面向流量的自适应高可用架构

淘宝应用架构升级实践

李鼎 (哲良) 淘宝

2019.11.16

个人简介

- 李鼎（花名：哲良） Id：oldratlee
  - Github(https://github.com/oldratlee) / oldratlee<at>gmail.com / 微博 / …
  - Vimer

- 经历：移动中间件 / 数据驱动设施 / 业务系统 / 后端中间件
  - 目前淘宝基础平台，负责 基础架构 与 移动基础中间件产品。
  - 2009年加入阿里巴巴，就职期间曾负责 淘宝用户增长设施与平台、天猫直播业务、阿里云移动云 / PAAS、服务端RPC / 消息等中间件。

- 注重系统设计、工程实践与代码味道；开源活跃者。

- 开源项目 alibaba/transmittable-thread-local 的 Owner
  - https://github.com/alibaba/transmittable-thread-local
- 翻译集 https://github.com/oldratlee/translations

- Dubbo核心开发者之一
  - https://github.com/apache/dubbo

大纲

0. 关于高可用 HA

1. 面向流量的高可用

2. 自适应限流 实现方案与落地关键

3. 自适应高可用 后续规划与展望

0. 关于高可用 HA 及其关注点

High Availability

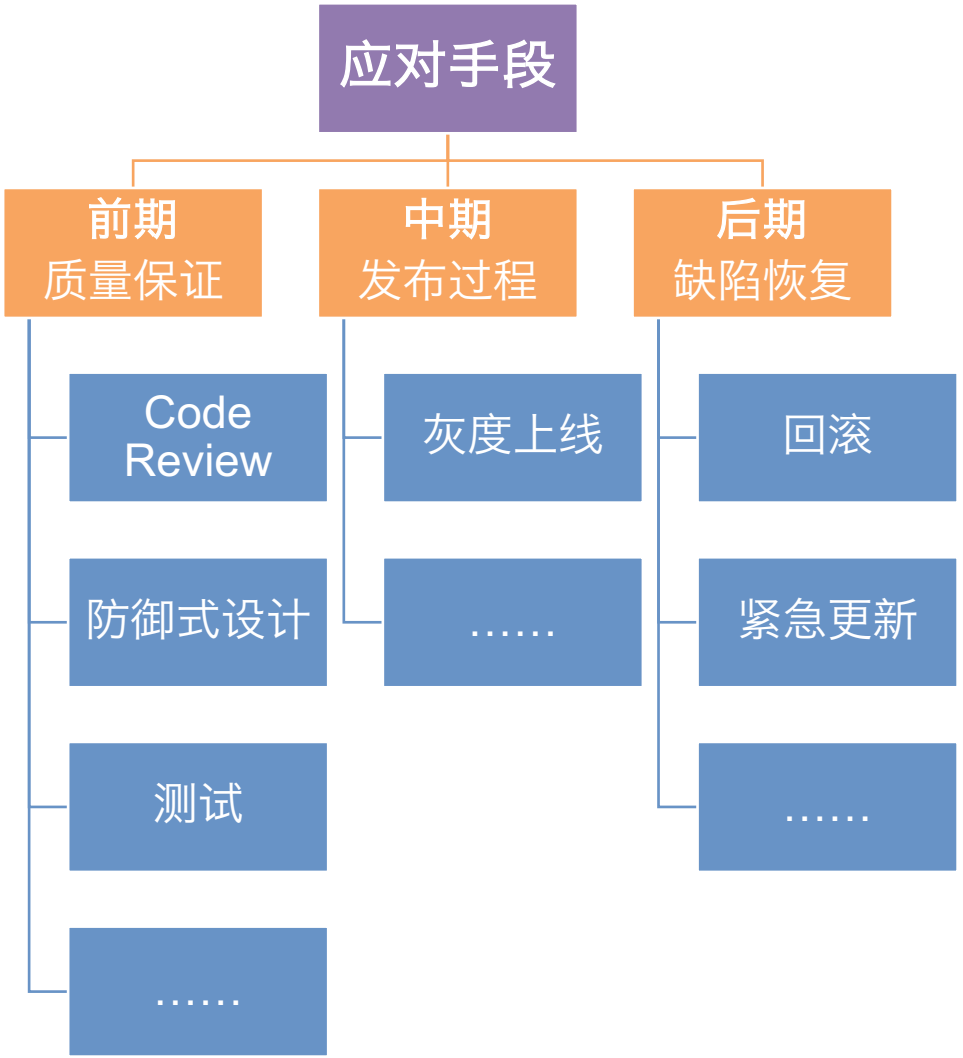
聚焦讨论高可用，不包含软件正确性的部分

软件正确性问题

- 应对 涉及
  - 质量保障(QA)
  - 代码质量
  - 发布流程
  - ……
- 业务代码逻辑问题，
  主要由各个应用来实现保障，
  而不是 一个基础功能 / 架构能力 能解决好的

