二

31 如何应对接口级的故障?

你好，我是华仔。

前几讲我介绍了异地多活方案。它主要用来应对系统级的故障，例如机器宕机、机房故障和网络故障等问题。这些系统级的故障虽然影响很大，但发生概率较小。在实际业务运行过程中，还有另外一种故障影响可能没有那么大，但发生的概率较高，这就是今天我要跟你聊的接口级的故障。

接口级故障的典型表现就是，系统并没有宕机、网络也没有中断，但业务却出现问题了，例如业务响应缓慢、大量访问超时和大量访问出现异常（给用户弹出提示“无法连接数据库”）。

这类问题的主要原因在于系统压力太大、负载太高，导致无法快速处理业务请求，由此引发更多的后续问题。最常见的情况就是，数据库慢查询将数据库的服务器资源耗尽，导致读写超时，业务读写数据库时要么无法连接数据库、要么超时，最终用户看到的现象就是访问很慢，一会儿访问抛出异常，一会儿访问又是正常结果。

如果进一步探究，导致接口级故障的原因可以分为两大类：

内部原因：包括程序 bug 导致死循环，某个接口导致数据库慢查询，程序逻辑不完善导致耗尽内存等。

外部原因：包括黑客攻击，促销或者抢购引入了超出平时几倍甚至几十倍的用户，第三方系统大量请求，第三方系统响应缓慢等。

解决接口级故障的核心思想和异地多活基本类似，都是**优先保证核心业务**和**优先保证绝大部分用户**。常见的应对方法有四种，降级、熔断、限流和排队，下面我会一一讲解。

1. 降级

降级指系统将某些业务或者接口的功能降低，可以是只提供部分功能，也可以是完全停掉所有功能。

例如，论坛可以降级为只能看帖子，不能发帖子；也可以降级为只能看帖子和评论，不能发评论；而 App 的日志上传接口，可以完全停掉一段时间，这段时间内 App 都不能上传日志。

降级的核心思想就是丢车保帅，优先保证核心业务。

例如，对于论坛来说，90% 的流量是看帖子，那我们就优先保证看帖的功能；对于一个 App 来说，日志上传接口只是一个辅助的功能，故障时完全可以停掉。

常见的实现降级的方式有两种：

1.1 系统后门降级

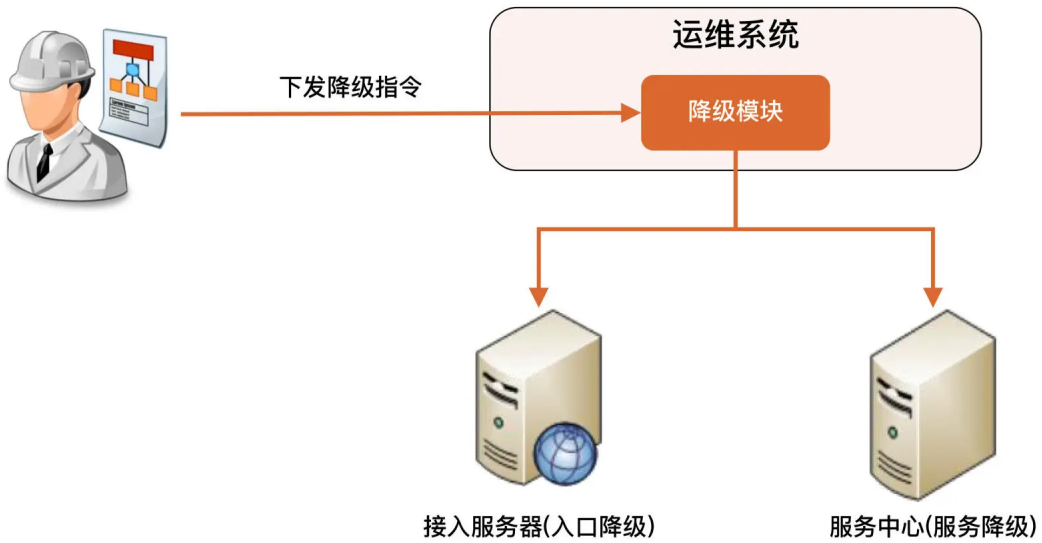
简单来说，就是系统预留了后门用于降级操作。例如，系统提供一个降级 URL，当访问这个 URL 时，就相当于执行降级指令，具体的降级指令通过 URL 的参数传入即可。这种方案有一定的安全隐患，所以也会在 URL 中加入密码这类安全措施。

