Go 进阶训练营
第5课
Go 架构实践 - 微服务可用性答疑

大明

大纲

1. 如何用毛老师的内容来面试
2. 服务治理的大一统模型（大明版）

怎么保证微服务的可用性

- 分析：这个问题，就是回答毛老师这节课目录：隔离、超时控制、过载保护、限流、降级、重试、负载均衡。尽量全部记住，如果记不住全部，至少记住过载保护、限流、降级三个，这是最基本的最常见最经常讨论的，然后再**额外记住至少两个**作为亮点！
- 回答的时候，隔离要回答到具体的子类，也就是动静分离、读写分离、轻重隔离、物理隔离四个。等后面面试官具体问怎么做这些隔离；
- 超时控制，要注意回答到链路超时控制，进程内的控制是基本功，没啥好说的。但是要防着面试官让你手写一个 Go context 超时控制的代码。
- 过载保护要回到几种常见的算法，但是不需要聊细节。（后面如果掌握了这些算法的优劣，也可以在这里讨论）
- 限流要讨论单机限流、分布式限流和针对业务的限流
- 重试要强调关键在于重试的时机，但是不需要聊细节，等面试官问你怎么挑合适的时机
- 负载均衡也是大概聊一下算法，等面试官面细节

怎么保证微服务的可用性

- 答：保证微服务的可用性是一个系统性问题（提纲挈领，先来依据确立理论高度的废话）。一般来说，站在开发者角度，保证微服务的可用性主要可以从这些角度出发：
  - 隔离。主要包括动静分离、读写分离、轻重隔离和物理隔离四个措施；
  - 超时控制。主要是做到链路超时控制，防止某些异常业务长时间占据资源
  - 过载保护。有固定窗口、滑动窗口、令牌桶、漏桶等算法
  - 限流。限流可以是单机限流，也可以是集群限流，也可以是针对业务对象的限流
  - 降级。降级可以是服务间降级，即丢弃不重要服务的请求，也可以是服务降级，即走快速路径，比如说直接返回默认响应等
  - 重试。重试的关键在于选择重试的时机，比如说很多时候，如果下游出问题了，立刻重试是没有用的
  - 负载均衡。负载均衡应当说是，如果做好了，大概都不需要别的手段了。负载均衡的算法也是多种多样的，比如说常见的轮询、基于权重的轮询

怎么保证微服务的可用性

- 类似问题
  - 服务治理怎么做？
  - 你们公司做了什么来治理服务（or 保证可用性，保证稳定性，降低错误率...）

你们是怎么做隔离的

- 分析：如果在简历里面有提到什么服务治理，那么面试官可能直接问。比较大可能是在聊服务治理的时候带过来的。

  在这个问题之下，基本的回答是毛老师课程里面的那四点：动静分离、读写分离、轻重隔离和物理隔离。

  这种带着实践的问题，最好是要结合自己公司，举至少一个例子出来，这样会更加有说服力。特别是这里问了“你们”是怎么做的，所以要轻理论重实践。如果是宽泛的问如何做隔离，那么就可以重理论轻实践。

  如果自己公司确实没有做怎么办？

你们是怎么做隔离的

- 回答：隔离从大方向上来说，可以做动静分离、读写分离、轻重隔离和物理隔离。
  - 动静隔离主要就是利用 CDN 来分发资源，（我司已经接入了 CDN）。读写分离一方面是指数据库读写分离的那种主从式的隔离，但是还有一种是针对读写是两种角色的操作，而进行的分发，（这种读写分离一般是和公司业务直接相关的，所以有些学员应该不太可能接触到这种搞法，那么就可以忽略）典型的就是所谓草稿库的设计，（刷个亮点）这种隔离还有一个好处，就是可以做到草稿库同步到线上库的时候，同时刷新缓存，保证缓存一直都是最新的，而且不必设置过期时间（数据多还是要淘汰的）
  - 轻重隔离则是五花八门。核心都是根据资源的重要性，将重要资源和不重要资源隔离开来，同时重要资源之间也相互隔离，以保证一些资源崩溃的时候，另外一些资源不会受到影响。资源一般就是服务和数据。而判定是否需要隔离的标准则很多，可以是服务的业务价值，服务的 QPS，数据的价值，数据被访问 QPS 等，由此引申出来了核心隔离，快慢隔离，热点隔离 （这一段回答属于个人风格，你们可以直接按照毛老师 PPT 举例来回答）。比如说我司就用了 xxx （随便说几个例子）
  - 物理隔离，比较多是指进程隔离、线程隔离和集群隔离。（刷个 go 语言特性）线程隔离在 Java 这种语言比较常见，一般是线程池隔离。 go 的框架在这方面使用得比较少，主要是因为 goroutine 非常轻量。在一些特别复杂的框架里面也会有人应用 gooutine 池，但是总体来说并不是主流，复杂性太多而收益太小。目前来说大规模采用的 docker 之类的容器技术应该算最为典型的。我司这方面用的是 xxxx

你们是怎么做隔离的

- 类似问题
  - 为什么 go 不用线程池隔离？没线程池，类似的 goroutine 那么轻量，没太大池化的价值，性价比太低
  - 你们用过 goroutine 池吗？有就是有，没有就是没有。没有就回答性价比低，有就回答性能提升，降低资源消耗
  - 你们怎么做冷热数据隔离？or 预热数据？这个问题其实在缓存里面讨论会更加适合
  - 怎么保证热点服务的可用性？隔离是一方面，还是很重要的一方面，基本上就是热点服务啥都给它单独的一分，不和任何服务共享资源，避免相互影响。其实这个问题基本等价于“如何治理微服务”，毕竟一个不怎么重要的服务，我们也懒得费神那么多精力上那么多组件

