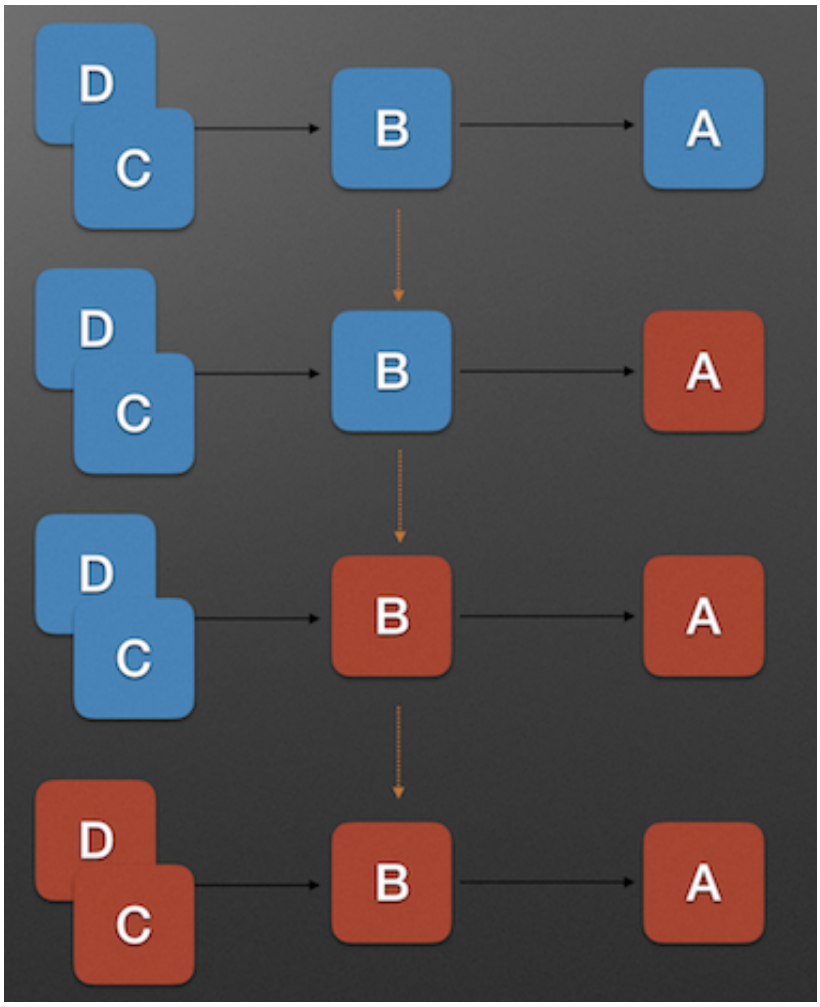
前言

分布式系统中经常会出现某个基础服务不可用造成整个系统不可用的情况, 这种现象被称为服务雪崩效应. 为了应对服务雪崩, 一种常见的做法是手动服务降级. 而Hystrix的出现,给我们提供了另一种选择.

服务雪崩效应的定义

服务雪崩效应是一种因 **服务提供者** 的不可用导致 **服务调用者** 的不可用,并将不可用 **逐渐放大** 的过程.如果所示:



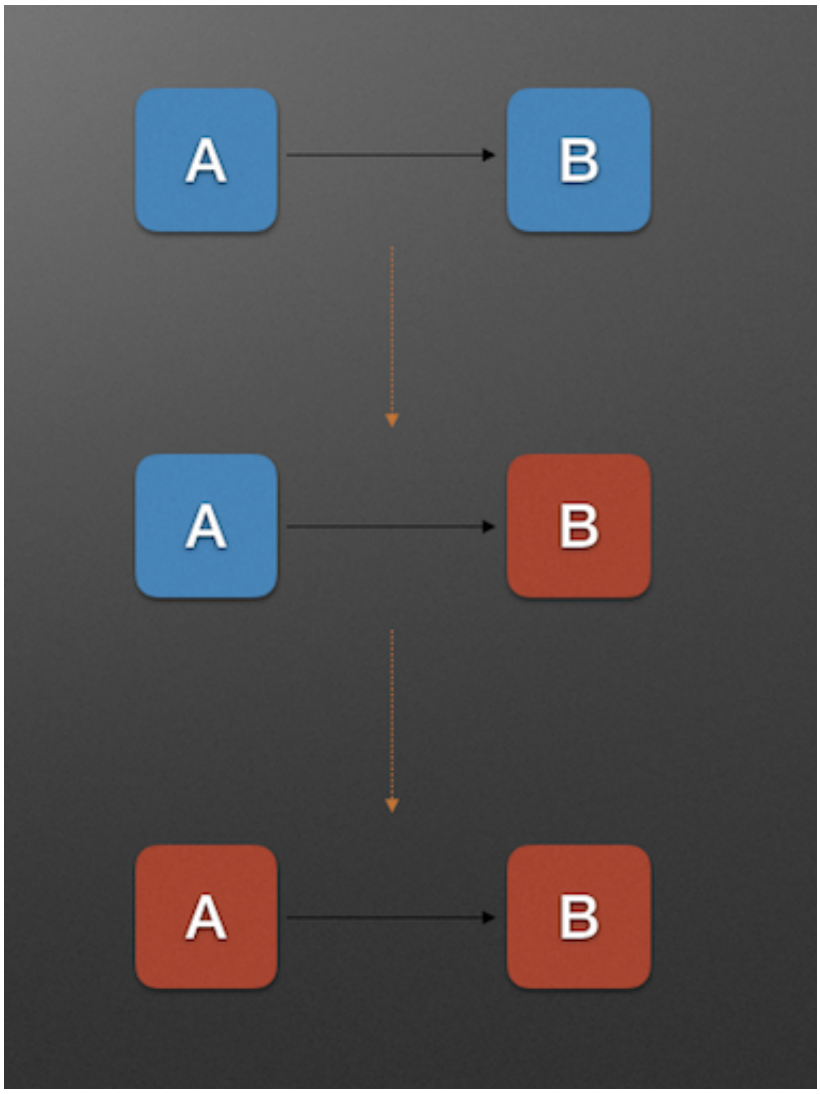
上图中, A为服务提供者, B为A的服务调用者, C和D是B的服务调用者. 当A的不可用,引起B的不可用,并将不可用逐渐放大C和D时, 服务雪崩就形成了.

服务雪崩效应形成的原因

我把服务雪崩的参与者简化为 **服务提供者** 和 **服务调用者**, 并将服务雪崩产生的过程分为以下三个阶段来分析形成的原因:

1. 服务提供者不可用

2. 重试加大流量
3. 服务调用者不可用



服务雪崩的每个阶段都可能由不同的原因造成, 比如造成 **服务不可用** 的原因有:

- 硬件故障
- 程序Bug
- 缓存击穿
- 用户大量请求

硬件故障可能为硬件损坏造成的服务器主机宕机, 网络硬件故障造成的服务提供者的不可访问.

缓存击穿一般发生在缓存应用重启, 所有缓存被清空时,以及短时间内大量缓存失效时. 大量的缓存不命中, 使请求直击后端,造成服务提供者超负荷运行,引起服务不可用.

在秒杀和大促开始前,如果准备不充分,用户发起大量请求也会造成服务提供者的不可用.

而形成 **重试加大流量** 的原因有:

- 用户重试
- 代码逻辑重试

在服务提供者不可用后, 用户由于忍受不了界面上长时间的等待,而不断刷新页面甚至提交表单.

服务调用端的会存在大量服务异常后的重试逻辑.

这些重试都会进一步加大请求流量.

最后, **服务调用者不可用** 产生的主要原因是:

- 同步等待造成的资源耗尽

当服务调用者使用 **同步调用** 时, 会产生大量的等待线程占用系统资源. 一旦线程资源被耗尽,服务调用者提供的服务也将处于不可用状态, 于是服务雪崩效应产生了.

服务雪崩的应对策略

针对造成服务雪崩的不同原因, 可以使用不同的应对策略:

1. 流量控制
2. 改进缓存模式
3. 服务自动扩容
4. 服务调用者降级服务

流量控制 的具体措施包括:

- 网关限流
- 用户交互限流
- 关闭重试

因为Nginx的高性能, 目前一线互联网公司大量采用Nginx+Lua的网关进行流量控制, 由此而来的OpenResty也越来越热门.

用户交互限流的具体措施有: 1. 采用加载动画,提高用户的忍耐等待时间. 2. 提交按钮添加强制等待时间机制.

改进缓存模式 的措施包括:

- 缓存预加载
- 同步改为异步刷新

服务自动扩容 的措施主要有:

- AWS的auto scaling

服务调用者降级服务 的措施包括:

- 资源隔离

- 对依赖服务进行分类
- 不可用服务的调用快速失败

资源隔离主要是对调用服务的线程池进行隔离.

