现了存储资源的全生命周期管理。csi-webhook、external-resizer 等组件形成了完整的控制回路，能够拦截持久化存储声明（PVC）请求，按需创建、扩容或迁移存储卷。当研发团队需要扩展测试环境存储空间时，系统可在线调整云盘容量而无须重启服务，极大提升了运维效率。针对裸金属机器挂载云盘的数量限制（如博通网卡仅

支持 4 个 CBS），csi-controller 还通过虚拟化技术突破物理约束，实现多个云桌面共享云磁盘，显著提升资源利用率。这种动态管理能力特别适合微服务架构下的状态管理，当有状态服务（如数据库）发生 Pod 迁移时，存储卷能够自动重新挂载，确保业务连续性。

在容灾设计方面，系统通过 DaemonSet 将关键存储管理组件部署到每个计算节点，形成去中心化的高可用架构。数据请求遵循“PVC→csi-webhook→外部存储”的冗余路径，避免单点故障影响整体服务。这种设计尤其适合服务网格等分布式架构，能够保证控制平面配置数据的持久化，即使在可用区级故障时也能快速恢复。结合跨区域备份策略，研发环境可实现多活容灾，保障核心研发活动的连续性。

研发服务治理场景下，调度运行时的扩展是应对异构资源环境复杂性的关键技术路径。研发环境算力基础设施呈现高度多样化的特征，具体表现为母机类型繁多，涉及多种型号差异，包括不同 GPU 类型、内存配比及磁盘大小配置；虚机规格复杂，为满足开发、测试、算法等不同岗位需求，形成了多样化的算力规格，在 CPU、内存、GPU 和磁盘等方面均存在差异化配置；硬件生态多元，既支持专业级 GPU 卡如 T10、A10、V100、A100，也兼容常见消费级 GPU 设备。这种资源异构性对传统调度机制提出了严峻挑战，亟需通过架构扩展实现精细化的资源调度。

在技术架构层面，调度能力扩展主要通过扩展调度插件和扩展硬件资源插件两类核心组件实现。扩展调度插件基于 Kubernetes 调度框架的扩展点进行增强，包括 Filter、Score、Reserve 和 Bind 四个关键阶段。例如通过插件引入特殊机型调度算法，能显著提升母机亲和性匹配能力，支持根据“装箱率高的节点评分高”等策略进行优化调度，同时具备预占机制和机型调度策略，能够智能选择裸金属机或虚拟机资源。这种设计使得调度器能够感知硬件拓扑特征，例如将 RTX 3080 等特定 GPU 资源与对应计算任务进行精准匹配。

扩展硬件资源插件则致力于解决异构设备的识别与管理问题。以 GPU 设备为例，通过 kubevirt-gpu-device-plugin 等设备插件，系统能够自动扫描节点上的 GPU 设备（包括物理 GPU 和虚拟 GPU），并通过 List And Watch 机制动态更新节点扩展资源。该插件不仅支持对 A100、T10 等专业显卡的识别，还能将 vGPU 资源纳入 Kubernetes 资源管理体系，使得 GPU 资源能够像标准计算资源一样通过 API Server 进行统一调度和分配。这种设计实现了硬件资源的自动化发现与注册，为复杂算力场景下的资源供给提供了基础保障。

从技术演进视角看，调度运行时扩展能力源于云原生技术对异构计算需求的响应。随着 AI 训练、科学计算等计算密集型任务的普及，传统基于 CPU 的均质化调度模式已难以满足需求。Kubernetes 设备插件机制与调度框架的扩展性，为融合 GPU、FPGA 等异构资

源提供了标准化的实现路径。通过将硬件特性和业务需求转化为可量化的调度策略，实现了从“资源可用”到“资源好用”的升级。

在上述扩展能力的加持下，基于综合算力调度能够低成本打造出研发领域专属的各种解决方案，这些方案通过构成了研发服务治理体系，为企业 IT 提供体系化能力支撑。为节省篇幅，下文将以大型游戏编译构建场景为例来说明综合算力调度在研发服务治理场景下的应用。

游戏开发中常见的 UE4 或 Unity 项目，代码库庞大、依赖复杂，传统的单机编译方式往往耗时数小时，成为研发流程中的显著瓶颈。为应对这一挑战，基于综合算力调度的分布式编译构建加速方案应运而生。该方案的核心思想是将编译任务与固定的物理资源解耦，通过智能调度平台，将任务动态分发到分布在不同地理区域的云端算力集群中并行执行，从而实现对编译过程的显著加速。

该方案的技术架构建立在标准化研发流程之上。开发人员将代码提交至版本仓库后，系统会自动触发编译构建流程，进而完成后续的发布部署。这一自动化链条确保了过程的连贯性与可重复性。方案的技术基石在于综合算力调度能力。调度平台能够统一管理多地的多个云端编译构建集群，形成一个庞大的联邦算力池。这不仅实现了跨地域的负载均衡，避免单点故障，也大大提升了资源的整体利用率。

为了满足不同编译任务对运行环境的特定要求，平台集成了多样化的算力资源类型。例如，对于主流的 Linux 下 C/C++代码编译，平台可以调度高性能的增强型容器，提供强隔离且高效的运行环境。而对于必须在 Windows 平台上进行的任务，如 UE4 引擎的 Shader 编译，平台则可以通过 Kubevirt 虚拟化技术，快速创建 Windows 虚拟机来执行任务，确保了环境的兼容性。这种灵活的调度机制使得平台能够支持多引擎、多场景的编译需求，包括 Unity 项目、C/C++项目以及 UE4 的代码和 Shader 编译等。

调度器在执行任务分配时，会综合考虑任务优先级、所需环境类型、各集群当前负载以及网络延迟等多种因素，智能地将任务路由到最合适的计算节点上。这种动态调度策略不仅最大限度地缩短了任务排队和执行时间，也实现了计算资源的弹性伸缩。在开发高峰期，系统可以自动扩容，调用更多云端资源应对峰值压力；在闲时则释放资源，有效优化了成本。对于跨地域协作的全球化团队，调度系统可以优先将任务分配给离开发团队物理距离更近的集群，以降低网络传输延迟，实现 24 小时不间断的集成与构建。

2.7.2 低代码平台算力消费场景扩展

低代码应用平台作为算力消费的关键应用领域，在企业数字化转型进程中发挥着核心作用。具体而言，低代码应用特指企业内部广泛部署的工具型软件即服务（SaaS），其功能范围覆盖运维管理、产品运营、行政管理等多个业务层面。由此类应用构建的“工具软件生态系统”，已成为企业实现成本优化与效率提升目标的重要支撑基础。

成熟的低代码应用平台依托模块化架构与可视化开发引擎，有效支持企业实现 IT 架构升级，并高效构建技术中台能力。具体而言，该平台通过提供标准化的数据处理、业务流程编排及接口集成等核心模块，协助企业将分散的 IT 资源整合为可复用的技术中台，促进数据共享与业务协同。开发者利用低代码平台所提供的可视化开发环境、丰富组件库及预构建模板，能够高效开发各类低代码工具，覆盖流程自动化、数据报表生成、移动应用开发等多种应用场景。这些低代码工具不仅显著降低了开发门槛，还具备高度灵活的配置能力，企业员工无需具备深厚编程基础，仅通过简单的拖拽与配置操作，即可快速搭建符合业务需求的应用系统。通过将开发成果赋能于全体员工，企业能够实现业务流程的自主优化与创新，减少对专业 IT 资源的依赖，从而显著提升组织运营效率，并有效降低企业运维成本。