你们是怎么做隔离的

- 案例
  - 草稿库和线上库，比如说博客、商户信息、商品信息，都是有一个录入、审核、发布的过程，就非常适合
  - B 端和 C端服务隔离部署。同一个服务，如果同时需要提供给商家和用户，那么就可以单独部署，必要时刻还可以切掉 B 端流量，全力支撑 C 端流量
  - VIP 和普通用户隔离，VIP 有专门的高性能资源服务
  - 仿照 Case Study 自己想想公司有什么可以做隔离的。关键是识别出来，什么是重要的，什么是不重要的，而且必要时候，弃车保帅里面，谁是车，谁是帅，以及，谁是小兵。当从重要性的角度考虑隔离的时候，就和熔断降级非常接近了

隔离 - Case Stduy

- 早期转码集群被超大视频攻击，导致转码大量延迟。
- 缩略图服务，被大图实时缩略吃完所有 CPU，导致正常的小图缩略被丢弃，大量 503。
- 数据库实例 cgroup 未隔离，导致大 SQL 引起的集体故障。
- INFO 日志量过大，导致异常 ERROR 日志采集延迟。

为什么要有超时

分析：这又是一个主打出乎预料的题目。就是大家天然会觉得超时很好超时很棒，但是为什么要超时？

超时还要考虑真超时和伪超时的问题。所谓真超时，就是指当超时发生的时候，会中断当前的动作，并且后面的动作都不会发生。伪超时是指，要么是无法中断当前的动作，要么是无法中断后续动作。

超时总体来说是一种保护机制，同时保护客户端和服务端的机制。对于客户端（进程内就是调用方）来说，它总是可以在期望的时间内得到响应。

但是在分布式系统里面，“总是可以在期望的时间内得到响应”更加像是一种美好的希望，因为如果客户端不自己维持超时计数（即客户端并不依赖与网络超时机制，也不依赖于服务端超时机制，而是自己维系了时钟，在超时发生后自己直接断开链接），你永远也不知道什么时候才能拿到响应。

而中断后续动作对于两端来说，都能减少资源消耗。因为客户端已经不管你了，即便执行下去得到结果，也已经没有人需要这个结果了。也就是毛老师说的快速失败

但是某些时候我们会期望即便超时了，它也能继续执行完所有的后续事情，而不是中断在某一步。而后通过重试和幂等机制，保证业务的唯一性。因此超时在这里就和重试、幂等关联上来了

为什么要有超时

答案：超时是一种保护机制，对于客户端来说，它可以期望在一定的时间内得到响应，虽然这个响应可能仅仅是一个超时信号，用户体验会好很多（为什么好很多？因为用户宁愿你告诉他失败了，也不想一直看到页面啥反应也没有）。

而对于服务端来说，更加重要的是，服务端可以在判断超时之后，直接丢弃当前请求。这在某些场景下，能够极大减少错误发生。比如说整个负载太高，那么超时会保证一部分请求在前面的步骤就被中断掉，不再需要消耗后续的资源。

（刷亮点1，链路超时控制和设置超时时间的简单原则）要想做到在整条链路的超时控制，往往意味着，需要在各个端之间传递超时信息，每一个服务端收到请求之后，都要重新计算剩余时间。因此在设置下游的超时时间的时候，也要兼顾考虑上游的超时时间。例如如果上游调用自己的服务，超时时间是1s，那么自己调用下游的超时时间就肯定要小于1s，不然毫无意义。

（刷亮点2，非中断式的超时控制）要么有些框架不支持链路超时控制，要么是我们业务希望超时之后服务端依旧继续按照正常流程处理请求，这种时候超时控制就仅仅对客户端有意义了。往往这种做法是要伴随重试、幂等，或者轮询机制，以便客户端有机会知道真正的执行情况。

（刷亮点3，讨论如何中断）一般来说，中断正在执行的动作是比较难的，即便 go 依托于 context 在超时控制上比较有优势，想真正执行一个正在运行中的方法，还是做不到的。不过已经比其他语言强很多了。而中断后续动作，则很多时候依赖于框架，框架不支持，业务层面上支持，做起来也很艰难。（这里要小心，面试官可能让你手写一个能够中断的代码，参考上节课毛老师 context 和 channel 的部分，或者网上自己背下来一个例子）

为什么要有超时

相关问题

- 如何用 context.Context 来实现超时控制？

- 如何实现链路超时控制？就是依托于 RPC 框架在端之间传递超时时间（进程内就是context传递），每次服务端收到请求、准备调用下游的时候，都要重新计算剩余超时时间。

- 超时了怎么办？还能怎么办，重试。一聊到重试就要聊幂等，然后就会聊到幂等怎么做。

- 超时了说明什么？啥也说明不了，服务端可能还在跑，也可能没在跑，也可能压根没收到你的请求。但是频繁收到超时，就要么是网络崩了，要么服务崩了。

如何设置合理的超时时间

分析：这算是一个理论与实践兼备的问题。一般如果在面试过程中聊到了超时，是比较容易被问到。这个问题从根本上来说，应该是从**用户体验的角度**去看待的。举例来说，为了用户体验，APP 的开屏首页的响应时间，一般是要求在 50ms 以内。这是最强的约束，直接导致后面的微服务接口响应时间基本上要控制在十几ms内。

