**高可用可伸缩架构实用经验谈**

移动互联网、云计算和大数据的成熟和发展，让更多的好想法得以在很短的时间内实现为产品。此时，如果用户需求抓得准，用户数量将很可能获得爆发式增长，而不需要像以往一样需要精心运营几年的时间。然而用户数量的快速增长（尤其是短时间内的爆发式增长），通常会让应用开发者有些吃不消，不得不面临一些严峻的技术挑战：如何避免因为单台机器当机导致服务不可用；如何避免在服务容量不足时，用户体验下降，等等。在系统构建之初就采用高可用和可伸缩架构，将能有效避免这些问题。

如何构建高可用和可伸缩架构呢？七牛云存储首席架构师李道兵在 3 月 22 的「开发者最佳实践日」第十期沙龙活动上给出了自己的想法。他结合自己多年的实践经验，针对一些不太复杂的业务场景，从入口层、业务层、缓存层和数据库层四个层面细致讲述了如何构建高可用和可伸缩系统。希望大家读完这篇文章，能觉得高可用和可伸缩不是一个高不可攀的东西，投入不高的成本就能在项目早期把高可用和可伸缩纳入架构设计之中。

**如何实现高可用**

**入口层**

入口层，通常指 Nginx 和 Apache 等层面的东西，负责应用（不管是 Web 应用还是移动应用）的服务入口。我们通常会将服务定位在一个 IP，如果这个 IP 对应的服务器当机了，那么用户的访问肯定会中断。此时，可以用 keepalived 来实现入口层的高可用。例如，机器 A 的 IP 是 1.2.3.4，机器 B 的 IP 是 1.2.3.5, 那么再申请一个 IP 1.2.3.6（称为⼼跳 IP）,平时绑定在机器A 上,如果 A 当机，IP 会自动绑定在机器 B 上；如果 B 当机，IP 会自动绑定在机器 A 上。对于这种形式，我们将 DNS 绑定到心跳 IP 上，即可实现入口层的高可用。

但这个方案有一点小问题。第一，它的切换可能会有一到两秒的中断，也就是说，如果不是要求到非常严格的毫秒级就不会有问题。第二，对入口的机器会有些浪费，因为买了两台机器的入口，可能就只有一台机器用上。对一些长连接的应用可能会导致服务中断，这时候就需要客户端做配合做一些重新创建连接的工作。简单来说，对于比较普通的业务来说，这个方案就能解决一部分问题。

这里要注意，keepalived 在使用上会有一些限制。

**·**两台机器必须在同一个网段，不是在同一个网段，没有办法实现互相抢 IP。

**·**内网服务也可以做心跳，但需要注意的是，以前为了安全我们会把内网服务绑定在内网 IP上，避免出现安全问题。但为了使用 keepalived，必须监听在所有 IP 上（如果监听在心跳IP 上，那么机器没有持有该 IP 时，服务无法启动），简单的方案是启用 iptables, 避免内网服务被外网访问。

**·**服务器利用率下降，这时可以考虑做混合部署来改善这一点。

比较常见的一个错误是，如果有两台机器，两个公网 IP，DNS 上把域名同时定位到两个 IP，就觉得已经做了高可用了。这完全不是高可用，因为如果一台机器当机,那么就有一半左右的用户无法访问。

除了 keepalive，lvs 也能用来解决入口层的高可用问题。不过，与 keepalived 相比，lvs 会更复杂一些，门槛也会高一些。

**业务层**

业务层通常是由 PHP、Java、Python、Go 等写的逻辑代码构成的，需要依赖于后台数据库及一些缓存层面的东西。如何实现业务层的高可用呢？最核心的就是，业务层不要有状态,将状态分散到缓存层和数据库。目前大家通常喜欢将以下几种数据放入业务层。

第一个是 session，即用户登录相关的数据，但好的做法是将 session 放在数据库里，或者一个比较稳定的缓存系统中。

第二个是缓存，在访问数据库时，如果一个查询很慢，就希望将这些结果暂时放到进程里，下次再做查询时就不用再访问数据库了。这种做法带来的问题是，当业务层服务器不只一台时，数据很难做到一致，从缓存拿到的数据就可能是错误的。。

一个简单的原则就是业务层不要有状态。在业务层没有状态时，一台业务层服务器当掉了之后，Nginx/Apache 会自动将所有的请求打到另外一台业务层的服务器上。由于没有状态，两台服务器没有任何差异，所以用户完全感受不到。如果把 session 放在业务层里面的话，那么面临的问题是，这个用户以前是登录在一台机器上的，这个进程死掉后，用户就会被登出了。

友情提醒：有一段时间比较流行 cookie session，就是将 session 中的数据加密之后放在客户的cookie 里，然后下发到客户端，这样也能做到与服务端完全无状态。但这里面有很多坑，如果能绕过这些坑就可以这样使用。第一个坑是怎么保证加密的密钥不泄露，一旦泄露就意味着攻击者可以伪造任何人的身份。第二个坑是重放攻击，如何避免别人通过保存 cookie 去不停地尝试的验证码，当然也还有其他一些攻击手段。如果没有好办法解决这两方面的问题，那么cookie session 尽量慎用。最好是将 session 放在一个性能比较好的数据库中。如果数据库性能不行，那么将 session 放在缓存中也比放在 cookie 里要好一点。

**缓存层**

非常简单的架构里是没有缓存这个概念的。但在访问量上来之后，MySQL 之类的数据库扛不住了，比如在 SATA 盘里跑 MySQL，QPS 到达 200、300 甚至 500 时，MySQL 的性能会大幅下降，这时就可以考虑用缓存层来挡住绝大部分服务请求，提升系统整体的容量。