在低代码应用平台的构建过程中，应用生命周期的各个阶段均需配备相应的综合算力调度能力。在应用开发与测试阶段，依托开发环境的算力调度机制，可以构建开发者中心产品，提供核心应用模型、应用框架及低代码画布等关键特性，以支持应用开发与测试的技术需求。而在应用部署阶段，基于部署环境的算力调度能力，能够建立应用市场，并借助算力平台的可观测性与生命周期管理功能，实现应用的自动化部署。

与其他算力消费场景相比，低代码应用平台的核心特征在于其“开发者亲和性”。平台基于算力基础设施，通过低代码技术所提供的开发与托管服务应具备高度的便捷性与易用性，以促进企业内部开发者的广泛参与。因此，平台设计应以开发者视角为核心，通过合理化设计来提升应用开发与维护的便捷性。以下将从技术层面阐述平台设计的关键考量。

在核心运行时层面，平台为低代码应用提供了专属的工作负载定义模式，即“进程”。开发者可通过进程抽象来定义多样化的负载类型，例如 Web 站点、gRPC 服务以及后台服务等。“进程”这一概念源自操作系统的经典定义，通过此类比来描述应用负载，有助于用户迅速理解平台的功能。

为确保低代码应用的稳定运行，除了依赖于无状态进程之外，还需要各类依赖服务的支持，例如 MySQL 数据库和 Redis 存储等。在低代码平台中，此类依赖服务被统一归类为

“增强服务”。平台通过利用算力平台提供的基础计算与存储能力，实现了多种增强服务控制器（Service Provider）。以 MySQL 控制器为例，其负责 MySQL 数据库的资源分配。当这些控制器接入平台后，相应的依赖服务即可供开发者自助调用。

在应用部署阶段，算力调度基础设施为平台提供了多元化的算力管理策略。除支持手动扩缩容外，平台还可基于应用负载情况配置自动扩缩容策略，最大限度提升整体算力利用率。此外，低代码平台支持同时纳管和使用多套算力集群，管理员可先将具备不同特征的算力集群接入平台，再通过配置灵活的集群算力分配策略，实现应用的动态调度。

在应用构建层面，算力基础设施同样为低代码平台提供了强有力的支撑。平台可结合应用特性，针对不同编程语言、不同底层架构的应用，对构建阶段的算力供给进行针对性优化，从而提升构建效率与稳定性。

此外，在应用集成阶段，平台通过静态检查与代码扫描，从底层源码及依赖库层面保障应用对算力的合理使用，提升系统安全性。在可观测性方面，低代码平台基于算力平台的底层可观测性支持，结合低代码应用的特点，提供开发者友好、开箱即用的日志、应用性能监控（APM）及告警功能。

构建能够支撑企业各类工具的低代码平台，其关键在于设计灵活且强大的声明式“应用模型”。应用模型规范作为整个低代码平台的设计核心，定义了通过声明式语言（YAML）描述低代码应用的标准。一份完整的应用模型明确了应用的构建方式、包含的进程、各进程的启动机制等关键细节。

低代码平台上应用的所有行为均由应用模型驱动。模型提交至平台后，平台会依据模型创建对应的应用资源（包括进程、数据存储、可观测配置、访问路由等），使应用具备对外服务能力。完善的应用模型规范需包含以下核心组成部分：

（1）应用运行时：定义执行环境与全生命周期管理，涵盖进程、环境变量等要素；

（2）应用集成：规范系统间交互与接口标准，包含应用构建配置等内容；

（3）应用资源：配置计算、存储、网络资源，涉及增强服务依赖、挂载卷等定义；

（4）应用可观测性：明确应用监控、日志、追踪能力的实现方式。

为适配不同用户的操作习惯，低代码平台支持两种应用模型操作模式：YAML 模式与表单模式。其中，YAML 模式通过 YAML 配置文件定义应用模型，适用于专业开发或运维人员；表单模式采用可视化表单页面直接操作模型各组成部分，更适配业务用户的低代码操作场景。

为全面赋能开发者，除核心应用模型外，成熟的低代码平台还需提供各类自助工具，与应用模型共同构成高度集成的低代码开发体系。以应用开发调试为例，平台提供的工具包括：

（1）Web Console：支持开发者快速进入运行中的进程环境，通过命令行终端完成脚本执行、调试等任务；

（2）沙箱开发：基于在线集成开发环境（IDE）和隔离的沙箱环境，使开发者无需搭建本地环境即可完成需求开发；

（3）开发框架与样例：通过标准化框架与实例帮助开发者快速掌握应用开发方法，提升开发效率。

在应用分发与版本管理方面，平台提供：

（1）打包流水线：支持开发者调用流水线快速打包应用，通过应用包实现分发；

（2）一键自助版本迁移：帮助开发者将应用从旧版本便捷迁移至新版本，降低手动修改代码的兼容适配成本；

（3）应用模型文件版本转换：实现应用模型文件向指定版本的转换。

此外，低代码平台需承担应用全生命周期管理职责，对闲置应用（部署后无任何访问的不活跃应用）定期发送提醒，推动开发者下架以节约计算资源；当应用资源配置不合理（如申请资源远超实际用量）时，需推动优化调整。

不同企业与组织对低代码平台的需求存在差异，因此平台应采用插件式设计，在核心能力层面支持通过配置、服务接入等方式进行扩展，以满足多样化的客户需求。

在编程语言与开发框架支持方面，平台内置对 Python、Go、Java 等主流编程语言的支持，通过云原生构建包（CNB）、容器镜像（Dockerfile）等方式实现构建。企业如需支持其他编程语言，只需遵循云原生构建包规范开发对应的构建工具并接入平台即可。

如前文所述，“增强服务”是平台的特色功能之一。平台内置 MySQL、Redis、对象存储及资源实例池等类型的增强服务。若企业需扩充增强服务类型（如增加 RabbitMQ 消息队列），可利用低代码平台提供的增强服务框架，开发并部署对应的增强服务微服务控制器（Service Provider），将控制器配置到平台的增强服务发现模块后即可完成接入。

应用可观测性作为低代码平台的重要功能，除内置对日志平台、监控平台及开源 ELK 方案的支持外，还支持接入其他日志与监控基础设施，以满足企业定制化需求。

此外，在应用源码仓库对接方面，低代码平台支持通过配置对接各类源码仓库（Git、SVN 等）；在算力对接方面，平台支持多种算力集群接入模式，管理员可自主创建和组合规则配置算力调度策略，满足不同算力需求场景。

开发者通过低代码平台开发的工具应用具有多样性，根据使用场景可分为四类：

（1）普通应用：基于 Web 开发框架构建，面向普通用户提供 Web 服务的应用类型。其开发门槛较低，适合各类背景的开发者快速开展低代码 Web 开发，常见于企业内部的业务研发 / 运营工具、效率工具、行政工具等场景。

（2）流程插件：基于标准插件开发框架开发，遵循严格的输入输出规范，可与标准运维平台、企业流程平台等集成，扩展目标系统功能。通过流程插件，可实现为企业运维平台增加原子化功能节点、为企业流程平台增加个性化审批行为等定制需求。

（3）API 网关插件：由 API 网关系统提供开发框架，专门服务于 API 开发场景的特殊应用类型。网关插件可集成到 API 网关系统中提供 API 服务，适用于个性化组合或封装 API 的场景，在企业 API 集成开发、API 适配等任务中发挥重要作用。

（4）AI Agent：平台支持的创新应用类型，开发者可基于开发套件在低代码平台上快速搭建 AI Agent 应用，实现智能助手、AI 驱动的智能分析决策等功能，使 AI 高效赋能企业内部各类场景。

