# Hystrix简介和使用

在分布式系统中，依赖之间会不可避免的调用失败，比如超时、异常等，如何能够保证在一个依赖出问题的情况下，不会导致整体服务失败，这就需要断路器。其中Hystrix是目前最常用的断路器系统，其提供熔断、隔离、Fallback、Cache及监控等功能，保证系统在一个或者多个依赖同时出现问题时保证系统依然可用。

# Hystrix系统架构

在微服务体系结构中应用程序依赖其他服务，依赖项在某些时候不可避免的失败，如下图所示：

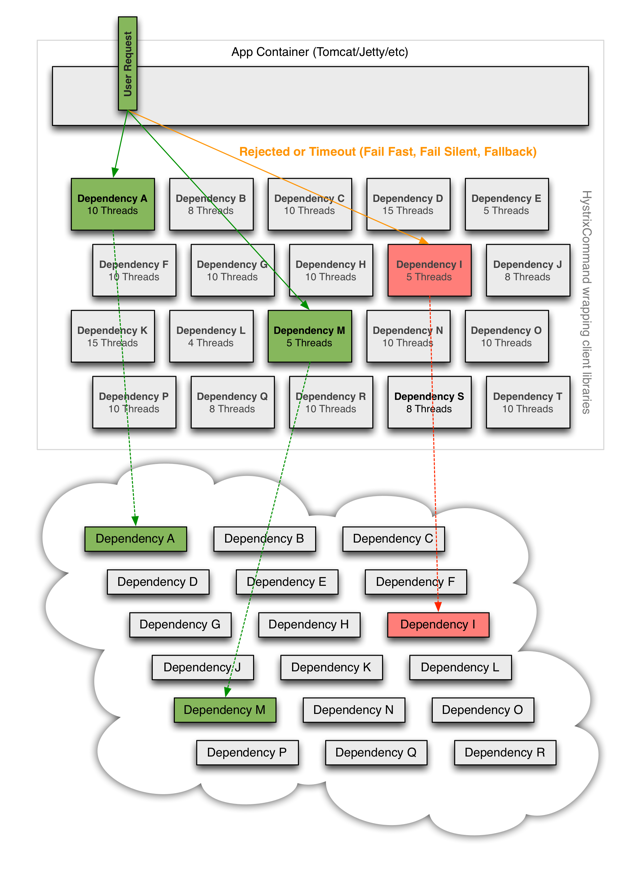
|  |  |
| --- | --- |
| **874963-20180730172725624-245631738** | 874963-20180730172821821-960520983 |

Hystrix的设计目标是：

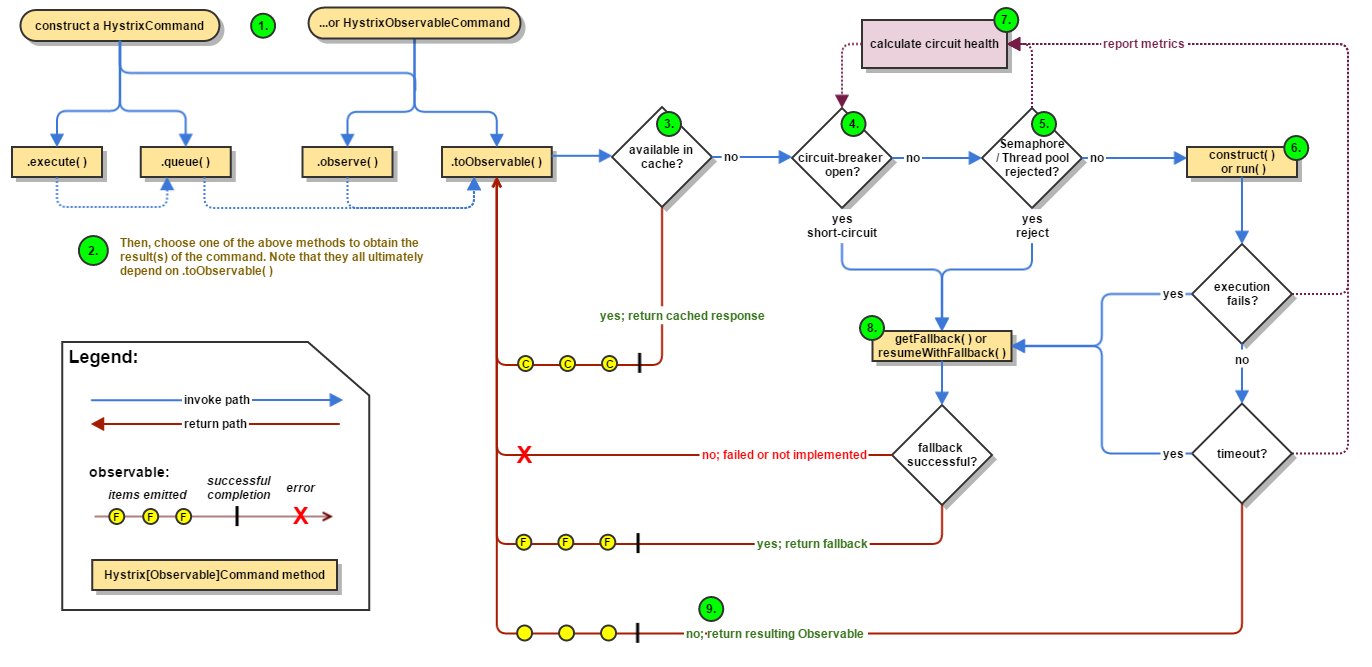
* 对通过第三方客户端库访问的依赖项的延迟和故障进行保护和控制
* 在复杂的分布式系统中阻止级联故障
* 快速失败及快速恢复
* 回退，尽可能优雅的降级
* 启用近实时监控、警报和操作控制

