

谈谈分布式系统建设

工作已经过3年了，看过、做过、也听过很多系统。

越来越多的名词，方案，架构在我脑中游荡翻滚，人的精力与记忆力是有限的，尤其是结婚生子后，时常感慨。

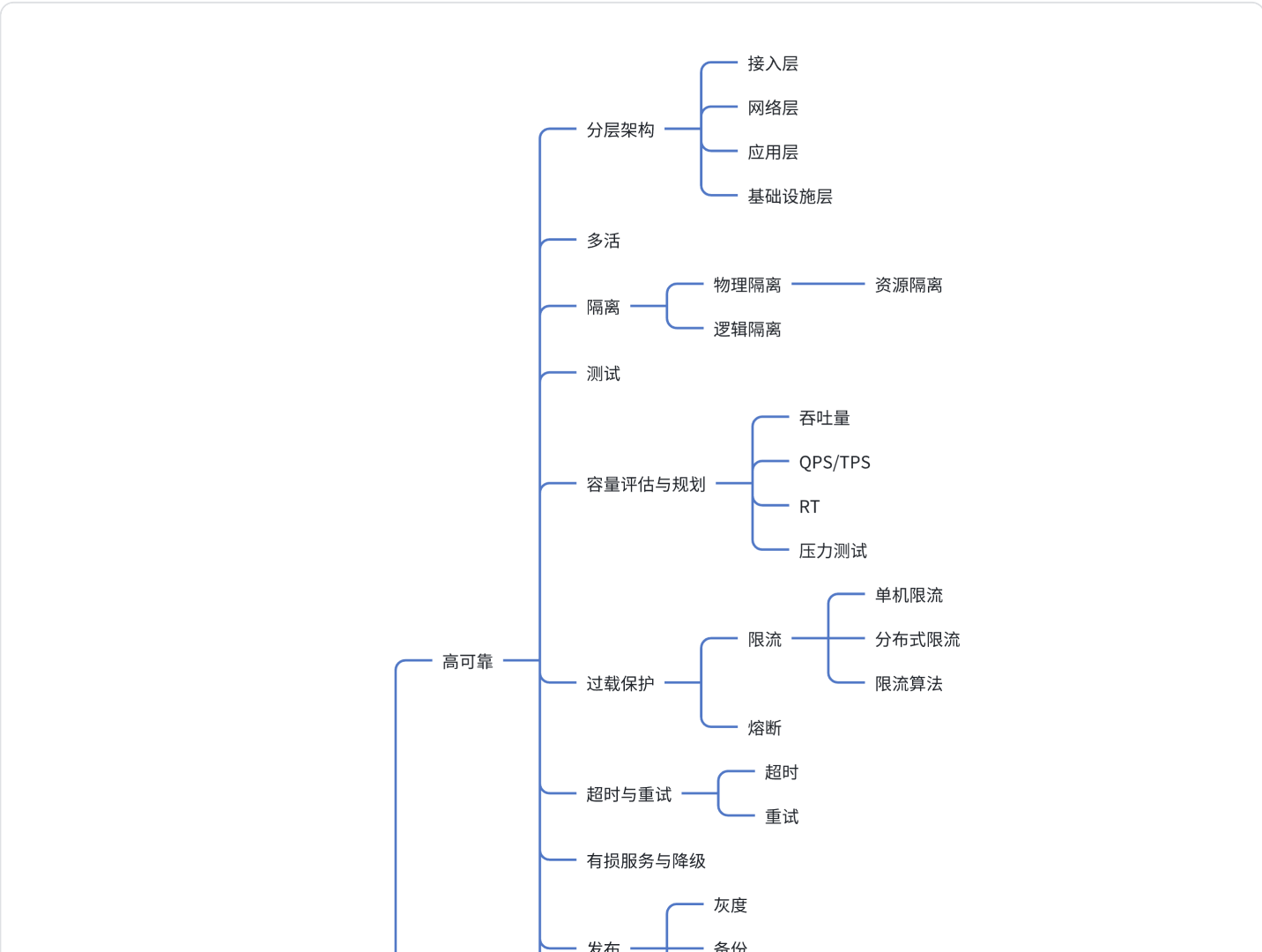
一个很简单的 outbox 方案，经常看了忘，做了忘。

人们常说常看常新，但是铺天盖地的内容又该如何看起，我想，是该总结出一套自己的知识体系了。

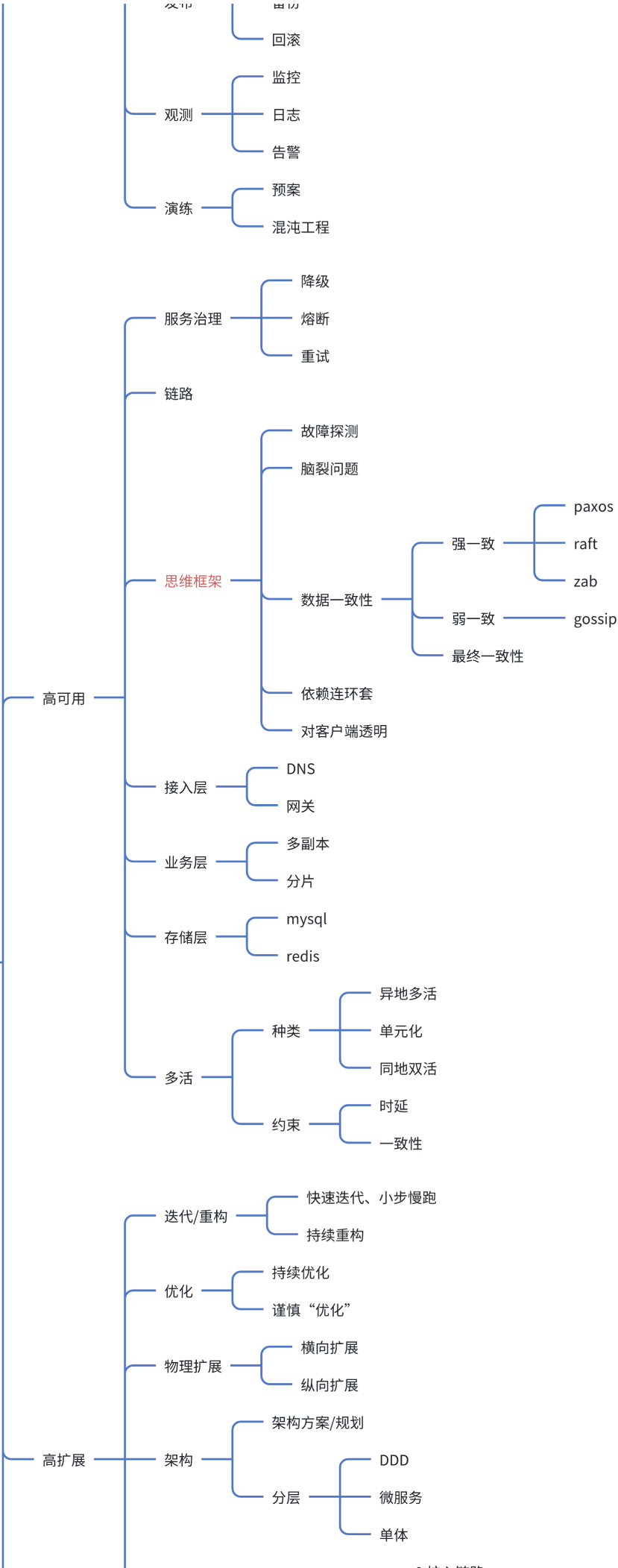
这套知识体系，必须满足下面三点：

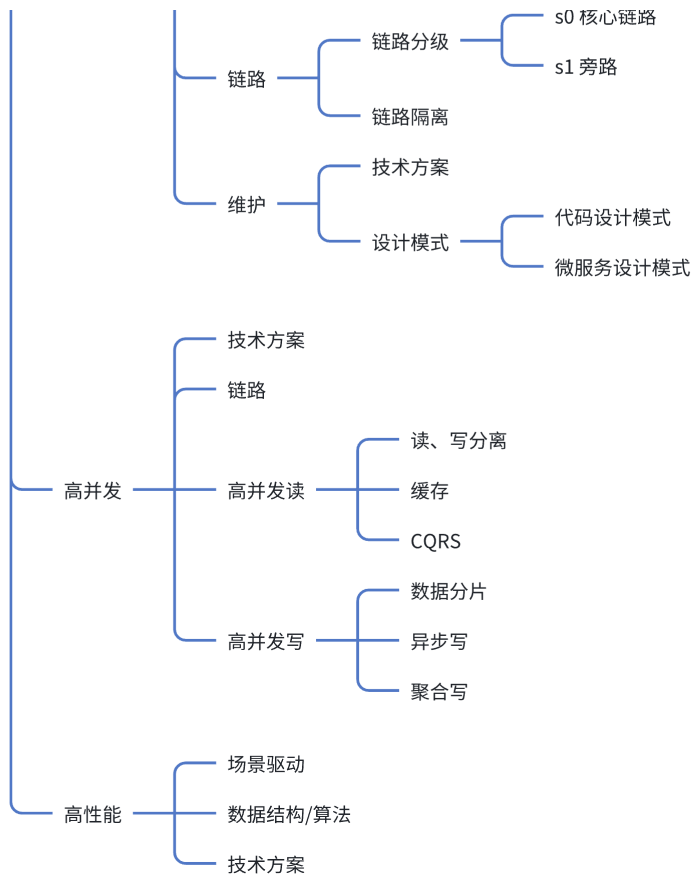
1. 适合自己，符合我的 review 习惯，当需要思考方案时，能迅速找到对应的路径；
2. 持续迭代，知识体系需要迭代，适合昨天未必适合明天；
3. 贴近真实案例，不能空谈误己，实践出来的真知需要着重体现，听到的、推演出来的、看到的要警示。

全景图



分布式系统架构特征





背景

以下纯属个人偏见

分布式系统很难定义，我们的愿景是其具有五高特性：

特性	核心定义（大白话）	路径