但是这属于政治正确的废话。大多时候，超时时间的确定是依据你下游的接口的 SLO 来确定的。比如说，使用90线，99线。如果90到99，响应时间大增，那么可能考虑90，如果相差不多，那么99就比较好。

我们的亮点是落在讨论新接口的超时时间。新接口缺乏线上运行数据。理论上的做法是压测来确定99线。但是压测比较难做，也很难模拟真实的线上环境。因此另外一个可选的方案就是预估。

预估一般是基于代码来预估的，考虑代码中发起的数据库操作，RPC 调用，或者和远程第三方组件打交道。

例如一个接口如果操作了两次数据库，而公司内数据库操作的平均时间是7ms，那么你可以预期，这个接口响应时间的平均值是在15ms左右——1ms基本上足够完成各种计算了。那么保守取值到20ms，就会比较合适了。

如何设置合理的超时时间

答：从根本上来说，是从用户体验的角度来确定整个链路的超时时间，而后按照这个值来预分配给这条链路上的接口。

或者说，如果公司有响应时间的硬性指标，那么就根据指标来设置。比如说大多数时候，面向C端的接口，其超时时间都不能超过 100ms， 那么把 100 ms设置为超时时间就是很合理的。

但是大多数时候产品对这方面没有要求，那么我们的超时时间就是根据下游的 SLO 来确定，比如说用 99 线。

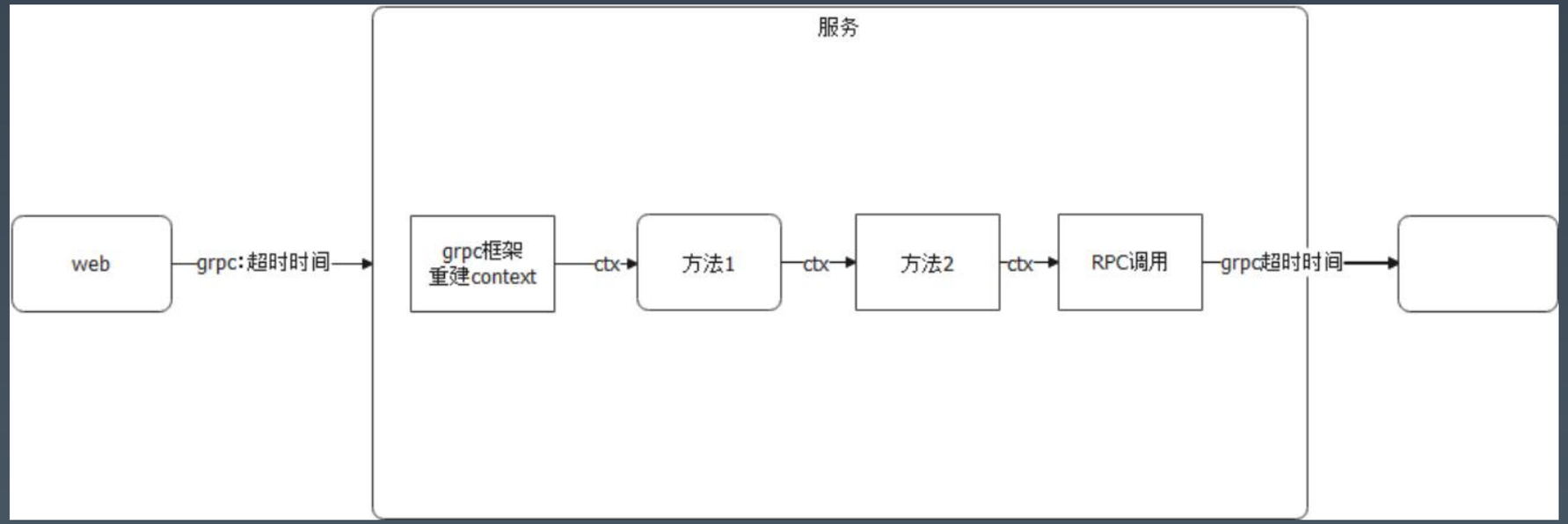
（要考虑新接口）如果调用的接口是新接口，缺乏线上性能信息，也无法模拟线上环境做压测，那么可以根据代码来进行评估，主要是计算代码中的数据库操作、RPC 调用或者和第三方框架打交道的次数来预估，并且出于保守可以在这几预估基础上加一点。后面在拿到了线上运行数据之后再来设置

如何实现链路超时控制

分析：这个问题与其说是考察超时，不如说是考察分布式链路构建。所以我们可以先回答超时是怎么做的，后面引申到利用类似机制实现的功能。

链路构建的核心就是传递 context，进程内传递和跨进程传递。进程内传递就是在所有的方法接口里面加上`context.Context`参数；跨进程传递，如果底层是 HTTP，那么基本上就是放在 Header 里面。如果是基于 TCP 直接传输，则是放在二进制流的某个部分。

难点就在于，使用了不同的框架，有些时候需要手动设置。比如说一个微服务内部，又使用了gRPC，还访问 Redis，gRPC 只能解决自己的超时，还需要自己手动设置访问 Redis 的超时



