高并发和高可用的一点思考

kriszhang 高效运维 2018-05-28

> *本文的架子参考张开涛的《亿级流量网站架构核心技术》这本书分为四个部分：指导原则，高可用，高并发，实践案例。这篇文章说一说前三个部分，大部分内容都是我自己的思考，书只作为参考。*

- 指导原则
- 高可用

  事前

  - 副本技术
  - 隔离技术
  - 配额技术
  - 探知技术
  - 预案

  事发

  - 监控和报警

  事中

  - 降级
  - 回滚
  - failXXX系列

  事后
- 高并发

  提高处理速度

  - 多线程

- 扩容

增加处理人手

- 缓存
- 异步

指导原则

书中所列举的，里有一些可能并不是原则，而是技巧。我理解的原则如下：

高并发原则：

1. 无状态设计：因为有状态可能涉及锁操作，锁又可能导致并发的串行化。
2. 保持合理的粒度：无论拆分还是服务化，其实就是服务粒度控制，控制粒度为了分散请求提高并发，或为了从管理等角度提高可操性。
3. 缓存、队列、并发等技巧在高并发设计上可供参考，但需依场景使用。

高可用原则：

1. 系统的任何发布必须具有可回滚能力。
2. 系统任何外部依赖必须准确衡量是否可降级，是否可无损降级，并提供降级开关。
3. 系统对外暴露的接口必须配置好限流，限流值必须尽量准确可靠。

业务设计原则：

1. 安全性：防抓取，防刷单、防表单重复提交，等等等等。
2. at least 消费，应考虑是否采用幂等设计
3. 业务流程动态化，业务规则动态化
4. 系统owner负责制、人员备份制、值班制
5. 系统文档化
6. 后台操作可追溯

以上原则只是大千世界中的一小部分，读者应当在工作学习中点滴积累。

高可用

我们先说高可用的本质诉求：高可用就是抵御不确定性，保证系统7*24小时健康服务。关于高可用，我们其实面对的问题就是对抗不确定性，这个不确定性来自四面八方。比如大地震，会导致整个机房中断，如何应对？比如负责核心系统的工程师离职了，如何应对？再比如下游接口挂了，如何应对？系统磁盘坏了，数据面临丢失风险，如何应对？

我想关于上述问题的应对方式，大家在工作中或多或少都有所了解，而这个不确定性的处理过程，就是容灾，其不同的‘灾难’，对应不同的容灾级别。

为了对抗这些不同级别的不确定性，就要付出不同级别的成本，因此可用性也应是有标准的。这标准就是大家常说的N个9。

随着N的增加，成本也相应增加，那如何在达到业务需要的可用性的基础上，尽量节省成本？这也是一个值得思考的话题。除此之外，100%减去这N个9就说所谓的平均故障时间（MTBF），很多人只关心那些9，而忽略了故障处理时间，这是不该的：

你的故障处理速度越快，系统的可用性才有可能越高。

上面扯了一些可用性概念上的东西，下面来说一下技巧。开涛的书中没有对可用性技巧做出一个分类，我这里则尝试使用‘事情’来分个类。这里的‘事’就是故障，分为：事前（故障发生以前）、事发（故障发生到系统或人感知到故障）、事中（故障发生到故障处理这段时间）、事后（故障结束之后）。

按照上述分类，不同的阶段应有着不同的技巧：

事前：副本、隔离、配额、提前预案、探知
事发：监控、报警
事中：降级、回滚、应急预案，failXXX系列
事后：复盘、思考、技改

事前

副本技术

大自然是副本技术当之无愧的集大成者，无论是冰河时代，还是陨石撞击地球所带来的毁灭性打

击，物种依然绵绵不绝的繁衍，这便是基因复制的作用。副本是对抗不确定性的有力武器，把副本技术引入计算机系统，也会带来高可用性的提升。

无状态服务集群便是副本的一个应用，因为没有状态，便可水平伸缩，而这些无状态服务器之间需要一层代理来统一调度管理，这便有了反向代理。

当代理通过心跳检测机制检测到有一台机器出现问题时，就将其下线，其他'副本'机器继续提供服务；存储领域也是经常使用副本技术的，比如OB的三地三中心五副本技术等，mysql主备切换，rabbitMQ的镜像队列，磁盘的RAID技术，各种nosql中的分区副本，等等等等，数不胜数，几乎所有保证高可用的系统都有冗余副本存在。

