Circuit Breaker: Hystrix Clients

**服务熔断的理解**



服务熔断和电路熔断是一个道理，如果一条线路电压过高，保险丝会熔断，防止出现火灾，但是过后重启仍然是可用的。

而服务熔断则是对于目标服务的请求和调用大量超时或失败，这时应该熔断该服务的所有调用，并且对于后续调用应直接返回，从而快速释放资源，确保在目标服务不可用的这段时间内，所有对它的调用都是立即返回，不会阻塞的。再等到目标服务好转后进行接口恢复。

客户端在调用服务端之前加上熔断控制器，进行目标服务的调用，如果在规定的次数内都为成功，则认为该服务不可用，对于该服务的所有调用返回一个容错值，并设置一个定时线程，在一定的时间后重新恢复服务。

**与服务降级有什么区别呢？**

比如一个业务场景的服务调用链中调用多个服务，有些服务不是该业务场景的必须服务，则可以进行降级处理。服务降级，就是这么回事，整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来。

**所以从上述分析来看，两者其实从有些角度看是有一定的类似性的：**

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；

**而两者的区别也是明显的：**

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；

**熔断器实现的三个状态机**

Closed：熔断器关闭状态，调用失败次数积累，到了阈值（或一定比例）则启动熔断机制；

Open：熔断器打开状态，此时对下游的调用都内部直接返回错误，不走网络，但设计了一个时钟选项，默认的时钟达到了一定时间（这个时间一般设置成平均故障处理时间，也就是MTTR），到了这个时间，进入半熔断状态；

Half-Open：半熔断状态，允许定量的服务请求，如果调用都成功（或一定比例）则认为恢复了，关闭熔断器，否则认为还没好，又回到熔断器打开状态；

**为什么需要Hystrix?**

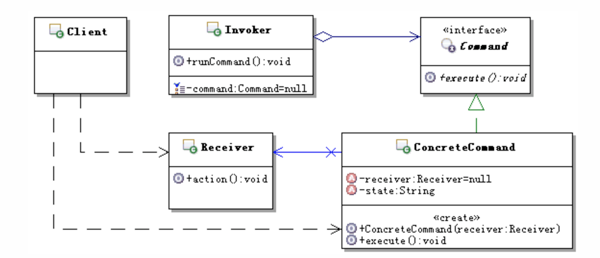
在大中型分布式系统中，通常系统很多依赖(HTTP,hession,Netty,Dubbo等)，在高并发访问下,这些依赖的稳定性与否对系统的影响非常大,但是依赖有很多不可控问题:如网络连接缓慢，资源繁忙，暂时不可用，服务脱机等。

当依赖阻塞时,大多数服务器的线程池就出现阻塞(BLOCK),影响整个线上服务的稳定性，在复杂的分布式架构的应用程序有很多的依赖，都会不可避免地在某些时候失败。高并发的依赖失败时如果没有隔离措施，当前应用服务就有被拖垮的风险。

解决问题方案:对依赖做隔离

**Hystrix设计理念**

想要知道如何使用，必须先明白其核心设计理念，Hystrix基于命令模式，通过UML图先直观的认识一下这一设计模式



可见，Command是在Receiver和Invoker之间添加的中间层，Command实现了对Receiver的封装。那么Hystrix的应用场景如何与上图对应呢？

API既可以是Invoker又可以是reciever，通过继承Hystrix核心类HystrixCommand来封装这些API（例如，远程接口调用，数据库查询之类可能会产生延时的操作）。就可以为API提供弹性保护了。