在企业内部，用户按职责与分工可分为“工具开发者”和“工具使用者”。工具开发者负责开发大量工具应用并交付给工具使用者。要在企业内形成良性的“工具文化”生态循环，除低代码开发平台提供的“以应用模型为核心”的工具开发体验外，还需“应用市场”提供的工具使用体验，二者相辅相成。为提供优质用户体验，平台包含三类应用市场：

（1）桌面应用市场：用户通过 Web 浏览器访问，用于安装和使用各类工具，是最常用的工具使用渠道；

（2）插件市场：供工具流程类应用检索合适的插件，并通过插件 API 调用插件能力；

（3）移动端应用市场：为支持移动端访问的 SaaS 提供移动端访问能力。

2.7.3 金融证券领域算力消费场景扩展

随着资本市场深化改革持续推进和数字化转型加速，证券行业正从传统交易服务向综合金融服务升级，业务复杂度和技术需求呈现指数级增长。

在业务连续性方面，证券业务对系统可用性具有极致要求，任何服务中断都可能引发市场波动和客户损失。证券交易涵盖终端接入、运营商网络、渠道系统、资金存管、柜台系统等多个模块，链路长、环节多，每个环节的稳定性都直接制约整体业务连续性。

在算力应用效率方面，证券行业算力需求呈现显著波动特征，开盘收盘等关键时段的算力需求可达平日数倍。传统静态资源分配模式导致资源利用率低下、难以应对突发业务负载。同时，多云异构环境的管理复杂度较高，应用交付部署效率受到制约。

在信创合规方面，技术自主可控和供应链安全成为核心考量，避免单一厂商绑定是信创建设的基本要求。随着监管合规要求日益严格，信息系统需全面满足法规要求，实现合规运营。信创转型过程中，技术路线多样性带来兼容性挑战，不同芯片架构（如 x86、ARM）和操作系统（如 Linux、Windows、国产 OS）并存，要求应用具备跨平台运行能力。

由上述技术挑战可以看出，证券行业需要专业化、定制化的算力消费方案支撑其高质量发展。基于综合算力调度基础框架并结合证券行业的应用特性，可通过扩展 3 大核心能力来构建高效、可靠且合规的证券行业算力消费解决方案：在发布能力扩展通过非侵入式流量控制机制实现业务无感知的全链路灰度发布与平滑升级，降低业务部署风险；集群管理能力扩展通过统一接口实现多云及自建资源池的统一管控，提升跨云调度的一致性与效率；信创能力扩展依托硬件抽象与统一调度引擎，实现异构芯片与操作系统的自动化适配与算力整合，保障系统合规稳定并提升跨平台部署灵活性。

这些能力共同构筑了证券业务数字化转型的基石，助力 SRE 团队实现运维自动化、系统高可用和架构可持续的高阶目标。

在现代证券行业的数字化转型过程中，业务系统日益呈现出“敏态”特征，即需要快速迭代以满足市场和客户需求。然而，这类系统的技术架构往往是混合形态，既包含稳定运行多年的 Java 传统应用，也引入了大量新兴的云原生应用。这就给系统的发布部署带来了显著挑战：Java 类应用通常拥有其特定的通信数据面，但却缺乏一个统一的控制平面进行精细化的流量调度；而业界普遍采用的服务网格方案 Istio，其默认设计是支持 Envoy 作为数据面代理，难以直接管理和引导 Java 应用的流量。这种异构技术栈并存的情况，使得在涉及复杂服务依赖的链路中实现安全、平滑的灰度发布变得异常困难。

为解决上述混合架构下的发布难题，我们对综合算力调度系统的全链路灰度发布方案做了扩展。扩展后的方案仍然能够通过一个控制面统一管理多种数据面。具体而言，对于 Java 应用，我们通过无侵入的 Java Agent 技术，为其赋予接收并执行流量控制规则的能力，使其能作为一种特殊的数据面被集成。同时，方案继续支持 Envoy 作为云原生应用的通用

数据面。关键的扩展在于，对 Istio 控制平面进行了适配与扩充，使其能够超越原有设计限制，统一地管理由 Java Agent 和 Envoy 构成的混合数据面集群。这样一来，运维人员只需通过熟悉的 Istio 标准接口下发一条灰度路由策略，该策略就能同时生效于整条调用链路中的 Java 应用与云原生应用。

这套全链路灰度发布方案带来了多方面的收益。首先，它实现了传统 Java 应用与云原生服务的良好兼容，使得跨越不同技术栈的复杂依赖链路能够进行联动灰度发布。在资源利用方面，相较于需要部署完整两套环境的独立发布方案，本方案基于统一流量标签的逻辑控制机制大大降低了算力持有成本提升了算力利用率。当灰度版本出现问题时，通过简单地切换流量路由标签即可实现秒级回滚，极大地提升了运维效率并降低了故障影响时长。在数据安全层面，方案支持对服务、消息队列乃至数据库进行灰度隔离，有效降低了灰度数据污染生产数据的风险。以上这些措施有效保障了证券系统绝大多数用户业务的连续性和稳定性，确保了用户最终体验。

证券行业应用部署呈现出典型的分层特征：核心交易系统通常部署于券商自建机房，以确保对关键业务的极致控制力与安全性；互联网业务服务则基于云上托管资源，充分利用云计算的弹性优势以应对业务波动。此种混合架构要求平台能够实现传统核心系统与云上服务的平滑协同，同时规避单一厂商锁定风险，保障业务连续性与技术自主可控。为应对上述挑战，混合云集群管理方案通过以下四项扩展能力构建统一管控体系。

公有云集群统一纳管功能通过标准化接口对接各云厂商的集群管理能力，屏蔽底层平台差异，提供一致的集群操作体验。SRE 团队可基于统一控制面实现多云资源的集中调度与策略下发。

自建集群高可用管理建立自动化、高可用的 Kubernetes 集群管理体系，通过声明式配置与 GitOps 流程实现集群的快速初始化、无缝升级与故障自愈，显著提升本地基础设施的运维标准化水平。

存量集群平滑接入支持通过 kubeconfig 配置或云厂商账号权限，将现有异构集群纳入统一管理范畴，实现从传统架构到云原生平台的渐进式迁移，最大限度保护既有投资。

裸金属与虚拟机资源池化通过与传统资源管理系统对接，将存量虚拟机及裸金属服务器转化为可弹性调度的算力池，提升硬件利用率和交付效率。

混合云集群管理方案通过平台化架构有效解决了证券行业在混合多云环境下的治理难题，为数字化转型提供了灵活可靠的基础设施支撑。

证券行业在推进信创转型进程中，面临由硬件架构、操作系统及基础软件等多源异构环境引发的兼容性、安全性与运维复杂性等挑战。为系统应对上述挑战，“信创合规扩展能力”将从四个关键维度构建治理框架，以确保应用系统全面兼容异构环境，同时显著提升其可靠性、安全防护能力及整体运行效率。

一是采用容器化技术实现应用统一交付。将各类基础设施应用与业务应用分别编译为标准镜像，以确保应用在不同环境间的一致性和可移植性。此举有效规避了环境差异导致的部署风险，并提升了 SRE 团队在应用部署和制品管理方面的效率。

二是确保应用编排具备跨平台一致性。尽管底层芯片架构和操作系统存在差异，但应用层的编排逻辑及关键策略（如优雅退出机制、自动扩缩容策略等）保持统一。这种设计增强了应用在多样化软硬件环境中的行为可预测性，从而显著降低运维复杂度。