这里将 正确性 从 可用性 区分出来

后面讨论可用性时，都不包含 正确性



说到 HA 时，会先想到什么？

讨论/强调的是 **资源失效**

- 资源 (机器/机房/光缆/……) 一定会出现 失效
- 服务/应用，能抵御 资源失效，不会**挂**了
- 核心策略 是
  - 资源冗余 / 资源隔离
  - 热切换、多活
- 各种 高大上的 分布式架构模式
- ……

从 资源失效 到 高可用(HA) 的 关注

积极应对

- 一定会发生的、不可控的
- 让服务挂的

因素，

保证服务不挂(HA)

准确地说，不挂 其实是 大大降低 挂掉的概率

对于你自己开发的应用，
或听到某某服务挂了时，
你会担心/想到什么？

对于你自己开发的应用，
或听到某某服务挂了时，
你会担心/想到什么？

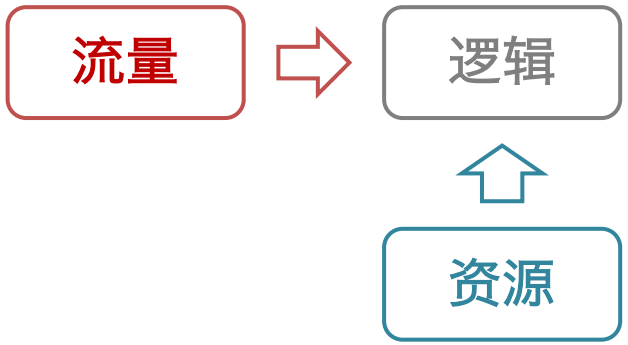
压跨了

大流量

突发流量

不偏科 的 高可用关注

高视角 的 系统模型



- 逻辑：对应 正确性问题，质量保障 / 发布流程
  各个业务应用是核心关注者
- 资源：需要应对**失效**，带来可用性问题
- 流量：需要应对**变化**，带来可用性问题

在应用架构上，业界对 **面向流量**的**可用性**，不如像 **面向资源** 一样

- 得到 充分的关注
- 有着 丰富成熟的思路
- 有着 广泛有效的实践

1. 面向流量的高可用

Flow Oriented HA

面向流量的可用性：现有的应对方式

静态限流

执行流程：

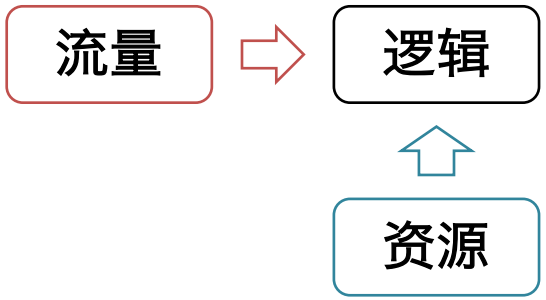
1. 录制/回放 流量
2. (全链路)压测出 **单机QPS**
3. 部署机器量
4. **设置 单机QPS限流阈值**