**Hystrix如何解决依赖隔离**

1:Hystrix使用命令模式HystrixCommand(Command)包装依赖调用逻辑，每个命令在单独线程中/信号授权下执行。

2:可配置依赖调用超时时间,超时时间一般设为比99.5%平均时间略高即可.当调用超时时，直接返回或执行fallback逻辑。

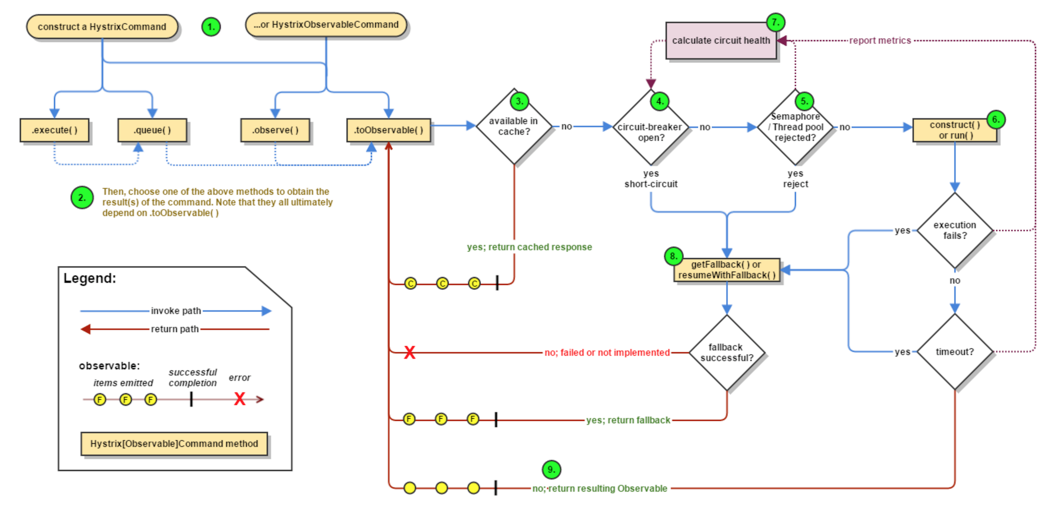
3:为每个依赖提供一个小的线程池（或信号），如果线程池已满调用将被立即拒绝，默认不采用排队.加速失败判定时间。

4:依赖调用结果分:成功，失败（抛出异常），超时，线程拒绝，短路。 请求失败(异常，拒绝，超时，短路)时执行fallback(降级)逻辑。

5:提供熔断器组件,可以自动运行或手动调用,停止当前依赖一段时间(10秒)，熔断器默认错误率阈值为50%,超过将自动运行。

6:提供近实时依赖的统计和监控

Hystrix流程结构解析



流程说明:

1:每次调用创建一个新的HystrixCommand,把依赖调用封装在run()方法中.

2:执行execute()/queue做同步或异步调用.

3:判断熔断器(circuit-breaker)是否打开,如果打开跳到步骤8,进行降级策略,如果关闭进入步骤.

4:判断线程池/队列/信号量是否跑满，如果跑满进入降级步骤8,否则继续后续步骤.

5:调用HystrixCommand的run方法.运行依赖逻辑

5a:依赖逻辑调用超时,进入步骤8.

6:判断逻辑是否调用成功

6a:返回成功调用结果

6b:调用出错，进入步骤8.

7:计算熔断器状态,所有的运行状态(成功, 失败, 拒绝,超时)上报给熔断器，用于统计从而判断熔断器状态.

8:getFallback()降级逻辑.