系统后门降级的方式实现成本低，但主要缺点是如果服务器数量多，需要一台一台去操作，效率比较低，这在故障处理争分夺秒的场景下是比较浪费时间的。

1.2 独立降级系统

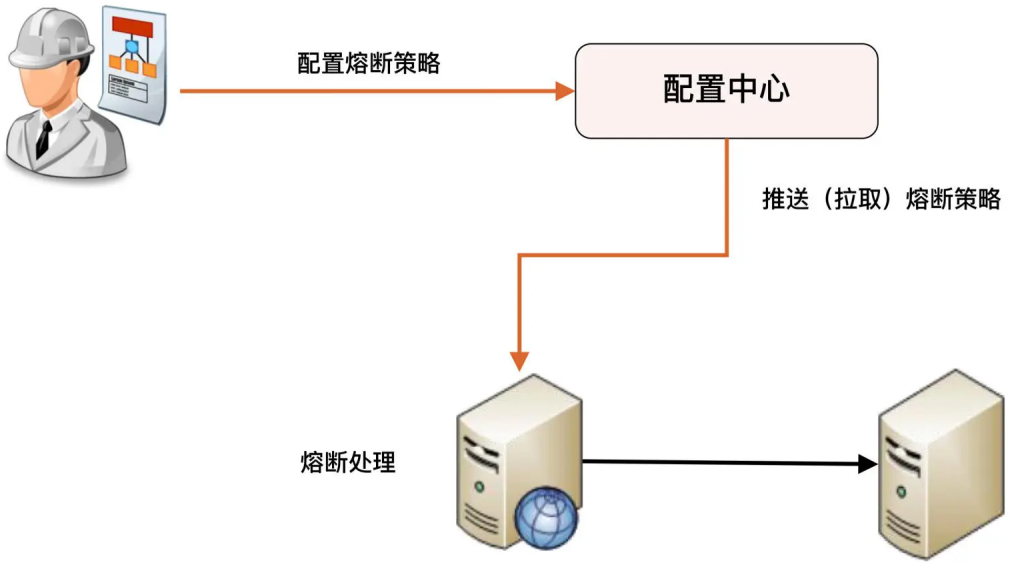
为了解决系统后门降级方式的缺点，我们可以将降级操作独立到一个单独的系统中，实现复杂的权限管理、批量操作等功能。

其基本架构如下：



2. 熔断

熔断是指按照规则停掉外部接口的访问，防止某些外部接口故障导致自己的系统处理能力急剧下降或者出故障。



熔断和降级是两个比较容易混淆的概念，因为单纯从名字上看，好像都有禁止某个功能的意思。但它们的内涵是不同的，因为降级的目的是应对系统自身的故障，而熔断的目的是应对依赖的外部系统故障的情况。

假设一个这样的场景：A 服务的 X 功能依赖 B 服务的某个接口，当 B 服务的接口响应很慢的时候，A 服务的 X 功能响应肯定也会被拖慢，进一步导致 A 服务的线程都被卡在 X 功能处理上，于是 A 服务的其他功能都会被卡住或者响应非常慢。

这时就需要熔断机制了：A 服务不再请求 B 服务的这个接口，A 服务内部只要发现是请求 B 服务的这个接口就立即返回错误，从而避免 A 服务整个被拖慢甚至拖死。

实现熔断机制有两个关键点：

一是需要有一个**统一的 API 调用层**，由 API 调用层来进行采样或者统计。如果接口调用散落在代码各处，就没法进行统一处理了。

二是**阈值的设计**，例如 1 分钟内 30% 的请求响应时间超过 1 秒就熔断，这个策略中的“1 分钟”“30%”“1 秒”都对最终的熔断效果有影响。实践中，一般都是先根据分析确定阈值，