高可靠	让系统更靠谱，减少系统故障发生次数	
高可用	减少系统故障后的恢复时间	
高扩展	系统能够伴随需求、用户增加快速扩展性能与容量	
高并发	让系统有效率，支撑大规模用户	
高性能	让系统能够以更短的时间内完成任务，提供快速、高校的响应	

 为什么要做高可靠，高可用放在前面？

从人的角度出发，无论是工作、生活，我们一般更加倾向于相信“靠谱”的人；

再从系统的角度上来看，“靠谱”就更加重要了，一个用户系统的登陆接口，调用10次就会失败1次，调用成功率90%，勉强还能忍；但如果调用1亿次，失败1000w次，用户体验极差，整个系统可以说是瘫痪状态，这样的系统跟“靠谱”完全不搭边。

谁也不愿意突然发不出消息，也不愿意突然下不了单，保证一个可靠、可用环境是系统的生命线。



高可靠与高可用如何区分？

核心点在于故障是事前还是事后

比如，技术方案评审，代码CR在故障前review，是为了高可靠；出现问题，针对故障模块进行动态摘除，或隔离是高可用



如何理解高性能？

个人视角来看，这个词几乎已经被滥用到任何地方了，基本所有系统、框架、中间件、算法都号称自己高性能。

虽然它们会提供一大堆benchmark图表来证明自己，但本质还是因为衡量高性能的指标太多，大多数人都“主观”地晒出自己好的部分。

如何做到高性能？有两点：

- 理解场景，梳理特性；
- 不做低性能的事情；

比如在内存中缓存一份用户白名单，用户请求时，检查一下白名单，如果有则放行，否则拒绝，用那种数据结构性能更高？

如果这份名单只有100人，1000人，那么其实数组（列表）更快，虽然时间复杂度是 $O(n)$ ，但紧凑内存更加适合于CPU缓存，几个指令就找到了对应人；

如果这份名单有100W人，那么明显CPU缓存无法一次性缓存这么多，可能需要多次内存寻址，外加 $O(n)$ 的时间复杂度，那么明显是哈希表更快；

使用那种数据结构得要你根据业务特征、用户规模去确定。

再如，依次同步调用10个外部API，明显会导致低性能，那么直接选择不做，去推进被调用方改造API，提供一次性批量API。

那什么是低性能的事情？本质是因为高性能很难定义，也很难寻找，但低性能的点却很容易发现，比如在Java中用了 BeanUtils.copy，会大量使用反射特性，在请求多的情况下会拉满CPU，我在压测时经常发现QPS上不去，是因为某个同学忽略了这点，偷懒使用了。