三是构建异构环境下的应用智能化调度能力。该能力的核心在于实现应用需求与基础设施特性的精准匹配，通过动态感知芯片架构、操作系统版本等底层差异，自动将应用调度至最优计算节点。例如，在双轨运行模式下，针对国产化架构优化的应用可自动分配至相应芯片节点，而未适配国产化的应用则精准部署至存量环境。这种精细化调度机制有效规避了异构环境中的兼容性问题，显著提升了复杂应用场景的执行效率。

四是实现基础设施应用负载的全栈信创适配。基础设施需全面兼容主流操作系统和处理器架构，支持包括 CentOS、Windows、KYLIN、统信 UOS、OpenEuler 等操作系统，以及 Intel、AMD、海光、鲲鹏、飞腾等硬件架构，从而为证券行业信创转型提供坚实的底层技术支撑。

随着深度学习训练、实时推理等人工智能工作负载在智能投顾、高频交易及风险监控等业务场景中的广泛应用，GPU 资源的高效管理已成为提升业务效能的关键要素。“基于信创合规的 AI 场景 GPU 能力扩展”方案依托系统化的 GPU 资源与应用治理架构，构建了支持异构芯片、满足信创合规要求的高性能 GPU 算力消费体系，为证券业务提供稳定、弹性且高效的算力支撑。

该方案的核心技术架构通过三个层次实现能力扩展。在 GPU 池化扩展层面，采用节点池技术对物理 GPU 资源进行统一纳管，支持按需分配与弹性伸缩，从而显著提升资源利用效率；在虚拟化扩展层面，运用算力与显存的细粒度切分技术，将单张物理 GPU 划分为多个虚拟设备，实现资源的隔离分配与动态共享；在智能调度引擎层面，通过优先级调度与抢占机制，确保关键任务获得优质算力，并结合跨节点远程调用功能，实现数据中心级资源的灵活调配。

基于上述扩展能力，本方案能够有效支持证券业务中多种智能化场景的算力需求。在训练任务场景中，针对需要大规模 GPU 进行分布式训练的深度学习模型（如投资策略模型和欺诈检测模型），GPU 池化能力可快速弹性供给大规模算力，显著缩短模型训练周期；在推理服务场景中，面对智能投顾、实时行情分析等具有潮汐特性的业务，虚拟化扩展能力可实现 GPU 资源的动态共享与复用，在保障低延迟响应的同时，大幅提升资源利用效率，以应对业务波动；在风控评估场景中，针对复杂实时风控模型对算力供给稳定性及突发承载力的高要求，智能调度引擎通过优先级保障和弹性扩容机制，确保在市场剧烈波动时风控系统获得充足算力，保障高效运行。

GPU 扩展方案通过资源池化、虚拟化切分与智能调度的协同作用，为证券行业构建了一套高效、灵活且符合信创要求的 GPU 算力消费体系。该方案不仅解决了异构 GPU 环境的管理挑战和资源利用率低下的问题，更重要的是，以平台化方式为各类 AI 业务提供按需取用、稳定可靠的算力供给，从而直接支持智能投研、智能风控、智能营销等业务创新，助力证券公司在数字化转型进程中提升核心竞争力。

2.7.4 游戏领域算力消费场景扩展

游戏服务治理作为综合算力调度体系在游戏领域的算力消费专项实践，致力于解决游戏业务特有的技术挑战。

随着游戏行业从简单的二维图形向复杂的三维场景及大型多人在线游戏发展，游戏质量和复杂度的不断提升带来了显著的算力需求增长。游戏业务呈现出与传统互联网服务显著不同的特征：其架构复杂度高，服务拓扑涉及登录、匹配、战斗等多个微服务模块；存在大量有状态服务，如玩家会话、实时游戏状态等数据需要持久化保障；对网络延迟具有极高的敏感性，实时竞技类游戏要求毫秒级响应能力；具备明显的地域亲和性需求，需要实现玩家就近接入；同时还需支持全球化多云部署架构，以应对不同地区用户的访问需求。

随着数字经济的蓬勃发展，强化算力基础设施建设、提供场景化的算力保障已成为各企业发力的重要方向，游戏行业作为算力消费的重要领域，亟需专业化的算力消费方案。

基于游戏业务面临的技术挑战，深入分析其架构特征是构建有效算力消费方案的前提。游戏业务架构呈现出典型的三层架构，各层级的职责与交互关系对综合算力调度体系提出了明确的定制化要求。

在架构分层上，接入层主要承担流量分发、网络加速、反 DDoS 防护、更新检测等安全与流量优化职能，主要通过负载均衡器实现流量的智能分发与高可用保障。局外服务层

涵盖登录、商城、社交、匹配等核心业务模块，依托游戏数据库与缓存中间件进行数据持久化与状态管理，构成游戏业务的基础服务单元。局内服务层则以专用游戏服务器为核心，由服务器集群管理节点统一调度资源池，负责实时对局的状态同步与运算处理。

该架构模式决定了算力调度方案需满足一系列特定要求：首先，必须保障有状态服务数据的持久化一致性，确保玩家会话与游戏进度的可靠存储；其次，需实现流量接入层面的高可用保障，避免单点故障影响全局服务；此外，应支持大规模版本发布与变更的平滑实施，最大限度减少运维操作对玩家体验的影响；尤为关键的是，需具备根据业务负载进行动态扩缩容的能力，以有效应对游戏业务固有的波峰波谷特征——例如，工作日与节假日间的玩家数量可能呈现数量级波动。这些特性共同构成了游戏场景下算力消费功能设计的核心考量维度。

在明确游戏业务架构特征及其对算力调度的特定要求基础上，需要构建一套针对性的核心扩展能力框架，深度整合的综合算力调度基础设施、算力市场与供给管理等核心基础，并通过定制化应用编排、应用网络管理、应用发布策略与弹性伸缩的完整技术体系，系统化应对游戏场景在弹性、性能、可靠性等方面的独特挑战。值得注意的是，在落地实践过程中，需确保各组件具备高内聚、低耦合的特性，支持独立演进与弹性扩缩容。

应用编排能力通过扩展自定义资源定义和专用控制器，为游戏有状态服务提供深度支撑；网络能力扩展聚焦南北向与东西向流量治理，满足低延迟与多云互联需求；应用发布策略建立标准化部署流程，保障全球业务的可靠交付；全局弹性扩缩容则通过多层监控指标驱动资源动态调整。

各核心能力相互配合，构成了游戏服务治理的技术基石。这种通过平台工程方法，将游戏场景的算力消费能力封装为游戏领域专用解决方案，不仅有效应对游戏业务在服务可靠性、网络性能与资源效率方面的核心需求，更为 SRE 团队提供了标准化、自动化的治理工具链，构建了可持续发展的技术治理框架。

接下来深入探讨应用编排与网络管理两大关键能力在游戏场景下的具体实现机制。这两项能力直接决定了游戏服务治理的精细度与可靠性水平，是保障玩家体验的技术基石。

在应用编排层面，游戏服务治理通过扩展 Kubernetes 自定义资源定义（CRD）与专用控制器，为游戏有状态服务提供深度支撑。其核心机制包括：服务状态保持功能，确保游戏服务器实例完全就绪后才将其纳入负载均衡池，从而实现业务无感知的平滑扩容过程；优雅退出功能通过监听游戏会话状态，确保仅在玩家对局结束后才执行服务器缩容操作，避免强制中断玩家体验；智能删减策略则根据实时在线玩家数量、服务器负载等业务指标，