① 保障 部署的资源 支撑 **业务容量**

② **保障 可用性 (不压挂/过载)**

TOP1

静态限流的问题

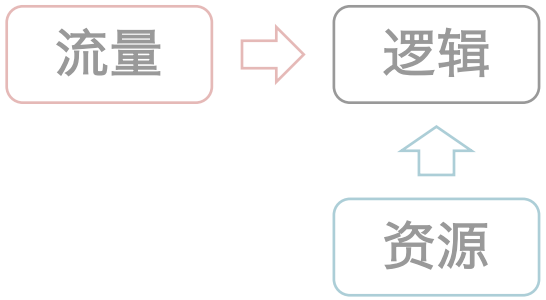


- **流量/请求**
  - 依赖：**请求模型** 的准确评估，即 测试流量的**请求大小**与实际一致
  - 热点流量，如热点用户被频繁访问/爆冷商品；运营导致用户动线走重逻辑分支
  - 依赖：来的**流量** 的准确预估
  - 要承认，流量一定无法准确评估，会被摸死（即拒绝操作导致过载）
- **逻辑**
  - 依赖：测试之后 **自己和下游依赖的逻辑** 不变，性能保持稳定不变
  - 业务总是在演进，除非全网封网，否则限流阈值一上线就潜在过时了
- **资源**
  - 依赖：各台 **机器性能** 一致 且 稳定不变
  - 机器型号不同，处理能力是不一致的
  - 虚拟化/容器化 之间有影响，你无法控制你的邻居会做什么
- **流程**
  - 依赖：**事先人工** 准确执行了评估过程
  - 只要人来执行就一定会 遗漏、出错
  - 对于 长尾应用 / 非核心应用，支持力度 没有保障；人力资源总是有限。
  - 说是 非核心应用，你确定？

100 CASE STUDIES

TOP1

静态限流的问题



- 流量/请求
  - 依赖：请求模型 的准确评估，即 测试流量的请求大小与实际一致
  - 热点流量，如热点用户被频繁访问/爆冷商品；运营导致用户动线走重逻辑分支
  - 依赖：来的流量 的准确预估
  - 要承认，流量一定无法准确评估，会被摸死（即拒绝操作导致过载）
- 逻辑
  - 依赖：测试之后 自己和下游依赖的逻辑 不变，性能保持稳定不变
  - 业务总是在演进，除非全网封网，否则限流阈值一上线就潜在过时了
- 资源
  - 依赖：各台 机器性能 一致 且 稳定不变
  - 机器型号不同，处理能力是不一致的
  - 虚拟化/容器化 之间有影响，你无法控制你的邻居会做什么
- 流程
  - 依赖：事先人工 准确执行了评估过程
  - 只要人来执行就一定会 遗漏、出错
  - 对于 长尾应用 / 非核心应用，支持力度 没有保障；人力资源总是有限。
  - 说是 非核心应用，你确定？

期望的解决方案

没有 **提前的** **人工的** 评估，

便没有 提前评估的过时 与 人的评估疏漏/错误！

自适应限流

实时 自动 评估QPS

业务流量 的 **不确定性**

与

技术方案 的 **自适应性**

天生一对！

Netflix 不大善于 预测 用户增长 或 设备增长。

这是采用原生能力公司背后的核心思考。

云原生思想的本质，就是

承认 无法对所提供服务能力的变化进行可靠的预测。

——《云原生Java》 Josh Long

TOP1

100 CASE STUDIES

业界的跟进与进展

- **Netflix**: Hystrix ⇒ Concurrency-Limits
- **Baidu**: BRPC auto concurrency limiter
- **Tencent**: Dagor
- **Google / Twitter**

- **Taobao: 诺亚(Noah)**

Hystrix Status

Hystrix is no longer in active development, and is currently in maintenance mode.

Hystrix (at version 1.5.18) is stable enough to meet the needs of Netflix for our existing applications. Meanwhile, our focus has shifted towards more adaptive implementations that react to an application's real time performance rather than pre-configured settings (for example, through adaptive concurrency limits). For the cases where something like Hystrix makes sense, we intend to continue using Hystrix for existing applications, and to leverage open and active projects like resilience4j for new internal projects. We are beginning to recommend others do the same.

TOP1

诺亚(Noah) 的收益效果

- **可用性提升**
  - 压垮 QPS 上限，最高可提升 **20 倍**于业务负载流量
  - 在大流量压力下降后，**1 秒快速** 恢复服务
  - 大流量压力下，仅需直接扩容机器一步即可，
    无需紧急调整限流
- **用户体验的优化**
  - 应用在负载情况下，服务成功率最高可提升 **2.7 倍**，
    同时响应时间维持正常水平**不劣化**
- **成本的优化**
  - 资源利用率最高可提升 **100%**
    （去除为了稳定性/不确定性而留的资源冗余）
- **效率提升**
  - 全链路压测/性能压测 更顺畅。无需人工限流阈值设置，
    避免人工评估错误导致系统被压垮后需要大量调整时间

