**限流、熔断、降级**

1 限流

限流解决了什么问题

在开发高并发系统时有三把利器用来保护系统：**缓存、降级和限流。缓存的目的是提升系统访问速度和增大系统能处理的容量，可谓是抗高并发流量的银弹**；而降级是当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开；而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页），因此需有一种手段来限制这些场景的并发/请求量，即限流。

限流的目的是通过对并发访问/请求进行限速或者一个时间窗口内的的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或告知资源没有了）、排队或等待（比如秒杀、评论、下单）、降级（返回兜底数据或默认数据，如商品详情页库存默认有货）。

**限流采用的算法-漏桶算法**

漏桶(Leaky Bucket)算法思路很简单,水(请求)先进入到漏桶里,漏桶以一定的速度出水(接口有响应速率),当水流入速度过大会直接溢出(访问频率超过接口响应速率),然后就拒绝请求,可以看出漏桶算法能强行限制数据的传输速率.示意图如下:

**限流采用的算法-令牌算法**

令牌桶算法(Token Bucket)和 Leaky Bucket 效果一样但方向相反的算法,更加容易理解.随着时间流逝,系统会按恒定1/QPS时间间隔(如果QPS=100,则间隔是10ms)往桶里加入Token(想象和漏洞漏水相反,有个水龙头在不断的加水),如果桶已经满了就不再加了.新请求来临时,会各自拿走一个Token,如果没有Token可拿了就阻塞或者拒绝服务.

令牌桶的另外一个好处是可以方便的改变速度. 一旦需要提高速率,则按需提高放入桶中的令牌的速率. 一般会定时(比如100毫秒)往桶中增加一定数量的令牌, 有些变种算法则实时的计算应该增加的令牌的数量.

### 漏桶算法与令牌桶算法的区别

令牌桶算法：以固定的速率(平均速率)生成对应的令牌放到桶中，客户端只需要在桶中获取到令牌后，就可以访问服务请求。

漏桶算法：以任意速率往桶中放入水滴，如果桶中的水滴没有满的话，可以访问服务。

在突发情况请求的时候，令牌桶中只需要客户端你能够拿到令牌就能访问服务。但是漏桶算法正好与令牌桶算法相反，令牌桶以平均速率访问，漏桶算法平滑访问。

### 1.漏桶限流算法的原理

   以固定速率从桶中流出水滴，以任意速率往桶中放入水滴，桶容量大小是不会发生改变的。

   流入：以**任意速率**往桶中放入水滴。

   流出：以**固定速率**从桶中流出水滴。

   水滴：是唯一不重复的标识。

   因为桶中的容量是固定的，如果流入水滴的速率>流出的水滴速率，桶中的水滴可能会溢出。那么溢出的水滴请求都是拒绝访问的，或者直接调用服务降级方法。前提是同一时刻。

### 2.令牌桶算法(Token)

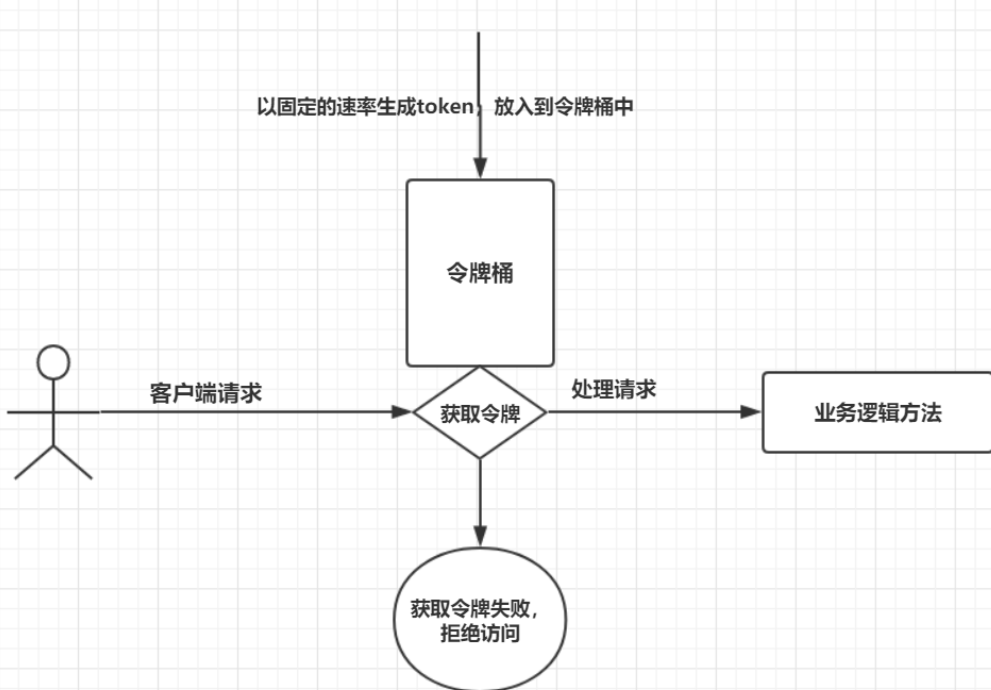
   令牌桶分为2个动作，动作1(固定速率往桶中存入令牌)、动作2(客户端如果想访问请求，先从桶中获取token)。guava 提供的RateLimiter类来进行限流处理。

