流量治理

- **前提**
  - **容错性设计：**是微服务的另一个核心原则，由于服务随时都有可能崩溃，因此快速的失败检测和自动恢复就显得至关重要。
  - **拆分服务越来越多会遇到的问题**
    - **雪崩效应：**由于某一个服务的崩溃，导致所有用到这个服务的其他服务都无法正常工作，一个点的错误经过层层传递，最终波及到调用链上与此有关的所有服务。
    - **突发流量：**服务没有崩溃，但由于处理能力有限，面临超过预期的突发请求时，大部分请求直至超时都无法完成处理。这种现象产生的后果跟交通堵塞是类似的，如果一开始没有得到及时的治理，后面就需要长时间才能使全部服务都恢复正常。
- **服务容错**
  - **容错策略**
    - **调用失败如何弥补**
      - **故障转移：**是指如果调用的服务器出现故障，系统不会立即向调用者返回失败结果，而是自动切换到其他服务副本，尝试能否返回成功调用的结果，保证整体的高可用性。故障转移应该有调用次数限制，超过次数报错依旧返回调用失败。因为重试是有执行成本的，且过度的重试反而可能让系统处于更加不利的状况。
        - **优点：**系统自动处理，调用者对失败的信息不可见
        - **缺陷：**增加调用时间，额外的资源开销
        - **应用场景：**调用幂等服务，对调用时间不敏感的场景
      - **快速失败：**尽快让服务报错，坚决避免重试，尽快抛出异常，由调用者自行处理。故障转移的前提是要求服务具备幂等性，对于非幂等的服务，重复调用就可能产生脏数据，引起的麻烦远大于服务调用直接失败。比如银行扣款错误，很难判断是扣款指令发送给银行时出现的网络异常，还是扣款后返回结果给服务时的网络异常。
        - **优点：**调用者有对失败的处理完全控制权，不依赖服务的幂等性
        - **缺陷：**调用者必须正确处理失败逻辑，如果一味只是对外抛异常，容易引起雪崩
        - **应用场景：**调用非幂等的服务，超时阈值较低的场景
      - **安全失败：**在一个调用链路中的服务通常有主路和旁路之分，有部分服务失败了也不影响核心业务的正确性。即使旁路逻辑调用实际失败了，也当作正确来返回，也可以返回一个符合要求的零值，然后自动记录一条服务调用出错的日志备查。
        - **优点：**不影响主路逻辑
        - **缺陷：**只适用于旁路调用
        - **应用场景：**调用链中的旁路服务
      - **故障恢复：**一般作为其他容错策略的补充措施，如果设置容错策略为故障恢复的话，通常默认会采用快速失败加上故障恢复的策略组合。当服务调用出错了以后，将该次调用失败的信息存入一个消息队列中，然后由系统自动开始异步重试调用。
        - 尽力促使失败的调用最终能够被正常执行，也为服务注册中心和负载均衡器及时提供服务恢复的通知信息。要求服务必须具备幂等性，由于重试是后台异步进行，一般用于对实时性要求不高的主路逻辑，也适合处理不需要返回值的旁路逻辑。为了避免内存中异步调用任务堆积，与故障转移一样，应该有最大重试次数的限制。
        - **优点：**调用失败后自动重试，也不影响主路逻辑
        - **缺陷：**重试任务可能产生堆积，重试仍然可能失败
        - **应用场景：**调用链中的旁路服务，对实时性要求不高的主路逻辑也可以使用
      - **沉默失败：**如果大量的请求需要等到超时或者长时间处理后才宣告失败，很容易由于请求堆积而消耗大量的线程、内存、网络等资源，进而影响到整个系统的稳定。此时合理的失败策略是当请求失败后，就默认服务提供者一定时间内无法再对外提供服务，不向它分配请求流量，将错误隔离开来，避免对系统其他部分产生影响。
        - **优点：**控制错误不影响全局
        - **缺陷：**出错的地方将在一段时间内不可用
        - **应用场景：**频繁超时的服务
    - **调用之前如何获得最大的成功概率**
      - **并行调用：**一开始就同时向多个服务副本发起调用，只要有其中任何一个返回成功，便宣告成功。在关键场景中使用更高的执行成本换取执行时间和成功概率。
        - **优点：**尽可能在最短时间内获得最高的成功率
        - **缺陷：**额外消耗机器资源，大部分调用可能都是无用功
        - **应用场景：**资源充足且对失败容忍度低的场景
      - **广播调用：**同时发起多个调用，广播调用要求所有的请求全部都成功，通常会被用于实现“刷新分布式缓存”这类的操作。
        - **优点：**支持同时对批量的服务提供者发起调用
        - **缺陷：**资源消耗大，失败概率高
        - **应用场景：**只适用于批量操作的场景
  - **容错设计模式**

