09 | 微服务治理的手段有哪些?

2018-09-11 胡忠想

09 | 微服务治理的手段有哪些?
朗读人：胡忠想　10′55″ | 5.01M

上一期我给你讲述了服务追踪的基本原理，有了分布式服务追踪系统，在服务出现问题的时候，我们就可以定位服务哪里出现了问题。一般单体应用改造成微服务架构后，还会增加哪些问题呢？又该如何应对呢？

前面我讲到单体应用改造为微服务架构后，服务调用由本地调用变成远程调用，服务消费者 A 需要通过注册中心去查询服务提供者 B 的地址，然后发起调用，这个看似简单的过程就可能会遇到下面几种情况，比如：

- 注册中心宕机；
- 服务提供者 B 有节点宕机；
- 服务消费者 A 和注册中心之间的网络不通；
- 服务提供者 B 和注册中心之间的网络不通；

- 服务消费者 A 和服务提供者 B 之间的网络不通；
- 服务提供者 B 有些节点性能变慢；
- 服务提供者 B 短时间内出现问题。

可见，一次服务调用，服务提供者、注册中心、网络这三者都可能会有问题，此时服务消费者应该如何处理才能确保调用成功呢？这就是服务治理要解决的问题。

接下来我们一起来看看常用的服务治理手段。

节点管理

根据我的经验，服务调用失败一般是由两类原因引起的，一类是服务提供者自身出现问题，如服务器宕机、进程意外退出等；一类是网络问题，如服务提供者、注册中心、服务消费者这三者任意两者之间的网络出现问题。

无论是服务提供者自身出现问题还是网络发生问题，都有两种节点管理手段。

1. 注册中心主动摘除机制

这种机制要求服务提供者定时的主动向注册中心汇报心跳，注册中心根据服务提供者节点最近一次汇报心跳的时间与上一次汇报心跳时间做比较，如果超出一定时间，就认为服务提供者出现问题，继而把节点从服务列表中摘除，并把最近的可用服务节点列表推送给服务消费者。

2. 服务消费者摘除机制

虽然注册中心主动摘除机制可以解决服务提供者节点异常的问题，但如果是因为注册中心与服务提供者之间的网络出现异常，最坏的情况是注册中心会把服务节点全部摘除，导致服务消费者没有可用的服务节点调用，但其实这时候服务提供者本身是正常的。所以，将存活探测机制用在服务消费者这一端更合理，如果服务消费者调用服务提供者节点失败，就将这个节点从内存中保存的可用服务提供者节点列表中移除。

负载均衡

一般情况下，服务提供者节点不是唯一的，多是以集群的方式存在，尤其是对于大规模的服务调用来说，服务提供者节点数目可能有上百上千个。由于机器采购批次的不同，不同服务节点本身的配置也可能存在很大差异，新采购的机器 CPU 和内存配置可能要高一些，同等请求量情况下，性能要好于旧的机器。对于服务消费者而言，在从服务列表中选取可用节点时，如果能让配置较高的新机器多承担一些流量的话，就能充分利用新机器的性能。这就需要对负载均衡算法做一些调整。

常用的负载均衡算法主要包括以下几种。

1. 随机算法

顾名思义就是从可用的服务节点中随机选取一个节点。一般情况下，随机算法是均匀的，也就是说后端服务节点无论配置好坏，最终得到的调用量都差不多。

2. 轮询算法

就是按照固定的权重，对可用服务节点进行轮询。如果所有服务节点的权重都是相同的，则每个节点的调用量也是差不多的。但可以给某些硬件配置较好的节点的权重调大些，这样的话就会得到更大的调用量，从而充分发挥其性能优势，提高整体调用的平均性能。

3. 最少活跃调用算法

这种算法是在服务消费者这一端的内存里动态维护着同每一个服务节点之间的连接数，当调用某个服务节点时，就给与这个服务节点之间的连接数加 1，调用返回后，就给连接数减 1。然后每次在选择服务节点时，根据内存里维护的连接数倒序排列，选择连接数最小的节点发起调用，也就是选择了调用量最小的服务节点，性能理论上也是最优的。

4. 一致性 Hash 算法

指相同参数的请求总是发到同一服务节点。当某一个服务节点出现故障时，原本发往该节点的请求，基于虚拟节点机制，平摊到其他节点上，不会引起剧烈变动。

这几种算法的实现难度也是逐步提升的，所以选择哪种节点选取的负载均衡算法要根据实际场景而定。如果后端服务节点的配置没有差异，同等调用量下性能也没有差异的话，选择随机或者轮询算法比较合适；如果后端服务节点存在比较明显的配置和性能差异，选择最少活跃调用算法比较合适。