     1.传统的方式整合RateLimiter 有很大的缺点：代码重复量特别大，而且本身不支持注解方式。

     2.如果限流代码可以放在网关中，相当于针对所有的服务接口都实现限流(可以使用排除法进行排除不进行限流的方法)，维护性不是很强。

     3.正常的互联网公司项目，不是所有的服务接口都需要实现限流方法的，一般只真针对于大流量接口。比如：秒杀抢购、12306抢票等。

     4.可以手动封装一个RateLimiter类 注解来解决这个方法。



以规定的速率往令牌桶中放入 token，用户请求必须获取到令牌桶中的 token才可以访问我们的业务逻辑方法，如果没有从令牌桶中获取到 token ，拒绝访问。

在高并发情况下，如果我们的请求过多 超出了令牌桶生成令牌的速度，这时候请求就会被驳回，提示请稍后重试！

优势：能够控制请求的速率。

**限流的目的：**为了保护服务，避免服务宕机。



### 漏桶算法与令牌桶算法的区别

令牌桶算法：以固定的速率(平均速率)生成对应的令牌放到桶中，客户端只需要在桶中获取到令牌后，就可以访问服务请求。

漏桶算法：以任意速率往桶中放入水滴，如果桶中的水滴没有满的话，可以访问服务。

在突发情况请求的时候，令牌桶中只需要客户端你能够拿到令牌就能访问服务。但是漏桶算法正好与令牌桶算法相反，令牌桶以平均速率访问，漏桶算法平滑访问。

**限流采用的算法-计数器算法**

计数器算法是限流算法里最简单也是最容易实现的一种算法。比如我们规定，对于A接口来说，我们1分钟的访问次数不能超过100个。那么我们可以这么做：在一开 始的时候，我们可以设置一个计数器counter，每当一个请求过来的时候，counter就加1，如果counter的值大于100并且该请求与第一个 请求的间隔时间还在1分钟之内，那么说明请求数过多；如果该请求与第一个请求的间隔时间大于1分钟，且counter的值还在限流范围内，那么就重置 counter

NGNIX限流

**Nginx主要有两种限流方式：**

（1）按连接数限流(ngx\_http\_limit\_conn\_module)（令牌算法实现）

（2）按请求速率限流(ngx\_http\_limit\_req\_module)（漏桶算法实现）

ngx\_http\_limit\_req\_module

模块提供限制请求处理速率能力，使用了漏桶算法(leaky bucket)。下面例子使用 nginx limit\_req\_zone 和 limit\_req 两个指令，限制单个IP的请求处理速率。在 nginx.conf http 中添加限流配置：

**Tomcat 线程池限流**

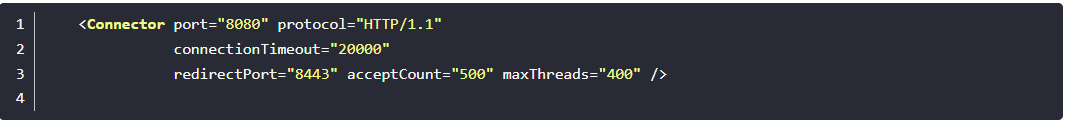
对于一个应用系统来说一定会有极限并发/请求数，即总有一个TPS/QPS阀值，如果超了阀值则系统就会不响应用户请求或响应的非常慢，因此我们最好进行过载保护，防止大量请求涌入击垮系统。

tomcat 通过以下三个配置参数来进行限流操作：

（1）maxThreads（最大线程数）：每一次HTTP请求到达Web服务，tomcat都会创建一个线程来处理该请求，那么最大线程数决定了Web服务可以同时处理多少个请求，默认200.

（2）accepCount（最大等待数）：当调用Web服务的HTTP请求数达到tomcat的最大线程数时，还有新的HTTP请求到来，这时tomcat会将该请求放在等待队列中，这个acceptCount就是指能够接受的最大等待数，默认100.如果等待队列也被放满了，这个时候再来新的请求就会被tomcat拒绝（connection refused）。

（3）maxConnections（最大连接数）：这个参数是指在同一时间，tomcat能够接受的最大连接数。一般这个值要大于maxThreads+acceptCount。



**Redis限流**

计数器算法：简陋的设计思路：假设一个用户（用IP判断）每分钟访问某一个服务接口的次数不能超过10次，那么我们可以在Redis中创建一个键，并此时我们就设置键的过期时间为60秒，每一个用户对此服务接口的访问就把键值加1，在60秒内当键值增加到10的时候，就禁止访问服务接口。在某种场景中添加访问时间间隔还是很有必要的。

令牌桶算法：基于 Redis 的 list接口可以实现令牌桶令牌补充和令牌消耗操作。

@SentinelResource限流

基于您已经引入了Spring Cloud Alibaba Sentinel为基础

使用Sentinel实现接口限流

Sentinel的使用分为两部分：

sentinel-dashboard：与hystrix-dashboard类似，但是它更为强大一些。除了与hystrix-dashboard一样提供实时监控之外，还提供了流控规则、熔断规则的在线维护等功能。