动态优化算力资源的回收优先级与时机。这些机制通过事件驱动架构进行异步解耦，共同保障了游戏会话的连续性、数据一致性及服务稳定性，有效应对游戏这一典型有状态业务的独特需求。

网络管理层面通过定义专门的 Ingress CRD 实现负载均衡器 端口分配与绑定规则的标准化管理。其关键技术能力主要包括：通过 CLB 端口池等游戏特化接入方案，满足海量游戏服独立寻址的需求；采用 CLB 直通架构减少网络传输跳数，显著降低端到端通信延迟；提供统一接入抽象层，屏蔽底层多云环境差异，实现跨云网络策略的一致性管理。网络架构全面支持 Underlay 模式，可适应物理机房及边缘节点的特殊网络环境，确保跨节点、跨机房流量的高效连通。应用编排与网络流量协同层面，玩家会话状态与连接生命周期管理深度耦合，需确保连接级精细化治理能力，特别是在版本发布过程中，必须维持长连接会话的连续性，避免因网络抖动或实例变更导致的玩家掉线。

在完成应用编排与网络能力的基础构建后，大规模游戏应用的发布机制成为保障全球业务高效交付的关键环节。该机制基于 GitOps 工作流实现，通过声明式配置与自动化管道，实现对多云多地域游戏服务的统一部署与精细化管理。

在具体实施中，针对游戏业务的全球化部署需求，采用共享 ArgoCD 集群与独立 ArgoCD 集群并存的混合管理模式，来确保不同业务线或区域部署的资源隔离与安全管控。整体架构同样需支持灵活设定目标发布集群范围，可根据业务容灾要求配置主备、两地三中心等差异化部署策略，实现数万级游戏服实例的精准调度。

通过 GitOps 的版本控制与不可变基础设施理念，所有部署配置均以代码形式存储于 Git 仓库，当进行全球同步发布时，仅需修改 Git 仓库中的配置值，即可触发自动化流水线完成跨区域部署，实现发布效率大幅提升。当单一区域发生故障时，可基于版本记录快速重建集群环境，实现跨地域业务恢复时间控制在分钟级别。结合全局负载均衡机制，确保版本发布过程中玩家流量自动调度至健康区域，维持全球业务的持续服务能力。

该发布机制已在实际业务场景中得到验证，在和平精英等头部游戏的全球同步更新中，实现超过 3 倍的发布效率提升，并保证版本发布期间玩家无感知。通过标准化、自动化的发布流程，为游戏业务提供了稳定可靠的全球化交付基础。

在完成应用编排、网络管理及大规模发布等基础能力构建后，需要建立精细化的弹性扩缩容机制以应对游戏业务的动态负载特征。游戏服务通过三层扩缩容体系实现服务等级协议保障与资源成本优化的平衡，该体系涵盖实例级、应用级与集群级弹性调度，形成基于游戏状态感知的交互式扩缩容架构。

游戏应用编排控制器通过智能删减规则引擎，在缩容过程中优先选择玩家数量最少或会话即将结束的游戏服务器实例进行操作，确保玩家对资源调整过程无感知。增强型水平自动扩缩容机制基于游戏业务状态指标（如实时玩家连接数、对局活跃度）设定实例扩缩策略，实现基于业务感知的精准弹性决策。集群自动扩缩容组件则根据全局资源利用率与节点负载情况，动态调整底层容器资源供给，实现从应用到基础设施的全栈弹性能力。

这种基于业务指标的弹性扩缩容策略，在大型多人在线游戏中得到充分验证。系统通过实时监控游戏服务器状态，在检测到玩家在线数量下降时自动触发缩容流程，同时确保正在进行的游戏会话不受影响。当玩家即将退出游戏时，系统会延迟缩容操作直至会话自然结束，避免强制中断玩家体验。这种精细化的状态管理机制，在保障服务质量的同时显著提升了算力使用效率。

该交互式扩缩容机制已在实际业务场景中验证其有效性，在保障游戏服务稳定性的同时，将资源利用率提升40%以上，为游戏业务提供了高效可靠的弹性算力支撑。

在完成弹性扩缩容机制构建后，需要进一步解决游戏业务固有波峰波谷特性带来的算力资源利用率问题。游戏业务普遍存在资源使用率波峰波谷现象，若没有有效的弹性能力和配置，波峰期需预留大量资源，而未达到预期时则会产生明显浪费。闲时算力复用机制通过分时复用策略，实现算力资源在游戏在线业务与离线业务间的智能调度，显著提升整体资源利用率。

该机制采用多层次策略体系实现精细化的算力调度。水位策略基于资源利用率阈值设定冗余资源缓冲池，确保突发流量场景下的业务稳定性；定时策略则根据游戏业务周期特性，预设算力流转时间计划，如工作日低峰期自动触发算力重分配。独立算力池的建立为定向供给提供基础架构支撑，通过优先级标签确保游戏在线业务享有最高级别消费保障，闲时算力则可定向供给AI训练、科学计算等特定业务场景。

在实际架构设计中，云原生技术为算力复用提供基础支撑。通过容器服务实现资源池化，并利用弹性伸缩能力自动应对游戏高峰期与波谷期，在保证游戏服务质量的同时提高资源利用率。平台可采用类似“算力券”的机制，通过补贴形式鼓励闲时算力消费，进一步优化资源配置。有平台实践表明，通过领先的调度算法整合分散算力，可实现细粒度的千卡级别算力共享，每天提供两万小时以上的稳定算力。

该策略的核心优势在于实现多项目算力池共享与成本分摊。设备资本支出根据分时复用模式分摊到不同业务线，使得游戏业务只需承担实际消费时段的成本，大幅降低总拥有

成本。通过构建完善的闲时算力复用体系，游戏企业可在保障主营业务服务质量的前提下，实现资源效率最大化，为可持续发展提供坚实基础。

2.8 算力能效与安全控制

随着人工智能、大模型推理训练等需求推动，算力产业规模高速扩张，算力中心的用电量占全国社会用电量比例持续上升，部分区域数据中心甚至成为电网峰值负荷的主要来源。若不加以调控，数据中心资源消耗的野蛮式增长比如对国计民生产生不良影响。国家层面已通过《数据中心绿色低碳发展专项行动计划》等文件，推动算力与电力协同优化，降低全国平均 PUE 值，若企业无法满足能效标准，将面临政策合规风险，其市场竞争力也会遭遇挑战。

算力作为数据经济的核心生产力，其安全直接关系到经济稳定、社会治理和国防安全。在当前社会，算力产业安全面临多维度复杂威胁，算力数据可能因为采集造假、存储泄露、分析投毒等原因导致失真，进而导致算力结果错误，引发决策失误；算法也可能存在设计缺陷，监管缺失而被恶意操控、资源滥用；算力调度系统由于其复杂性，可能出现流量被劫持，数据被篡改等情况。由于算力安全的重要性，国家政策文件《数字中国建设整体布局规划》及《算力基础设施高质量发展行动计划》均将算力安全列为国家安全治理的重要领域。

鉴于算力产业对能源、环境、安全的多维度影响，通过必要的技术手段对算力能效进行持续调控，对算力安全进行有效保护成为综合算力系统的必不可少的能力。

2.8.1 算力能效控制

在 SRE 平台工程实践中，算力能效控制是站在全局，通过统一的方式对算力使用效率进行度量和调优。在很多企业中，算力的供给方和消费方是分开的，供给方负责将算力卖出去，KPI 是卖出的量，至于消费方是谁，是否合理使用，供给方没有查看动力和途径；算力消费方只管购买算力满足应用部署需求，消费方虽然关注成本，但优化手段有限，只能通过不断优化应用来压榨算力空间。供给方和消费方虽然都是算力调度的重要参与方，确因为缺乏统一的目标和手段而导致算力存在不同程度的浪费。在 SRE 综合算力调度系统中，算力调度的参与方由三方构成：算力供给方、算力消费方、算力调度方（平台方）。算力调度方提供算力全局数据视图，并沟通协调供给方和消费方，提出算力利用率目标。算力供给方在卖出算力同时，需要通过算力全局数据合理控制水位，避免过度采购。算力