我们根据具体业务,将依赖服务分为: 强依赖和若依赖. 强依赖服务不可用会导致当前业务中止,而弱依赖服务的不可用不会导致当前业务的中止.

不可用服务的调用快速失败一般通过 **超时机制**, **熔断器** 和熔断后的 **降级方法** 来实现.

使用Hystrix预防服务雪崩

Hystrix [hɪst'rɪks]的中文含义是豪猪, 因其背上长满了刺,而拥有自我保护能力.

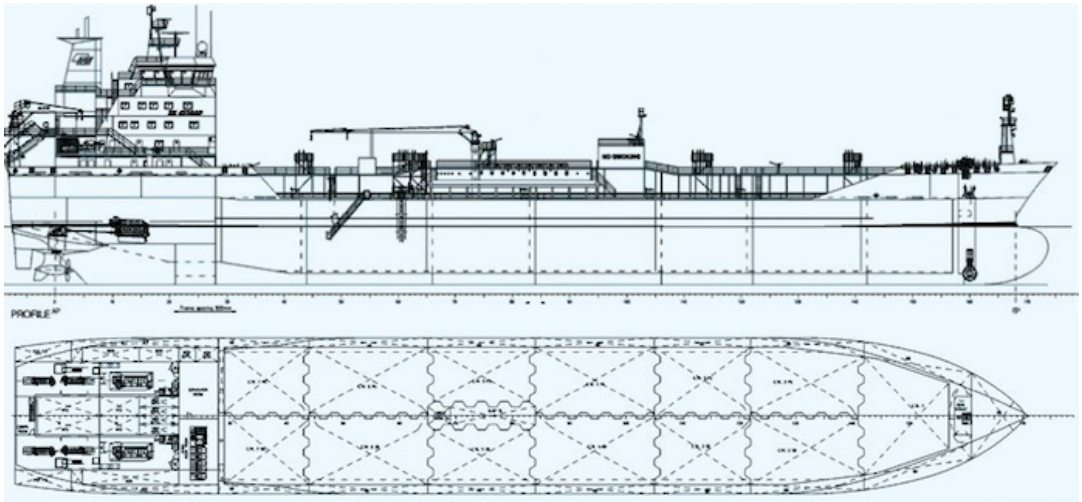
Netflix的 **Hystrix** 是一个帮助解决分布式系统交互时超时处理和容错的类库, 它同样拥有保护系统的能力.

Hystrix的设计原则包括:

- 资源隔离
- 熔断器
- 命令模式

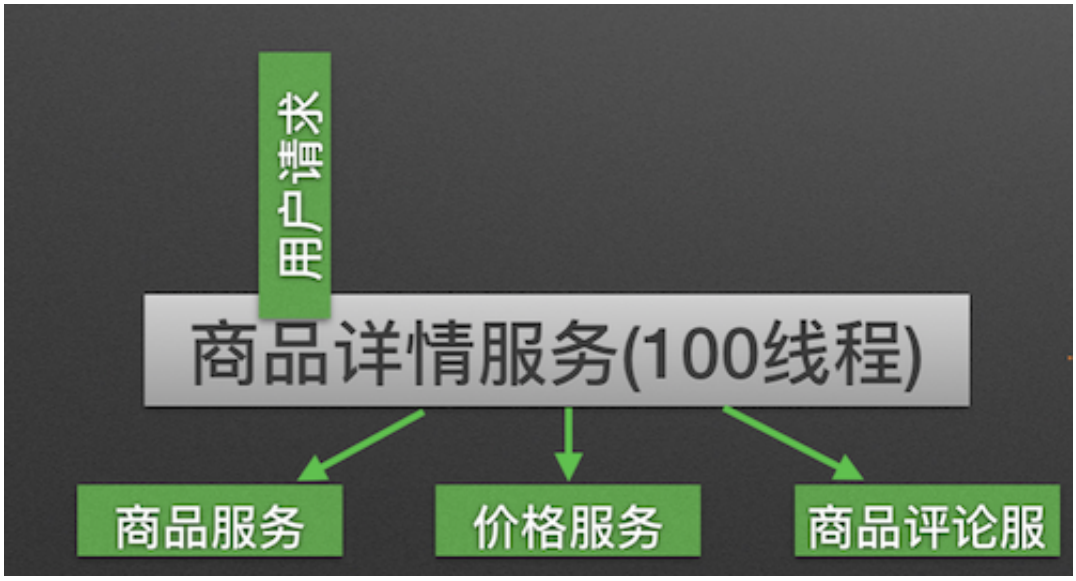
资源隔离

货船为了进行防止漏水和火灾的扩散,会将货仓分隔为多个, 如下图所示:

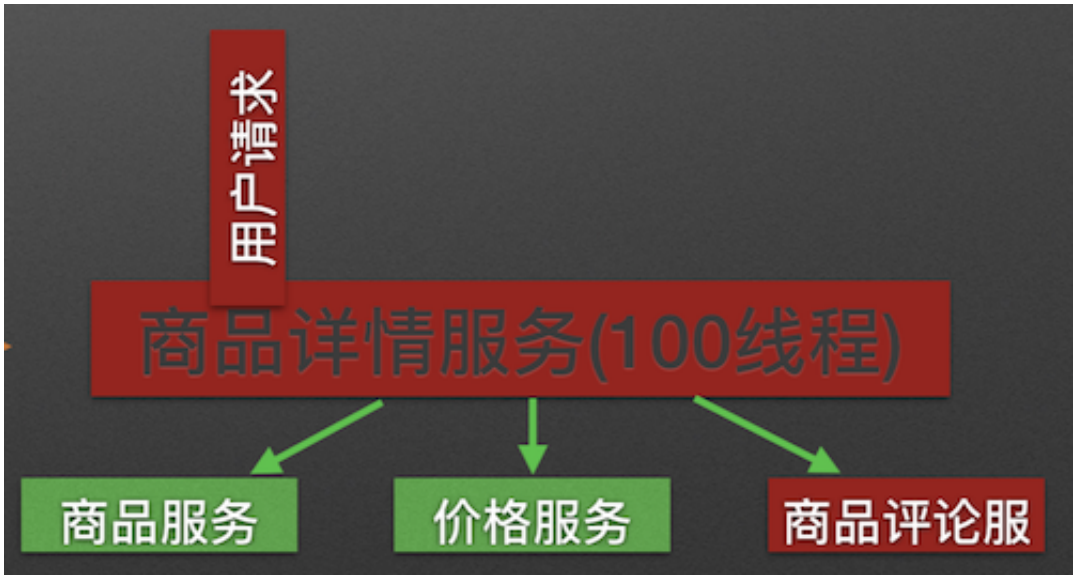


这种资源隔离减少风险的方式被称为:Bulkheads(舱壁隔离模式). Hystrix将同样的模式运用到了服务调用者上.

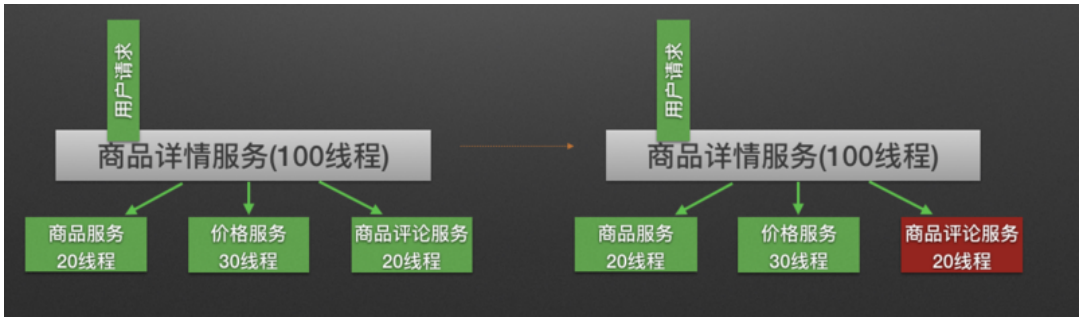
在一个高度服务化的系统中,我们实现的一个业务逻辑通常会依赖多个服务,比如:商品详情展示服务会依赖商品服务, 价格服务, 商品评论服务. 如图所示:



调用三个依赖服务会共享商品详情服务的线程池. 如果其中的商品评论服务不可用, 就会出现线程池里所有线程都因等待响应而被阻塞, 从而造成服务雪崩. 如图所示:

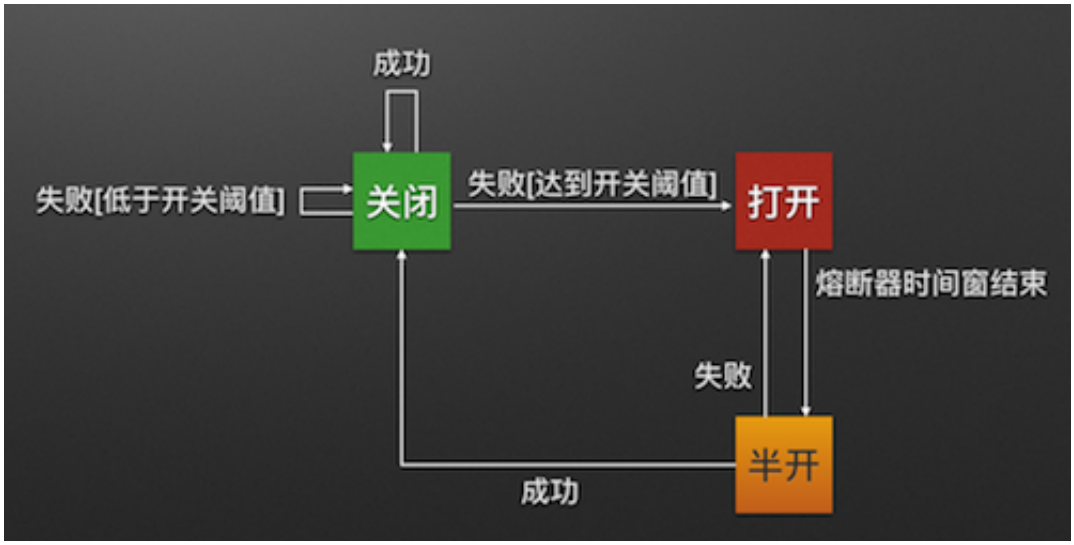


Hystrix通过将每个依赖服务分配独立的线程池进行资源隔离, 从而避免服务雪崩. 如下图所示, 当商品评论服务不可用时, 即使商品服务独立分配的20个线程全部处于同步等待状态,也不会影响其他依赖服务的调用.



熔断器模式

熔断器模式定义了熔断器开关相互转换的逻辑:



服务的健康状况 = 请求失败数 / 请求总数. 熔断器开关由关闭到打开的状态转换是通过当前服务健康状况和设定阈值比较决定的.

1. 当熔断器开关关闭时, 请求被允许通过熔断器. 如果当前健康状况高于设定阈值, 开关继续保持关闭. 如果当前健康状况低于设定阈值, 开关则切换为打开状态.
2. 当熔断器开关打开时, 请求被禁止通过.
3. 当熔断器开关处于打开状态, 经过一段时间后, 熔断器会自动进入半开状态, 这时熔断器只允许一个请求通过. 当该请求调用成功时, 熔断器恢复到关闭状态. 若该请求失败, 熔断器继续保持打开状态, 接下来的请求被禁止通过.

熔断器的开关能保证服务调用者在调用异常服务时, 快速返回结果, 避免大量的同步等待. 并且熔断器能在一段时间后继续侦测请求执行结果, 提供恢复服务调用的可能.