为了实现以上目标，Hystrix提供了依赖隔离的方案：

* 使用命令模式HystrixCommand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
* 可配置依赖调用超时时间，超时时间一般设置为99.5%平均时间略高，当调用超时，直接返回或这些fallback逻辑
* 为每个依赖提供小的线程池，如果线程池已满则调用立即拒绝，默认不采用排队
* 依赖调用结果分为：成功，失败，超时，线程拒绝及短路等。请求失败时执行fallback逻辑
* 提供熔断器组件，可以自动运行或手动调用，停止当前依赖时间，熔断器默认错误率阈值为50%，超时则将自动运行
* 提供近实时依赖的统计和监控



Hystrix的流程结构如下图：



1. 每次调用创建新的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，如果关闭则进入步骤4
4. 判断线程池/队列/信号量是否跑满，如果跑满则进入降级步骤8，否则继续后续步骤
5. 调用HystrixCommand的run方法，运行依赖逻辑
   1. 依赖逻辑调用超时，则进入步骤8
6. 判断逻辑是否调用成功
   1. 返回成功调用结果
   2. 调用出错，进入步骤8
7. 计算熔断器状态，所有运行状态（成功/失败/拒绝/超时）上报给熔断器
8. getFallback降级逻辑，以下四种情况将触发getFallback调用

* run方法抛出非HystrixBadRequestException异常
* run方法调用超时
* 熔断器开启拦截调用
* 线程池/队列/信号量是否饱满

1. 没有实现getFallback的Command则直接抛出异常
2. fallback降级逻辑调用成功则直接返回
3. 降级逻辑调用失败则抛出异常

9）返回执行成功结果

# Hystrix使用示例

添加依赖，在pom.xml中

*<dependency>*

*<groupId>com.netflix.hystrix</groupId>*

*<artifactId>hystrix-core</artifactId>*

*<version>${hystrix.version}</version>*

*</dependency>*

*<dependency>*

*<groupId>com.netflix.hystrix</groupId>*

*<artifactId>hystrix-metrics-event-stream</artifactId>*

*<version>${hystrix-metrics-event-stream.version}</version>*

*</dependency>*

## 使用命令模式封装依赖逻辑

程序使用Hystrix，只需要简单继承Hystrix(Observable)Command，并重写run/construct方法，然后调用程序实例执行execute/queue/observe及toObservable方法，其示例如下：