然后上线观察效果，再进行调优。

3. 限流

降级是从系统功能优先级的角度考虑如何应对故障，而限流则是从用户访问压力的角度来考虑如何应对故障。限流指只允许系统能够承受的访问量进来，超出系统访问能力的请求将被丢弃。

虽然“丢弃”这个词听起来让人不太舒服，但保证一部分请求能够正常响应，总比全部请求都不能响应要好得多。

限流一般都是系统内实现的，常见的限流方式可以分为两类：基于请求限流和基于资源限流。

3.1 基于请求限流

基于请求限流指从外部访问的请求角度考虑限流，常见的方式有两种。

第一种是限制总量，也就是限制**某个指标的累积上限**，常见的是限制当前系统服务的用户总量，例如：某个直播间限制总用户数上限为 100 万，超过 100 万后新的用户无法进入；某个抢购活动商品数量只有 100 个，限制参与抢购的用户上限为 1 万个，1 万以后的用户直接拒绝。

第二种是限制时间量，也就是限制**一段时间内某个指标的上限**，例如 1 分钟内只允许 10000 个用户访问；每秒请求峰值最高为 10 万。

无论是限制总量还是限制时间量，共同的特点都是实现简单，但在实践中面临的主要问题是比较难以找到合适的阈值。例如系统设定了 1 分钟 10000 个用户，但实际上 6000 个用户的时候系统就扛不住了；或者达到 1 分钟 10000 用户后，其实系统压力还不大，但此时已经开始丢弃用户访问了。

即使找到了合适的阈值，基于请求限流还面临硬件相关的问题。例如一台 32 核的机器和 64 核的机器处理能力差别很大，阈值是不同的，可能有的技术人员以为简单根据硬件指标进行数学运算就可以得出来，实际上这样是不可行的，64 核的机器比 32 核的机器，业务处理性能并不是 2 倍的关系，可能是 1.5 倍，甚至可能是 1.1 倍。

为了找到合理的阈值，通常情况下可以采用性能压测来确定阈值，但性能压测也存在覆盖场景有限的问题，可能出现某个性能压测没有覆盖的功能导致系统压力很大；另外一种方式是逐步优化：先设定一个阈值然后上线观察运行情况，发现不合理就调整阈值。

基于上述的分析，根据阈值来限制访问量的方式更多的适应于业务功能比较简单的系统，例如负载均衡系统、网关系统、抢购系统等。

3.2 基于资源限流

基于请求限流是从系统外部考虑的，而基于资源限流是从系统内部考虑的，也就是找到系统内部影响性能的关键资源，对其使用上限进行限制。常见的内部资源包括连接数、文件句柄、线程数和请求队列等。

例如，采用 Netty 来实现服务器，每个进来的请求都先放入一个队列，业务线程再从队列读取请求进行处理，队列长度最大值为 10000，队列满了就拒绝后面的请求；也可以根据 CPU 的负载或者占用率进行限流，当 CPU 的占用率超过 80% 的时候就开始拒绝新的请求。

