12 | 高可用架构：如何让你的系统不掉链子？

2020-03-18 王庆友

讲述：王庆友
时长 15:52 大小 14.55M

你好，我是王庆友。今天我和你聊一聊，如何实现系统的高可用。

在实际工作中，我们平常更关注系统业务功能的实现，而对于系统是否会出故障，总觉得那是小概率事件，一开始不会考虑得太多。然而系统上线后，我们会发现系统其实很脆弱，每个地方都可能会出问题，处理线上事故的时间往往超过了开发功能的时间。

所以，对于系统的高可用，我想你经常会有这样的疑问：**系统的高可用真的很重要吗？如何实现系统的高可用，具体都有哪些手段呢？**

十年前，我还在 eBay，那时候，我们有几个数据来说明系统宕机对公司的影响，我记得其中一个是系统每宕掉 1 秒，公司将损失三千美金的收入；现在的大型外卖平台也是如此，

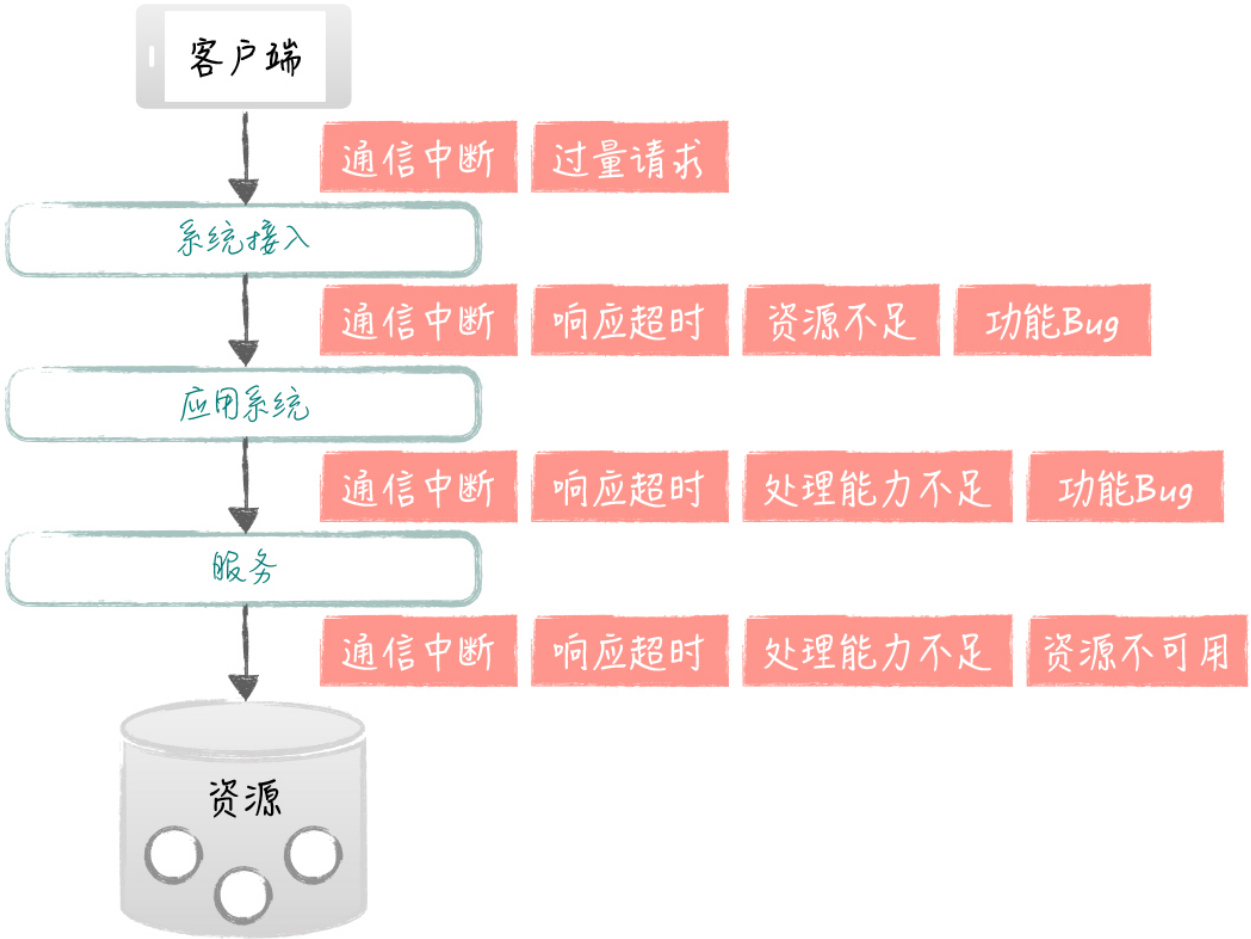
如果就餐高峰期宕掉 1 小时，平台至少损失几个亿的直接收入，更加不用说对公司品牌的影响。