- **断路器模式**
  - **基本思路：**通过断路器对象与远程服务一一对应接管服务调用者的远程请求，持续监控并统计服务返回的各种结果，当故障次数达到断路器的阈值时，断路器打开，后续代理的远程访问都将直接返回调用失败。通过断路器对远程服务的熔断，避免因持续的失败而消耗资源或持续超时而堆积请求，避免雪崩效应的出现。
  - **状态转变为OPEN**
    - 一段时间内请求数量达到一定阈值。这个条件的意思是如果请求本身就很少，那就用不着断路器介入。
    - 一段时间内请求的故障率到达一定阈值。这个条件的意思是如果请求本身都能正确返回，也用不着断路器介入。
  - **服务熔断与服务降级**
    - 服务熔断是一种快速失败的容错策略的实现方法。在明确反馈了故障信息给上游服务以后，上游服务必须能够主动处理调用失败的后果，而不是坐视故障扩散，这里的“处理”指的就是一种典型的服务降级逻辑，降级逻辑可以包括，但不应该仅仅限于是把异常信息抛到用户界面去，而应该尽力想办法通过其他路径解决问题。
    - 服务降级不一定是在出现错误后才被动执行的，许多场景里面降级更可能是指需要主动迫使服务进入降级逻辑的情况。如应对可预见的峰值流量，或者是系统检修等原因，关闭系统部分功能或部分旁路服务，此时有可能会主动迫使这些服务降级。
- **舱壁隔离模式**
  - **定义：**是常用的实现服务隔离的设计模式，符合容错策略中沉默失败策略
  - **调用外部服务故障**
    - **失败：**如400 Bad Request、500 Internal Server Error
    - **拒绝：**如 401 Unauthorized、403 Forbidden
    - **超时：**如 408 Request Timeout、504 Gateway Timeout
  - **超时带来全局性风险：**由于目前主流的网络访问大多是基于Thread per Request并发模型来实现的，只要请求不结束就一直占用着线程不能释放。而线程是典型的全局性资源，尤其是 Java 这类将线程映射为操作系统内核线程来实现的语言，为了不让某一个远程服务的局部失败演变成全局性的影响，就必须引入服务隔离。
  - **使用局部的线程池来控制服务的最大连接数：**为每个服务单独设立线程池，默认不预置活动线程。服务的超时故障就只会最多阻塞最大连接数的线程。
    - **优点：**当服务出问题时能够隔离影响，当服务恢复后，还可以通过清理掉局部线程池，瞬间恢复该服务的调用。
    - **缺陷：**额外增加了 CPU 的开销，每个独立的线程池都要进行排队、调度和下文切换工作
  - **更轻量的可以用来控制服务最大连接数办法：**信号量机制。如果不考虑清理线程池、客户端主动中断线程这些额外的功能，可以只为每个远程服务维护一个线程安全的计数器即可，并不需要建立局部线程池。
  - 舱壁隔离模式还可以在更高层、更宏观的场景中使用，不是按调用线程，而是按功能、按子系统、按用户类型等条件来隔离资源都是可以的，譬如，根据用户等级、用户是否 VIP、用户来访的地域等各种因素，将请求分流到独立的服务实例去。
- **重试模式**
  - **使用场景：**适合解决系统中的瞬时故障，即有可能自愈的临时性失灵，网络抖动、服务临时过载如503 Bad Gateway 错误等。实现故障转移和故障恢复容错策略。
  - **使用的前提条件**
    - **仅在主路逻辑的关键服务上进行同步的重试**，不是关键的服务，一般不把重试作为首选容错方案，尤其不该进行同步重试。
    - **仅对由瞬时故障导致的失败进行重试**。关于故障是否属于可自愈的瞬时故障可从 HTTP 的状态码上获得一些初步的结论。
    - **仅对具备幂等性的服务进行重试**。判断服务是否幂可以找到一些总体上通用的原则。如RESTful 服务中的 POST 请求是非幂等的，而 GET、PUT、HEAD、OPTIONS、TRACE 由于不会改变资源状态，这些请求应该被设计成幂等的。
    - **重试必须有明确的终止条件**，如超时终止或次数终止。
  - **风险：**由于重试模式可以在网络链路的多个环节中去实现，如客户端调用的、网关中的、负载均衡器中的自动重试等，若不注意可能导致重试次数达到指数级别。