基于资源限流相比基于请求限流能够更加有效地反映当前系统的压力，但实际设计时也面临两个主要的难点：如何确定关键资源，以及如何确定关键资源的阈值。

通常情况下，这也是一个逐步调优的过程：设计的时候先根据推断选择某个关键资源和阈值，然后测试验证，再上线观察，如果发现不合理，再进行优化。

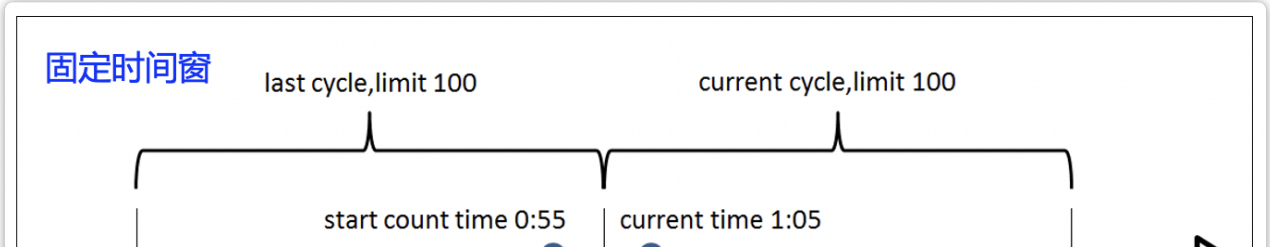
限流算法

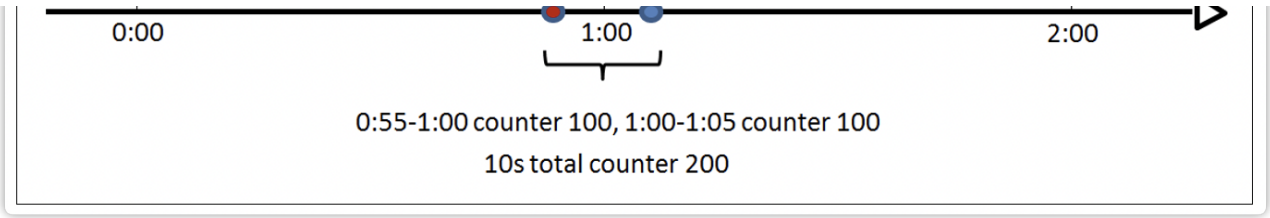
为了更好地实现前面描述的各种限流方式，通常情况下我们会基于限流算法来设计方案。常见的限流算法有两大类四小类，它们的实现原理和优缺点各不相同，在实际设计的时候需要根据业务场景来选择。

（1）时间窗

第一大类是时间窗算法，它会限制一定时间窗口内的请求量或者资源消耗量，根据实现方式又可以细分为“固定时间窗”和“滑动时间窗”。

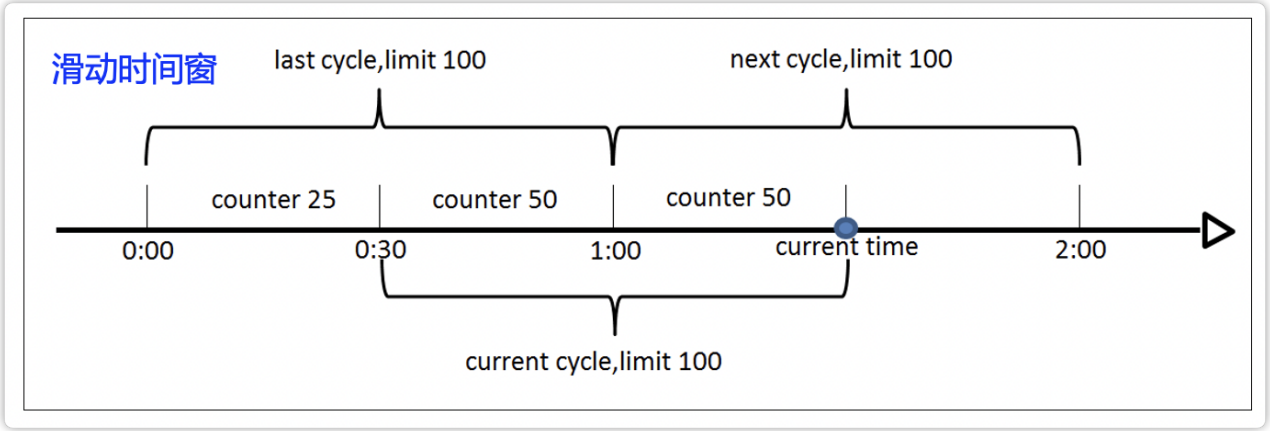
固定时间窗算法的实现原理是，统计固定时间周期内的请求量或者资源消耗量，超过限额就会启动限流，如下图所示：