但是我们知道，系统中包含了大量的软硬件设备，要保证所有的节点都可用，不是一件容易的事。所以今天这一讲，我会从系统高可用的角度出发，和你介绍如何才能做到让系统不掉链子。

系统有哪些故障点?

那么一个系统，它在运行的过程中，都可能会出现哪些故障呢？我们来看一个简化的系统处理过程。

首先，客户端在远程发起请求，经过接入系统处理后，请求被转发给应用系统；应用系统调用服务完成具体的功能；在这个过程中，应用和服务还会访问各种资源，比如数据库和缓存。这里，我用红色部分，标识出了整个处理过程中可能出现的故障点，如下图所示：



这些故障点可以归纳为三类：

1. **资源不可用**，包括网络和服务器出故障，网络出故障表明节点连接不上，服务器出故障表明该节点本身不能正常工作。
2. **资源不足**，常规的流量进来，节点能正常工作，但在高并发的情况下，节点无法正常工作，对外表现为响应超时。
3. **节点的功能有问题**，这个主要体现在我们开发的代码上，比如它的内部业务逻辑有问题，或者是接口不兼容导致客户端调用出了问题；另外有些不够成熟的中间件，有时也会有功能性问题。

下面，我们就来看看如何才能应对这些问题，实现系统的高可用。

高可用策略和架构原则

系统可能出问题的地方有很多，解决的方式也不一样，在讨论具体的解决手段之前，我想先说下高可用的总体解决思路，这样你就能更好地理解具体的实现方式。

要想让系统能够稳定可用，我们首先要考虑如何**避免问题的发生**。比如说，我们可以通过UPS（Uninterruptible Power System，不间断电源）来避免服务器断电，可以通过事先增加机器来解决硬件资源不足的问题。

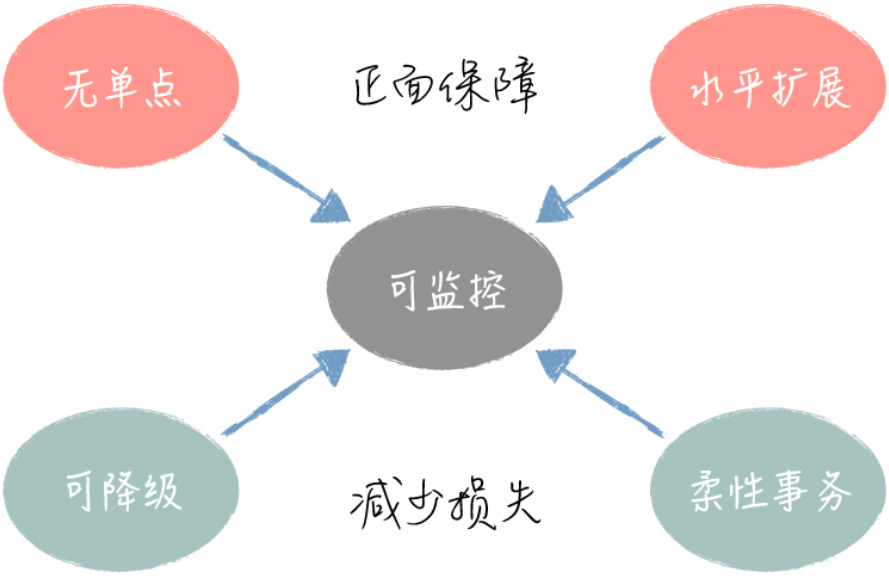
然后，如果问题真的发生了，我们就要考虑怎么**转移故障**（Failover）。比如说，我们可以通过冗余部署，当一个节点发生故障时，用其它正常的节点来代替问题节点。

