

(/apps/redi
utm_sourc
banner-clic

Hystrix技术解析



新栋BOOK (/u/f2fa1bce6780) + 关注

2017.08.29 13:30* 字数 3997 阅读 5606 评论 0 喜欢 21

(/u/f2fa1bce6780)

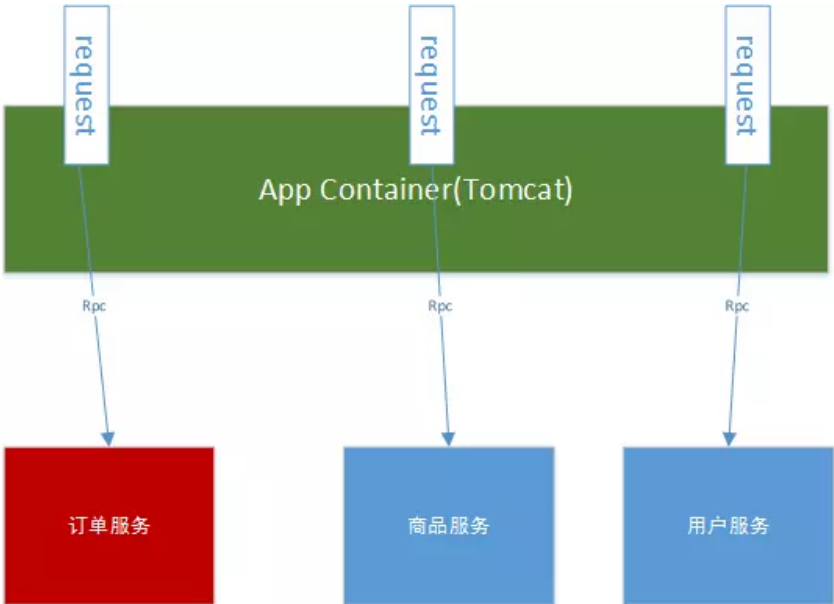
一、认识Hystrix

Hystrix是Netflix开源的一款容错框架，包含常用的容错方法：线程池隔离、信号量隔离、熔断、降级回退。在高并发访问下，系统所依赖的服务的稳定性对系统的影响非常大，依赖有很多不可控的因素，比如网络连接变慢，资源突然繁忙，暂时不可用，服务脱机等。我们要构建稳定、可靠的分布式系统，就必须要有这样一套容错方法。本文将逐一分析线程池隔离、信号量隔离、熔断、降级回退这四种技术的原理与实践。

二、线程隔离

2.1为什么要做线程隔离

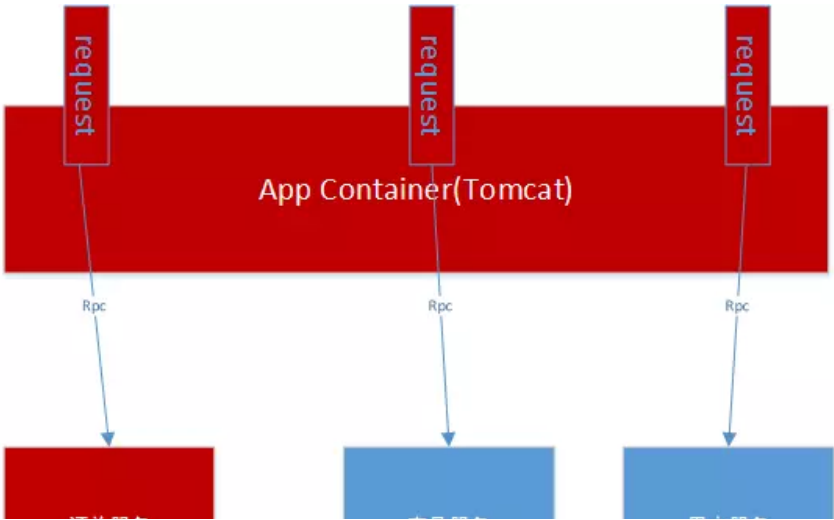
比如我们现在有3个业务调用分别是查询订单、查询商品、查询用户，且这三个业务请求都是依赖第三方服务-订单服务、商品服务、用户服务。三个服务均是通过RPC调用。当查询订单服务，假如线程阻塞了，这个时候后续有大量的查询订单请求过来，那么容器中的线程数量则会持续增加直致CPU资源耗尽到100%，整个服务对外不可用，集群环境下就是雪崩。如下图