如何实现链路超时控制

回答：链路超时控制，关键在于在链路之间传递超时信息，而超时信息基本上都是放在上下文中传递。在 Go 里面就是用 context 传递。包含进程内 context 传递和跨端传递 context。

具体步骤是：首先将超时时间塞进去 context，而后 context 在进程内传递，直到我们准备发起远程调用，比如说 HTTP 或者 RPC 调用。这一类的框架，会帮我们处理超时，它们会读取 context 里面超时时间，而后将超时时间传递到服务端，服务端的框架会重建这个 context，之后重复该过程。

（开始刷亮点）但是这种机制的难点就是，跨框架超时控制。比如说我服务同时使用 HTTP，gRPC和访问 Redis，那么初始设置超时时间的时候需要为三者都设置一遍。如果我们本身是一个 gRPC 服务，那么我们还需要从 gRPC 里面读出来原始的超时时间，然后为 Redis 设置超时时间。

所以普遍上我们都会为超时控制单独封装一下 context，使用统一的 key，然后为各个框架重新封装一遍设置超时时间和重建context的过程。

超时 - Case Stduy

- SLB 入口 Nginx 没配置超时导致连锁故障。
- 服务依赖的 DB 连接池漏配超时，导致请求阻塞，最终服务集体 OOM。
- 下游服务发版耗时增加，而上游服务配置超时过短，导致上游请求失败。

限流有哪些算法？

分析：其实在毛老师的内容组织里面，这部分被划过去了过载保护里面，但是依据我的一般性经验，大多数时候，面试官会认为这是限流的算法（应当说过载保护会被认为是限流）。要注意这一点，等面试官问到限流的时候就可以回答这些算法。

基本回答就是罗列算法。罗列算法也是有技巧的，一般来说，不建议聊算法的具体步骤，但是可以建议多聊一点算法的优劣，以及某些参数对算法的影响。这是因为，步骤是属于纯粹彻底的八股文，背就可以。而自己对算法的分析，除了背，还得有自己的理解（这里就是你们拥有了毛老师的理解，再加点自己的理解）

刷亮点可以落到三个方面：BBR，即动态限流的那个算法、均匀性问题，以及与 TCP 流量控制的。要注意，如果不能说清楚 BBR 算法，就不要刷这个亮点，因为大概率面试官会接着问 BBR。这里这里面，毛老师在课程并没有直接给你们讲 BBR，但是他课程里面讨论的就是类似的主题。

均匀性则是为了体现自己的思考，它讨论的是尖峰问题。比如说我设置限流的阈值是1000请求每分钟。但是好巧不巧，第一秒就来了一千请求，那么服务器也还是得崩。

讨论限流和 TCP 流量控制的相似点是一个更加罕见的角度。其实如果我们把请求对标到报文，那么两者要做的事情就是同一回事了。

限流有哪些算法？

答：（首先从大类开始分，这种回答还是不太常见的，属于从高到底回答）限流算法大体上可以分为两大类，静态限流算法和动态限流算法。

静态限流算法是指什么时候触发阈值，是事先配置好的，是静态的，典型的就是固定窗口，滑动窗口，令牌桶和漏桶。（简明扼要分析优缺点）它们都是事先指定好阈值，到了阈值就触发限流，并未实际考虑应用此时的真实状态。因此其难点就在于阈值难确定（这是为了引导面试官，如何确定阈值），而优点则在于，算法简单稳定可预期。

动态算法，是指那种实时评估当下应用状态的，然后确定是否要触发限流。典型的就是 BBR 的算法。（这里不要聊细节）它的特点就是近实时检测，本质上还是根据过去表现预测未来表现。它的有点就是自适应动态调整，缺点就是实现复杂，额外开销比较多。

（开始刷亮点，就是开始评价这些算法）限流算法里面，很重要的一个点是要考虑均匀性的问题，也就是，峰值流量过来，我是应该限制住，还是放开。举例来说，窗口算法和令牌桶算法，都会把峰值放过去。比如说固定窗口一秒内处理100请求，但是可能在0.01秒内，100个全来了，而实际上，这个峰值会导致大部分请求的响应时间突然增加。而漏桶这种均匀的，也就是你0.01秒过来100个请求，不好意思我这么一点时间只有一个令牌漏出来，也就是其余99个都会拒掉。（进一步解释）不过大多数时候我们也不太在意这个东西，只要把窗口设置小了，多半就没什么问题。窗口太大，比如说一分钟内1000个请求，结果一秒钟就来了1000个，那么服务器就会直接崩溃。