它的优点是实现简单，缺点是存在**临界点**问题。例如上图中的红蓝两点只间隔了短短 10 秒，期间的请求数却已经达到 200，超过了算法规定的限额（1 分钟内处理 100）。但是因为这些请求分别来自两个统计窗口，从单个窗口来看还没有超出限额，所以并不会启动限流，结果可能导致系统因为压力过大而挂掉。

为了解决临界点问题，滑动时间窗算法应运而生，它的实现原理是，两个统计周期部分重叠，从而避免短时间内的两个统计点分属不同的时间窗的情况，如下图所示：

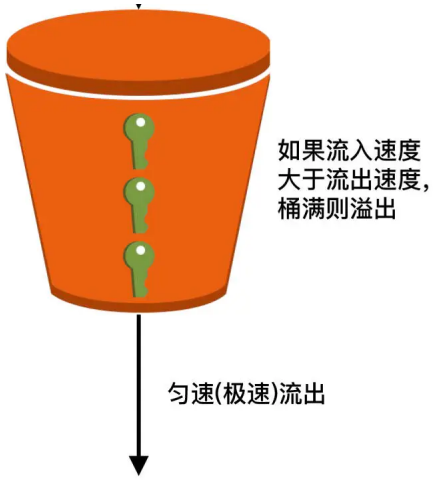


总体上来看，滑动时间窗的限流效果要比固定时间窗更好，但是实现也会稍微复杂一些。

（2）桶算法

第二大类是桶算法，用一个虚拟的“桶”来临时存储一些东西。根据桶里面放的东西，又可以细分为“漏桶”和“令牌桶”。

漏桶算法的实现原理是，将请求放入“桶”（消息队列等），业务处理单元（线程、进程和应用等）从桶里拿请求处理，桶满则丢弃新的请求，如下图所示：



我们可以看到漏桶算法的三个关键实现点：

流入速率不固定：可能瞬间流入非常多的请求，例如 0 点签到、整点秒杀。

匀速 (极速) 流出：这是理解漏桶算法的关键，也就是说即使大量请求进入了漏桶，但是从漏桶流出的速度是匀速的，速度的最大值就是系统的极限处理速度（对应图中的“极速”）。这样就保证了系统在收到海量请求的时候不被压垮，这是第一层的保护措施。需要注意的是：如果漏桶没有堆积，那么流出速度就等于流入速度，这个时候流出速度就不是匀速的。