消费方通过算力全局数据，合理预测资源需求，通过平台能力实现弹性算力消费。在此基础上，算力调度方也应提供一些列能力，帮助算力供给方和消费方之间、算力消费方和消费方之间实现算力售卖、共享、租借等流转方式，提升算力使用效率。

在具体技术实践上，算力能效控制主要包含能效评估和能效优化两点。

2.8.1.1 算力能效评估

能效评估体系的建设可以参考以下设计要点：

2.8.1.2 评估指标体系的设计

算力能效评估体系的构建需涵盖多维度的评估指标，如算力资源利用率、资源分布状况、单用户承载能力、算力预测执行精度以及新技术适应能力等。评估指标权重应具备场景相关性，可根据不同应用场景动态调整，以提升评估方案的可扩展性与适应性。同时，评估对象需构建分层可控体系，涵盖计算节点级、集群级、数据中心级、应用级直至业务单元级等层级，实现逐层细化评估，确保评估粒度可控。

2.8.1.3 能效数据采集和治理

数据源需实现多维度覆盖，涵盖以下类别，1）硬件相关数据：包括服务器级GPU/CPU 实时利用率、功耗（如通过 NVIDIA DCGM 监控）、温度、PUE 值、冷却系统效率（如水温、流量）等；其中部分关键指标（如 DCGM 监控的功耗）精度需达到±0.5%。2）应用软件相关数据：包括任务队列时延、任务成功率、作业完成率、容器资源配额占用率、集群装箱率、访问请求平均延迟、算法训练收敛速度等。能效数据应确保高准确性与完整性。需建立机制：对异常值进行自动识别与处理（自动剔除或修正）；对缺失值进行自动补偿（通过时间序列预测或邻近值关联修复），确保数据的连续性。数据标准化是评估指标具备合理性与有效性的关键因素。在 SRE 平台工程实践中，需实施指标归一化处理，保障单位统一。能效数据可通过 Prometheus、Kafka 或定制的可观测系统等渠道进行实时采集。数据采集的完备性与评估指标的精度呈正相关关系。

2.8.1.4 能效评估模型与结果反馈

能效评估体系采用五维分层架构，核心层由算效水平、智效水平、碳效水平、可获得水平及可持续水平构成垂直评估框架。算效水平通过量化硬件基础能效（涵盖理论算力、实测任务性能与资源利用效率，以 GFLOPS/W 为核心指标），形成系统能效基准；智效水平聚焦 AI 场景适配性，基于 AI 算力有效利用率、混合精度支持能力（如 FP16/INT8）及

模型精度衰减率等指标，评估算力与智能业务需求的匹配度；碳效水平整合能源与环境因子，包括碳排强度（$kgCO_2$ /kWh）、PUE 联动系数（制冷能耗占比）及碳利用率，驱动低碳计算实践；可获得水平保障服务连续性，依托分钟级资源可调度率、秒级故障恢复时延及跨域协同成功率（如多云/边缘节点调用）构建可用性度量；可持续水平支撑全生命周期管理，通过硬件折旧能效比（年 GFLOPS/W 衰减率）、扩容兼容性（单机柜算力密度提升空间）及绿色能源渗透率（风电/光伏占比）优化长期运维。

能效评估算法的标准化方法基于行业规范与场景化指标体系构建。在基础维度层面，严格遵循《大数据中心算力评估规范》（DB23/T 3512-2023）框架，确立五项核心指标：通用算力维度以 CPU 双精度浮点性能为基准（TFLOPS@FP64），智能算力维度量化 GPU 及 AI 加速卡的单精度/半精度算力（TFLOPS@FP32/FP16），算效能效比维度表征单位功耗的算力输出效率（GFLOPS/W），存储性能维度涵盖容量（TB）与每秒输入输出操作数（KIOPS），网络能力维度则测量带宽吞吐量（Gbit/s）。通过 TOPSIS 多属性决策算法对上述指标进行归一化处理与综合评价，最终输出 1~5 星级数据中心能效评级（综合评分≥0.85 分授予五星认证）。

为满足人工智能场景的专项评估需求，中国信息通信研究院（CAICT）提出"单位瓦特有效运算次数"（OPS/W）度量体系：训练场景采用浮点计算能效（FLOPS/W）与样本处理能效（Samples/W）作为核心观测值，推理场景则聚焦 Token 生成能效（Tokens/W）与任务吞吐能效（Tasks/W）。该体系通过动态适配机制，针对图像分类（ImageNet）、大语言模型推理（MMLU）等典型业务场景，依据其负载特征定制专属测试数据集与收敛判定标准，实现评估模型与实际业务需求的精准映射。

2.8.1.5 算力能效优化

算力能效评估能对当前算力的整体使用状况给出观测和分析数据，依据这些数据进行能效优化提升是算力能效控制的终极目标。算力能效优化的核心措施可以归纳为以下三个维度：

2.8.1.6 构建统一的算力池，整合碎片资源

为应对企业算力资源分散化与异构化挑战，需构建跨层级、多来源的算力整合体系。该体系需覆盖企业本地碎片化算力（如空闲 CPU 核心、闲置 GPU 卡）、公有云弹性算力及边缘节点算力，通过跨云管理平台实现多源异构资源的统一纳管与可视化监控。在此基础上，采用算力池化技术对异构资源进行抽象整合与归一化处理，形成逻辑统一的资源供

给层，消除底层硬件差异对上层业务的影响。通过池化架构，可建立全局编排调度引擎，实现跨地域、跨集群的智能任务分发与负载均衡，并基于统一服务目录提供标准化的算力计量与定价模型，支撑资源按需分配与成本透明化。为确保资源调度的可信性与安全性，需构建审计与隔离双重保障机制：一方面建立基于区块链或可信执行环境（TEE）的资源确权体系，实现算力归属与使用行为的不可篡改记录；另一方面通过硬件虚拟化（如vGPU 细粒度切分）与容器级隔离技术，保障多租户任务在共享异构算力池中的安全隔离与性能稳定性。

该体系最终形成“算力纳管-池化整合-智能调度-可信运营”的全链路闭环，显著提升碎片化算力的综合利用效率，降低企业用算成本，并为大规模异构算力协同提供技术基础。

2、优化基础设施，提升原子算力能效

基础设施优化的核心路径是通过跨层软硬件协同，实现单设备能效提升。典型策略主要包括两类：弹性算力技术与非业务算力消减。弹性算力的实现涵盖软件与硬件双重维度：在软件层面，通过基于负载的弹性扩缩容机制（如 Kubernetes HPA 水平扩缩容与 VPA 垂直资源调整）实现资源动态调度；在成熟的 SRE 平台工程实践中，依托 vCPU/vGPU 虚拟化技术，可支持 CPU/GPU 以 0.1 核（卡）为粒度的精细化分配。硬件层面则依赖动态电压频率调节技术（DVFS），通过实时负载感知动态调节设备电压与频率，有效降低动态功耗。另一方面，虚拟化环境中虚拟化管理层（如 Hypervisor）的固有开销会占用业务算力资源。采用轻量化替代方案（例如以容器化架构取代传统虚拟机），可显著减少管理层损耗，间接提升有效算力的能效比。

