漫谈高可用与负载均衡

当服务上线运行后，要尽量减少服务中断时间，也就是说让系统有较高的可用性。高可用往往与集群一起出现，因为高可用是由资源冗余来实现的。高可用本身又是一个很大的话题，从硬件资源的高可用，到依赖系统的高可用，再到具体应用设计的高可用。在产 品的初创阶段，高可用指标也不用太苛刻，我们只需要实现最基本的高可用，即用多点部署来达到基本的可用性保障。

高可用概念

衡量一个系统的可用性，使用的指标是给定时间内可用时间占用的百分比。这里可用 的意思是指用户可以正常地使用系统，比如，即使系统没有瘫痪，但是对用户来说响应延 迟远远超出预期，那么我们也可以认为此时系统处于不可用状态。系统的可用性反映了系 统的稳定程度。

高可用系统是指一个系统的可用性比较高，也就是说可用时间占用总体运行时间的百分比较高。一般我们会使用**“N 个 9”**这样的术语来描述高可用指标， **“1 个 9”**就是 90%， **“2 个 9”**就是 99%，以此类推。为了达到某一个高可用需求，就要尽可能降低系统的宕机时间。下表列出了不同的可用性百分比下，一年的最大宕机时间。

表 3-1　不同可用性下的最大宕机时间

可用性%	一年宕机时间	一月宕机时间	一周宕机时间	一天宕机时间
90%（“1 个 9”）	36.50d	72h	16.8h	2.4h
99%（“2 个 9”）	3.65d	7.2h	1.68h	14.4min
99.9%（“3 个 9”）	8.76h	43.8min	10.1min	1.44min
99.99%（“4 个 9”）	52.56min	4.38min	1.01min	8.66s
99.999%（“5 个 9”）	5.26min	25.9s	6.05s	864.3ms

(不同可用性下的最大宕机时间)

高可用的意义

下面，我们以一个 10 万 PV 的产品为例，来看看使用简单的异常→通知→人工处理异 常模式会造成的影响。

- 异常发现阶段：服务发生了异常，我们假定 30s 内触发了相应的监控报警，经过 系统处理，30s 之后（考虑到程序处理和运营商的延时，30s 已经是一个比较理想 的数字）发送到相关负责人的手机上。我们假定这个负责人十分负责，并且会及 时查看手机信息，因此他迅速查看了手机，发现了系统出现问题。假定他看信息，所花时间为 10s，那么从异常出现到运维人员得到通知，至少得 1min 的时间。
- 故障定位阶段：我们再假定报警短信十分详备，运维人员经验也非常丰富，并且 快速地完成了故障定位，花费时间假定为 1min。
- 问题处理：假设对应负责人手边恰好就是他的工作电脑，有现成的网络环境，迅 速连接到出问题的服务器上，并且找到对应的进程，完成进程重启或者配置调整。 这个过程假定时间 1min。我们完成了所有的故障处理后，服务开始恢复。假定服务通过简单的重启就可以恢复， 时间为 30s。

最终，我们的服务恢复了，在最好的情况下用了差不多 5min 的时间。按照产品高峰期 10h 占 90%的流量，其他时间占用 10%的流量来算，在业务高峰期，平均每秒的访问人数 为 90000/10/3600＝2.5 人，那么 5min 的服务异常时间，差不多会有 600 位用户受到影响。

但是上述都是假设在理想情况下，实际上任何一个环节都可能出现意外，运营商的报 警信息延迟、运维人员没有实时看报警信息、运维人员手边没有网络、运维人员没能快速 定位到错误所在等。从互联网产品运维的经验出发，在相关人员都是正常水平从业专业人员 的情况下，15min 已经是非常理想的情况，按照这个标准，高峰期影响的人数可能达到 2000 人左右。而且一般触发异常的情况，都是业务高峰期，只有在业务高峰期，才会对系统带 来压力，出现一些平时测试时没有遇到的情况；反而业务低谷的时候，更不容易出现异常。