所以这就是低性能的，要避免。

目的与路径

分布式拥有五高特性难吗？非常难，可以说**基本办不到**！

这里我其实很想果断的说根本办不到，但这似乎不符合正确价值观；

但回到现实而言，一个系统想要完美的做到这五点，需要极其庞大的人力、物力、财力；


要在迭代与维护上找到平衡，更要快速迭代证明系统价值与潜力，还要能容忍海量用户流量的冲击以及内部开发者、产品的破坏。

五高虽然未能达成，但却一直在路上。


如果分布式系统追求五高，那么需要回答下面三个问题：

- 五高目的（目标）是什么？
- 五高如何拆解？路径是什么？
- 五高如何落地？

目的

 追求五高，本质是让系统更加稳定、可靠（不然就是为了面试吹牛批）。

对于一般系统而言，稳定性做好做坏带来的影响可能不明显，因为有其他的地方能证明系统价值。所以不要过于追求五高，因为很多系统可能做的很稳定，但是业务跟不上市场，那么还是会死亡。

 谈目的首先不是要告诉我们去达成它？而是去思考是否真的需要去完成它。

比如部分C端能够容忍部分特性不稳定，登陆失败也可以重新登陆，发不出邮箱可以再来一遍，那么系统SLA可以不用那么严格，将时间花在功能迭代上，收益是更高的。

但如果是高敏性质的B端业务，比如签约，签不了客户得当场跳脚，交互做的再好，功能做的再多，但是签署容易失败，那都百搭，稳定性直接就是生命。

所以追求稳定性的目的是啥？当然是让你的系统（业务）能够更好、更优雅的活下去。

路径

这条路要走，很难走，要做的事情太多了。核心还是 by 场景的去看待问题，然后基于稳定性建设全景图来选择建设目标。

比如，如果是一个金融性质的系统，那么就需要城市级别的容灾，比如异地多活；


如果是一个用户系统，那么可能需要机房级别的容灾，比如同城双活；

如果是一个推送系统，那么可能需要副本级别的容灾，比如一主两备；

如果是一个个人博客系统，那么可能无需容灾，挂了等修复就行。

不同的场景建设路径不一样，但建设方法导致是差不多的，大概有这么几条线：

1. 通过冗余、副本来保证数据容灾；
2. 通过分片来保证计算容灾；
3. 通过隔离、分层来降低故障、问题范围；
4. 通过重试、降级等来保证链路可用性；

 实际解决方案很多，叫法也不统一，我个人更在意的思路，当遇到问题，先尝试分类，然后再看看能不用这种方式来解决。

例子

以下面某系统为例，对于稳定性，我们怎么拆分和评估。



核心评估点：

- 服务级别协议（SLA）； 99.9 => 99.99 可做
- 平均修复时间 MTTR（Mean Time To Repair）； 商务分档，故障分级
- 平均故障间隔 MTBF（Mean Time Between Failure）；

todo：

- SDK升级，拿到更多的指标和观测性；
- dashboard；
- webhook 可配置告警；
- 限流未部署；

故障预防：

资源隔离：DB、缓存、MQ、COS 隔离，已做；

容量评估/压力测试：创建等API已压，协同编辑、协同时延等未做；报表有测试：已做；

故障演练：未做；

故障发现：

- 监控/告警：
 - 服务、DB、缓存等CPU、内存、磁盘系统层面已有；
 - 子服务接口QPS、RT没有；报表有，核心暴露
 - 子服务error日志告警、统计没有；API网关层有

- TTV、TTU、白屏率、网络状态等前端指标，已有部分；报表有
- 协同时延，长链会话失败率，接口失败率等没有；报表有
- 协同OP追踪未知；—
- 整体大盘：没有；报表有

故障定位：

- 接口链路跟踪没有；报表有
- 日志追踪没有；报表有
- 长链追踪未知；日志系统

故障恢复：

- DB，缓存，MQ已有，依赖公司基建和云；
- 限流：网关入口有，子服务RPC/HTTP未知；升级后有
- 熔断：网关入口有，子服务RPC/HTTP未知；升级后有
- 降级：未知；没有

数据安全：

- 权限体系：依赖文档，已做；
- DB存储：公司单独集群，已做；
- 搜索：与文档一致，已做；
- 表格内容、附件、版本快照前端直读COS，外网读取需开白，改造成本；CDN改造；

参考资料

- 《架构设计2.0》
- 《亿级流量系统架构设计与实战》
- 《分布式架构原理与实战》
- 《凤凰架构》
- 《欧创新DDD实战课》
- 《微服务设计模式》

- 高可用还是高可靠，你区分得开吗? - 墨天轮