缓存层做高可用一个简单的方法就是，将缓存层分得细一点儿。比如说，缓存层就一台机器的话，那么这台机器当了以后，所有应用层的压力就会往数据库里压，数据库扛不住的话，整个网站（或应用）就会随之当掉。而如果缓存层分在四台机器上的话，每台只有四分之一，这台机器当掉了以后，也只有总访问量的四分之一会压在数据库上面，数据库能扛住的话，网站就能很稳定地等到缓存层重新起来。在实践中，四分之一显然是不够的，我们会将它分得更细，以保证单台缓存当机后数据库还能撑得住即可。在中小规模下，缓存层和业务层可以混合部署,这样可以节省机器。

**数据库层**

在数据库层面实现高可用，通常是在软件层面来做。例如，MySQL 有主从模式(Master-Slave)，还有主主模式(Master-Master)都能满足需求。MongoDB 也有 ReplicaSet 的概念,基本都能满足大家的需求。

总之，要想实现高可用，需要做到这几点：入口层做心跳，业务层服务器无状态，缓存层减小粒度，数据库做一个主从模式。对于这种模式来讲，我们做的高可用不需要太多服务器，这些东西都可以同时部署在两台服务器上。这时，两台服务器就能满足早期的高可用需求了。任何一台服务器当机用户完全无感知。

**如何实现可伸缩**

**入口层**

在入口层实现伸缩性，可以通过直接水平扩机器，然后 DNS 加 IP 来实现。但需要注意，尽管一个域名解析到几十个 IP 没有问题,但是很多浏览器客户端只会使用前几个 IP,部分域名供应商对此有优化(如每次返回的 IP 顺序随机),但这个优化效果不稳定。

推荐的做法是使用少量的 Nginx 机器作为入口,业务服务器隐藏在内网(HTTP 类型的业务这种方式居多)。另外,也可以把所有 IP 下发到客户端，然后在客户端做一些调度(特别是非 HTTP 型的业务,如游戏、直播)。 **业务层**

业务层的伸缩性如何实现?与做高可用时的解决方案一样,要实现业务层的伸缩性，保证无状态是很好的手段。此外，加机器继续水平部署即可。

**缓存层**

比较麻烦的是缓存层的伸缩性，最简单粗暴的方式是什么呢？趁着半夜量比较低的时候，把整个缓存层全部下线，然后上线新的缓存层。新的缓存层启动起来之后，再等这些缓存慢慢预热。

当然这里一个要求，你的数据库能抗住低估期的请求量。如果扛不住呢？取决于缓存类型，下面我们先可以将缓存的类型区分一下。

1. 强一致性缓存：无法接受从缓存拿到错误的数据 (比如用户余额，或者会被下游继续缓存这种情形)

2. 弱一致性缓存：能接受在一段时间内从缓存拿到错误的数据 (比如微博的转发数)。

3. 不变型缓存：缓存 key 对应的 value 不会变更 (比如从 SHA1 推出来的密码, 或者其他复杂公式的计算结果)。

那什么缓存类型伸缩性比较好呢？弱一致性和不变型缓存的扩容很方便,用一致性 Hash 即可；

强一致性情况稍微复杂一些，稍后再讲。使用一致性 Hash，而不用简单 Hash 的原因是缓存的失效率。如果缓存从 9 台扩容到 10 台,简单 Hash 情况下 90%的缓存会马上失效,而如果使用一致性 Hash 情况,只有 10%的缓存会失效。

那么，强一致性缓存会有什么问题？第一个问题是，缓存客户端的配置更新时间会有微小的差异,在这个时间窗内有可能会拿到过期的数据。第二个问题是，如果扩容之后再裁撤节点,会拿到脏数据。比如 a 这个 key 之前在机器 1，扩容后在机器 2，数据更新了，但裁撤节点后 key回到机器 1，这时候就会拿到脏数据。

要解决问题 2 比较简单，要么保持永不减少节点,要么节点调整间隔大于数据的有效时间。问题1 可以用如下的步骤来解决：

1. 两套 hash 配置都更新到客户端，但仍然使用旧配置；

2. 逐个客户端改为只有两套 hash 结果一致的情况下会使用缓存，其余情况从数据库读，但写入缓存；

3. 逐个客户端通知使用新配置。

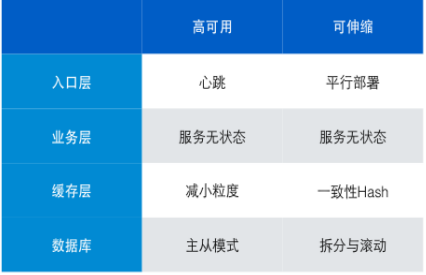
Memcache 设计得比较早，导致在伸缩性高可用方面的考虑得不太周到。Redis 在这方面有不少改进，特别是 @ngaut 团队基于 redis 开发了 codis 这个软件，一次性地解决了缓存层的绝大部分问题。推荐大家考察一下。

**数据库**

在数据库层面实现伸缩，方法很多，文档也很多，此处不做过多赘述。大致方法为：水平拆分、垂直拆分和定期滚动。

总之，我们可以在入口层、业务层面、缓存层和数据库层四个层面，使用刚才介绍的方法和技术实现系统高可用和可伸缩性。具体为：在入口层用心跳来做到高可用，用平行部署来伸缩；

在业务层做到服务无状态；在缓存层，可以减小一些粒度，以方便实现高可用，使用一致性 Hash将有助于实现缓存层的伸缩性；数据库层的主从模式能解决高可用问题，拆分和滚动能解决可伸缩问题。



本文中分享的这些技巧和方法，主要想帮助不太复杂的业务场景或者中小型应用快速搭建起高

可用可伸缩的系统。关于如何构建高可用和可伸缩系统还有很多更为细节的点和实践经验值得

探讨，望以后能与大家做更充分的交流。