流量控制

- **涉及的问题**
  - **限流的依据：**要不要控制流量，要控制哪些流量，控制力度要有多大，等等这些操作都没法在系统设计阶段静态地给出确定的结论，必须根据系统此前一段时间的运行状况，甚至未来一段时间的预测情况来动态决定。
  - **具体如何限流：**解决系统具体是如何做到允许一部分请求能够通行，而另外一部分流量实行受控制的失败降级，必须了解掌握常用的服务限流算法和设计模式。
  - **超额流量如何处理：**有不同的处理策略，可以直接返回失败如 429 Too Many Requests，或者被迫使它们进入降级逻辑（否决式限流）。也可能让请求排队等待，暂时阻塞一段时间后继续处理（阻塞式限流）。
- **流量统计指标**
  - **每秒事务数Transactions per Second：**TPS 是衡量信息系统吞吐量的最终标准。事务可以理解为一个逻辑上具备原子性的业务操作。
  - **每秒请求数Hits per Second：**HPS 是指每秒从客户端发向服务端的请求数。如果只要一个请求就能完成一笔业务，那 HPS 与 TPS 是等价的，但在一些场景（尤其常见于网页中）里，一笔业务可能需要多次请求才能完成。
  - **每秒查询数Queries per Second：**QPS 是指一台服务器能够响应的查询次数。如果只有一台服务器来应答请求，那 QPS 和 HPS 是等价的，但在分布式系统中，一个请求的响应往往要由后台多个服务节点共同协作来完成。
  - 整体上最希望能够基于 TPS 来限流，但真实业务操作的耗时受限于用户交互带来的不确定性。目前主流系统大多倾向使用 HPS 作为首选的限流指标，它是相对容易观察统计的，而且能够在一定程度上反应系统当前以及接下来一段时间的压力。
- **限流设计模式**
  - **流量计数器模式**
    - **思路：**设置计算器，根据当前时刻的流量计数结果是否超过阈值来决定是否限流。
    - **每一秒的统计流量都没有超过阈值不能说明系统没有遇到超过阈值的流量压力**。如果系统连续两秒都收到 60 TPS 的访问请求，但分别是前 1 秒里面的后 0.5 秒，以及后 1 秒中的前面 0.5 秒所发生的。虽然每个周期的流量都不超过 80 TPS 请求的阈值，但是系统确实曾

经在 1 秒内实在在发生了超过阈值的 120 TPS 请求。

- **连续若干秒统计流量都超过了阈值也不能说明流量压力就超过了系统承受能力**。如果 10 秒中，前 3 秒 TPS为100，而后 7 秒为30，此时系统仍然能够处理完这些请求。超时时间是 10 秒，而最慢的请求也能在 8 秒左右处理完毕。此时限制阈值，反而会误杀一部分请求，造成部分请求出现原本不必要的失败。
- **缺陷：**只是针对时间点进行离散的统计。