隔离技术

书上提到了很多种隔离：线程隔离、进程隔离、集群隔离、机房隔离、读写隔离、动静隔离、爬虫隔离、热点隔离、硬件资源隔离。

在我看来，这些隔离其实就是一种，即资源隔离，无论线程、进程、硬件、机房、集群都是一种资源；动态资源和静态资源也不过是资源的一种分类；热点隔离也即是热点资源和非热点资源的隔离；读写隔离不过仅仅是资源的使用方式而已，相同的两份资源，一份用来写，一份用来读。

因此，隔离的本质，其实就是对资源的独立保护。因为每个资源都得到了独立的保护，其中一个资源出了问题，不会影响到其他资源，这就提高了整体服务的可用性。人类使用隔离术也由来已久了，从农业养殖，到股票投资，甚至关犯人的监狱，都能找到隔离术的影子。

配额技术

配额技术通过限制资源供给来保护系统，从而提高整体可用性。限流是配额技术的一种，它通过调节入口流量水位上线，来避免供给不足所导致的服务宕机。

限流分为集群限流和单机限流，集群限流需要分布式基础设施配合，单机限流则不需要。大部分业务场景使用单机限流足以，特殊场景（类秒杀等）下的限流则需限制整个集群。除此之外，限流这里我们还需要考虑几点：

如何设置合理的限流值？限流值的设定是需要经过全链路压测、妥善评估CPU容量、磁盘、内存、IO等指标与流量之间的变化关系（不一定线性关系）、结合业务预估和运维经验后，才能确定。

对于被限流的流量如何处理？有几种处理方式，其一直接丢弃，用温和的文案提醒用户；其二，

静默，俗称的无损降级，用缓存内容刷新页面；其三，蓄洪，异步回血，这一般用于事务型场景。

会不会导致误杀？单机限流会导致误杀，尤其当负载不均衡的情况下，很容易出现误杀；单机限流值设定过小也容易出现误杀的情况。

探知技术

其只用于探知系统当前可用性能力，无法切实提高系统可用性，做不好甚至还会降低系统可用性。压测和演练和最常见的探知技术 。压测分为全链路压测和单链路压测，全链路压测用于像双十一大促活动等，需要各上下游系统整体配合，单链路压测一般验证功能或做简单的单机压测提取性能指标。

全链路压测的一般过程是：压测目标设定和评估，压测改造，压测脚本编写部署，压测数据准备，小流量链路验证，通知上下游系统owner，压测预热，压测，压测结果评估报告，性能优化。以上过程反复迭代，直到达到压测目标为止；

演练一般按规模划分：比如城市级别的容灾演练，机房级别的容灾演练，集群规模的容灾演练（DB集群，缓存集群，应用集群等），单机级别的故障注入，预案演练等。演练的作用无需过多强调，但演练一般发生在凌晨，也需要各系统owner配合排错，着实累人，一般都是轮班去搞。

预案

预案一般分为提前预案（事前）和应急预案（事中）。提前预案提前执行，比如将系统临时从高峰模式切换成节能模式；应急预案关键时刻才执行，主要用于止血，比如一键容灾切换等。预案技术一般要配合开关使用，推预案一般也就是推开关了。除此之外，预案也可和限流、回滚、降级等相结合，并可以作为一个定期演练项目。