如果故障无法以正面的方式解决，我们就要**努力降低故障带来的影响**。比如说流量太大，我们可以通过限流，来保证部分用户可以正常使用，或者通过业务降级的手段，关闭一些次要功能，保证核心功能仍旧可用。

最后是要**快速恢复系统**。我们要尽快找到问题的原因，然后修复故障节点，使系统恢复到正常状态。

这里我要强调的是，**处理线上事故的首要原则是先尽快恢复业务**，而不是先定位系统的问题，再通过解决问题来恢复系统。因为这样做往往比较耗时，这里给出的处理顺序也体现了这个原则。

那么结合前面介绍的系统故障点和高可用的解决思路，我们在做架构设计时，就可以从 **正面保障** 和 **减少损失** 两个角度来考虑具体的应对手段。下面，我就来和你分享一下高可用的设计原则。



正面保障

第一个设计原则是冗余无单点。

首先，我们要保证系统的各个节点在部署时是冗余的，没有单点。比如在接入层中，我们可以实现负载均衡的双节点部署，这样在一个节点出现问题时，另一个节点可以快速接管，继续提供服务。

还有远程网络通信，它会涉及到很多节点，也很容易会出现问题，我们就可以提供多条通信线路，比如移动 + 电信线路，当一条线路出现问题时，系统就可以迅速切换到另一条线路。

甚至，我们可以做到机房层面的冗余，通过系统的异地多 IDC 部署，解决自然灾害（如地震、火灾）导致的系统不可用问题。

第二个设计原则是水平扩展。

很多时候，系统的不可用都是因为流量引起的：在高并发的情况下，系统往往会整体瘫痪，完全不可用。

在前面的故障点介绍中，你可以看到，在应用层、服务层、资源层，它们的处理压力都是随着流量的增加而增加。上一讲中，我也提到过，由于硬件在物理上存在瓶颈，通过硬件升级（垂直扩展）一般不可行，我们需要通过增加机器数量，水平扩展这些节点的处理能力。

对于无状态的计算节点，比如应用层和服务层来说，水平扩展相对容易，我们直接增加机器就可以了；而对于有状态的节点，比如数据库，我们可以通过水平分库做水平扩展，不过这个需要应用一起配合，做比较大的改造。

减少损失

第三个原则是柔性事务。

我们知道，系统的可用性经常会和数据的一致性相互矛盾。在 CAP 理论中，系统的可用性、一致性和网络容错性，三个最多只能保证两个，在分布式系统的情况下，我们只能在 C 和 A 中选一个。

在很多业务场景中，**系统的可用性比数据的实时一致性更重要**，所以在实践中，我们更多地使用 BASE 理论来指导系统设计。在这里，我们努力实现系统的基本可用和数据的最终一致。

> 知识拓展：关于 BASE 理论的详细信息，你可以参考一下隔壁专栏《分布式协议与算法实战》的这篇文章，这里就不详细展开了。

我们平时对单个数据库事务的 ACID 特性非常熟悉，因为这里不存在 P，所以 C 和 A 都能得到很好地保证，这是一种**刚性事务**。但在复杂的分布式场景下，基于 BASE 理论，我们通常只能实现部分的 C（软状态和最终一致）和部分的 A（基本可用），这是一种**柔性事务**。

柔性事务具体的实现方式有很多，比如说，通过异步消息在节点间同步数据。当然，不同的方式，对 C 和 A 的支持程度是不一样的，我们在设计系统时，要根据业务的特点来决定具体的方式。

第四个原则是系统可降级。

当系统问题无法在短时间内解决时，我们就要考虑尽快止损，为故障支付尽可能小的代价。具体的解决手段主要有以下这几种。

限流：让部分用户流量进入系统处理，其它流量直接抛弃。

降级：系统抛弃部分不重要的功能，比如不发送短信通知，以此确保核心功能不受影响。

熔断：我们不去调用出问题的服务，让系统绕开故障点，就像电路的保险丝一样，自己熔断，切断通路，避免系统资源大量被占用。比如，用户下单时，如果积分服务出现问题，我们就先不送积分，后续再补偿。