订单服务不可用.png

^

🔗



(/apps/redi
utm_sourc
banner-clic

整个tomcat容器不可用.png

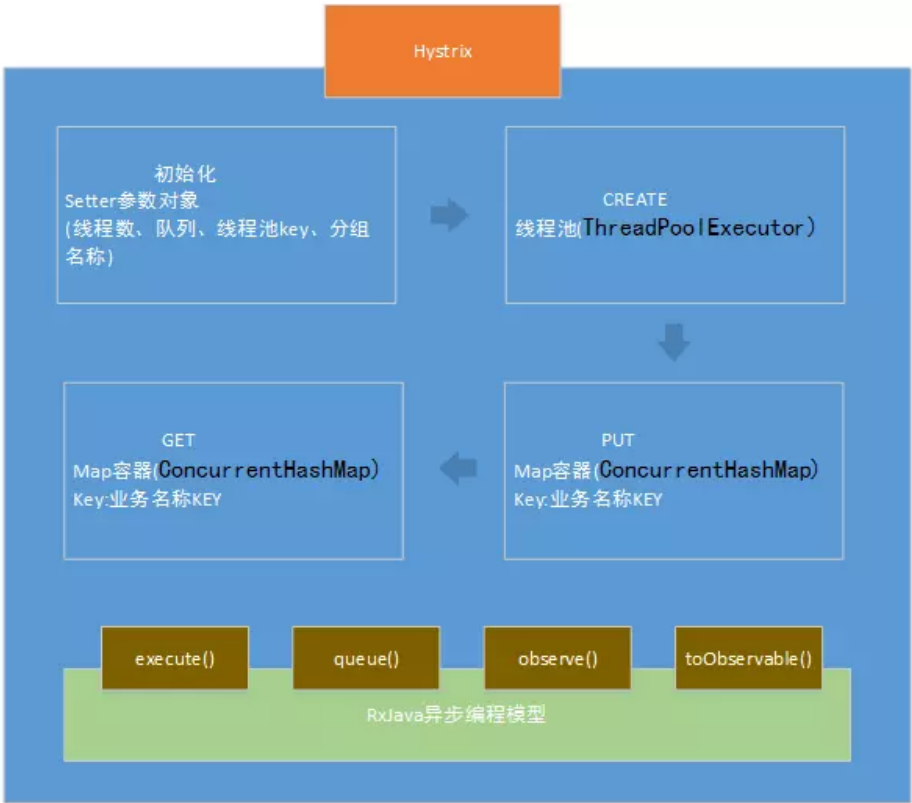
2.2、线程隔离-线程池

2.2.1、Hystrix是如何通过线程池实现线程隔离的

Hystrix通过命令模式，将每个类型的业务请求封装成对应的命令请求，比如查询订单-> 订单Command，查询商品->商品Command，查询用户->用户Command。每个类型的Command对应一个线程池。创建好的线程池是被放入到ConcurrentHashMap中，比如查询订单：

```
final static ConcurrentHashMap<String, HystrixThreadPool> threadPools = new Concurrent
threadPools.put("hystrix-order", new HystrixThreadPoolDefault(threadPoolKey, properti
```

当第二次查询订单请求过来的时候，则可以直接从Map中获取该线程池。具体流程如下
图：



Navigation icons: an upward arrow and a share icon.

```
(/apps/redi
utm_sourc
banner-clic
```

```
(/apps/redi
utm_sourc
banner-clic
```

```
(/apps/redi
utm_sourc
banner-clic
```

```
(/apps/redi
utm_sourc
banner-clic
```

```
(/apps/redi
utm_sourc
banner-clic
```

- ```
(/apps/redi
utm_sourc
banner-clic
```