一个产品的可用率，不但会直接影响到用户业务，还会直接影响用户对产品的信心。 如果说服务不可用对用户造成的经济损失随时可以通过代金券、免费物品等经济方式来弥 补，那服务不可用造成对产品信心的损失则几乎不可恢复。特别是在初创期，获取用户已经 非常不容易，如果由于服务的接二连三不可用导致用户流失，更加得不偿失。

因此，我们不仅要解决服务能不能恢复的问题，还需要解决服务如何快速恢复，并且 在服务节点部分失效的情况下保证系统仍然可以正常工作的问题。

一个高可用的系统会为我们带来最长恢复时间的保证，由于没有人为介入，异常恢复 处理是自动进行的，因此保证系统的最长恢复时间，可以大大减少对用户的影响。

另外，由于一般的请求都有重试机制，因此在使用高可用的情况下，当单个节点异常 时，对用户来说，往往是出现短暂的延时，而不会失败。即使没有重试机制，一般用户也 会通过重试来尝试解决问题。由于服务可以快速恢复，在用户重试几次之

后就会发现服务恢复正常，在这种情况下，用户并不会认为是服务的问题，觉得只是单纯的网络抖动或者 其他原因，并不影响用户对产品的信心。

系统健康检查

先要知道系统是不是健康可用的，才可能在出问题的时候进行自动恢复。传统上，如 果一个系统的进程无法工作，或者端口监听不在了，就肯定不健康，也很容易就被检测到。

实际上，更多情况下，我们要面对的不健康情况，是由于系统的一些异常情况处理不 严谨或者随着负载增加，造成系统的响应不满足预期。比如，由于系统的 bug，使得系统 运行进入死循环，此时虽然进程没停止，端口监听完好，但是却无法对外提供服务。再比 如，某一个页面，正常情况下加载会在几秒内完成，业务要求在 10s 内完成。如果由于系 统问题，页面加载需要 1min，那么此时虽然页面还能加载完成，但是达不到业务需求，系 统也是不健康的。

因此，这里的健康检查，实际上更多是对系统的业务健康度的健康检查，往往与业务 有关。一般情况下，这种场景下的健康检查有两种方式：脚本和协议。

脚本的方式，是每隔一段时间运行一次某一个定制的本地脚本，根据脚本的返回码确认 系统是否健康。在脚本执行中，可以去检查进程状态，发一些模拟请求等。还有一种比较常 见的方式，就是在系统中插入一些检查点，然后在脚本中对这些检查点进行检查。比如，可 以在系统中加入，每隔一段时间，就把当前时戳写到一个本地文件中，然后在脚本中对这个 文件进行检查，一旦发现这个本地文件更新不正常，就可以认为原来的系统已经不健康了。

对于绝大部分应用来说，一般都是 Web 服务，Web 服务直接基于 HTTP 协议进行健康 检查更合适。比如，对于一个论坛，每隔一段时间，请求一次其主页，看看是否能够请求 成功，响应时间是否满足其需求。对于 HTTP 健康检查，需要配置其健康检查的 URL 及响 应超时时间。对于一个长连接应用来说，配置 TCP 的健康检查，需要指定端口号及连接建 立超时时间。

高可用实现方式

在初创期，我们的服务架构是一个单体架构，这里的高可用主要是指我们服务器进程 的高可用，一般是采用服务器冗余来实现的，当部分节点异常时，其他节点仍然可以正常 工作，从而保证系统正常运行。同时，多点部署也能够实现初步的水平扩展，以应对用户的增长。

多个节点的服务部署构成一个集群，也就是高可用集群，这也是我们在阅读一些文章的时候，会发现高可用总是与集群一起出现的原因。
在一个高可用集群中，我们可以配置对各个节点的健康检查，当有节点出现异常的时 候，只要其他节点还在正常工作，系统整体的可用性就不会受到影响。我们可以将系统部 署在容器中，通过 Kubernetes 进行管理，多点部署实际对应的就是 Kubernetes 中的多个副本的概念。Kubernetes 同时也支持对其上运行的容器进行健康检查，并且可以在健康检查 失败后，自动重启相关的 Pod。

除了系统 bug 过多导致系统频繁异常，造成自我 DoS（Denial of Service，拒绝服务） 攻击外，使用这种多点部署的方式，基本上可以保证"2 个 9"到"3 个 9"的可用性。