服务路由

对于服务消费者而言，在内存中的可用服务节点列表中选择哪个节点不仅由负载均衡算法决定，还由路由规则确定。

所谓的路由规则，就是通过一定的规则如条件表达式或者正则表达式来限定服务节点的选择范围。

为什么要制定路由规则呢？主要有两个原因。

1. 业务存在灰度发布的需求

比如，服务提供者做了功能变更，但希望先只让部分人群使用，然后根据这部分人群的使用反馈，再来决定是否做全量发布。这个时候，就可以通过类似按尾号进行灰度的规则限定只有一定比例的人群才会访问新发布的服务节点。

2. 多机房就近访问的需求

据我所知，大部分业务规模中等及以上的互联网公司，为了业务的高可用性，都会将自己的业务部署在不止一个 IDC 中。这个时候就存在一个问题，不同 IDC 之间的访问由于要跨 IDC，通过专线访问，尤其是 IDC 相距比较远时延迟就会比较大，比如北京和广州的专线延迟一般在 30ms 左右，这对于某些延时敏感性的业务是不可接受的，所以就要一次服务调用尽量选择同一个 IDC 内部的节点，从而减少网络耗时开销，提高性能。这时一般可以通过 IP 段规则来控制访问，在选择服务节点时，优先选择同一 IP 段的节点。

那么路由规则该如何配置呢？根据我的实际项目经验，一般有两种配置方式。

1. 静态配置

就是在服务消费者本地存放服务调用的路由规则，在服务调用期间，路由规则不会发生改变，要想改变就需要修改服务消费者本地配置，上线后才能生效。

2. 动态配置

这种方式下，路由规则是存在注册中心的，服务消费者定期去请求注册中心来保持同步，要想改变服务消费者的路由配置，可以通过修改注册中心的配置，服务消费者在下一个同步周期之后，就会请求注册中心来更新配置，从而实现动态更新。

服务容错

服务调用并不总是一定成功的，前面我讲过，可能因为服务提供者节点自身宕机、进程异常退出或者服务消费者与提供者之间的网络出现故障等原因。对于服务调用失败的情况，需要有手段自动恢复，来保证调用成功。

常用的手段主要有以下几种。

- FailOver：失败自动切换。就是服务消费者发现调用失败或者超时后，自动从可用的服务节点列表总选择下一个节点重新发起调用，也可以设置重试的次数。这种策略要求服务调用的操作必须是幂等的，也就是说无论调用多少次，只要是同一个调用，返回的结果都是相同的，一般适合服务调用是读请求的场景。

- FailBack：失败通知。就是服务消费者调用失败或者超时后，不再重试，而是根据失败的详细信息，来决定后续的执行策略。比如对于非幂等的调用场景，如果调用失败后，不能简单地重试，而是应该查询服务端的状态，看调用到底是否实际生效，如果已经生效了就不能再重试了；如果没有生效可以再发起一次调用。

- FailCache：失败缓存。就是服务消费者调用失败或者超时后，不立即发起重试，而是隔一段时间后再次尝试发起调用。比如后端服务可能一段时间内都有问题，如果立即发起重试，可

能会加剧问题，反而不利于后端服务的恢复。如果隔一段时间待后端节点恢复后，再次发起调用效果会更好。

- FailFast：快速失败。就是服务消费者调用一次失败后，不再重试。实际在业务执行时，一般非核心业务的调用，会采用快速失败策略，调用失败后一般就记录下失败日志就返回了。

从我对服务容错不同策略的描述中，你可以看出它们的使用场景是不同的，一般情况下对于幂等的调用，可以选择 FailOver 或者 FailCache，非幂等的调用可以选择 FailBack 或者 FailFast。

总结

上面我讲的服务治理的手段是最常用的手段，它们从不同角度来确保服务调用的成功率。节点管理是从服务节点健康状态角度来考虑，负载均衡和服务路由是从服务节点访问优先级角度来考虑，而服务容错是从调用的健康状态角度来考虑，可谓是殊途同归。

在实际的微服务架构实践中，上面这些服务治理手段一般都会在服务框架中默认集成了，比如阿里开源的服务框架 Dubbo、微博开源的服务框架 Motan 等，不需要业务代码去实现。如果想自己实现服务治理的手段，可以参考这些开源服务框架的实现。

思考题

上面讲述的这些服务治理手段，哪些是你的业务场景中可能需要的？你可以描述下你的业务场景，以及思考下为什么这些服务治理手段可以解决你的问题。

欢迎你在留言区写下自己的思考，与我一起讨论。

通过留言可与作者互动