将创建新线程非堵塞执行run()，调用程序不必等待run()；如果继承的是HystriObservableCommand，将以调用程序线程堵塞执行construct()，调用程序等待construct()执行完才能继续往下走)，如果run()/construct()执行成功则触发onNext()和onCompleted()，如果执行异常则触发onError()

注：

execute()和queue()是HystrixCommand中的方法，observe()和toObservable()是HystriObservableCommand 中的方法。从底层实现来讲，HystrixCommand其实也是利用Observable实现的（如果我们看Hystrix的源码的话，可以发现里面大量使用了RxJava），虽然HystrixCommand只返回单个的结果，但HystrixCommand的queue方法实际上是调用了toObservable().toBlocking().toFuture()，而execute方法实际上是调用了queue().get()。

(/apps/redi  
utm\_sourc  
banner-clc

### 2.2.2、如何应用到实际代码中

```
package myHystrix.threadpool;

import com.netflix.hystrix.*;
import org.junit.Test;

import java.util.List;
import java.util.concurrent.Future;

/**
 * Created by wangxindong on 2017/8/4.
 */
public class GetOrderCommand extends HystrixCommand<List> {

 OrderService orderService;

 public GetOrderCommand(String name){
 super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ThreadPoolTest"))
 .andCommandKey(HystrixCommandKey.Factory.asKey("testCommandKey"))
 .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey(name))
 .andCommandPropertiesDefaults(
 HystrixCommandProperties.Setter()
 .withExecutionTimeoutInMilliseconds(5000)
)
 .andThreadPoolPropertiesDefaults(
 HystrixThreadPoolProperties.Setter()
 .withMaxQueueSize(10) //配置队列大小
 .withCoreSize(2) // 配置线程池里的线程数
)
);
 }

 @Override
 protected List run() throws Exception {
 return orderService.getOrderList();
 }

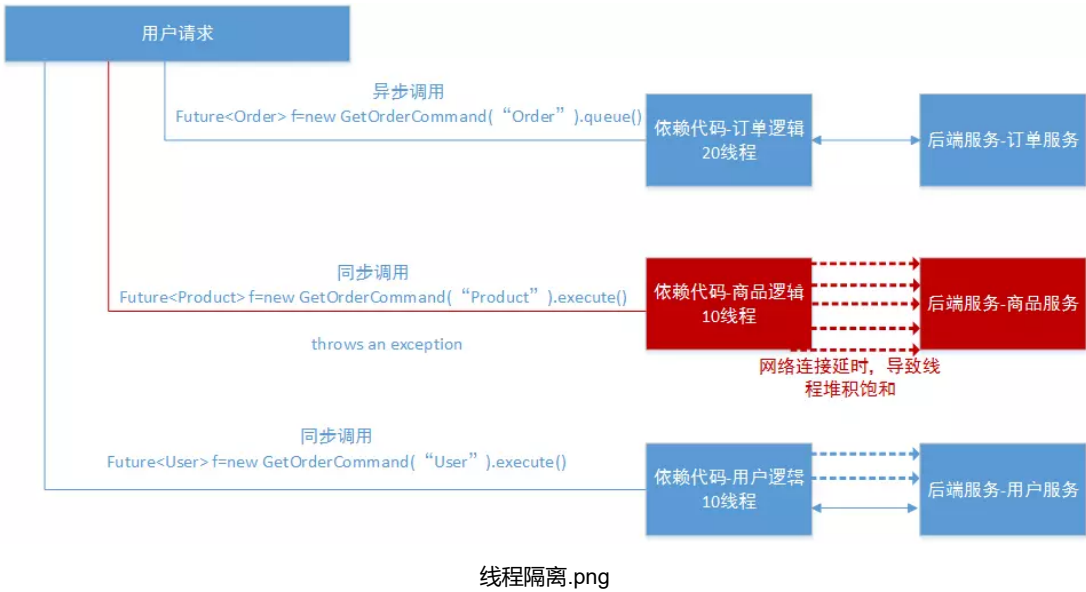
 public static class UnitTest {
 @Test
 public void testGetOrder(){
 // new GetOrderCommand("hystrix-order").execute();
 Future<List> future =new GetOrderCommand("hystrix-order").queue();
 }
 }
}
```