负载均衡

在对系统进行多点部署后，我们需要将用户请求按照不同的转发策略，发送到部署的 不同的节点上，负责完成请求转发的设备，我们称之为负载均衡。负载均衡可以完成流量 分发，负载均衡的服务示例如图所示。

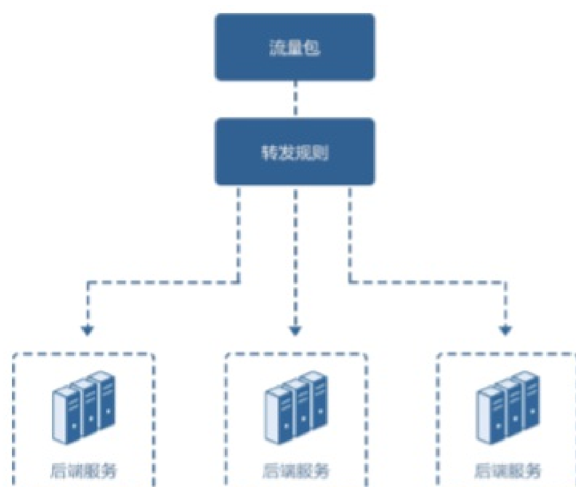

图 3-11 负载均衡服务示例

(负载均衡服务示例)

负载均衡作为用户服务的统一入口，将用户流量分摊到不同的后端服务上去，可以实现服务的初步水平扩展，以应对支持增长

的业务压力。另外，负载均衡设备除了提供流量 分发外，还会提供故障隔离。一般负载均衡都提供了对后端服务的健康检查机制，如果检 测到后端节点异常，负载均衡可以不再向其转发请求，完成故障隔离，提高系统的可用性。

对于 Web 应用来说，我们一般可以使用 Apache、Nginx 等来实现负载均衡。云原生架构下，可以直接使用这些软件的镜像，一键部署，即可启动一个负载均衡节点。不过，假如我们使用 Nginx 来实现后端节点的负载均衡，那么我们的 Nginx 又成为了单点，我们不 得不再启用多个 Nginx 节点来进行高可用，就算我们完成了 Nginx 的多点部署，交换机也 可能挂掉。因此，自己去实现一个高可用的负载均衡是困难的。不过，幸运的是，在云计算中，基本上所有的云服务提供商都把负载均衡作为基础服务提供，如 AWS 的 ELB、阿 里云的 SLB、网易云基础服务的 NLB 等。

由于负载均衡服务偏底层，因此不同云计算服务商提供的负载均衡服务虽然底层实现技术各不相同，但是实现的功能都是类似的。

粘性会话（会话保持）

如果一个服务是有状态的，服务中维护了用户某一系列请求的 Session 信息，以电商为例，假如把购物车的商品清单保存到了 Session 中。当只有单点部署时，用户的每一 次请求，都会落到这个服务的部署节点上，最后提交订单时，可以从 Session 中直接获取 购物车信息。但是当部署了多点后，如果不加任何限制，可能最后提交订单的请求落到 了其他节点上，而在其他节点上有没有这个 Session 信息，从而查不到购物车信息，导致 问题。

解决这个问题的一种方案是将所有这种 Session 信息都保存到共享的一个外部缓存服 务中，这样多节点之间可以共享这些数据。该方案额外引入了一个外部依赖，从而引入了 复杂性，我们还可以通过在负载均衡上面配置粘性会话（Sticky Session）来实现，粘性会 话有时也称作会话保持。

使用一个 Cookie 字段给负载均衡使用，负载均衡设备会根据这个 Cookie 字段，来决 定往哪个节点转发流量。对于同一个请求会话来说，其用于负载均衡的 Cookie 字段值都是 相同的，负载均衡会把属于这个会话的所有请求都转发到同一个节点上，从而避免了上述 提到的问题。这个 Cookie 字段可以是用户应用中使用的某一个 Cookie 字段，比如 Tomcat 中的 JSESSIONID，也可以由负载均衡自动生成，从而对应用透明。几乎所有的云负载均衡服务都提供了对粘性会话的支持，我们只需要对负载均衡进行一下配置即可。