限流有哪些算法？

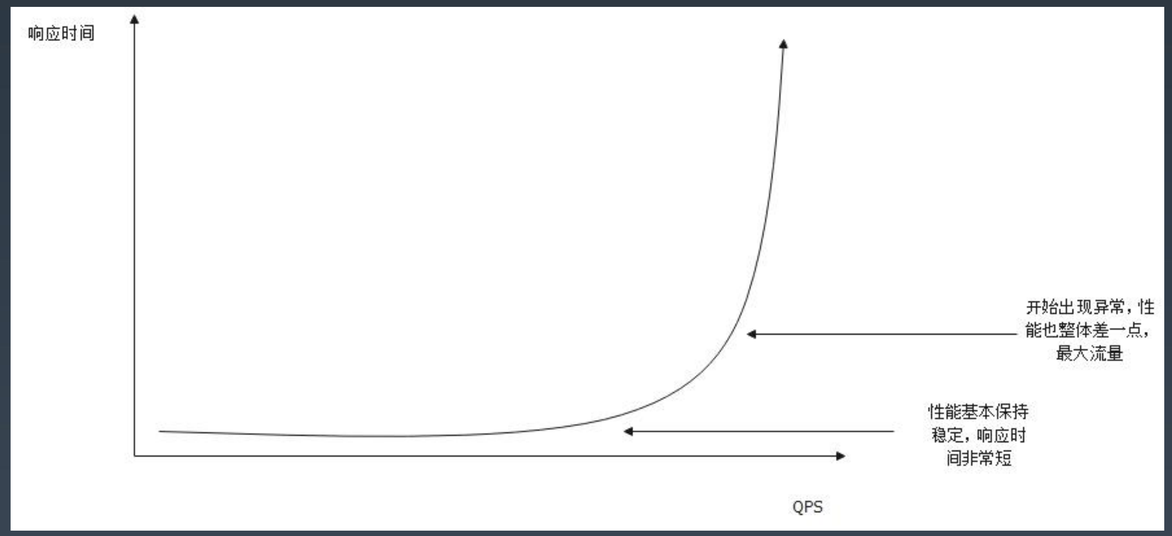
限流总体上和 TCP 的流量控制有非常多类似的点，比如说 BBR 算法其实最开始就是被设计为 TCP 流量控制的。后来微服务借鉴了这个算法。因为从本质上来说，如果我们把请求等同于报文，那么 TCP 流量控制和微服务流量控制，那就基本上是一回事。所不同的是，TCP 不需要考虑集群的问题，而微服务还需要讨论整体的集群问题。

限流有哪些算法？

类似问题：

- 怎么选择限流算法？大多数时候就是考虑要不要处理峰值流量，框架支不支持动态限流算法
- 限流算法各有什么优缺点？
- 如何评价这些限流算法？
- 你能手写 XXX 算法吗？所以最好是大概了解这些算法的实现，手写的时候不必强求写对，大概意思到了就可以——如果有很长时间准备面试，就可以练习到写正确
- 令牌桶和漏桶的区别？
- 固定窗口和滑动窗口的区别？
- BBR 算法是怎么实现的？这个不强求，不是做中间件的话，直接认怂都可以
- 为什么要限流？保护系统

如何确定限流的阈值



分析：这也是一个理论和实践都很有意义的问题。基本上，这里有两个选择：使用性能最好的值和使用最大流量的值。

什么意思呢，就是如果我们对一个接口做压测，会发现它的图形是类似于上图。根据自己的需要，挑一个节点。

有些时候公司会有性能要求，那么就在图形上找到响应时间对应的 QPS 来选择。

万一要是没有压测，根本得不出性能数据怎么办？只能是预估，参考类似业务的 QPS 来设置。

如何确定限流的阈值

答：最好的做法就是压测，然后根据压测结果来确定限流的阈值。（大概描述一下前面的图）压测的结果，如果按照 QPS 和响应时间来构成一个坐标系，那么曲线大概是随着QPS增长，首先会有一段平滑的曲线，而后在逐渐接近性能瓶颈的时候，响应时间开始急剧增长。 QPS 继续增长之后，达到某一个值，就会开始出现报错。如果这个时候 QPS 继续增长，那么错误比率和响应时间都急剧上升。

因此可以考虑选两个点：第一个点是在响应时间开始上升之前的那个点，作为限流的阈值，保证最好的用户体验；第二个点是在出错之前的那个点，保证最大的吞吐量。

某些时候，如果公司对响应时间有规定，那么可以考虑采用该响应时间对应的 QPS 作为阈值。

（讨论没有压测的情况）如果没有压测结果来参考，那么基本上就只能依赖于个人对业务的理解来判断阈值了。

首先可以考虑的是根据一般情况下的流量来设置阈值。比如说公司业务的峰值流量是 200 QPS，而且没啥错误，那么直接设置为 200 QPS 就可以。

其次可以参考相关业务的阈值。比如说两个功能接近的接口，或者说场景相同的接口，那么它们的阈值应该也是接近的。

怎么做限流？

分析：这算是一个非常大问题。从这里我给出一般的回答思路。限流可以限流对象、限流算法，拒绝策略和恢复策略四个角度讨论。

限流对象，即针对什么来限流。可以是集群限流，单机限流。也可以是针对业务对象的限流，比如说限制普通用户的访问频率和流量大小，或者针对 IP 限流来做一些安全保证。

限流算法之前我们已经讨论过了，到时候只需要稍微列举一下就可以。

拒绝策略则是，触发限流了怎么办？大多数时候就是直接丢弃请求，也可以回答毛老师说的，丢给另外一个backup集群，backup集群不行了就只能丢了（这算是一个很大的亮点，一般的公司根本不会考虑这么做）。又或者如果不是实时请求，可以考虑缓存起来，返回给调用方 Accepted 的状态，后面慢慢处理。

恢复策略其实毛老师课程也讨论了很多，主要就是要避免抖动问题。也就是恢复不能立刻恢复，而是要有一个逐步恢复的过程。比较多的就是等待+试探。即触发阈值之后，维持当前状态一段时间，而后在恢复的时候尝试放一点流量过去，如果正常处理了。