*public class HelloWorldCommand extends HystrixCommand<String> {*

*private final String name;*

*public HelloWorldCommand(String name) {*

*super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));*

*this.name = name;*

*}*

*@Override*

*protected String run() {*

*return “Hello ” + name;*

*}*

*}*

测试程序，执行如下：

*HelloWorldCommand hwCommand = new HelloWorldCommand("Synchronous-hystrix");*

*String result = hwCommand.execute();*

同步调用执行command.execute其等同于command.queue().get。异步调用执行：

*Command.queue().get(timeout,TimeUnit.MILLISECODS)*

## **注册异步事件回调机制**

注册观察者事件拦截器，示例如下：

*Observable<String> fs = new HelloWorldCommand("Hystrix").observe();*

*fs.subscribe(new Action1<String>() { //注册结果回调事件*

*public void call(String s) {*

*System.out.println(s);*

*}*

*});*

注册完整执行生命周期事件

*fs.subscribe(new Observer<String>() {*

*@Override*

*public void onCompleted() {*

*// onNext/onError完成之后最后回调*

*System.out.println("execute onCompleted");*

*}*

*@Override*

*public void onError(Throwable e) {*

*// 当产生异常时回调*

*System.out.println("onError " + e.getMessage());*

*e.printStackTrace();*

*}*

*@Override*

*public void onNext(String v) {*

*// 获取结果后回调*

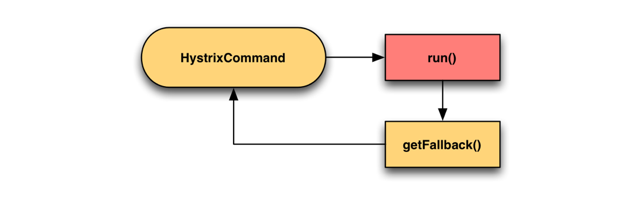
*System.out.println("onNext: " + v);*

*}*

*});*

## **使用Fallback提供降级策略**

执行流程如下：

****

在HystrixCommand中除了HystrixBadRequestException异常之外，所有从run方法抛出的异常都算作失败，并触发降级getFalback和断路器逻辑。

HystrixBadReqeustException用在非法参数或非系统故障异常等不应触发回退逻辑的场景。执行示例如下：

*public class HelloWorldCommand extends HystrixCommand<String> {*

*private final String name;*

*public HelloWorldCommand(String name) {*

*super(*

*Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))*

*.andCommandPropertiesDefaults( //设置执行时间timeout*

*HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(500)));*

*this.name = name;*

*}*

*@Override*

*protected String run() throws Exception {*

*TimeUnit.MILLISECONDS.sleep(1000); //超时*

*return "Hello " + name + " Thread: " + Thread.currentThread().getName();*

*}*

*@Override*

*protected String getFallback() {*

*return "Execute Falled";*

*}*

*}*

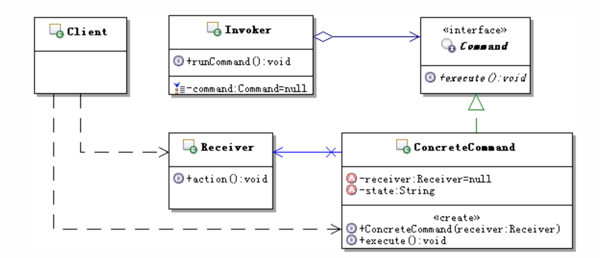
执行结果如下：

*Result: Execute Fallback*

超时触发了Fallback

# Hystrix源码分析

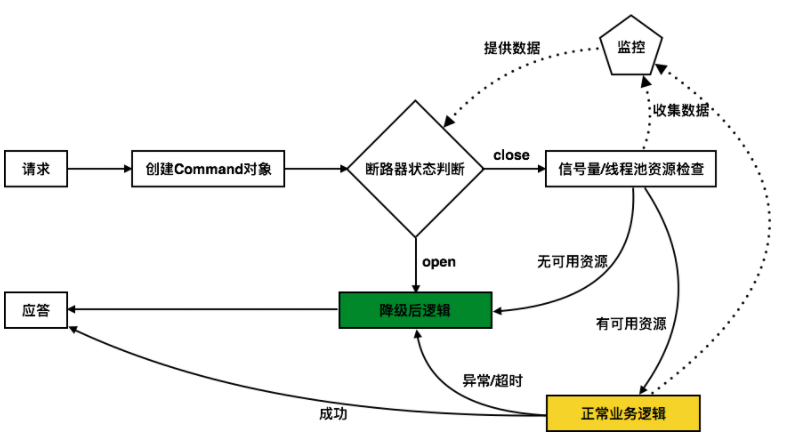
Hystrix基于命令模式，通过UML图



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。

## **Hystrix命令调用流程执行**

Hystrix使用命令模式HystrixCommand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，流程如下：