100 CASE STUDIES

诺亚(Noah) 的落地实践进展

- 上线 超过 **9 个月**
- 部署容器 **15K+**
- 上线 大促会场、直播、导购、群聊等 核心业务场景
- 涉及 淘宝、天猫、优酷、盒马、聚划算、猫超、达摩院等等多个业务
- 经历了 618、99、双11 的大流量规模实战验证

面向流量的自适应高可用 的 思路、方案与实现

完成了

在**广泛**业务**核心**场景上 **大规模 大流量**的 验证与实战。

2. 自适应限流
实现方案与落地关键

Adaptive Flow Limit

诺亚自适应限流实现方案

算法原理

- P 比例，着眼现在（目标差距）
- I 积分，回看过去（偏差积累）
- D 微分，估计未来（趋势变化）

负反馈闭环

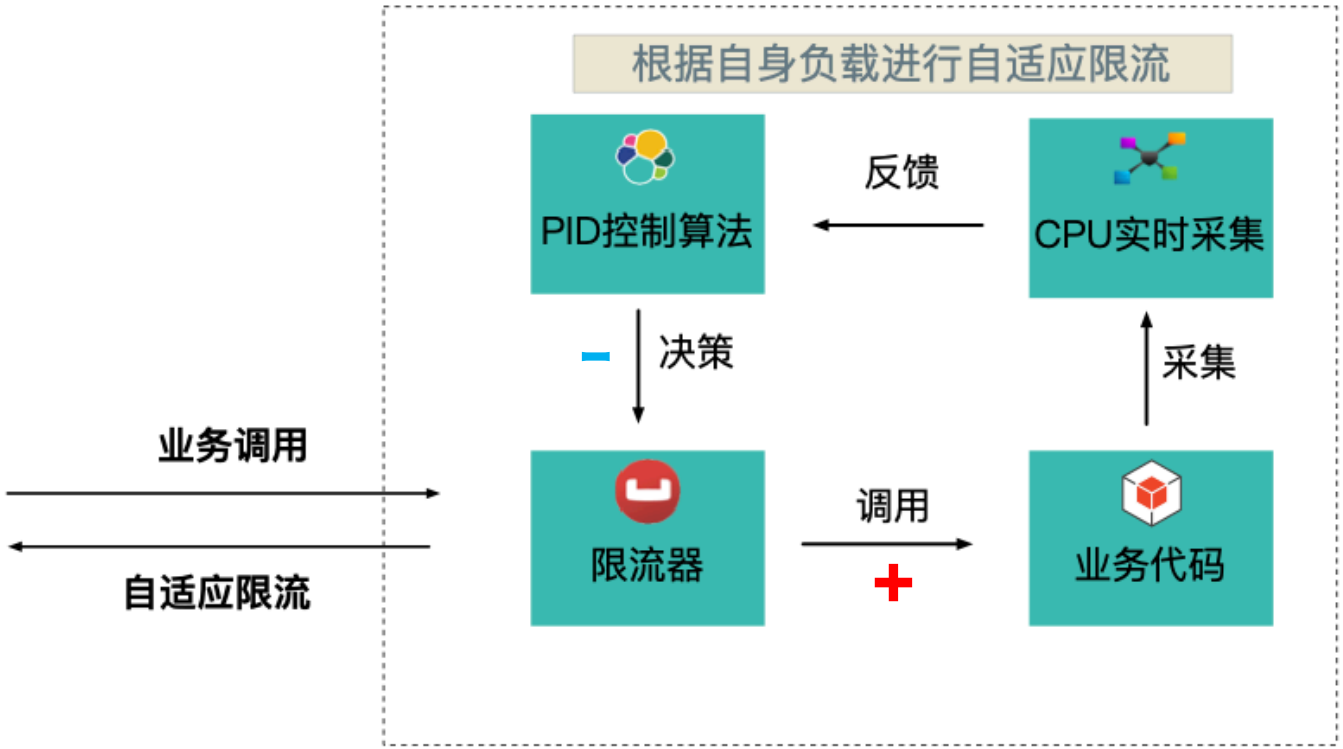
- CPU采样 + PID 控制器 + 流量控制器
- 流量控制器 控制 进入系统的流量
- 进入系统的流量改变 反馈成 CPU 利用率变化

容量感知

- 流量控制器 根据 PID 输出的可接受流量

控制手段

- 实时就地 拒绝 过载流量



实现关键难点

稳定性能力，实现质量有最高要求！

- **完善的验证Case**
  - 全面场景覆盖
  - 充分测试验证
- **数据驱动**
  - 效果指标体系
  - 数据分析
- **工程效率 敏捷的迭代工程体系**
  - 调参之后，全量回归
  - 回归测试自动执行
  - 数据分析要 (半)自动完成