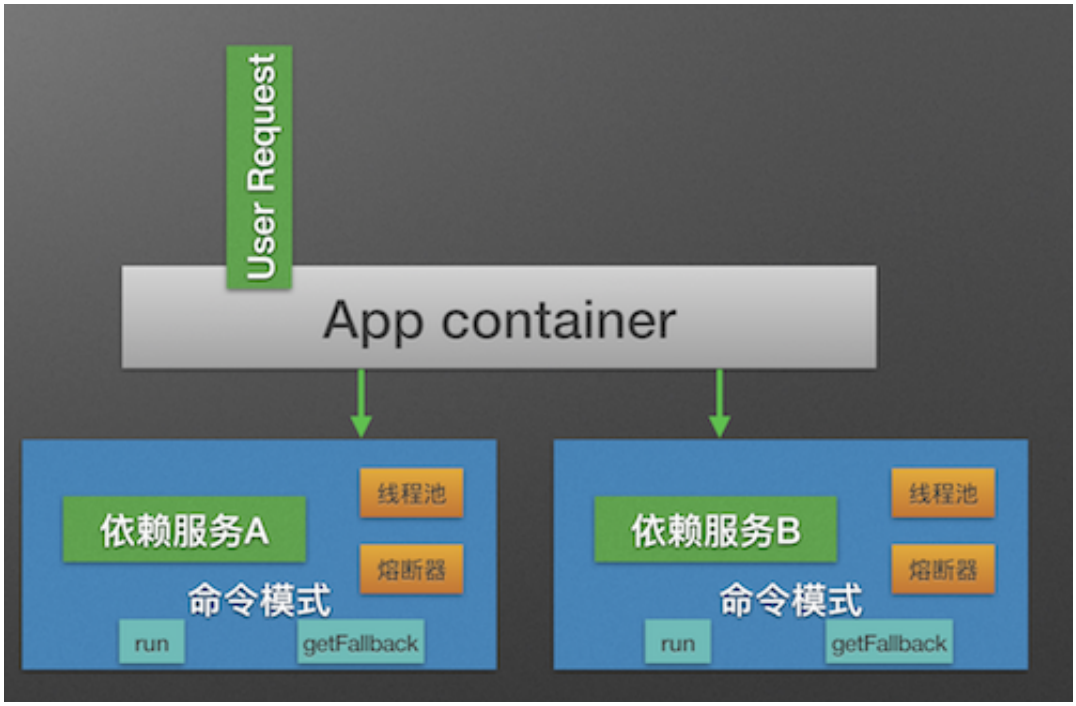
命令模式

Hystrix使用命令模式(继承HystrixCommand类)来包裹具体的服务调用逻辑(run方法), 并在命令模式中添加了服务调用失败后的降级逻辑(getFallback).同时我们在Command的构造方法中可以定义当前服务线程池和熔断器的相关参数. 如下代码所示:

```
publicclassService1HystrixCommandextendsHystrixCommand<Response>
{
  private Service1 service;
  private Request request;

  publicService1HystrixCommand(Service1 service, Request
request){
```

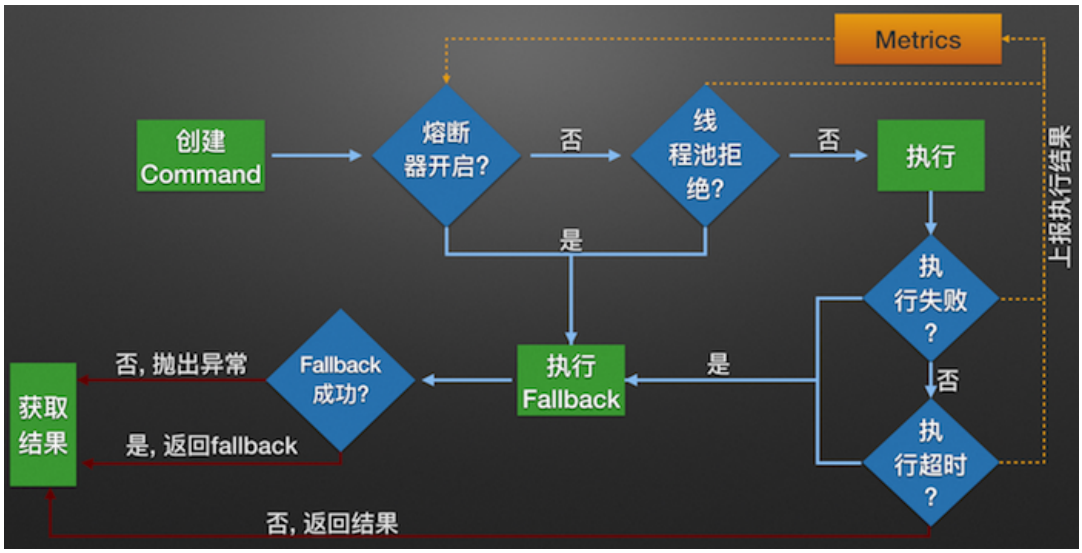
```
    supper(

Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Servic
eGroup"))

.andCommandKey(HystrixCommandKey.Factory.asKey("servcie1query"))

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("service1Th
readPool"))

.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Set
ter()
            .withCoreSize(20))//服务线程池数量

.andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
            .withCircuitBreakerErrorThresholdPercentage(60)//熔断
器关闭到打开阈值
            .withCircuitBreakerSleepWindowInMilliseconds(3000)//
熔断器打开到关闭的时间窗长度
      ))
      this.service = service;
      this.request = request;
    );
  }

  @Overrideprotected Response run(){
    return service1.call(request);
  }

  @Overrideprotected Response getFallback(){
    return Response.dummy();
  }
}
```

在使用了Command模式构建了服务对象之后, 服务便拥有了熔断器和线程池的功能.