客户端整合：每个微服务客户端都需要整合sentinel的客户端封装与配置，才能将监控信息上报给dashboard展示以及实时的更改限流或熔断规则等。

@Slf4j

@Service

public class TestService {

@SentinelResource(value = "doSomeThing", blockHandler = "exceptionHandler")

public void doSomeThing(String str) {

log.info(str);

}

// 限流与阻塞处理

public void exceptionHandler(String str, BlockException ex) {

log.error( "blockHandler：" + str, ex);

}

}

主要做了两件事：

通过@SentinelResource注解的blockHandler属性制定具体的处理函数

实现处理函数，该函数的传参必须与资源点的传参一样，并且最后加上BlockException异常参数；同时，返回类型也必须一样。

如果熟悉Hystrix的读者应该会发现，这样的设计与HystrixCommand中定义fallback很相似，还是很容易理解的。

这里个帖子做了很详细的讲解：

https://www.cnblogs.com/didispace/p/11114788.html

**分布式限流**

分布式限流最关键的是要将限流服务做成原子化，而解决方案可以使使用redis+lua或者nginx+lua技术进行实现，通过这两种技术可以实现的高并发和高性能。

首先我们来使用redis+lua实现时间窗内某个接口的请求数限流，实现了该功能后可以改造为限流总并发/请求数和限制总资源数。Lua本身就是一种编程语言，也可以使用它实现复杂的令牌桶或漏桶算法。

有人会纠结如果应用并发量非常大那么redis或者nginx是不是能抗得住；不过这个问题要从多方面考虑：你的流量是不是真的有这么大，是不是可以通过一致性哈希将分布式限流进行分片，是不是可以当并发量太大降级为应用级限流；对策非常多，可以根据实际情况调节；像在京东使用Redis+Lua来限流抢购流量，一般流量是没有问题的。

对于分布式限流目前遇到的场景是业务上的限流，而不是流量入口的限流；流量入口限流应该在接入层完成，而接入层笔者一般使用Nginx。

**2 熔断**

在介绍熔断机制之前，我们需要了解微服务的雪崩效应。在微服务架构中，微服务是完成一个单一的业务功能，这样做的好处是可以做到解耦，每个微服务可以独立演进。但是，一个应用可能会有多个微服务组成，微服务之间的数据交互通过远程过程调用完成。这就带来一个问题，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其它的微服务，这就是所谓的“扇出”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”。

**熔断机制是**应对雪崩效应的一种微服务链路保护机制。我们在各种场景下都会接触到熔断这两个字。高压电路中，如果某个地方的电压过高，熔断器就会熔断，对电路进行保护。股票交易中，如果股票指数过高，也会采用熔断机制，暂停股票的交易。同样，在微服务架构中，熔断机制也是起着类似的作用。当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应信息。当检测到该节点微服务调用响应正常后，恢复调用链路。

@SentinelResource熔断

@SentinelResource注解除了可以用来做限流控制之外，还能实现与Hystrix类似的熔断降级策略

@Slf4j

@Service

public class TestService {

// 熔断与降级处理

@SentinelResource(value = "doSomeThing2", fallback = "fallbackHandler")

public void doSomeThing2(String str) {

log.info(str);

throw new RuntimeException("发生异常");

}

public void fallbackHandler(String str) {

log.error("fallbackHandler：" + str);

}

}

在Sentinel中定义熔断的降级处理方法非常简单，与Hystrix非常相似。只需要使用@SentinelResource注解的fallback属性来指定具体的方法名即可。

**熔段解决如下几个问题：**

    当所依赖的对象不稳定时，能够起到快速失败的目的

    快速失败后，能够根据一定的算法动态试探所依赖对象是否恢复

**3 降级**

降级是指自己的待遇下降了，从RPC调用环节来讲，就是去访问一个本地的伪装者而不是真实的服务

当双11活动时，把无关交易的服务统统降级，如查看蚂蚁深林，查看历史订单，商品历史评论，只显示最后100条等等。

区别

相同点：

目的很一致，都是从可用性可靠性着想，为防止系统的整体缓慢甚至崩溃，采用的技术手段；

最终表现类似，对于两者来说，最终让用户体验到的是某些功能暂时不可达或不可用；

粒度一般都是服务级别，当然，业界也有不少更细粒度的做法，比如做到数据持久层（允许查询，不允许增删改）；

自治性要求很高，熔断模式一般都是服务基于策略的自动触发，降级虽说可人工干预，但在微服务架构下，完全靠人显然不可能，开关预置、配置中心都是必要手段；

区别：

触发原因不太一样，服务熔断一般是某个服务（下游服务）故障引起，而服务降级一般是从整体负荷考虑；

管理目标的层次不太一样，熔断其实是一个框架级的处理，每个微服务都需要（无层级之分），而降级一般需要对业务有层级之分（比如降级一般是从最外围服务开始）

实现方式不太一样；服务降级具有代码侵入性(由控制器完成/或自动降级)，熔断一般称为自我熔断。

————————————————