### 2.2.3、线程隔离-线程池小结



执行依赖代码的线程与请求线程(比如Tomcat线程)分离，请求线程可以自由控制离开的时间，这也是我们通常说的异步编程，Hystrix是结合RxJava来实现的异步编程。通过设置线程池大小来控制并发访问量，当线程饱和的时候可以拒绝服务，防止依赖问题扩散。

(/apps/redi  
utm\_sourc  
banner-lic



线程池隔离的优点:

- [1]:应用程序会被完全保护起来，即使依赖的一个服务的线程池满了，也不会影响到应用程序的其他部分。
- [2]:我们给应用程序引入一个新的风险较低的客户端lib的时候，如果发生问题，也是在本lib中，并不会影响到其他内容，因此我们可以大胆的引入新lib库。
- [3]:当依赖的一个失败的服务恢复正常时，应用程序会立即恢复正常的性能。
- [4]:如果我们的应用程序一些参数配置错误了，线程池的运行状况将会很快显示出来，比如延迟、超时、拒绝等。同时可以通过动态属性实时执行来处理纠正错误的参数配置。
- [5]:如果服务的性能有变化，从而需要调整，比如增加或者减少超时时间，更改重试次数，就可以通过线程池指标动态属性修改，而且不会影响到其他调用请求。
- [6]:除了隔离优势外，hystrix拥有专门的线程池可提供内置的并发功能，使得可以在同步调用之上构建异步的外观模式，这样就可以很方便的做异步编程（Hystrix引入了Rxjava异步框架）。

**尽管线程池提供了线程隔离，我们的客户端底层代码也必须要有超时设置，不能无限制的阻塞以致线程池一直饱和。**

线程池隔离的缺点:

- [1]:线程池的主要缺点就是它增加了计算的开销，每个业务请求（被包装成命令）在执行的时候，会涉及到请求排队，调度和上下文切换。不过Netflix公司内部认为线程隔离开销足够小，不会产生重大的成本或性能的影响。

The Netflix API processes 10+ billion Hystrix Command executions per day using thread isolation. Each API instance has 40+ thread-pools with 5–20 threads in each (most are set to 10).  
Netflix API每天使用线程隔离处理10亿次Hystrix Command执行。每个API实例都有40多个线程池，每个线程池中有5-20个线程（大多数设置为10个）。



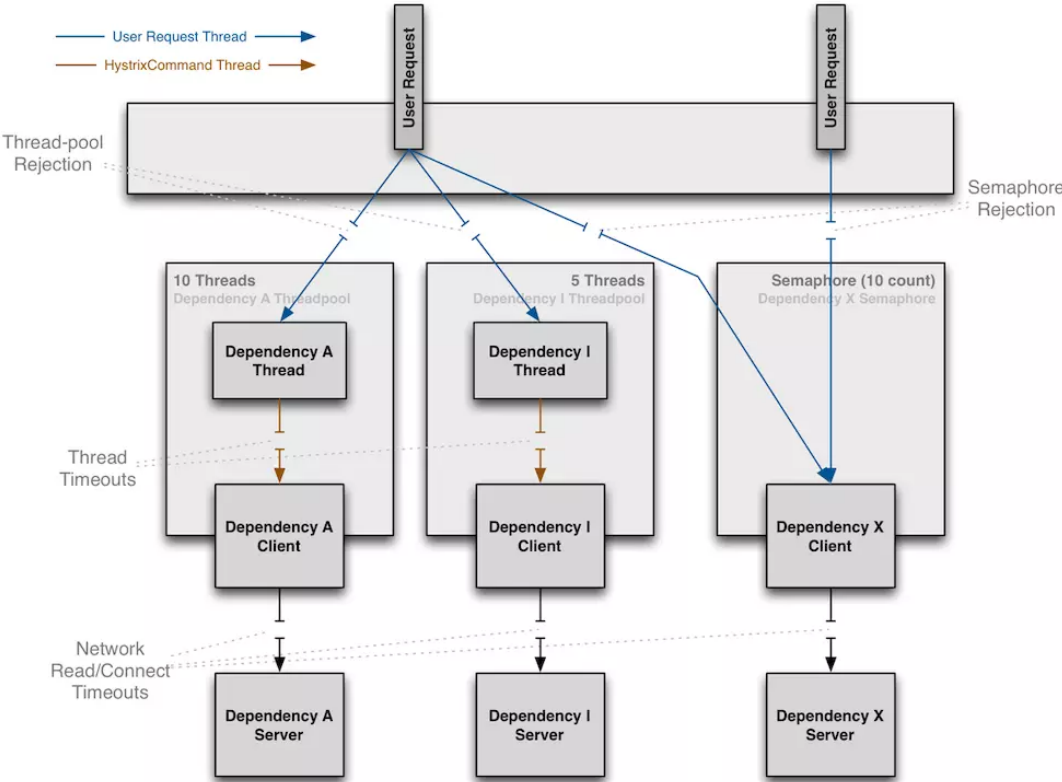
对于不依赖网络访问的服务，比如只依赖内存缓存这种情况下，就不适合用线程池隔离技术，而是采用信号量隔离。