多维度的测试验证

覆盖 20+类 测试场景

- 请求流量（QPS）类型
  - 阶跃 / 脉冲 / 斜坡流量
- 计算类型
  - 占用 CPU时长
  - 纯计算 / 等待 时间比例
- 应用 单服务 vs. 多服务
- 后台任务消耗
  - 进程内 vs. 进程外 任务
- 宿主机/虚拟化
  - CPU Set vs. CPU Share
- 容器处理能力
  - 容器规格 / 宿主机性能
- ……

充分测试验证

- 执行 **1300+次** 场景压测评估
  - Case压测执行
    1小时 / 30分钟 / 5分钟
  - 通过 评估系统
    自动化执行 指标评估
- 历时 **5个月** 测试与评估

自动化的测试验证 与 指标评估

评估系统

指标数据数值分析

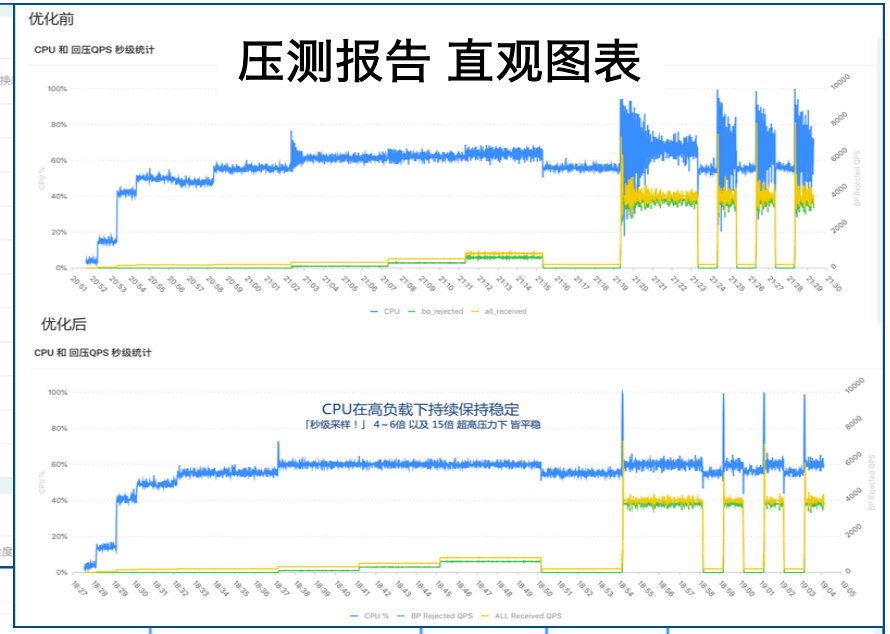
分析报告 目标值(targetValue) : CPU 60%

序号	进入调节的时间点	完成调节的时间点	调节时长	稳定值的误差率	调节中抖动的幅度	调节中的最小值	调节中的最大值	调节前的稳定值	调节后的稳定值	调节前接收QPS	调节后接收QPS	QPS切换开始时刻	QPS切换完成时刻	QPS切换
0					~0.00 %	42.26%	67.38%	42.26%	67.38%	100	160	12:07:43	12:07:51	8秒
1					~0.00 %	65.06%	99.27%	65.06%	99.27%	160	256	12:12:42	12:12:51	9秒
2					~0.00 %	99.27%	99.27%	99.27%	99.27%	253	438	12:18:24	12:20:29	125秒
3					~136.93 %	41.90%	99.27%	99.27%	41.90%	375	100	12:22:57	12:23:42	45秒
4					~0.00 %	41.90%	99.65%	41.90%	99.65%	100	1934	12:28:44	12:28:47	3秒
5	12:32:41	12:33:40	0 秒	-66.08 %	0.00 %~0.00 %	99.65%	99.65%	99.65%	99.65%	1912	1160	12:32:54	12:33:40	46秒
6	12:33:41	12:34:40	0 秒	-12.41 %	0.00 %~0.00 %	67.45%	67.45%	67.45%	67.45%	100	104	12:33:41	12:34:40	0秒
7	12:34:41	12:35:40	0 秒	-12.41 %	0.00 %~0.00 %	67.45%	67.45%	67.45%	67.45%	100	1408	12:34:45	12:35:40	55秒
8	12:35:41	12:36:40	0 秒	-12.41 %	0.00 %~0.00 %	67.45%	67.45%	67.45%	67.45%	100	100	12:35:41	12:36:40	0秒
9	12:36:41	12:37:40	0 秒	-12.41 %	0.00 %~0.00 %	67.45%	67.45%	67.45%	67.45%	100	1995	12:36:46	12:36:49	3秒
10	12:37:41	12:38:40	0 秒	-12.41 %	0.00 %~0.00 %	67.45%	67.45%	67.45%	67.45%	248	100	12:37:41	12:37:42	1秒