- **滑动时间窗模式**
  - **思路：**与流量计数器模式区别，实现平滑的基于时间片段统计。在向前流淌的时间轴上，漂浮着固定大小的窗口。任何时刻静态地通过窗口内观察到的信息，都等价于一段长度与窗口大小相等、动态流动中时间片段的信息。
  - **工作过程**
    - 1、将数组最后一位的元素丢弃掉，并把所有元素都后移一位，然后在数组第一个插入一个新的空元素。这个步骤即为“滑动窗口”。
    - 2、将计数器中所有统计信息写入到第一位的空元素中。
    - 3、对数组中所有元素进行统计，并复位清空计数器数据供下一个统计周期使用。
  - **优点：**滑动时间窗口模式的限流完全解决了流量计数器的缺陷，可以保证任意时间片段内，只需经过简单的调用计数比较，就能控制住请求次数一定不会超过限流的阈值，在单机限流或者分布式服务单点网关中的限流中很常用。
  - **缺陷：**通常只适用于否决式限流，超过阈值的流量就必须强制失败或降级，很难进行阻塞等待处理，也就很难在细粒度上对流量曲线进行整形，无法削峰填谷。
- **流量整型-漏桶模式**
  - **思路：**其实就是一个以请求对象作为元素的先入先出队列，队列长度就相当于漏桶的大小，当队列已满时便拒绝新的请求进入。
  - **缺陷：**困难在于如何确定漏桶的两个参数：桶的大小和水的流出速率。如果桶设置得太大，那服务依然可能遭遇到流量过大的冲击，不能完全发挥限流的作用；如果设置得太小，那很可能就会误杀掉一部分正常的请求。
- **流量整型-令牌桶模式**
  - **思路：**与漏桶模式刚好相反，前者是从水池里往系统出水，后者是系统往排队机中放入令牌。
  - **工作过程**
    - 1、让系统以一个由限流目标决定的速率向桶中注入令牌，譬如要控制系统的访问不超过100 次/秒，速率即设定为 100 个令牌/秒，每个令牌注入间隔为10 毫秒。
    - 2、桶中最多可以存放 N 个令牌，N 的具体数量是由超时时间和服务处理能力共同决定的。如果桶已满，第 N+1 个进入的令牌会被丢弃掉。
    - 3、请求到时先从桶中取走 1 个令牌，如果桶已空就进入降级逻辑。
  - **实现：**不需要真的用一个专用线程或者定时器来每隔一段时间放入令牌，只要在令牌中增加一个时间戳记录，每次获取令牌前，比较一下时间戳与当前时间，就可以轻易计算出这段时间需要放多少令牌进去，然后一次性放入即可。

- **分布式限流**
  - **背景：**前面讨论过的那些限流算法，直接使用在单体架构的集群上是完全可行的，但到了微服务架构下，它们就最多只能应用于集群最入口处的网关上，对整个服务集群进行流量控制，而无法细粒度地管理流量在内部微服务节点中的流转情况。
  - **与单机限流差别：**核心差别在于如何管理限流的统计指标，单机限流指标存储在服务内存当中，而分布式限流无论是将限流功能封装为专门的远程服务，抑或是在系统采用的分布式框架中有专门的限流支持，都需要将原本在每个服务节点自己内存当中的统计数据给开放出来，让全局的限流服务可以访问到才行。
  - **一种分布式限流方法是将所有服务的统计结果都存入集中式缓存如 Redis中**，以实现在集群内的共享，并通过分布式锁、信号量等机制解决读写访问时的并发控制。此时单机限流模式也可以应用于分布式环境中。代价是每次服务调用都必须要额外增加一次网络开销，流量压力大时反倒会显著降低系统的处理能力。
  - **缓解集中式缓存性能损耗：**在令牌桶限流模式基础上把令牌看作数值形式的货币额度。当请求进入集群时，首先在 API 网关处领取到一定数额的货币，访问每个服务时都要求消耗一定量的货币。
    - **限流指标：**把剩余额度作为内部限流的指标，只要一旦剩余额度≤0 时，就不再允许访问其他服务了。此时必须先发生一次网络请求，重新向令牌桶申请一次额度，成功后才能继续访问，不成功则进入降级逻辑。
    - **缺陷：**基于额度的限流方案对限流的精确度有一定的影响，可能存在业务操作已经进行了一部分服务调用，却无法从令牌桶中再获取到新额度导致业务操作失败，白白浪费了部分已经完成了的服务资源。但总体来说，它仍是一种并发性能和限流效果上都相对折衷可行的分布式限流方案。