超时、重试和抖动回退

Marc Brooker

故障时有发生

每当一个服务或系统调用另一个服务或系统时，都可能会发生故障。造成故障的因素可能多种多样。它们包括服务器、网络、负载均衡器、软件、操作系统，甚至是系统操作员失误。我们的系统设计致力于减少故障发生的可能性，但是没有任何人能构建出永不中断的系统。因此，在 Amazon，我们的系统设计竭力提升容忍故障的能力并降低发生故障的可能性，避免将原本较小比例的故障放大成一次全面停机。为了构建弹性系统，我们使用了三种必备工具：超时、重试和回退。

许多类型的故障都会显露出明显的迹象，比如请求花费的时间比平时更长，并且可能永远无法完成。如果客户端等待一项请求完成的时间比平常更长，它也会因将该资源用于处理请求而将资源保留更长时间。如果大量请求长时间占用资源，服务器的相应资源就可能耗尽。这些资源可能包括内存、线程、连接、临时端口或任何其他有限的资源。为了避免这种情况，客户端可以设置*超时*。超时是客户端等待请求完成的最长时间。

通常，再次尝试相同的请求会使得请求成功。这是因为我们构建的系统类型通常不会作为一个整体失败。相反，它们会遭受部分或瞬态故障。部分故障是指一定百分比的请求成功。瞬态故障是指请求在短时间内失败。*重试*允许客户端通过再次发送相同的请求来幸免于这些随机的部分故障和瞬态故障。

重试并非总是安全的。如果系统已经因过载而出现故障，那么重试只会增加被调用系统的负载。为避免此问题，我们的客户端实现可使用*回退*。这增加了后续重试之间的时间，从而使后端的负载保持均匀。重试的另一个问题是某些远程调用会产生副作用。超时或失败并不一定意味着没有产生副作用。如果不希望多次产生副作用，则最佳实践是将 API 设计为*幂等*形式，也就是说可以安全地重试它们。

最后，流量不会以恒定的速度进入 Amazon 服务。请求的到达率经常发生大突发。这些突发事件可能是由客户端行为、故障恢复，甚至是由诸如定期 cron 作业之类的简单事件引起的。如果错误是由负载引起的，那么如果所有客户端同时重试，则重试可能无效。为了避免这个问题，我们采用*抖动*。这是指发出或重试请求之前的随机时间，目的在于通过分散到达率来帮助防止大规模突发事件。

以下各节中将讨论每种解决方案。

超时

Amazon 的最佳实践是对任何远程调用设置超时，并且通常在跨多个进程的任何调用上设置超时（即便这些进程位于相同机器上）。这包括连接超时和请求超时。许多标准客户端都提供了强大的内置超时功能。

通常，最困难的问题在于选择要设置的超时值。超时设置得太高会降低其有效性，因为在客户端等待超时期间仍会消耗资源。超时设置得太低有两个风险：

- 由于重试了太多请求，因此增加了后端流量并增加了延迟。
- 小规模后端延迟增加，导致完全中断，因为所有请求都开始重试。

为 AWS 区域内的呼叫选择超时的一种好方式是从下游服务的延迟指标开始。因此，在 Amazon，当我们使用一项服务调用另一项服务时，我们会选择一个可接受的错误超时率（例如 0.1%）。随后，我们查看下游服务的相应延迟百分比（在此示例中为 p99.9）。这种方法在大多数情况下都能很好地发挥作用，但存在一些陷阱，如下所述：

- 在客户端存在严重网络延迟（例如通过互联网访问）的情况下，这种方法不起作用。在这些情况下，我们会考虑到合理最坏情况下的网络延迟，但不要忘记，客户端可能遍布全球。

- 此方法也不适用于延迟时间边界范围狭窄的服务，即 p99.9 接近 p50 的情况。在这些情况下，添加一些填充可以帮助我们避免导致大量超时的细微延迟增加。

- 实施超时的时候，我们遇到了一个常见的陷阱。Linux 的 SO_RCVTIMEO 非常强大，但是有一些缺点，使其不适合作为端到端套接字超时。某些语言（例如 Java）直接公开了这种控件。其他语言（例如 Go）提供了更强大的超时机制。

- 在某些实现中，超时不包括所有远程调用，例如 DNS 或 TLS 握手。通常，我们更喜欢使用经过良好测试的客户端内置的超时时间。如果我们实现自己的超时，我们会特别注意超时套接字选项的确切含义以及正在执行的工作。

在我在 Amazon 负责过的一个系统中，我们发现少量超时在部署后立即与依赖关系通信。超时设置为非常低的值，大约为 20 毫秒。在部署之外，即使超时值很低，我们也没有看到定期发生超时的情况。深入研究后，我发现计时器包括建立一个新的安全连接，该连接将在后续请求中重复利用。由于建立连接的时间超过 20 毫秒，所以当新服务器在部署后投入使用时，我们看到少量请求超时。在某些情况下，请求重试并成功。我们最初通过在建立连接的情况下增加超时值来解决此问题。后来，我们通过在进程启动时、接收流量之前建立这些连接来改进系统。这让我们完全解决了超时问题。

重试和回退

重试是“自私的”。 换句话说，在客户端重试时，它将花费更多的服务器时间来获得更大的成功几率。在故障很少发生或瞬态发生的情况下，这并不是问题。这是因为重试请求的总数很小，并且

增加表面可用性的权衡效果也很好。如果故障是由过载引起的，重试会增加负载，导致情况进一步恶化。在原始问题得到解决后，它们甚至可能会通过保持较高的负载造成恢复延迟。重试类似于强大的药物，应该以正确的剂量使用，使用过量只会造成严重损害。遗憾的是，在分布式系统中，几乎无法在所有客户端之间进行协调以实现正确的重试次数。