功能禁用：针对具体的功能，我们设置好功能开关，让代码根据开关设置，灵活决定是否执行这部分逻辑。比如商品搜索，在系统繁忙时，我们可以选择不进行复杂的深度搜索。

做好监控

最后一个设计原则，是系统可监控。

在实践中，系统的故障防不胜防，问题的定位和解决也非常的困难，所以，要想全面保障系统的可用性，最重要的手段就是监控。

当我们在做功能开发的时候，经常会强调功能的可测试性，我们通过测试来验证这个功能是否符合预期，而系统可监控，就像业务功能可测试一样重要。**通过监控，我们可以实时地了解系统的当前状态**，这样很多时候，业务还没出问题，我们就可以提前干预，避免事故；而当系统出现问题时，我们也可以借助监控信息，快速地定位和解决问题。

好，为了帮助你更好地理解，我对这些架构原则做个小结。

无单点和水平扩展是从正面的角度，直接保障系统的可用性。**无单点设计针对的是节点本身的故障，水平扩展针对的是节点处理能力的不足。**

柔性事务和可降级是通过提供有损服务的方式来保证系统的可用性。**柔性事务保证功能的基本可用和数据的最终一致，可降级通过损失非核心功能来保证核心功能的可用。**

最后，无论我们采取了多么强大的高可用措施，我们还是不能充分相信系统，还需要借助额外的监控来及时发现系统的问题并加以解决。**监控是我们的第二条保命措施。**

高可用手段

好了，通过前面的介绍，你应该已经了解了系统的故障点，以及高可用的设计原则。下面我们就一起来看下，在实践中都有哪些手段来保障系统的高可用。这里，我会按照系统的处理顺序来给你做详细介绍。

客户端 -> 接入层

客户端到服务端通常是远程访问，所以我们首先要解决网络的可用性问题。

针对网络的高可用，我们可以拉多条线路，比如在企业私有的 IDC 机房和公有云之间，同时拉移动和电信的线路，让其中一条线路作为备份，当主线路有问题时就切换到备份线路上。

在接入层，也有很多成熟的 HA 方案，比如说，你可以选择 Nginx、HAProxy、LVS 等负载均衡软件，它们都能很好地支持双节点 +Keepalived 部署。这样当一个节点出了问题，另一个节点就可以自动顶上去，而且两个节点对外是共享一个虚拟 IP，所以节点的切换对外部是透明的。

这里，我们通过冗余和自动切换避免了单点的故障。

接入层 ->Web 应用

Web 应用通常是无状态的，我们可以部署多个实例，很方便地通过水平扩展的方式，提升系统的处理能力；接入层的负载均衡设备，可以通过各种算法进行多个 Web 实例的路由，并且对它们进行健康检测，如果某个实例有问题，请求可以转发到另一个实例进行处理，从而实现故障的自动转移。

通常情况下，我们还可以在接入层做限流，比如，在 Nginx 中设置每秒多少个并发的限制，超过这个并发数，Nginx 就直接返回错误。

这里，我们同时支持了 Web 节点的水平扩展、自动故障转移以及系统的可降级（限流）。

Web 应用 -> 内部服务

服务通常也是无状态的，我们也可以通过部署多个实例进行水平扩展。

有多种方式可以支持服务实例的发现和负载均衡，比如说，我们可以使用传统的代理服务器方式，进行请求分发；另外，很多的微服务框架本身就支持服务的直接路由，比如在 Spring Cloud 中，我们就可以通过 Eureka 进行服务的自动注册和路由。

应用通常会访问多个服务，我们在这里可以做服务的隔离和熔断，避免服务之间相互影响。

比如在 Spring Cloud 的 Hystrix 组件（开源熔断框架）中，我们可以为不同服务配置不同的线程池，实现资源隔离，避免因为一个服务响应慢，而占用所有的线程资源；如果某个服务调用失败，我们可以对它进行熔断操作，避免无谓的超时等待，影响调用方的整体性能。