3、通过多维度智能化调度，实现全局最优算力分配

多维智能调度系统通过 AI 算法预测任务负载波动趋势，动态优化算力分配策略，从而提升整体调度效率与算力利用率。该技术作为全局性优化手段，其核心策略涵盖时间维度、任务类型、地理分布及成本因素的智能协同调度，具体应用场景如下：

在时间维度上，系统基于业务负载的周期性波动规律实施动态算力分配。例如，在业务高峰期优先保障关键任务的算力供给，而在负载低谷期则主动整合闲置算力处理轻量任务，实现算力利用率最大化。此种调度模式通过"削峰填谷"机制有效平衡算力供需矛盾，避免算力闲置与过载风险。

在任务类型维度，系统通过调度亲和性优化提升任务与算力的匹配精度：将高带宽需求的 AI 训练任务绑定至 GPU 集群节点；将低延迟要求的实时推理任务调度至边缘计算节

点；利用批量离线任务填充非高峰时段的闲置算力。这种差异化调度策略显著降低任务等待时间，同时提高异构算力的适配效率。

在地理维度，系统遵循数据局部性原则，依据数据传输成本与时延敏感度动态选择计算节点。对于依赖本地数据的任务，优先调度至数据存储节点或邻近区域节点，大幅减少跨域数据传输开销。该策略尤其适用于分布式计算环境，可降低网络带宽压力并提升端到端计算效率。

在成本维度，系统集成动态定价模型与算力成本图谱，通过实时比选不同算力节点的单位成本，在保障服务质量的前提下自动选择最具成本效益的算力组合。该机制可有效平衡性能需求与经济性目标，降低企业整体算力运营成本。

通过 AI 驱动的跨维度协同调度，企业可进一步拓展应用场景。例如结合强化学习实现多目标动态权衡，或基于联邦学习构建跨域算力池协同机制。实际部署需结合业务特性定制策略，持续优化调度系统的适应性与扩展性。

2.8.2 算力安全控制

当前算力环境面临多重安全挑战：首先，分布式架构的复杂性显著增加了安全管理难度——随着系统节点数量激增，跨节点数据传输过程易受篡改威胁；而在云原生环境中，容器的动态编排特性使得安全策略难以实时匹配算力资源调度需求，进一步扩大了系统的暴露面。其次，数据安全与隐私合规压力持续加剧：集中化存储的数据资产成为攻击者重点目标，同时跨境数据流动需满足多国差异化法规（如 GDPR、CCPA），企业承担高昂的合规成本与持续防护压力。最后，算力资源滥用风险日益严峻：恶意挖矿软件会非法占用闲置计算资源导致运营成本激增，更严重的 DDoS 攻击则可利用算力发起大规模流量攻击，造成关键业务瘫痪（例如游戏服务平台遭攻击导致玩家无法登录）。这些核心挑战共同凸显了构建多层次、动态化算力安全防护体系的迫切性。

2.8.2.1 算力安全概述

算力安全控制的核心设计遵循多层次纵深防御原则，在运行时、网络通信、环境隔离及数据存储等基础层面构建统一的安全策略支撑体系。核心架构依托容器化环境展开，重点强化应用安全防护，涵盖 API 安全（包括网关鉴权与应用隔离机制）与软件供应链安全保障（如镜像漏洞扫描与签名验证）；同步夯实基础架构安全，通过运行时入侵检测、零信任网络架构（实现微服务间加密通信与策略隔离）以及容器编排平台（如 Kubernetes）的安全加固措施，确保资源调度与执行环境可控。

横向层面深度融合 DevSecOps 实践，将代码审计、合规检查等流程自动化嵌入开发周期；强化可观测性能力，实现对服务拓扑、数据流及攻击面的实时监测；并通过安全管理模块建立持续威胁响应、策略治理与应急编排的闭环机制。底层能力上，该框架适配多元化云环境，整合隐私计算技术保障数据"可用不可见"，依托自研技术栈降低供应链风险，最终形成从硬件基础设施到上层应用、贯穿开发运维全生命周期的动态防护体系。

2.8.2.2 算力基础安全控制

算力基础安全控制体系通过多层次技术手段构建防护框架，贯穿"边界防护—入侵防御—资源隔离"三层架构，既应对外部网络攻击，也化解内部虚拟化环境风险，形成纵深防御的完整闭环。。在网络边界层，防火墙策略实现对南北向流量的精细化管控，依据预设规则阻断外部攻击并防止内部数据泄露，企业需定期更新策略以适应新型威胁。入侵检测与防御系统（IDS/IPS）形成第二道防线：IDS 实时监控网络流量并识别异常行为（如端口扫描），触发告警机制；IPS 则主动拦截攻击行为（如阻断恶意软件传播），实现动态防护。针对大规模分布式拒绝服务（DDoS）攻击，综合运用流量清洗技术过滤恶意流量，并通过带宽弹性扩容保障核心业务可用性。

在虚拟化资源层，虚拟机隔离依托 Hypervisor 加固技术阻断虚拟机逃逸攻击，同时通过资源分配策略避免计算/存储资源抢占，确保多租户环境稳定性。容器安全则聚焦全生命周期防护：容器镜像需经过漏洞扫描与恶意代码检测，运行时则持续监控异常操作（如未授权进程启动）。最终通过资源分组与隔离策略，根据业务需求对计算、存储及网络资源实施逻辑划分，配合细粒度访问控制策略，从根源上遏制资源滥用与越权访问行为。

2.8.2.3 访问控制与身份验证

在数字化时代，算力系统面临未授权访问、数据泄露、横向攻击等多重安全威胁，构建严格的访问控制与身份认证体系已成为保障系统安全稳定运行的基石。该体系通过三层防护机制确保业务安全：在身份核验层抵御凭证盗用风险，在权限管控层防止越权操作，在行为监控层阻断攻击渗透路径；同时满足 GDPR 等法规对敏感信息保护的强制要求，避免业务中断及合规处罚带来的双重损失。

算力系统的访问控制与身份验证体系构建需遵循分层防御与动态授权原则，核心涵盖多因素认证（MFA）、角色属性协同控权及零信任架构三大技术支柱。多因素认证（MFA）通过生物识别技术（如指纹/面部识别）与静态密码的双因子组合强化身份核验，典型应用场景包括计算节点登录时的人脸+密码双重验证；同时引入硬件令牌与动态验证码的二次

防护机制（例如调度 API 调用前需物理令牌生成的一次性密码叠加短信验证），显著提升攻击者破解成本，从根本上解决单点认证脆弱性问题。

在权限控制层面，采用角色基访问控制（RBAC）与属性基访问控制（ABAC）协同模型实现精细化授权。RBAC 按职能预设标准化权限组：运维角色获得节点启停等高权限操作，开发角色限制作业提交等基础功能，大幅降低权限配置错误率；ABAC 则动态评估三元组属性——用户属性（部门/职级）、资源属性（节点安全等级）、环境属性（访问时间/IP 位置），实现场景自适应授权（如财务部员工仅能在办公时段通过指定 IP 访问计费系统）。二者协同形成“静态角色定基准、动态属性调细节”的弹性权限框架。

零信任架构通过持续验证与微隔离技术构建纵深防御：每次资源访问均强制重新认证身份（如查询任务日志时需令牌重验），消除传统边界信任过度问题；微隔离技术将计算网络划分为独立安全域（如隔离 GPU 计算集群与存储集群），通过微分段策略限制攻击横向扩散半径，并基于软件定义边界（SDP）实施网络隐身，默认拒绝所有连接，按需开放最小通信路径。

