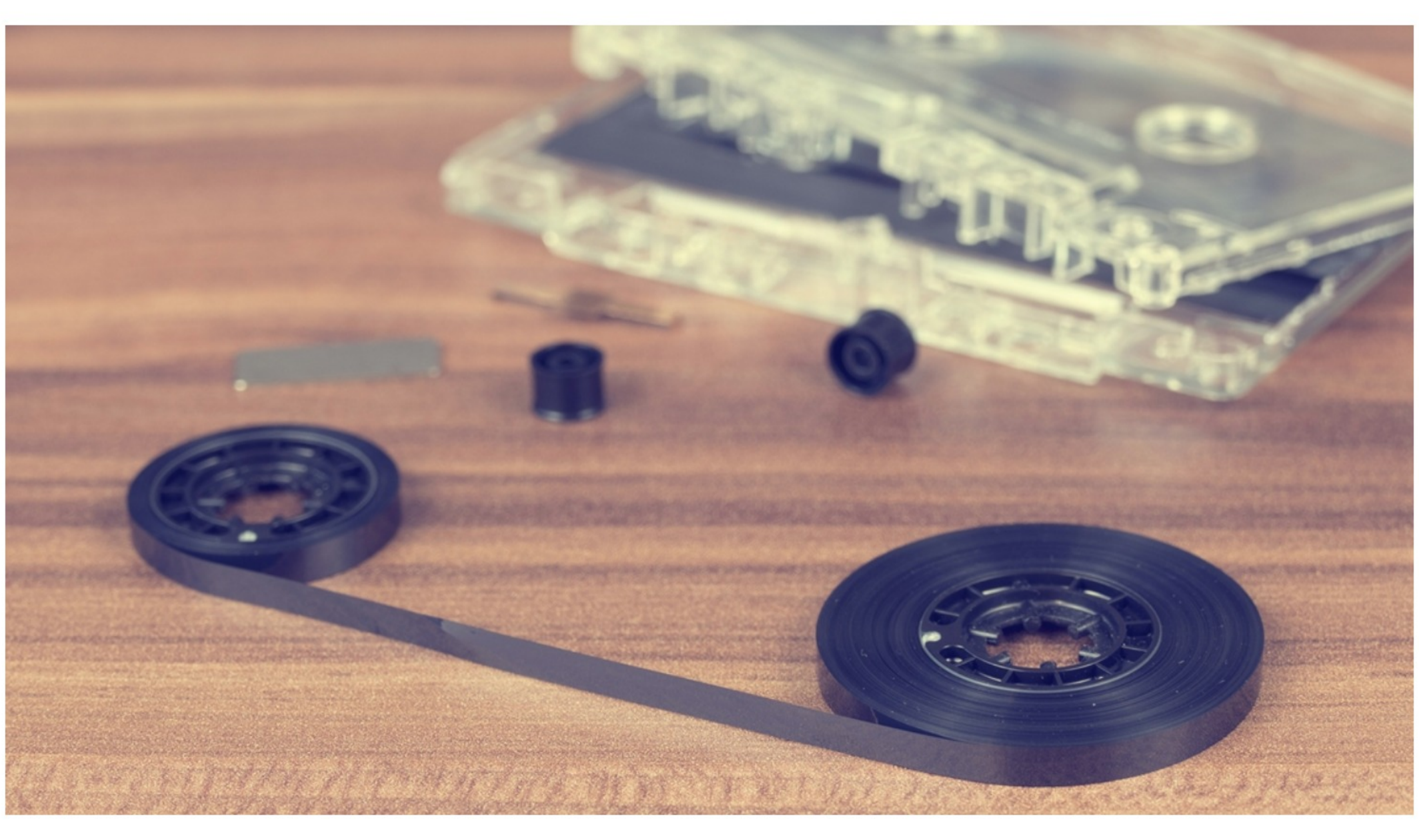
弹力设计篇之“服务的状态”

2018-03-06 陈皓



弹力设计篇之“服务的状态”
朗读人：柴巍 10′56″ | 5.01M

之前在我们讲的幂等设计中，为了过滤掉已经处理过的请求，其中需要保存处理过的状态，为了把服务做成无状态的，我们引入了第三方的存储。而这一篇中，我们将谈论一下服务的状态，只有清楚地了解了状态这个事，我们才有可能设计出更好或是更有弹力的系统架构。