以下四种情况将触发getFallback调用：

(1):run()方法抛出非HystrixBadRequestException异常。

(2):run()方法调用超时

(3):熔断器开启拦截调用

(4):线程池/队列/信号量是否跑满

8a:没有实现getFallback的Command将直接抛出异常

8b:fallback降级逻辑调用成功直接返回

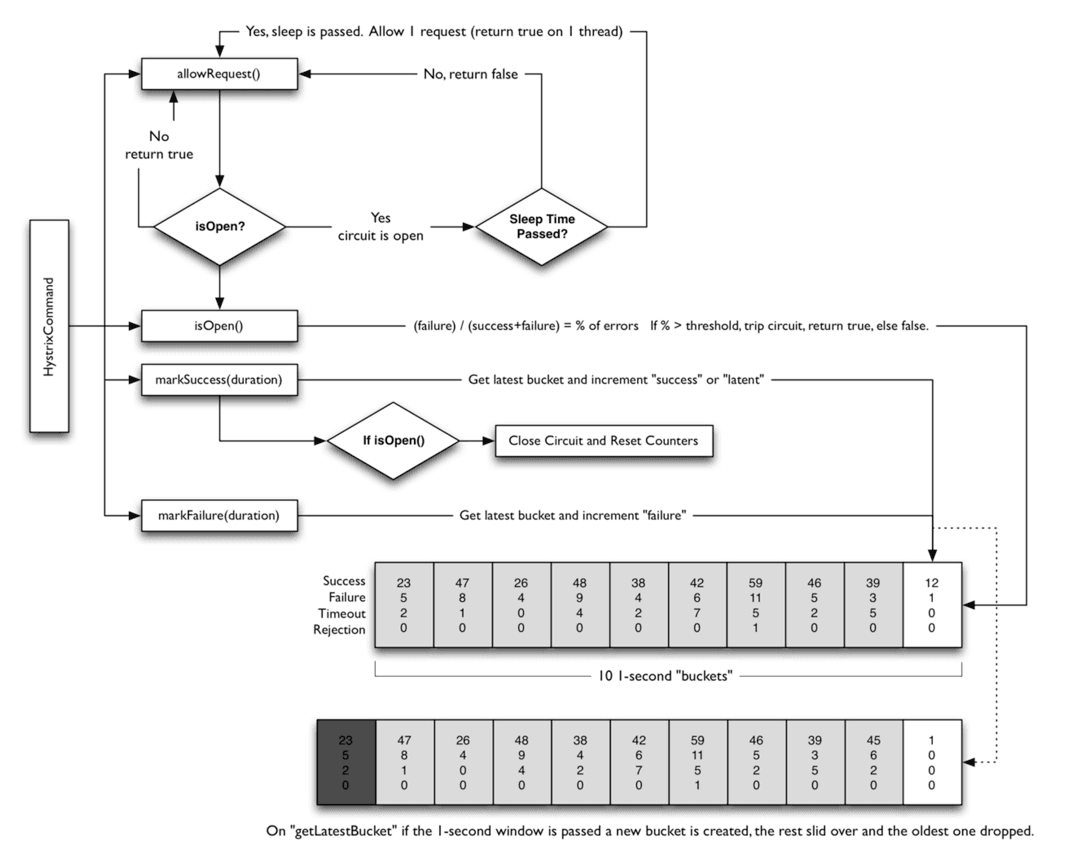
8c:降级逻辑调用失败抛出异常

9:返回执行成功结果

**熔断器:Circuit Breaker**

每个熔断器默认维护10个bucket,每秒一个bucket,每个bucket记录成功,失败,超时,拒绝的状态，

默认错误超过50%且10秒内超过20个请求进行中断拦截.



## Hystrix隔离分析

Hystrix隔离方式采用线程/信号的方式,通过隔离限制依赖的并发量和阻塞扩散.

\* (1)线程隔离

把执行依赖代码的线程与请求线程(如:jetty线程)分离，请求线程可以自由控制离开的时间(异步过程)。

通过线程池大小可以控制并发量，当线程池饱和时可以提前拒绝服务,防止依赖问题扩散。

线上建议线程池不要设置过大，否则大量堵塞线程有可能会拖慢服务器。

\* (2)线程隔离的优缺点

\* 线程隔离的优点:

\* [1]:使用线程可以完全隔离第三方代码,请求线程可以快速放回。

\* [2]:当一个失败的依赖再次变成可用时，线程池将清理，并立即恢复可用，而不是一个长时间的恢复。

\* [3]:可以完全模拟异步调用，方便异步编程。

\* 线程隔离的缺点:

\* [1]:线程池的主要缺点是它增加了cpu，因为每个命令的执行涉及到排队(默认使用SynchronousQueue避免排队)，调度和上下文切换。

\* [2]:对使用ThreadLocal等依赖线程状态的代码增加复杂性，需要手动传递和清理线程状态。

\* NOTE: Netflix公司内部认为线程隔离开销足够小，不会造成重大的成本或性能的影响。

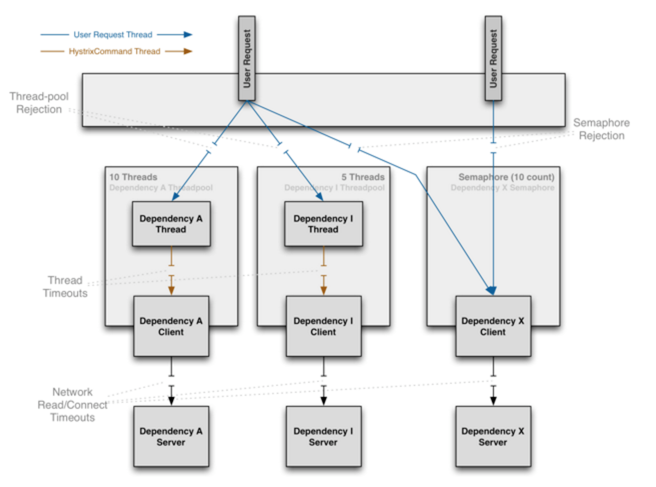
\* Netflix 内部API 每天100亿的HystrixCommand依赖请求使用线程隔，每个应用大约40多个线程池，每个线程池大约5-20个线程。

\* (3)信号隔离

\* 信号隔离也可以用于限制并发访问，防止阻塞扩散, 与线程隔离最大不同在于执行依赖代码的线程依然是请求线程（该线程需要通过信号申请）,

\* 如果客户端是可信的且可以快速返回，可以使用信号隔离替换线程隔离,降低开销.