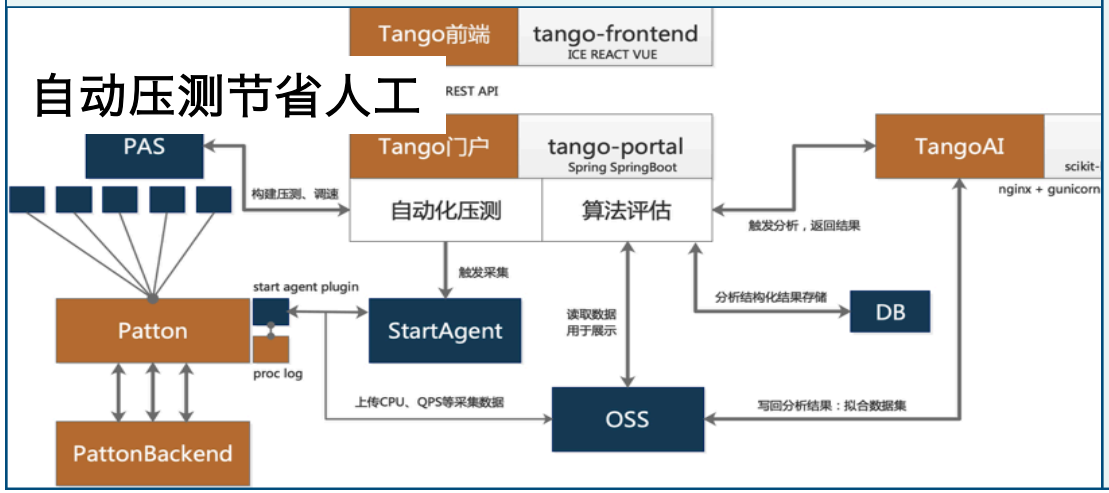
时序数据特征指标【指标定义】

CPU时序复杂度，越小越好	CPU到60%的均方根误差，越小越好	CPU的标准差，越小越好	成功QPS的均方根误差，越大越好	成功请求的RT的时序复杂度
52.99	32.72	15.86	223.61	