所谓“状态”，就是为了保留程序的一些数据或是上下文。比如之前幂等性设计中所说的需要保留下每一次请求的状态，或是像用户登录时的 Session，我们需要这个 Session 来判断这个请求的合法性，还有一个业务流程中需要让多个服务组合起来形成一个业务逻辑的运行上下文 Context。这些都是所谓的状态。

我们的代码中基本上到处都是这样的状态。

无状态的服务 Stateless

一直以来，无状态的服务被当成分布式服务设计的最佳实践和铁律。因为无状态的服务对于扩展性和运维实在是太方便了。没有状态的服务，可以随意地增加和减少结点，同样可以随意地搬迁。而且，无状态的服务可以大幅度降低代码的复杂度以及 Bug 数，因为没有状态，所以也没有明显的“副作用”。

基本上来说，无状态的服务和“函数式编程”的思维方式如出一辙。在函数式编程中，一个铁律是，函数是无状态的。换句话说，函数是 immutable 不变的，所有的函数只描述其逻辑和算法，根本不保存数据，也不会修改输入的数据，而是把计算好的结果返回出去，哪怕要把输入的数据重新拷贝一份并只做少量的修改（关于函数式编程可以参看我在 CoolShell 上的文章《函数式编程》）。

但是，现实世界是一定会有状态的。这些状态可能表现在如下的几个方面。

- 程序调用的结果。
- 服务组合下的上下文。
- 服务的配置。

为了做出无状态的服务，我们通常需要把状态保存到一个第三方的地方。比如，不太重要的数据可以放到 Redis 中，重要的数据可以放到 MySQL 中，或是像 ZooKeeper/Etcd 这样的高可用的强一致性的存储中，或是分布式文件系统中。

于是，我们为了做成无状态的服务，会导致这些服务需要耦合第三方有状态的存储服务。一方面是有依赖，另一方面也增加了网络开销，导致服务的响应时间也会变慢。

所以，第三方的这些存储服务也必须要做成高可用高扩展的方式。而且，为了减少网络开销，还需要在无状态的服务中增加缓存机制。然而，下次这个用户的请求并不一定会在同一台机器，所以，这个缓存会在所有的机器上都创建，也算是一种浪费吧。

这种“转移责任”的玩法也催生出了对分布式存储的强烈需求。正如之前在《分布式系统架构的本质》系列文章中谈到的关键技术之一的“状态 / 数据调度”所说的，因为数据层的 scheme 众多，所以，很难做出一个放之四海皆准的分布式存储系统。

这也是为什么无状态的服务需要依赖于像 ZooKeeper/Etcd 这样的高可用的有强一致的服务，或是依赖于底层的分布式文件系统（像开源的 Ceph 和 GlusterFS）。而现在分布式数据库也开始将服务和存储分离，也是为了让自己的系统更有弹力。

有状态的服务 Stateful

在今天看来，有状态的服务在今天看上去的确比较“反动”，但是，我们也需要比较一下它和无状态服务的优劣。

正如上面所说的，无状态服务在程序 Bug 上和水平扩展上有非常优秀的表现，但是其需要把状态存放在一个第三方存储上，增加了网络开销，而在服务内的缓存需要在所有的服务实例上都有（因为每次请求不会都落在同一个服务实例上），这是比较浪费资源的。

而有状态的服务有这些好处。

- **数据本地化（Data Locality）**。一方面状态和数据是本机保存，这方面不但有更低的延时，而且对于数据密集型的应用来说，这会更快。

- **更高的可用性和更强的一致性**。也就是 CAP 原理中的 A 和 C。

为什么会这样呢？因为对于有状态的服务，我们需要对于客户端传来的请求，都必需保证其落在同一个实例上，这叫 Sticky Session 或是 Sticky Connection。这样一来，我们完全不需要考虑数据要被加载到不同的结点上去，而且这样的模型更容易理解和实现。

可见，最重要的区别就是，无状态的服务需要我们把数据同步到不同的结点上，而有状态的服务通过 Sticky Session 做数据分片（当然，同步有同步的问题，分片也有分片的问题，这两者没有谁比谁好，都有 trade-off）。