怎么做限流？

类型	优点	缺点	现有实现
单机限制	1. 实现简单 2. 稳定可靠 3. 性能高	1. 流量不均匀会引发误限制 2. 机器数变化时 配额要手动调整 容易出错	各个语言都有相应的实现 如golang的令牌桶: golang/x/time/ratelimit
动态流控	根据服务情况 动态限流 不用调整额度	1. 需要主动搜集请求的性能数据 (cpu load 成功率 耗时) 2. 客户端主动善意限流 3. 一般只限于接口调用 支持的范围小 应用场景狭窄	BBR限流 广义上各种连接池也属于这类
全局限流	1. 流量不均不会误触发限流 2. 机器数变动时候无需调整 3. 应用场景丰富 接口 DB 等任何资源都可以使用	1. 实现较复杂 2. 需要手动配置	无

怎么做限流？

答：做限流需要综合的考虑。

首先要考虑限流的对象。主要考虑是需要对整个集群进行限流，还是针对单机限流。另外在一些特殊的业务场景下，需要针对业务对象限流，比如说针对普通用户进行限流。或者出于安全考虑，针对 IP 进行限流。

特别的，如果是在集群维度上限流，那么主要依赖于网关，或者 Redis 这种中间件。

而从算法上来说，有固定窗口、滑动窗口、令牌桶、漏桶和 BBR这种动态限流算法。选择算法的时候要考虑，是否需要处理峰值流量，是否能够准确预估阈值。

而在触发阈值之后，则要考虑处理措施，或者说拒绝策略。大部分情况下，触发阈值代表服务已经到达了处理能力的上限，因此直接丢弃请求，或者返回简单的默认值，都是合理的。某些时候，我们如果有backup集群，则可以转发请求到 backup 集群。也可以考虑返回客户端 Accepted 状态，后面自己慢慢处理。

当触发限流之后，要考虑恢复的问题，主要是要避免抖动的问题。一般做法是保持限流状态一小段时间，而后尝试放部分流量过去查看是否能够正常拿到响应。如果可以再继续加流量，直到完全恢复正常。

怎么做限流？

类似问题：

- 你怎么保护你的服务不会被压垮？
- 集群限流和单机限流，有什么区别？

怎么做降级？

分析：差不多就是问降级该怎么处理。降级可以从两方面聊：

1. 提供有损服务

2. 服务间停掉不重要的服务

所以问题就归结为：有损，我该损到什么地步；我怎么识别这些服务的重要性。

第一点其实比较简单，也就是我们常说兜底。大体上就是，我们认为业务有所谓的快路径和慢路径。或者说，可以非常复杂，也可以非常简单。典型的就是所谓的首页。在正常情况下我们是走个性化推荐，但是显然这个个性化推荐是一个很重要的逻辑，因此我们可以考虑在降级的时候退化到走比较简单的逻辑，比如说就是返回运营预先配置的某些数据（也可能是事先计算好的某些兜底数据）。如果要是还是不行，那么就直接退化到某个静态页面。

很多时候降级的麻烦之处在于要预先设置好这些兜底的数据。

其次就是不同服务之间降级，即我优先保证高业务价值的接口正常运作，其它我就可以暂时关闭。比如说万一服务器崩了，我可以先保证 VIP 用户的服务，普通用户就随便了。又或者我可以直接把退款之类的反向流程关掉，资源用于确保下单之类的更有价值的服务。

怎么做降级？

答：降级大体上有两种做法：

1. 在单一服务内部，可以将业务逻辑降级为简单的做法。最典型的做法就是可能只是返回一个默认值而已。（用自己公司的例子，没有的话就用这个或者毛老师的例子）比如说网站首页，APP 首屏。正常是走个性化推荐，如果个性化推荐崩了，就是返回预先计算好的数据。这些数据可能是依赖于大数据分析每天计算一次（这个频率是看个人需要的），如果还是不行，则直接切换到静态页面（前端或者说BFF那里就直接切走了）。该做法要求我们要事先准备好各种兜底措施。

2. 在不同服务之间，把不重要的服务停掉，把资源腾出来。该做法依赖于全局的服务重要性的配置，让服务治理的中间件知道在必要的时候可以关掉哪些服务，全力保障核心服务。最为典型的例子，就是某些平台会在大促前夕直接把退款之类的反向功能关掉，等过去了大促之后再打开。

怎么做降级？

类似问题：

- 怎么判断业务的优先级？看业务价值
- 为什么要做降级？有损服务总比没服务好，部分不可用好过全部不可用
- 降级之后怎么恢复？还是要小心抖动，基本上就是试探，加大流量，完全恢复

降级 - Case Study

- 客户端解析协议失败，app 奔溃。
- 客户端部分协议不兼容，导致页面失败。
- local cache 数据源缓存，发版失效 + 依赖接口故障，引起的白屏。
- 没有 playbook，导致的 MTTR 上升。

怎么重试？

分析：重试这个话题，属于简单回答可以很简单，复杂回答也可以吹一波的东西，全靠个人感悟和公司基础设施。

比如说，简单的重试就是直接重试 N 次，要么成功，要么打印日志。稍微改进一点，就是重试 N 次之后也没有成功，那么直接告警，人手工介入。

再复杂一点，就是要考虑重试的时机。比如说失败了立刻重试，如果是因为网络之类的硬件类，或者远程组件类引起的失败，那么立刻重试多半还是失败。于是比较好的做法就是我间隔一段时间重试。于是就要考虑间隔的时间怎么设置，是固定，还是按照某个规律增加重试间隔。