整个体系在构建和实施中需严格贯彻两大安全原则：一是最小权限原则，要求初始仅开放必要权限（如测试人员仅访问开发环境），并通过周期审查机制依岗位变动回收冗余权限；二是动态授权原则，通过实时监测用户行为特征（如登录地点/频率异常），自动触发权限调整（如异地登录时临时关闭敏感操作）。二者共同形成“权限最小化→行为监测→动态调控”的闭环模型，在某行业算力平台实践中使未授权访问尝试下降 92%，权限配置错误率减少 75%（依据 NIST SP 800-207 标准验证），实现了安全防护与业务灵活性的有机统一。

2.8.2.4 数据安全

当前数据安全防护体系的构建受双重驱动力推动：其一，数字化业务跨网络边界交互的常态化显著扩展了数据攻击面，传统防护机制难以应对传输窃取与存储介质劫持等新型威胁；其二，GDPR 等法规对个人身份信息（PII）的严格管控要求企业建立精细化防护架构。为应对这些挑战，现代数据安全体系采用分层加密与动态脱敏双轨并行的技术路线。

分层加密技术贯穿数据全生命周期：在传输环节强制实施 SSL/TLS 协议（如全站 HTTPS 加密），通过加密通道防御中间人攻击与传输窃取风险，并借助定期协议更新与密钥轮转机制保障传输机密性与完整性；在数据处理环节融合对称加密（AES）与非对称加密（RSA）技术，兼顾批量数据的高效处理与密钥安全交换需求，重点防护内存计算过程中的敏感信息泄露；针对静态存储介质（数据库、云存储及物理磁盘），部署符合 NIST

标准的 AES-256 强加密算法，依托密钥管理系统实现自动化密钥轮转与细粒度访问控制，确保持久化数据的本质安全。

动态脱敏技术聚焦敏感信息智能管控：首先依据 GDPR、CCPA 等法规建立数据分类分级体系，精准识别 PII、金融账户等核心敏感字段；在此基础上，通过数据库代理网关实时解析查询请求，基于访问者角色属性动态生成差异化数据视图（如在客户服务场景仅返回手机号后四位）。该技术在不中断业务流程的前提下实现"可用不可见"（Visible but Inaccessible）目标，有效降低测试开发、数据共享等场景的原始数据暴露风险。

整套体系通过传输加密、处理防护、存储加固的三级纵深防御覆盖数据全生命周期，并基于加密策略动态优化机制（根据数据类型、敏感级别与访问场景自适应调整防护强度），实现业务效率与合规要求的有机平衡，最终形成以法规为基准、技术为支撑、业务可持续为目标的数据安全新范式。

2.8.2.5 安全审计与合规

算力安全审计与合规体系的构建需系统整合多维审计机制与标准化实施流程，形成贯穿算力全生命周期的安全保障闭环。在审计类型层面，首要实施系统操作审计，完整跟踪用户登录、操作行为及退出全链路，通过实时监测异常操作（如非常规时段权限变更）及时阻断未授权访问，保障操作合规性，例如追踪运维人员配置误操作事件。其次开展数据访问审计，精确记录数据访问、修改及删除轨迹，结合敏感数据标签（如个人身份信息字段）强化监控粒度，有效防范数据泄露与非法篡改，典型实现方案包括数据库审计网关实时解析 SQL 指令。同时必须执行合规性审计，周期性验证系统对法律法规（GDPR）、行业标准（ISO 27001）及内控政策的符合度，重点检查数据跨境传输加密强度与留存周期合规性，输出可落地的整改建议。此外需强化安全策略审计，动态评估防火墙规则、访问控制列表等策略有效性，依托 MITRE ATT&CK 等威胁情报库优化防御配置，通过季度攻防演练快速响应新型攻击手法。

审计实践需遵循三阶标准化流程：准备阶段首先界定审计边界（如限定边缘节点或核心集群范围），配置漏洞扫描器（Nessus）、配置核查工具（OpenSCAP）等基础工具链，金融企业可聚焦支付业务集群实施针对性审计。执行阶段通过主动漏洞扫描匹配 CVE 漏洞库识别系统缺陷，并借助 SIEM 平台（如 Splunk）关联分析操作日志与网络流量，具体实践包括检测未修复的 Log4j 漏洞、识别非常规数据导出行为，为风险处置提供实证依据。最终报告阶段采用 DREAD 模型量化风险等级（高/中/低），制定补丁升级、访问控制策略重构等整改路径，向管理层交付具备可执行性的优化路线图。

该体系通过操作审计（行为溯源）、数据审计（内容管控）、合规审计（规则治理）的三维联动构建纵深防御机制，并依托 OpenSCAP-Nessus-SIEM 工具链实现 70%审计流程自动化。在证券交易系统等场景应用中，该框架达成未授权操作 100%识别率，合规整改周期压缩至 72 小时内，印证其平衡安全性与业务连续性的核心价值。

3 SRE 团队的数据与 AI 场景（编写中）

4 SRE 团队的基础监控与应用观测 （编写中）

附录

1 参考文献

出版书籍：《SRE Google 运维揭秘》 ISBN：9787121297267

出版书籍：《Google SRE 工作手册》 ISBN：9787519845858

2 术语

此术语表描述了我们在白皮书行文中使用到的一些术语。

术语	定义
SRE	Site Reliability Engineering，即网站可靠性工程，是一套原则和实践，旨在创建和维护可扩展、高可靠和高效的软件系统。
SLI	Service Level Indicator，即服务水平指标，是衡量服务性能的具体指标。
SLO	Service Level Objective，即服务水平目标，是服务期望达到的性能水平。
SLA	Service Level Agreement，即服务水平协议，是服务提供者和客户之间关于服务水平的正式协议。
DevOps	Development and Operations，即开发与运维，是一套实践，旨在使软件开发（Dev）和软件运维（Ops）更加协同高效。

术语	定义
CI/CD	Continuous Integration/Continuous Delivery，即持续集成/持续交付，是一种软件开发实践，通过自动化的构建、测试和部署来加快软件交付过程。
AIOps	Artificial Intelligence for IT Operations，即 IT 运维的人工智能，是使用人工智能技术来改善 IT 运维的实践。
MTTR	Mean Time To Recover，即平均恢复时间，是从服务中断到恢复正常所需的平均时间。
MTBF	Mean Time Between Failures，即平均故障间隔时间，是系统在两次故障之间正常运行的平均时间。
PaaS	Platform as a Service，即平台即服务，是一种云计算服务，提供开发、运行和管理应用程序所需的平台和环境。
IaaS	Infrastructure as a Service，即基础设施即服务，是一种云计算服务，提供虚拟化的计算资源。
SaaS	Software as a Service，即软件即服务，是一种软件分发模型，用户通过互联网访问应用程序，无需安装和维护软件。
CDN	Content Delivery Network，即内容分发网络，是一种分布式网络，用于更有效地向用户分发内容。
API	Application Programming Interface，即应用程序编程接口，是一组规则和定义，允许不同的软件应用程序相互通信。
SDK	Software Development Kit，即软件开发工具包，是一组软件开发工具，用于帮助开发者创建应用程序。
BPM	Business Process Management，即业务流程管理，是一种系统化的方法，用于优化和管理组织的业务流程。
ECC	Enterprise Command Center，即企业指挥中心，是组织内部用于监控和管理 IT 运营的中心。
GOC	Global Operations Center，即全球运营中心，是组织内部用于全球范围内监控和管理 IT 运营的中心。