事发

事发是指当故障发生了到系统或人感知到故障准备处理的这段时间，核心诉求即是如何快速、准确的识别故障。

监控和报警

一般出现故障的时候，老板大多会有三问：为什么才发现？为什么才解决？影响有多大？即使故障影响面较大，如果能迅速止血，在做复盘的时候多少能挽回一些面子，相反如果处理不及时，

即使小小的故障，都可能让你丢了饭碗。越早识别故障，就能越早解决问题，而这眼睛便是监控和报警了。监控报警耳熟能详，这里不多赘述。

事中

事中是指当故障发生时，为了保证系统可用性，我们可以或必须做的事情。分为降级、回滚、应急预案（见上文，这里不多数了），faillXXX系列。

降级

降级的内涵丰富，我们只从链路角度去思考。降级的本质是弃车保帅，通过临时舍弃部分功能，保证系统整体可用性。降级虽然从整体上看系统仍然可用，但由于取舍的关系，那么可知所有的降级一定是有损的。不可能有真正的无损降级，而常说的无损降级指的是用户体验无损。

降级一定发生在层与层之间（上下游），要么a层临时性不调用b层，这叫做熔断，要么a层临时调用c层（c层合理性一定<b层），这叫备用链路。

无论是哪一种方式，都会面临一个问题：如何确定什么时候降级，什么时候恢复？一般有两种方式，

其一是人工确认，通过监控报警等反馈机制，人工识别故障，推送降级，待故障恢复后在手动回滚；

其二是自适应识别，最常用的指标有超时时间、错误次数、限值流等等，当达到阈值时自动执行降级，恢复时自动回滚。

这两种方式无需对比，它们都是经常采用的高可用技巧。

除此之外，我们还要注意降级和强弱依赖的关系。强弱依赖表示的是链路上下游之间的依赖关系，是'是否可降级'的一种专业表述。

我们再来看书中的一些降级的例子：

①读写降级，实际上是存储层和应用层之间的降级，采用备用链路切换方式，损失了一致性；

②功能降级，将部分功能关闭，实际上是应用层和功能模块层之间的降级，采用熔断方式，损失了部分功能。

③爬虫降级，实际上是搜索引擎爬虫和应用系统之间的降级，采用备用链路切换方式，将爬虫引

导到静态页面，损失是引擎索引的建立和页面收录。

回滚

当执行某种变更出现故障时，最为稳妥和有效的办法就是回滚。虽然回滚行之有效，但并不简单，因为回滚有一个大前提：变更必须具有可回滚性。

而让某一种变更具有可回滚的特性，是要耗费很大力气的。索性的是，大部分基础服务已经帮我们封装好了这一特性，比如DB的事务回滚（DB事务机制），代码库回滚（GIT的文件版本控制），发布回滚（发布系统支持）等等。

我们在日常变更操作的时候，必须要确定你的操作是否可回滚，并尽力保证所有变更均可回滚。如果不能回滚，是否可以进行热更新（比如发布应用到app store）或最终一致性补偿等额外手段保证系统高可用。

failXXX系列

当出现下游调用失败时，我们一般有几种处理方式：

1. failretry，即失败重试，需要配合退避时间，否则马上重试不一定会有效果。
2. failover，即所谓的故障转移。比如调用下游a接口失败，那么RPC的负载均衡器将会调用a接口提供方的其他机器进行重试；在比如数据库x挂了，应用自适应容灾将对x库的调用切换到y库调用，此y库即可以是faillover库（流水型业务），也可以备库（状态型业务）。
3. failsafe，即静默，一般下游链路是弱依赖的时候，可以采用failsafe，即可和failover相结合，比如failover了3次还是失败，那么执行failsafe。
4. failfast，立即报错，failfast主要让工程师快速的感知问题所在，并及时进行人工干预。
5. failback，延迟补偿（回血），一般可以采用消息队列或定时扫描等。