\* 信号量的大小可以动态调整, 线程池大小不可以.



## 参数说明

@HystrixCommand(groupKey = "productStockOpLog", commandKey = "addProductStockOpLog", fallbackMethod = "addProductStockOpLogFallback",

commandProperties = {

@HystrixProperty(

//指定多久超时，单位毫秒。超时进fallback

name = "execution.isolation.thread.timeoutInMilliseconds", value = "400"),

//判断熔断的最少请求数，默认是10；只有在一个统计窗口内处理的请求数量达到这个阈值，才会进行熔断与否的判断

@HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),

//判断熔断的阈值，默认值50，表示在一个统计窗口内有50%的请求处理失败，会触发熔断

@HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "10"),

}

)

public void addProductStockOpLog(Long sku\_id, Object old\_value, Object new\_value) throws Exception {

if (new\_value != null && !new\_value.equals(old\_value)) {

doAddOpLog(null, null, sku\_id, null, ProductOpType.PRODUCT\_STOCK, old\_value != null ? String.valueOf(old\_value) : null, String.valueOf(new\_value), 0, "C端", null);

}

}

public void addProductStockOpLogFallback(Long sku\_id, Object old\_value, Object new\_value) throws Exception {

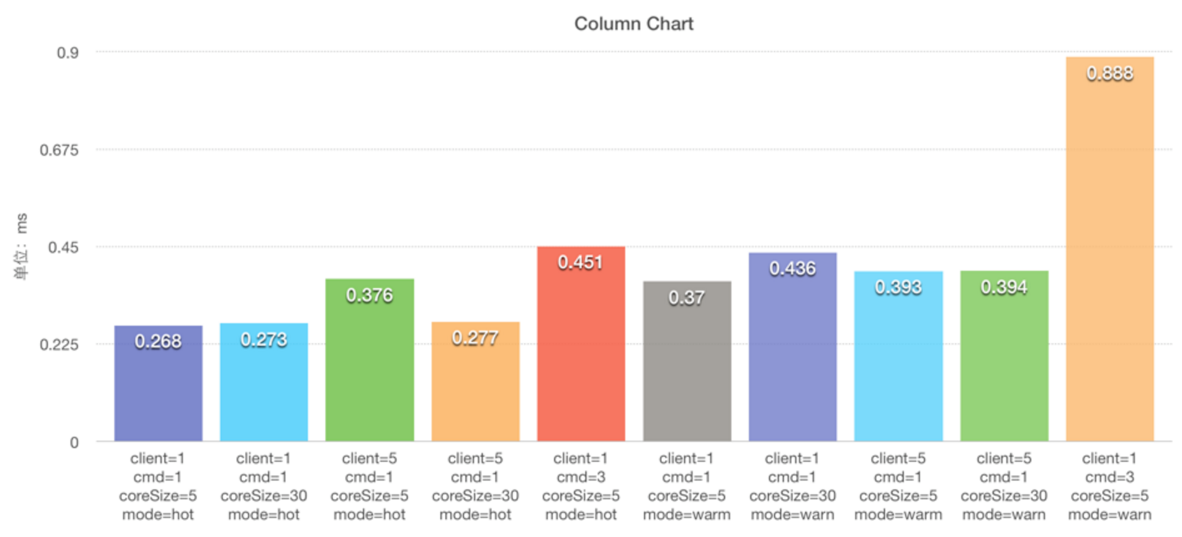
LOGGER.warn("发送商品库存变更消息失败,进入Fallback,skuId:{},oldValue:{},newValue:{}", sku\_id, old\_value, new\_value);

}

说明：   
hystrix函数必须为public，fallback函数可以为private。两者需要返回值和参数相同 详情。

hystrix函数需要放在一个service中，并且，在类本身的其他函数中调用hystrix函数，是无法达到监控的目的的。

## 性能测试



去除Cold状态的第一个异常点后，1-10测试场景的Hystrix平均耗时如上图所示， 可以得出结论：

1. 单个HystrixCommand的额外耗时基本稳定处于0.3ms左右，和线程池大小无关，和client数量无关

2. hystrix的额外耗时和执行的HystrixCommand数量有关系，随着command数量增多，耗时增加，但是增量较小，没有比例关系

3. App刚启动时，第一个请求耗时300+ms，随后请求的耗时降低至1ms以下；刚启动的一小段时间内耗时略大于Hot状态时耗时，总体不超过1ms