2.3、线程隔离-信号量。

(/apps/redi  
utm\_sourc  
banner-clc

2.3.1、线程池和信号量的区别

上面谈到了线程池的缺点，当我们依赖的服务是极低延迟的，比如访问内存缓存，就没有必要使用线程池的方式，那样的话开销得不偿失，而是推荐使用信号量这种方式。下面这张图说明了线程池隔离和信号量隔离的主要区别：**线程池方式下业务请求线程和执行依赖的服务的线程不是同一个线程；信号量方式下业务请求线程和执行依赖服务的线程是同一个线程**



信号量和线程池的区别.png

2.3.2、如何使用信号量来隔离线程

将属性execution.isolation.strategy设置为SEMAPHORE，象这样  
ExecutionIsolationStrategy.SEMAPHORE，则Hystrix使用信号量而不是默认的线程池来做隔离。



(/apps/redi  
utm\_sourc  
banner-clic

```
public class CommandUsingSemaphoreIsolation extends HystrixCommand<String> {

 private final int id;

 public CommandUsingSemaphoreIsolation(int id) {
 super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"
 // since we're doing work in the run() method that doesn't involve ne
 // and executes very fast with low risk we choose SEMAPHORE isolation
 .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
 .withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)
 this.id = id;
 }

 @Override
 protected String run() {
 // a real implementation would retrieve data from in memory data structure
 // or some other similar non-network involved work
 return "ValueFromHashMap_" + id;
 }

}
```