再复杂一点就是考虑重试万一这期间应用挂掉了，我重启之后还会不会把之前重试的东西重建起来，继续进行下去。如果能，又要考虑，会不会我每次应用启动，立刻触发一大波重试，导致我这个应用立刻就崩掉。

同时还要考虑，我这些重试数据存在哪里。存在内存，宕机就没了。如果存在数据库或者 Redis，那么我能不能用别的实例来调度重试？

所以很多时候，分布式的重试，就跟分布式任务调度密切相关。基本上可以认为，分布式环境下重试，就相等于创建一个分布式定时执行任务。

最后我们讨论一下重试风暴。也就是因为某个东西故障引起所有大部分请求重试，每一次重试调用好几个接口，这些接口也失败，最终引起整个集群都被重试的流量给占据了。

怎么重试？

答：重试属于简单好用，但是也要小心使用的机制。

简单的重试，就是直接进程内重试，也不需要等待。大部分框架，如果要访问网络，它们的重试机制就是立刻重试。

但是，某些时候，立刻重试几乎可以肯定百分百会失败。比如说因为网络问题引起的失败，再次重试的时候，网络没有恢复，依旧失败。这个时候就要考虑间隔一段时间重试。如果是间隔一段时间重试，可以是固定间隔，比如说每分钟重试一次；也可以是动态计算的重试。很常见的策略是，我失败了立刻重试一次，如果还失败，我就等一分钟再重试；如果依旧失败，我可以考虑等十分钟再重试一次；还是失败，我就要考虑再等一小时。但是如果还是失败，我就按照每小时重试一次的频率尝试下去。如果一直失败，最终就是放弃或者告警，需要人工介入。

（从这一部分开始，就属于刷亮点部分了）而如果我们业务要求很高，那么重试就要考虑宕机的问题，特别是在引入了间隔重试机制之后。一般来说，比较好的做法就是结合分布式任务系统，重试就相当于创建一个新的定时任务。在失败之后再次创建一个任务，直到放弃或者人工介入。但是要防止因为挤压大量了重试任务，导致某次重启之后应用直接被重试流量淹没的问题。

（更加高级的讨论）即便如此，也要考虑重试风暴的问题。比如说某个下游失败引起了重试，连锁反应导致它的上游，上上游...都开始重试，很容易就导致整个集群资源都被重试占据了。

（更进一步的罕见场景）有些时候，我们可能为了保证服务的可用性，会禁止掉重试。这种有点类似熔断的场景。

怎么重试？

类似问题

- 失败了立刻重试吗？看情况
- 怎么设置重试的时机？
- 怎么防止宕机丢失重试请求？
- 如果重试了依旧失败怎么办？凉拌，只能人处理了。它其实相当于不断重试将失败率降低到一个非常非常低的比例，但是最终最终肯定是人手工介入。
- 重试怎么保证幂等？这其实属于幂等的话题，大多数时候都是依赖于唯一索引，或者唯一标识符去重

重试 - Case Study

- Nginx upstream retry 过大，导致服务雪崩。
- 业务不幂等，导致的重试，数据重复。
  - *全局唯一 ID: 根据业务生成一个全局唯一 ID，在调用接口时会传入该 ID，接口提供方会从相应的存储系统比如 redis 中去检索这个全局 ID 是否存在，如果存在则说明该操作已经执行过了，将拒绝本次服务请求；否则将相应该服务请求并将全局 ID 存入存储系统中,之后包含相同业务 ID 参数的请求将被拒绝。*
  - *去重表: 这种方法适用于在业务中有唯一标识的插入场景。比如在支付场景中，一个订单只会支付一次，可以建立一张去重表,将订单 ID 作为唯一索引。把支付并且写入支付单据到去重表放入一个事务中了，这样当出现重复支付时，数据库就会抛出唯一约束异常,操作就会回滚。这样保证了订单只会被支付一次。*
  - *多版本并发控制: 适合对更新请求作幂等性控制,比如要更新商品的名字，这是就可以在更新的接口中增加一个版本号来做幂等性控制。*
- 多层级重试传递，放大流量引起雪崩。

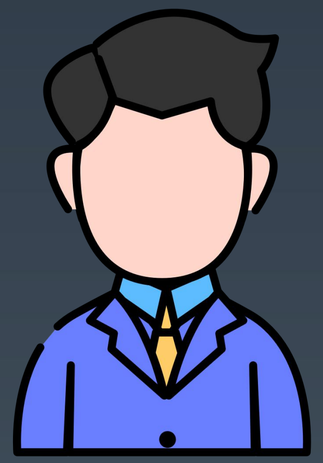
最佳实践

- 变更管理:
  - *70%的问题是由变更引起的，恢复可用代码并不总是坏事。*
- 避免过载:
  - *过载保护、流量调度等。*
- 依赖管理:
  - *任何依赖都可能故障，做 chaos monkey testing，注入故障测试。*
- 优雅降级:
  - *有损服务，避免核心链路依赖故障。*
- 重试退避:
  - *退让算法，冻结时间，API retry detail 控制策略。*
- 超时控制:
  - *进程内 + 服务间 超时控制。*
- 极限压测 + 故障演练。
- 扩容 + 重启 + 消除有害流量。