在应用和服务的内部，针对具体的功能，我们还可以做一些**功能开关**。开关实际上是一个标志变量，它的值可以是 on/off， 我们在代码中可以根据它的值，来确定某一段逻辑是否要执行。开关的值可以在数据库或配置系统里定义，这样我们就能够通过外部的开关值，控制应用内部的行为，这个在 eBay 有大量的落地。

这里，我们同时支持了服务节点的水平扩展、自动故障转移以及系统的可降级（熔断和业务开关）。

访问基础资源

常见的资源包括关系数据库、缓存和消息系统，我就以它们为例来介绍一下。

关系数据库属于有状态服务，它的水平扩展没有那么容易，但还是有很多手段能够保障数据库的可用性和处理能力。

首先，我们可以做数据库的主从部署，一方面通过读写分离，提升数据库**读**的性能，减轻主库压力；另一方面，数据库有成熟的 MHA 方案，支持主库故障时，能够自动实现主从切换，应用可以通过 VIP 访问数据库，因此这个切换过程对应用也是透明的。

另外，我们也可以通过物理的水平分库方式，对数据进行分片，这样就有多个主库支持写入。水平分库会涉及较多的应用改造，后面会有一篇文章专门介绍 1 号店的订单水平分库

项目，到时我们再详细讨论。

再说下**缓存**。在数据读写比很高的情况下，我们可以利用缓存优化数据库的访问性能，包括进程内部缓存和分布式缓存，缓存是应对高并发的有效武器。

很多缓存方案，比如 Redis 本身就支持集群方式，它可以通过多节点支持处理能力的水平扩展，通过数据的多副本来支持故障转移。

最后说下**消息系统**。消息系统有很多成熟的 MQ 组件，比如说 Kafka，它可以通过多节点部署来支持处理能力的水平扩展，也能通过数据的多分区，实现故障的自动切换，保证系统的可用性。

最后我想说的是，明天和意外你永远不知道哪个先到来，即使有了这些高可用措施，还是会有各种各样的意外等待着我们。所以，**系统的监控非常重要，只有准确地了解系统当前的状况，我们在面对问题时，才能快速响应，处理到点子上。**

总结

今天，我和你介绍了保障系统高可用都有哪些策略和设计原则，相信你现在对高可用的整体处理思路有了清楚的认识。

另外，我还针对典型的系统处理过程，和你介绍了各个环节都有哪些具体的高可用手段，希望你可以在工作中，结合系统的实际情况去落地它们。

接下来，我会通过几个实际的案例，来具体说明如何实现系统的高可用，你可以跟着课程继续学习，然后尝试着在实际的工作中去参考和灵活运用。

最后，给你留一道思考题：处理事故有三板斧的说法，你知道它们都是什么吗？你是怎么评价它们的呢？

欢迎在留言区和我互动，我会第一时间给你反馈。如果觉得有收获，也欢迎你把这篇文章分享给你的朋友。感谢阅读，我们下期再见。

上一篇　11 | 技术架构：作为开发，你真的了解系统吗?

下一篇　13 | 高可用架构案例（一）：如何实现O2O平台日订单500万?

精选留言 (5)

写留言

孙同学
2020-03-18

https://www.processon.com/view/link/5e51378ce4b0c037b5f9d1e3 整理学习更新

1

约书亚
2020-03-23

第一次从这种抽象层面看待问题，之前太low了，受益颇多。
我感觉“隔离”也应该算是一种解决思路，尽管和可降级有重复的地方。

Din
2020-03-22

是 重启、下线、回滚 这三个吗?

感觉这些手段和老师说的「处理线上事故的首要原则是先尽快恢复业务 」是一致的，都是先恢复业务，将业务损失降到最低，然后再定位具体的问题。

展开

作者回复: 差不多，下线和回滚差不多意思，还有一个是加机器。

孙同学

2020-03-19

是 主备切换 重启设备 回推操作吗

展开

Jeff.Smile

2020-03-18

老师，课程看到现在觉得确实都是实际的架构经验，不过更偏重于设计角度，有个疑问，对于有志于成为架构师的开发工程师来说，是需要多花精力在软件本身的使用或者说落地上呢？还是思考架构如何设计上而对软件达到基本能上手使用就行？

展开

作者回复: 工程师到架构师，是一个从实现到设计的过程，设计比实现更难，会设计，实现自然不是问题