桶满则丢弃请求：这是第二层保护措施，也就是说漏桶不是无限容量，而是有限容量，例如漏桶最多存储 100 万个请求，桶满了则直接丢弃后面的请求。

漏桶算法的技术本质是**总量控制**，桶大小是设计关键，具体的优缺点如下：

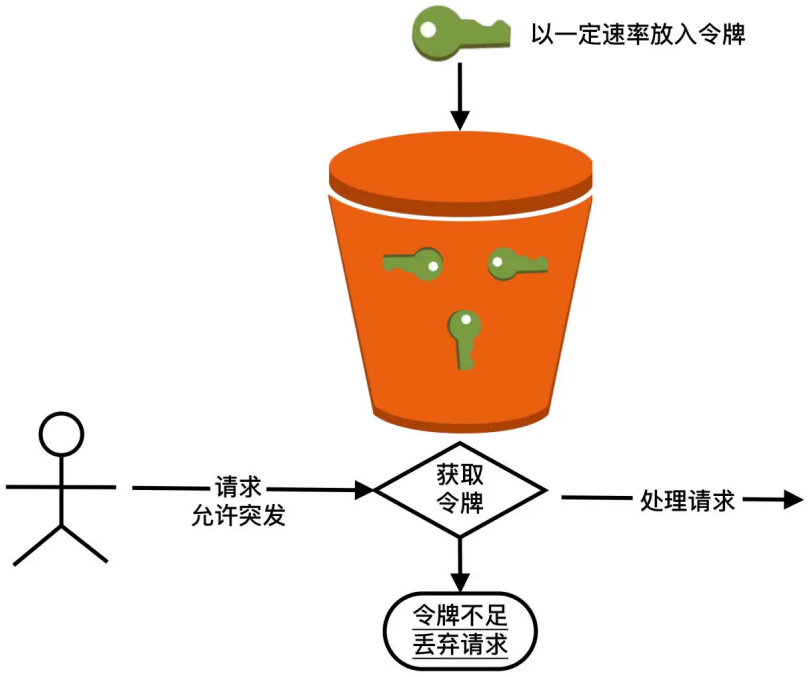
突发大量流量时丢弃的请求较少，因为漏桶本身有缓存请求的作用。

桶大小动态调整比较困难（例如 Java BlockingQueue），需要不断的尝试才能找到符合业务需求的最佳桶大小。

漏桶算法主要适用于瞬时高并发流量的场景（例如刚才提到的 0 点签到、整点秒杀等）。在短短几分钟内涌入大量请求时，为了更好的业务效果和用户体验，即使处理慢一些，也要做到尽量不丢弃用户请求。

令牌桶算法和漏桶算法的不同之处在于，桶中放入的不是请求，而是“令牌”，这个令牌就是业务处理前需要拿到的“许可证”。也就是说，当系统收到一个请求时，先要到令牌桶里面拿“令牌”，拿到令牌才能进一步处理，拿不到就要丢弃请求。

它的实现原理是如下图所示：



我们可以看到令牌桶算法的三个关键设计点：

有一个处理单元往桶里面放令牌，放的速率是可以控制的。

桶里面可以累积一定数量的令牌，当突发流量过来的时候，因为桶里面有累积的令牌，此时的业务处理速度会超过令牌放入的速度。

令牌桶算法的技术本质是**速率控制**，令牌产生的速率是设计关键，具体的优缺点如下：

突发大量流量的时候可能丢弃很多请求，因为令牌桶不能累积太多令牌。

令牌桶算法主要适用于两种典型的场景，一种是需要控制访问第三方服务的速度，防止把下游压垮，例如支付宝需要控制访问银行接口的速率；另一种是需要控制自己的处理速度，防止过载，例如压测结果显示系统最大处理 TPS 是 100，那么就可以用令牌桶来限制最大的处理速度。

刚才介绍漏桶算法的时候我提到漏桶算法可以应对**瞬时高并发**流量，现在介绍令牌桶算法的时候，我又说令牌桶允许**突发**流量。

你可能会问，这两种说法好像差不多啊，它们到底有什么区别，到底谁更适合做秒杀呢？

其实，令牌桶的“允许突发”实际上只是“允许一定程度的突发”，比如系统处理能力是每秒 100 TPS，突发到 120 TPS 是可以的，但如果突发到 1000 TPS 的话，系统大概率就被

压垮了。所以处理秒杀时高并发流量，还是得用漏桶算法。

令牌桶的算法原本是用于网络设备控制传输速度的，而且它控制的目的是保证一段时间内的平均传输速度。之所以说令牌桶适合突发流量，是指在网络传输的时候，可以允许某段时间内（一般就几秒）超过平均传输速率，这在网络环境下常见的情况就是“网络抖动”。

但这个短时间的突发流量并不会导致雪崩效应，网络设备也能够处理得过来。对应到令牌桶应用到业务处理的场景，就要求即使有突发流量来了，系统自己或者下游系统要真的能够处理的过来，否则令牌桶允许突发流量进来，结果系统或者下游处理不了，那还是会被压垮。

因此，令牌桶在实际设计的时候，桶大小不能像漏桶那样设计很大，需要根据系统的处理能力来进行仔细的估算。例如，漏桶算法的桶容量可以设计为 100 万，但是一个每秒 30 TPS 的令牌桶，桶的容量可能只能设计成 40 左右。海外有的银行给移动钱包提供的接口 TPS 上限是 30，压测到了 40 就真的挂了。