1. HystrixCommand的创建，初始化如下：

*HystrixCommand command = new HystrixCommand(arg1, arg2);*

对于HystrixObservalbeCommand返回Observable，其发送响应信息。

1. 命令的执行，有4种执行方法

* execute，从dependency获取响应信息，遇到问题是抛出error
* queue，返回Future
* observe，订阅Observable，从dependency中获取响应，并返回Observe
* toObservale，返回Observale，其执行Hystrix命令并发送响应

*K value = command.execute();*

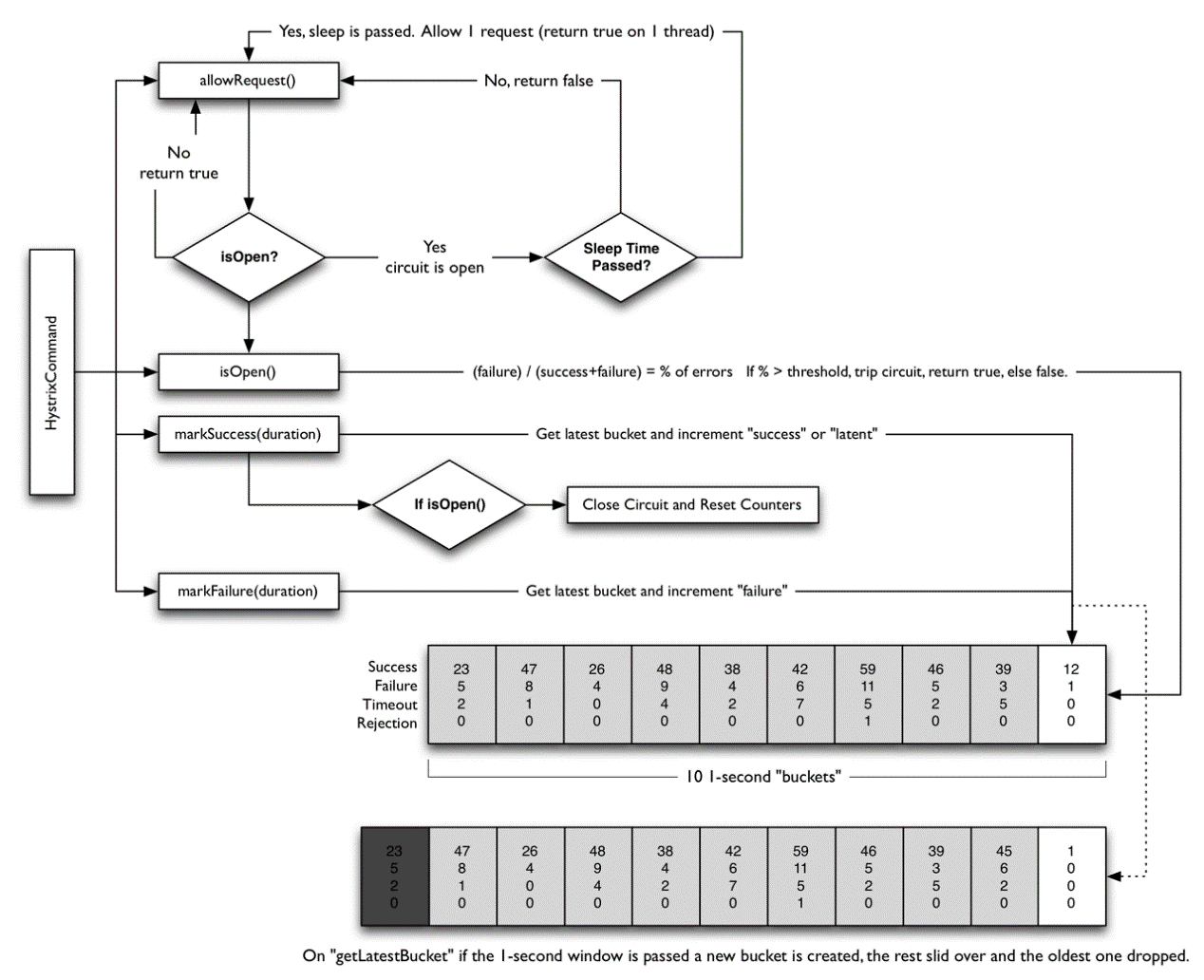
*Future<K> fValue = command.queue();*

*Observable<K> ohValue = command.observe(); //hot Observale*

*Observable<K> ocValue = command.toObservable(); //cold observabl*e

## **熔断器**

断路器的工作原理，如下图：



断路器开启或者关闭的条件

* 当满足一定阈值，默认10秒内超过20个请求次数
* 当失败率达到一定的时候
* 到达以上阈值，断路器就会开启

HystrixCircuitBreaker的类定义如下：

*public interface HystrixCircuitBreaker {*

*boolean allowRequest();*

*boolean isOpen();*

*void markSuccess();*

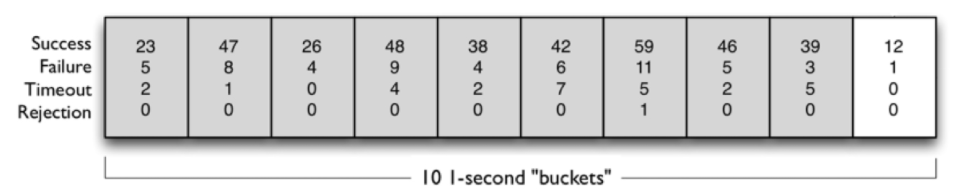
*void markNonSuccess();*

*boolean attemptExecution();*