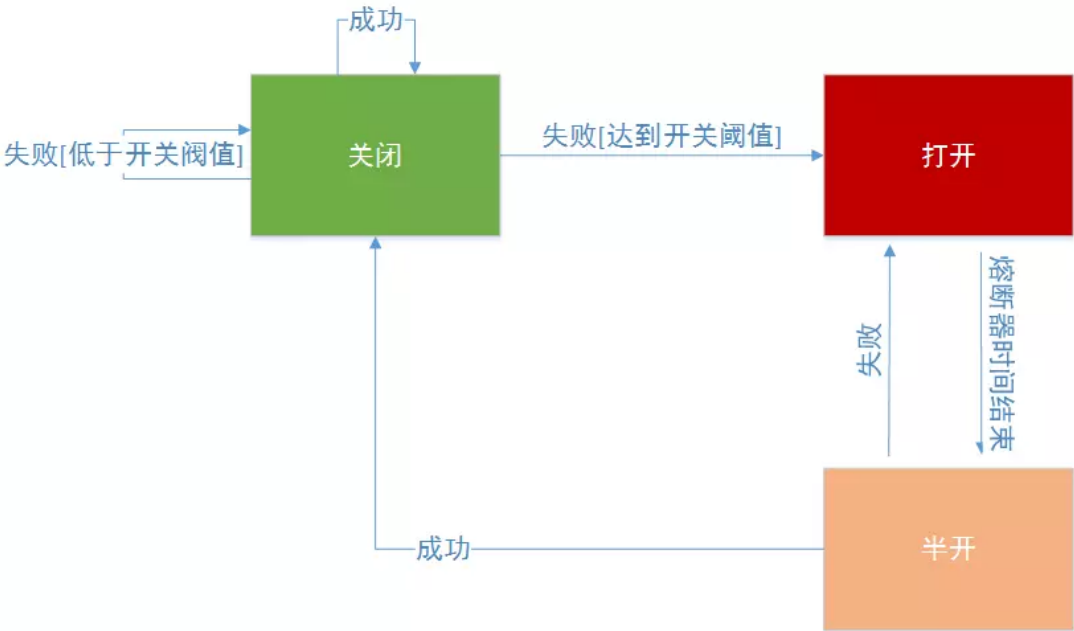
2.3.4、线程隔离-信号量小结

信号量隔离的方式是限制了总的并发数，每一次请求过来，请求线程和调用依赖服务的线程是同一个线程，那么如果不涉及远程RPC调用（没有网络开销）则使用信号量来隔离，更为轻量，开销更小。

三、熔断

3.1、熔断器(Circuit Breaker)介绍

熔断器，现实生活中有一个很好的类比，就是家庭电路中都会安装一个保险盒，当电流过大的时候保险盒里面的保险丝会自动断掉，来保护家里的各种电器及电路。Hystrix中的熔断器(Circuit Breaker)也是起到这样的作用，Hystrix在运行过程中会向每个commandKey对应的熔断器报告**成功、失败、超时和拒绝**的状态，熔断器维护计算统计的数据，根据这些统计的信息来确定熔断器是否打开。如果打开，后续的请求都会被截断。然后会隔一段时间默认是5s，尝试半开，放入一部分流量请求进来，相当于对依赖服务进行一次健康检查，如果恢复，熔断器关闭，随后完全恢复调用。如下图：

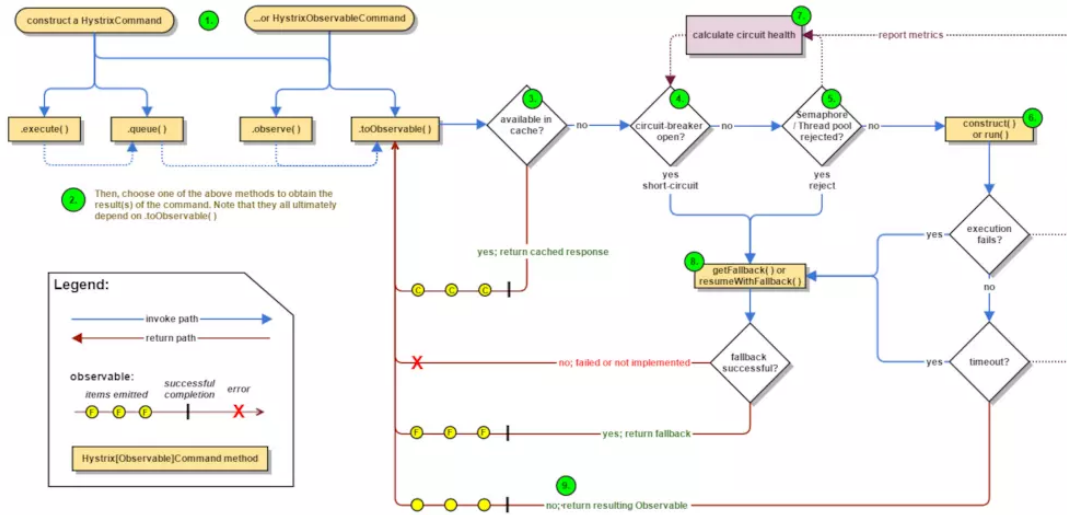


熔断器开关图.png

说明，上面说的commandKey，就是在初始化的时候设置的  
andCommandKey(HystrixCommandKey.Factory.asKey("testCommandKey"))