上面的1，2，4是属于重试策略，即书中《超时与重试》章节所讲到的重试。重试有个问题：退避间隔是多少？重试几次？一般在下游临时抖动的情况下，很短时间内就可以恢复；但当下游完全不可用，那么很有可能重试多少次都不会成功，反而会对下游造成了更大的压力，那这种情况就应当做用熔断了。

所以正确设定重试次数、选择退避时间等都是需要仔细思考的。我们在来说一下超时，超时只是

一种预防机制，不是故障应对策略，其主要为了防止请求堆积——资源都用于等待下游请求返回了。堆积的后果自不用多说，重要的是如何选择正确的超时时间？书上只说了链路每个部分超时时间怎么配置，却不知道应配置多少，这是不够全面的。

事后

复盘、思考、技改。不多赘述。

高并发

如果仅是追求高可用性，这其实并不难做，试想如果一年只有一个人访问你的系统，只要这一个人访问成功，那你系统的''可用性'就是100%了。

可现实是，随着业务的发展，请求量会越来越高，进而各种系统资源得以激活，那潜在风险也会慢慢的暴露出来。

因此，做系统的难点之一便是：如何在高并发的条件下，保证系统的高可用。上文已经说了一些保证高可用的技巧，这节将结合开涛的书，说说高并发。

上图是我们生活中常见的一个场景——排队购物。收银员就是我们的服务，每一个在队列中的顾客都是一个请求。我们的本质诉求是让尽可能多的人都在合理的等待时间内完成消费。

如何做到这一点呢？其一是提高收银员的处理速度，他们处理的越快，单位时间内就能服务更多

的顾客；其二是增加人手，一名收银员处理不过来，我们就雇十名收银员，十名不够我们就雇佣一百名（如果不计成本）；其三是减少访问人数，也即分流过滤，将一些人提前过滤掉，或做活动预热（比如双十一预热），在高峰之前先满足一部分人的需求。因此，想要高并发无外乎从以下几个方面入手：

提高处理速度：缓存、异步
增加处理人手：多线程（多进程）、扩容
减少访问人数：预处理（本文不涉及）

提高处理速度

缓存

缓存之所以能够提高处理速度，是因为不同设备的访问速度存在差异。缓存的话题可以扯几本书不带重样的。从CPU可以一直扯到客户端缓存，即从最底层一直到扯到最特近用户的一层，每一层都可能或可以有缓存的存在。我们这里不扯这么多，只说简单服务端缓存。现在从几个不同角度来看一下缓存：

①从效果角度。命中率越高越好吗？10万个店铺数据，缓存了1000个，命中率稳定100%，那是不是说，有99000个店铺都是长尾店铺？缓存效果评估不能单看命中率。

②从回收策略。如果把缓存当做数据库一样的存储设备去用，那就没有回收的说法了（除非重启或者宕机，否则数据依然有效）；如果只存储热数据，那就有回收和替换的问题。回收有两种方式，一种是空间配额，另一种是时间配额。替换也有几种方式，LRU，FIFO，LFU。

③从缓存使用模式角度：用户直接操作缓存和db；用户直接操作缓存，缓存帮助我们读写DbB；

④从缓存分级角度。java堆内缓存、java堆外缓存、磁盘缓存、分布式缓存，多级缓存。

⑤从缓存使用角度。null穿透问题、惊群问题、缓存热点问题、缓存一致性问题、读写扩散问题……

⑥更新方式。读更新、写更新、异步更新。

如果缓存集群涉及到异地多集群部署，再结合大数据量高并发业务场景，还会遇到很多更加复杂的问题，这里就不一一列举了。

异步

异步这里有几点内涵，

其一是将多个同步调用变成异步并发调用，这样就将总响应时间由原来的t1+t2+t3+.....+tn变成了max(t1,t2,t3....,tn)，这也叫异步编排；