*}*

## **滑动窗口**

断路器需要时间窗口请求量和错误率（固定窗口时间内的统计数据），根据这些统计数据决定下一步的行为，默认情况下滑动窗口包含10个桶，每个桶时间宽度是1秒，负责统计1秒内的数据统计，



每个小矩形代表一个桶，每个桶都记录4个指标：成功量、失败量、超时量和拒绝量（信号量/线程资源检查）中被拒绝的流量。10个桶合起来是完整的滑动窗口。桶的定义如下：

*static class Bucket {*

*final long windowStart;*

*final LongAdder[] adderForCounterType;*

*final LongMaxUpdater[] updaterForCounterType;*

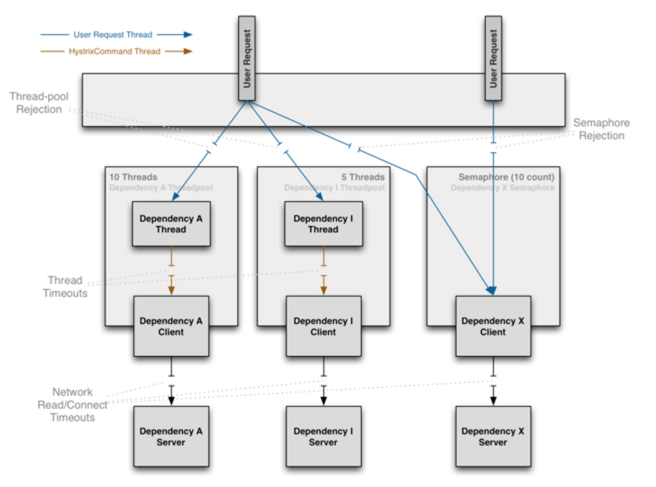
*}*

统计信息分为两类：

* 简单自增计数器，比如请求量，错误量
* 并发最大值

## **线程隔离**

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。线程隔离把执行依赖代码的线程和请求线程分离，请求线程可以自由控制离开的时间

****

在Hystrix中使用bulkhead pattern模式，采用多个线程池来管理线程，这样使得1个线程池资源出现问题时，不会造成另一个线程池资源问题。

参考链接：

https://www.cnblogs.com/duanxz/p/7521009.html

Hystrix状态转换： http://ifeve.com/hystrix/

https://github.com/Netflix/Hystrix/wiki/How-it-Works

http://www.importnew.com/25704.html

http://hot66hot.iteye.com/blog/2155036

https://www.cnblogs.com/gaoyanqing/p/7470085.html