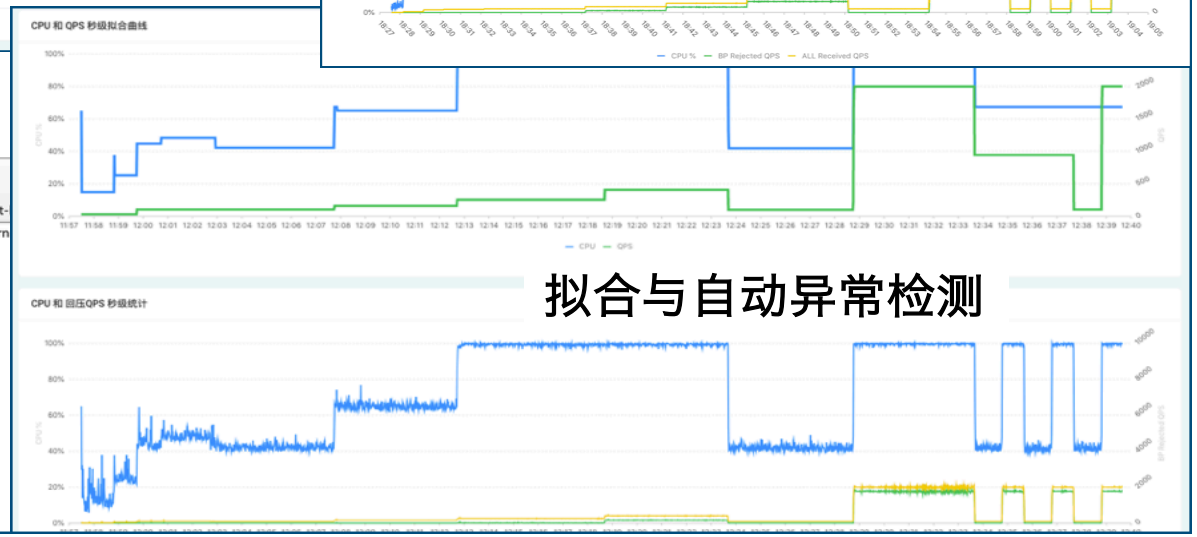
压测报告 直观图表



自动压测节省人工



拟合与自动异常检测



落地关键与打法

- **珍视 口碑**
  - 因为是稳定性能力，关乎业务生死，容忍度低。
  - 一旦有问题，口碑很难挽回，整体落地推广受阻。
  - 宁可实施计划后延，也不激进上线；尤其是前期的实施Case上线。
- **拥抱 现有稳定性体系**，如静态限流
  - 现有稳定性体系 多年实践，可靠可用。
  - 避免 在前期就需要完整体系，尽量复用 避免战线过长。
- **注重 专业算法同学加持**
  - 对于算法及时的思路纠偏 与 迭代分析思路 至关重要！
- **发现 痛点匹配的甜蜜应用**：逻辑多变、流量变化大难于预测的场景
  - 活动运营类应用，会场投放
  - 直播、群聊
- **重视 长尾应用，积极撬动 核心应用**
  - 因为核心应用的高保障力度，痛点可能暴露不明显，且心态上严谨用新
  - 长尾应用缺乏保障投入，有整体上稳定保障诉求

3. 自适应高可用
后续规划与展望

Future Plan

诺亚(Noah)的规划

- **自适应**能力 由 **限流** 拓展到 **隔离/熔断**等更多稳定性能力，如
  - 自适应 线程资源隔离
  - 自适应 服务比例
  - 自适应服务熔断 等等。
- 由 **单机的**自适应限流 拓展到 **链路级**，尤其是 客户端流量入口接入层
  - 与接入层协同，让 入口流量 与 应用的处理容量 自适应匹配
  - 确定性地保障 面向流量的高可用
- 自适应 **控制流量** 拓展到 自适应 **伸缩容**
  - 流量控制 与 处理资源控制/伸缩容 协同
  - 无论是流量的控制还是资源的控制，都是为了让 处理容量 与 资源容量 匹配
  - 保证 系统不过载稳定性 与 业务请求的成功率
- 诺亚(Noah) 开源

核心观点

1. 高可用(HA) 不单是应对 **资源失效的高可用**，还有 **面向流量的高可用**。

   在今天对于应用架构来说，后者更是痛点，但业界的关注、思路、实践要落后很多。

2. 业务流量 的 **不确定性** 与 技术方案 的 **自适应性**，**天生一对**！

『淘系技术』公众号

让我们一起

更多地关注 **面向流量的高可用**!

更多正面思考与解决这只**房间里的大象**:

流量不确定性带来的稳定性问题!

Thanks!

相关资料

- 淘系双11技术介绍 2019-11-12，包含了 **诺亚(Noah)** 的『自适应限流技术』介绍
  实力为2019年双11而战！稳！ https://mp.weixin.qq.com/s/q3kSWp5DTgo6i6vp3p9MuQ
- Netflix相关
  - Hystrix ⇒ **Concurrency-Limits** https://github.com/Netflix/concurrency-limits
  - Hystrix is no longer in active development, and is currently in maintenance mode.
    Our focus has **shifted** towards **more adaptive** implementations that
    react to an application's **real time** performance rather than **pre-configured** settings
    (for example, through adaptive concurrency limits).
- Baidu: **BRPC**, auto concurrency limiter, 自适应限流:
  https://github.com/apache/incubator-brpc/blob/master/docs/cn/auto_concurrency_limiter.md
- Tencent: **Dagor** 微信微服务过载控制系统
  - 论文导读《Overload control for scaling WeChat microservices》
    https://www.infoq.cn/article/bAveV*B7GTH123tLwyDR

微信官方公众号：壹佰案例
关注查看更多精彩实践案例

100

TOP 100 CASE STUDIES OF THE YEAR

全球软件案例研究峰会