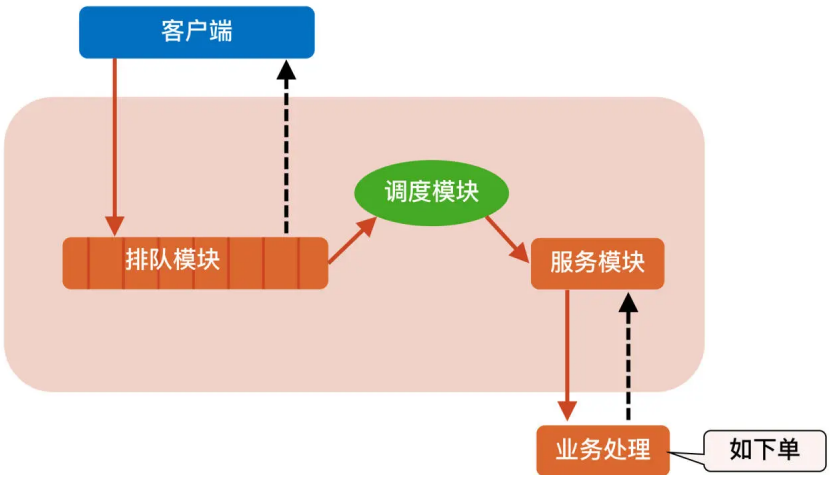
4. 排队

排队实际上是限流的一个变种，限流是直接拒绝用户，排队是让用户等待一段时间，全世界最有名的排队当属 12306 网站排队了。

排队虽然没有直接拒绝用户，但用户等了很长时间后进入系统，体验并不一定比限流好。

由于排队需要临时缓存大量的业务请求，单个系统内部无法缓存这么多数据，一般情况下，排队需要用独立的系统去实现，例如使用 Kafka 这类消息队列来缓存用户请求。

下图是 1 号店的“双 11”秒杀排队系统架构：



它的基本实现摘录如下：

【排队模块】

负责接收用户的抢购请求，将请求以先入先出的方式保存下来。每一个参加秒杀活动的商品保存一个队列，队列的大小可以根据参与秒杀的商品数量（或加点余量）自行定义。

【调度模块】

负责排队模块到服务模块的动态调度，不断检查服务模块，一旦处理能力有空闲，就从排队队列头上把用户访问请求调入服务模块，并负责向服务模块分发请求。这里调度模块扮演一个中介的角色，但不只是传递请求而已，它还担负着调节系统处理能力的重任。我们可以根据服务模块的实际处理能力，动态调节向排队系统拉取请求的速度。

【服务模块】

负责调用真正业务来处理服务，并返回处理结果，调用排队模块的接口回写业务处理结果。

小结

今天我为你讲了接口级故障的四种应对方法，分别是降级、熔断、限流和排队，希望对你有所帮助。

方法	实现原理	场景	案例
降级	停掉故障接口，收到接口请求后直接返回错误，避免不重要的接口故障后产生链式反应，导致系统整体故障	应对系统自身的接口故障	论坛可以降级为只能看帖子，不能发帖子 淘宝双十一的时候将“领淘金币”降级
熔断	按照规则停掉外部接口的访问，防止某些外部接口故障导致自己的系统处理能力急剧下降或者出故障	依赖的外部系统的接口故障	支付宝双十一访问某银行的支付接口，一分钟内失败率超过设定的阈值，接下来10分钟内不再访问此接口，10分钟后再试
限流	只允许系统能够承受的访问量进来，超出系统访问能力的请求将被丢弃，防止系统被压垮	应对访问压力过大的情况	0点签到、整点秒杀 支付宝控制访问银行接口的速率
排队	尽量接收用户的请求，但是没有立即处理，而是先将用户请求缓存起来，需要用户等待一段时间才会真正开始处理	应对用户访问压力过大的情况，相比限流可以有更好的用户体验，但实现更复杂	12306购票 1号店双十一秒杀

这就是今天的全部内容，留一道思考题给你吧，如果你来设计一个整点限量秒杀系统，包括登录、抢购、支付（依赖支付宝）等功能，你会如何设计接口级的故障应对手段？

上一页 下一页