服务治理三部曲

- 故障检测
- 故障处理
- 故障恢复

服务治理三部曲——故障检测



我怎么知道，服务出问题了？

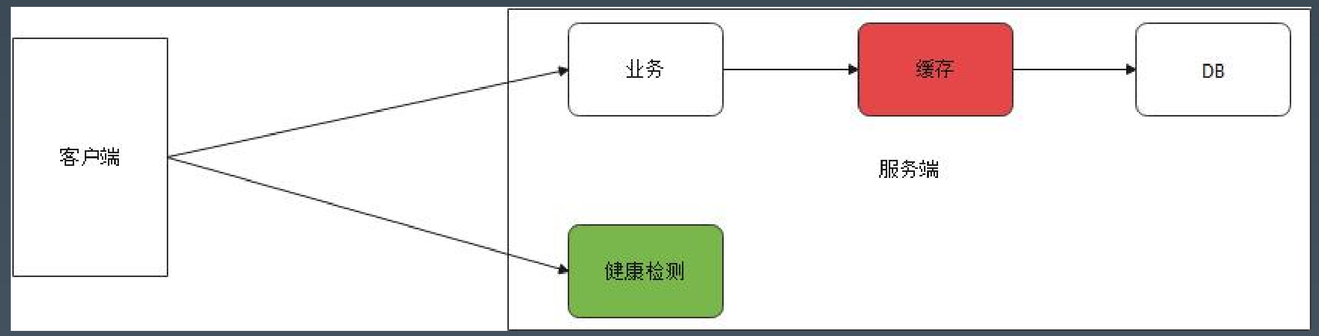
服务治理三部曲——故障检测

- 我看到了 5xx 之类的错误码，或者看到了错误响应
- 我看到了错误日志
- 我迟迟拿不到响应
- ……

- 这个过程自动化，就是故障检测手段

服务治理三部曲——故障检测

- 健康检测：最为朴素的检测手段。
- 健康检测究竟检测的是什么？我健康检测通过了，能否证明我的服务的就是好的？
  - 健康检测要注意是什么维度的检测：你使用的框架层面，还是你业务层面？
  - 健康检测大多数时候只能代表你应用正常运行了，但是你服务可能崩了
  - 健康检测甚至只能代表你们网络连通了
- 我该多久发起一次健康检测？



服务治理三部曲——故障检测

- 硬件指标：不管你实际业务怎样，我只看你的硬件指标
  - CPU
  - 磁盘 IO
  - 网络流量
  - 连接数

服务治理三部曲——故障检测

- 服务指标：检查服务的运行情况。因为服务往往很复杂，所以我们不可能发一个请求过去试试能不能得到期望的响应。所以检测服务指标，实际上是统计一段时间内的服务数据，包括：
  - 响应时间
  - 错误数量
  - ...

服务治理三部曲——统一模型

$$health = f(x_1, \ldots, x_n)$$

也就是我们用 0-100 来描述服务是否健康，100 是满分，0 就是彻底崩溃。

x 就是影响健康度的各种指标。或者说，我们能够用来评估的因素。那么前面讨论的硬件指标、服务指标都可以作为一个因子。它们不是独立变量，互相之间是有影响的。

- 如果我们只考虑连接数，那么就退化为根据连接数推断服务状态的算法。例如负载均衡里面的最少连接数算法
- 如果我们只考虑当前处理的请求数，那么就会退化为轮询、最少请求数之类的算法
- 如果我们只考虑响应时间，那么就会退化为最快响应算法
- 如果我们综合考虑，并且实时检测，那么典型的就是 BBR
- 如果我们综合考虑，但是完全不实时检测，那么就是预先设置阈值一类的限流、熔断算法

服务治理三部曲——统一模型

$$health = f(x_1, \ldots, x_n)$$

基于这个模型，我们只需要列举所有觉得可能会影响这个 health 值的指标，然后将不同的算法一个个对应过去。

目前来说，业界之前有人考虑过用大数据来分析哪些指标是真实有用的，影响很大的。不过没下文。

服务治理三部曲——统一模型

$$health = f(x_1, \ldots, x_n)$$

$$impact_i = g(t, x_i)$$

毛老师课程上，多次讨论到采集到的指标，越新就越有价值。这可以看做是一个和时间有关的函数 g。

这个 g 满足单调递减的性质。猜测大多数时候应该是满足一种指数衰减的规律（不确定……缺乏相关论文数据支撑）

它所体现的是一种时间局部性原理。

服务治理三部曲——故障处理

过载保护、超时控制、熔断、限流、降级、重试都可以看做是故障处理的一种手段。

也就是说我们在检测到故障发生了（或者将要发生），而后采取措施，依据目的不同，被分成了这几个。本质上可以理解为他们都是故障处理的手段。

过载保护、限流、熔断和降级其实界限并不是那么清晰。比如说当我检测到故障之后，我返回默认值，那么算是限流？熔断？还是降级？

服务治理三部曲——故障恢复

故障恢复，是最不被我们重视的步骤，因而可用的方案非常的少。

1. 不管不顾，到了我就放流量，我可以一点点放，也可以一次性全部放掉

2. 我尝试放流量，稍微放一点，看看是否可行。可行我再加点。可以参考 TCP 的快开始慢恢复策略

服务治理五部曲

- 故障检测
- 故障预防
- 故障处理
- 故障转移
- 故障恢复