Hystrix的内部处理逻辑

下图为Hystrix服务调用的内部逻辑:



1. 构建Hystrix的Command对象, 调用执行方法.
2. Hystrix检查当前服务的熔断器开关是否开启, 若开启, 则执行降级服务getFallback方法.
3. 若熔断器开关关闭, 则Hystrix检查当前服务的线程池是否能接收新的请求, 若超过线程池已满, 则执行降级服务getFallback方法.
4. 若线程池接受请求, 则Hystrix开始执行服务调用具体逻辑run方法.
5. 若服务执行失败, 则执行降级服务getFallback方法, 并将执行结果上报

Metrics更新服务健康状况.

6. 若服务执行超时, 则执行降级服务getFallback方法, 并将执行结果上报Metrics更新服务健康状况.

7. 若服务执行成功, 返回正常结果.

8. 若服务降级方法getFallback执行成功, 则返回降级结果.

9. 若服务降级方法getFallback执行失败, 则抛出异常.

Hystrix Metrics的实现

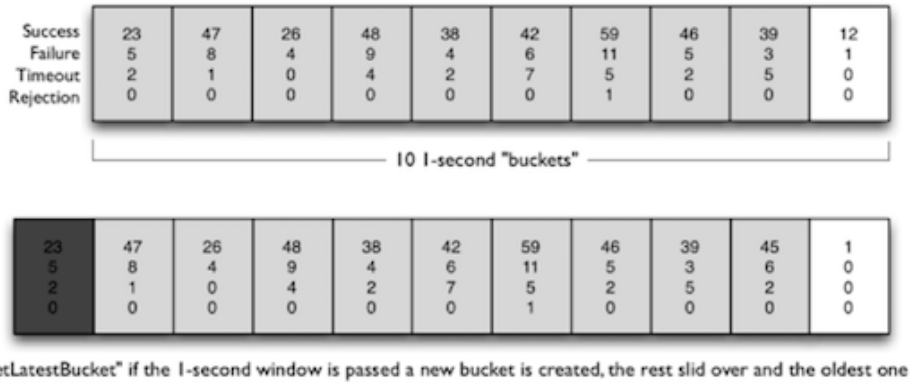
Hystrix的Metrics中保存了当前服务的健康状况, 包括服务调用总次数和服务调用失败次数等. 根据Metrics的计数, 熔断器从而能计算出当前服务的调用失败率, 用来和设定的阈值比较从而决定熔断器的状态切换逻辑. 因此Metrics的实现非常重要.

1.4之前的滑动窗口实现

Hystrix在这些版本中的使用自己定义的滑动窗口数据结构来记录当前时间窗的各种事件(成功,失败,超时,线程池拒绝等)的计数.

事件产生时, 数据结构根据当前时间确定使用旧桶还是创建新桶来计数, 并在桶中对计数器经行修改.

这些修改是多线程并发执行的, 代码中有不少加锁操作,逻辑较为复杂.



On "getLatestBucket" if the 1-second window is passed a new bucket is created, the rest slid over and the oldest one dropped.

1.5之后的滑动窗口实现

Hystrix在这些版本中开始使用RxJava的Observable.window()实现滑动窗口.

RxJava的window使用后台线程创建新桶, 避免了并发创建桶的问题.

同时RxJava的单线程无锁特性也保证了计数变更时的线程安全. 从而使代码更加简洁.

以下为我使用RxJava的window方法实现的一个简易滑动窗口Metrics, 短短几行代码便能完成统计功能,足以证明RxJava的强大:

```
@Testpublicvoidtime WindowTest()throws Exception{
  Observable<Integer> source = Observable.interval(50,
TimeUnit.MILLISECONDS).map(i -> RandomUtils.nextInt(2));
  source.window(1, TimeUnit.SECONDS).subscribe(window -> {
```

```
    int[] metrics = newint[2];
    window.subscribe(i -> metrics[i]++,
      InternalObservableUtils.ERROR_NOT_IMPLEMENTED,
      () -> System.out.println("窗口Metrics:" +
JSON.toJSONString(metrics)));
  });
  TimeUnit.SECONDS.sleep(3);
}
```

总结

通过使用Hystrix,我们能方便的防止雪崩效应, 同时使系统具有自动降级和自动恢复服务的效果.