Amazon 采用的首选解决方案是回退。客户端不会立即积极地重试，而是在两次尝试之间等待一段时间。最常见的模式是*指数回退*，每次尝试后的等待时间都呈指数级延长。指数回退可能导致很长的回退时间，因为指数函数增长很快。为了避免重试太长时间，实现通常会将回退设置为最大值。可以预见，这称为*上限指数回退*。但是，这带来了另一个问题。现在，所有客户端都会不断尝试重设上限速率。我们的解决方案几乎在所有情况下都限制客户端重试的次数，并在面向服务的架构中更早地处理由此导致的故障。大多数时候，客户端都会放弃调用，因为它有着自己的超时时间。

重试还有其他问题，描述如下：

- 分布式系统通常具有多个分层。想一下有这样一个系统：该系统中的客户调用导致了五层深的服务调用堆栈。它最终执行一次对数据库的查询，并在每层重试三次。如果在负载下数据库开始使查询失败，会发生什么？ 如果每层单独进行重试，则数据库上的负载将增加 243 倍，导致恢复几乎成为不可能完成的任务。这是因为每一层的重试次数都会成倍增加，首先是三次尝试，然后时九次尝试，依此类推。相反，在堆栈的最高层重试可能会浪费以前的调用工作，从而降低效率。通常，对于低成本控制平面和数据层面操作，我们的最佳实践是在堆栈中的单个点执行重试。

- 负载。即使在单层重试，错误开始时流量仍然会显著增加。为了解决此问题，我们广泛推行了*断路机制*，也就是在超过错误阈值时，对下游服务的调用将完全停止。遗憾的是，断路机制将模态行为引入了可能难以测试的系统，可能会导致恢复时间延长。我们发现，可以通过使用令牌桶在本地限制重试，从而减轻这种风险。只要有令牌，就可以重试所有调用，然后在令牌耗尽时以固定的速率重试。2016 年，AWS 将此行为添加到了 AWS 开发工具包中。因此，使用该开发工具包的客户内置了这种限制行为。

- 确定何时重试。通常，我们认为具有副作用的 API 必须要提供幂等性才能保证重试的安全性。这样可以确保无论您重试多少次，副作用都只会发生一次。只读 API 通常是幂等的，而资源创建 API 则可能不是。某些 API（例如 Amazon Elastic Compute Cloud (Amazon EC2) RunInstances API）提供了基于令牌的显式机制，以提供幂等性并保证重试的安全。为了防止重复出现副作用，需要良好的 API 设计，并在实现客户端时要格外小心谨慎。

- 了解哪些故障值得重试。HTTP 明确区分了*客户端*和*服务器*错误。它指示不应对相同的请求重试客户端错误，因为它们后续也不会成功，而服务器错误可能在后续尝试中成功。遗憾的是，系统的最终一致性极大地模糊了这条界线。随着状态的传播，客户端错误可能在下一刻变为成功。

尽管存在这些风险和挑战，重试仍然是在遇到瞬态和随机错误时提供高可用性的一种强大机制。为了找到适合每种服务的合理权衡，需要一定的判断力。根据我们的经验，一个理想的起点就是谨记“重试是自私的”。重试是让客户端断言其请求的重要性并要求该服务花费更多资源来处理它的一种方法。如果客户端过于自私，则会造成大范围的问题。

抖动

如果故障是由过载或争用引起的，回退通常无法带来应有的帮助。其原因就在于相关性。如果所有失败的调用都回退到同一时间，则重试这些调用会导致争用或过载。我们的解决方案就是抖动。抖动会给回退增加一定程度的随机性，以使重试在时间上有所分散。如需详细了解添加多少抖动以及添加抖动的最佳方法，请参见指数回退和抖动。

抖动不仅用于重试。在运营方面的经验告诉我们，我们的服务（包括控制平面和数据层面）的流量往往会产生很多峰值。这些流量峰值可能非常短，而且通常不体现在聚合指标中。在构建系统时，我们考虑为所有计时器、周期性作业和其他延迟的工作增加一些抖动。这有助于分散工作峰值，并使下游服务更容易扩展以适应工作负载。

向计划的工作添加抖动时，我们不会随机选择各主机上的抖动。相反，我们使用一致的方法，每次在同一主机上产生相同的数字。这样，如果服务过载或出现争用状况，则它始终以相同的方式、遵循一种模式发生。人类善于识别模式，这让我们更有可能确定根本原因。使用随机方法可以确保，如果资源不堪重负，那么就会发生过载，而且是随机的。这就使得问题排查变得更加困难。

在我从事过的系统上，例如 Amazon Elastic Block Store (Amazon EBS) 和 AWS Lambda，我们发现客户端经常以固定的时间间隔发送请求，例如每分钟一次。但是，如果一个客户端有多台服务器采用相同的行为方式，则它们可以排起来同时触发其请求。这可以是一分钟的前几秒，也可以是午夜后处理日常作业的前几秒。通过关注每秒的负载，并与客户端合作以对其周期性的工作负载实施抖动，我们就能以更少的服务器容量完成相同数量的工作。

我们对客户流量峰值的控制更少。但是，即使对于客户触发的任务，在不影响客户体验的情况下添加抖动也是一个好主意。

结论

在分布式系统中，瞬态故障或远程交互中的延迟不可避免。超时会导致系统异常长时间地挂起，重试可能会掩盖这些故障，而回退和抖动可以提高利用率并减少系统拥塞问题。

在 Amazon，我们了解到，谨慎重试非常重要。重试会加重从属系统上的负载。如果对系统的调用超时，且该系统过载，则重试会导致过载问题恶化，而非好转。我们仅在观察到依赖关系运行状况良好时才会重试，从而避免了这种加重负载的问题。当重试无助于提高可用性时，我们将停止重试。