这种 Sticky Session 是怎么实现的呢？

最简单的实现就是用持久化的长连接。就算是 HTTP 协议也要用长连接。或是通过一个简单的哈希（hash）算法，比如，通过 uid 求模的方式，走一致性哈希的玩法，也可以方便地做水平扩展。

然而，这种方式也会带来问题，那就是，结点的负载和数据并不会很均匀。尤其是长连接的方式，连上了就不断了。所以，玩长连接的玩法一般都会有一种叫“反向压力 (Back Pressure)”。也就是说，如果服务端成为了热点，那么就主动断连接，这种玩法也比较危险，需要客户端的配合，否则容易出 Bug。

如果要做到负载和数据均匀的话，我们需要有一个元数据索引来映射后端服务实例和请求的对应关键，还需要一个路由结点，这个路由结点会根据元数据索引来路由，而这个元数据索引表会根据后端服务的压力来重新组织相关的映射。

当然，我们可以把这个路由结点给去掉，让有状态的服务直接路由。要做到这点，一般来说，有两种方式。一种是直接使用配置，在节点启动时把其元数据读到内存中，但是这样一来增加或减少结点都需要更新这个配置，会导致其它结点也一同要重新读入。

另一种比较好的做法是使用到 Gossip 协议，通过这个协议在各个节点之间互相散播消息来同步元数据，这样新增或减少结点，集群内部可以很容易重新分配（听起来要实现好真的好复杂）。

在有状态的服务上做自动化伸缩的是有一些相关的真实案例的。比如，Facebook 的 Scuba，这是一个分布式的内存数据库，它使用了静态的方式，也就是上面的第一种方式。Uber 的 Ringpop 是一个开源的 Node.js 的根据地理位置分片的路由请求的库（开源地址为：https://github.com/uber-node/ringpop-node ）。

还有微软的 Orleans，Halo 4 就是基于其开发的，其使用了 Gossip 协议，一致性哈希和 DHT 技术相结合的方式。用户通过其 ID 的一致性哈希算法映射到一个节点上，而这个节点保存了这个用户对应的 DHT，再通过 DHT 定位到处理用户请求的位置，这个项目也是开源的（开源地址为：https://github.com/dotnet/orleans ）。

关于可扩展的有状态服务，这里强烈推荐 Twitter 的美女工程师 Caitie McCaffrey 的演讲 Youtube 视频《Building Scalable Stateful Service》(演讲 PPT)，其文字版是在 High Scalability 上的这篇文章《Making the Case for Building Scalable Stateful Services in the Modern Era》

服务状态的容错设计

在容错设计中，服务状态是一件非常复杂的事。尤其对于运维来说，因为你要调度服务就需要调度服务的状态，迁移服务的状态就需要迁移服务的数据。在数据量比较大的情况下，这一点就变得更为困难了。

虽然上述有状态的服务的调度通过 Sticky Session 的方式是一种方式，但我依然觉得理论上来说虽然可以这么干，这实际在运维的过程中，这么干还是件挺麻烦的事儿，不是很好的玩法。

很多系统的高可用的设计都会采取数据在运行时就复制的方案，比如：ZooKeeper、Kafka、Redis 或是 ElasticSearch 等等。在运行时进行数据复制就需要考虑一致性的问题，所以，强一致性的系统一般会使用两阶段提交。

这要求所有的结点都需要有一致的结果，这是 CAP 里的 CA 系统。而也有的系统采用的是大多数人一致就可以了，比如 Paxos 算法，这是 CP 系统。

但我们需要知道，即使是这样，当一个结点挂掉了以后，在另外一个地方重新恢复这个结点时，这个结点需要把数据同步过来才能提供服务。然而，如果数据量过大，这个过程可能会很漫长，这也会影响我们系统的可用性。

所以，我们需要使用底层的分布式文件系统，对于有状态的数据不但在运行时进行多结点间的复制，同时为了避免挂掉，还需要把数据持久化在硬盘上，这个硬盘可以是挂载到本地硬盘的一个外部分布式的文件卷。

这样当结点挂掉以后，以另外一个宿主机上启动一个新的服务实例时，这个服务可以从远程把之前的文件系统挂载过来。然后，在启动的过程中就装载好了大多数的数据，从而可以从网络其它结点上同步少量的数据，因而可以快速地恢复和提供服务。