其二是在操作系统层面，使用asyc io以提高io处理性能；

其三是将请求'转储'，稍后异步进行处理，一般使用队列中间件。其中的异步编排，可以使用CompletableFuture；异步IO一般框架都做了封装；而队列中间件则是最为常用的技术之一，也是我们重点关注的对象。

业务允许延迟处理，是使用队列中间件的大前提，即非实时系统或准实时系统更适合使用。主要作用有：异步处理（增加吞吐），削峰蓄洪（保障稳定性），数据同步（最终一致性组件），系统解耦（下游无需感知订阅方）。

- 缓冲队列：一般使用环形缓冲队列，控制缓冲区大小。
- 任务队列：一般用于任务调度系统，比如线程池等，disrupter
- 消息队列：一般意义上的消息中间件，不同业务场景需要的中间件能力不同，有的需要高吞吐，有的需要支持事务，有的需要支持多客户端，有的需要支持特定协议等等，妄图开发一个大而全的消息队列，个人觉得不如提供多种队列按需选型，之后在统一提供一个通信中台，全面整合消息能力。
- 请求队列：就是处理请求的队列，有点像流程引擎，可以做一些前置后置的入队出队处理，比如限流、过滤等等
- 数据总线队列：比如canal，datax等数据（异构或同构）同步用的。
- 优先级队列：一般大根堆实现
- 副本队列：如果队列支持回放，副本队列有些冗余。
- 镜像队列：一般用于做队列系统的高可用切换的。

  有时候也跨集群跨机房做复制，提供更多消费者消费，增加投递能力。

队列的消费端有pull模式或者push模式的选取。pull模式可以控制进度，push模式则实时性更高一些；pull能支持单队列上的有序，push很难支持。除了消费模式，队列还有一系列其他问题请参考其他书籍，这里不多说明了。

这里在补充一点关于异步的说明。同步转异步，可以提高吞吐量；异步转同步，可以增加可靠

性。

增加处理人手

多线程

多线程（多进程）技术是‘增加处理人手’技术中最容易想到的，一般我们也广泛采用。无论是web服务容器、网关、RPC服务端、消息队列消费和发送端等等都有使用多线程技术。其优点也无需过多说明。

这里我们只说一件重要的事情，即线程数的设置问题，如果线程数过高则可能会吃光系统资源，如果过低又无法发挥多线程优势。

一般设置的时候，会参考平均处理时长、并发峰值、平均并发量、阻塞率、最长可容忍响应时间、CPU核心数等等，然后做一定的运算，计算出线程数、core和max，以及阻塞队列大小。

具体算法可以自行谷歌。

扩容

在无状态服务下，扩容可能是迄今为止效果最明显的增加并发量的技巧之一。

有临时性并发问题时，可以直接提扩容工单，立竿见影。但扩容的最大问题就是成本了，想想动辄几万的实体机，如果只是为了支撑一个小时的大促，而平常利用率几乎为0，那确实是浪费。

因此便有了弹性云，当需要扩容时，借别人机器（阿里云）用完再还回去；以及离线在线混部，充分利用资源。

从扩容方式角度讲，分为垂直扩容（scale up）和水平扩容（scale out）。

垂直扩容就是增加单机处理能力，怼硬件，但硬件能力毕竟还是有限；水平扩容说白了就是增加机器数量，怼机器，但随着机器数量的增加，单应用并发能力并不一定与其呈现线性关系， 此时就可能需要进行应用服务化拆分了。

从数据角度讲，扩容可以分为无状态扩容和有状态扩容。

无状态扩容一般就是指我们的应用服务器扩容；有状态扩容一般是指数据存储扩容，要么将一份数据拆分成不同的多份，即sharding，要么就整体复制n份，即副本。

sharding遇到的问题就是分片的可靠性，一般做转移、rehash、分片副本；副本遇到的问题是一致性性，一般做一致性算法，如paxos，raft等。

> 注：本文转自
>
> http://kriszhang.com/high_performance/

文章已于2018-05-28修改

阅读原文