这一点，对于有状态的服务来说非常关键。所以，使用一个分布式文件系统是调度有状态服务的关键。

小结

好了，我们来总结一下今天分享的主要内容。首先，我讲了无状态的服务。无状态的服务就像一个函数一样，对于给定的输入，它会给出唯一确定的输出。它的好处是很容易运维和伸缩，但需要底层有分布式的数据库支持。

接着，我讲了有状态的服务，它们通过 Sticky Session、一致性 Hash 和 DHT 等技术实现状态和请求的关联，并将数据同步到分布式数据库中；利用分布式文件系统，还能在节点挂掉时快速启动新实例。下篇文章中，我们讲述补偿事务。希望对你有帮助。

也欢迎你分享一下你所实现的分布式服务是无状态的，还是有状态的？用到了哪些技术？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
    - 认识故障和弹力设计
    - 隔离设计 Bulkheads
    - 异步通讯设计 Asynchronous
    - 幂等性设计 Idempotency
    - 服务的状态 State
    - 补偿事务 Compensating Transaction
    - 重试设计 Retry
    - 熔断设计 Circuit Breaker
    - 限流设计 Throttle
    - 降级设计 degradation
    - 弹力设计总结
- 管理设计篇
    - 分布式锁 Distributed Lock
    - 配置中心 Configuration Management
    - 边车模式 Sidecar
    - 服务网格 Service Mesh
    - 网关模式 Gateway
    - 部署升级策略
- 性能设计篇
    - 缓存 Cache
    - 异步处理 Asynchronous
    - 数据库扩展
    - 秒杀 Flash Sales
    - 边缘计算 Edge Computing

精选留言

北极点 2
这里的这些方案，总体下来感觉架构上需要超丰富的经验！协调一大帮人来弄这个。现在接触到的系统没有这么复杂！有问题一般一个小点一个小点的优化。读了文章还是感觉很收益！谢谢！
2018-03-09

mingshun 0
这是近几年所在的团队维护的其中一个重要系统的过程：最初为提高处理性能和水平扩展性，就从有状态往无状态发展。但随着数据量越来越大时，分布式存储就成了瓶颈，经常因为存储系统同步不及时导致不同节点读到的数据不一致。而后又回到有状态，但不在节点间使用分布式存储，因为数据量实在太大。为了在出现故障能快速恢复，每个节点做成双机热备。每改一次架构都要分好几次在周末深夜做数据迁移，多么痛的领悟啊！
理想情况下，计算密集型系统应该让存储向计算方移动，即做成无状态的；存储密集型系统应该让计算向存储移动，即做成有状态的。而上述架构来回修改的最终体会是，当计算和存储都密集的时候，应该整台机器地加，而且每台机器都要求有高的计算和存储性能。
然而说到底，存储其实又是由独立于CPU之外的一个计算单元来处理，本质似乎又回到了计算。现在设计分布式系统时总是先想着怎么分离计算和存储，最终发现计算和存储又是如此密不可分。
感觉我后半段的表述不太成立，打码久了不写人话的后果啊，望谅解！
2018-05-22

道 0
期待工作流引擎实现细节
2018-05-10

流迷的咸菜 0
虽然有状态的服务可以通过sticky session的方式将数据本地化，但是当这个sticky session expire的时候，或者服务处理完成之后，其相关的数据仍然是要同步到数据库中的吧？
2018-04-12

作者回复
不用，一般来说就是一个客户端cookie，或是uid分布一下，或是服务端的一个缓存。
2018-04-19

流迷的咸菜 0
文中提到，同步有同步的问题，分片也有分片的问题。同步的话主要是数据一致性和效率问题。那么分片的话，请问，主要问题是数据在其他片区上，需要远程获取数据，和对其他片区数据的写问题吗？
2018-04-12

kingeasternsun 0
sixcky session和DHT哪里有详细的介绍，这篇文章里好多名词都不认识
2018-04-03

作者回复
自行Google吧
2018-04-06

夜行观星 0
皓哥，这篇是在讲是在讲Service Mesh的思想吗？
2018-03-06

作者回复
不是，SM后面的文章会讲
2018-03-08