(/apps/redi  
utm\_sourc  
banner-clc

再来看下熔断器在整个Hystrix流程图中的位置，从步骤4开始，如下图：



Hystrix流程图.png

Hystrix会检查Circuit Breaker的状态。如果Circuit Breaker的状态为开启状态，Hystrix将不会执行对应指令，而是直接进入失败处理状态（图中8 Fallback）。如果Circuit Breaker的状态为关闭状态，Hystrix会继续进行线程池、任务队列、信号量的检查（图中5）

### 3.2、如何使用熔断器(Circuit Breaker)

由于Hystrix是一个容错框架，因此我们在使用的时候，要达到熔断的目的只需配置一些参数就可以了。但我们要达到真正的效果，就必须要了解这些参数。Circuit Breaker一共包括如下6个参数。

#### 1、circuitBreaker.enabled

是否启用熔断器，默认是TURE。

#### 2、circuitBreaker.forceOpen

熔断器强制打开，始终保持打开状态。默认值FLASE。

#### 3、circuitBreaker.forceClosed

熔断器强制关闭，始终保持关闭状态。默认值FLASE。

#### 4、circuitBreaker.errorThresholdPercentage

设定错误百分比，默认值50%，例如一段时间（10s）内有100个请求，其中有55个超时或者异常返回了，那么这段时间内的错误百分比是55%，大于了默认值50%，这种情况下触发熔断器-打开。

#### 5、circuitBreaker.requestVolumeThreshold

默认值20.意思是至少有20个请求才进行errorThresholdPercentage错误百分比计算。比如一段时间（10s）内有19个请求全部失败了。错误百分比是100%，但熔断器不会打开，因为requestVolumeThreshold的值是20。这个参数非常重要，熔断器是否打开首先要满足这个条件，源代码如下





```
// check if we are past the statisticalWindowVolumeThreshold
if (health.getTotalRequests() < properties.circuitBreakerRequestVolumeThreshold().get
 // we are not past the minimum volume threshold for the statisticalWindow so we'll
 return false;
}

if (health.getErrorPercentage() < properties.circuitBreakerErrorThresholdPercentage())
 return false;
}
```

(/apps/redi  
utm\_sourc  
banner-clic

## 6、circuitBreaker.sleepWindowInMilliseconds

半开试探休眠时间，默认值5000ms。当熔断器开启一段时间之后比如5000ms，会尝试放过去一部分流量进行试探，确定依赖服务是否恢复。

**测试代码（模拟10次调用，错误百分比为5%的情况下，打开熔断器开关。）：**



(/apps/redi  
utm\_sourc  
banner-clc

```
package myHystrix.threadpool;

import com.netflix.hystrix.*;
import org.junit.Test;

import java.util.Random;

/**
 * Created by wangxindong on 2017/8/15.
 */
public class GetOrderCircuitBreakerCommand extends HystrixCommand<String> {

 public GetOrderCircuitBreakerCommand(String name){
 super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ThreadPoolTest"))
 .andCommandKey(HystrixCommandKey.Factory.asKey("testCommandKey"))
 .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey(name))
 .andCommandPropertiesDefaults(
 HystrixCommandProperties.Setter()
 .withCircuitBreakerEnabled(true)//默认是true, 本例中为true
 .withCircuitBreakerForceOpen(false)//默认是false, 本例中为false
 .withCircuitBreakerForceClosed(false)//默认是false, 本例中为false
 .withCircuitBreakerErrorThresholdPercentage(5)//(1)错误率
 .withCircuitBreakerRequestVolumeThreshold(10)//(2)10秒内请求数
 .withCircuitBreakerSleepWindowInMilliseconds(5000)//(3)5秒
)
 .withExecutionTimeoutInMilliseconds(1000)
);
 .andThreadPoolPropertiesDefaults(
 HystrixThreadPoolProperties.Setter()
 .withMaxQueueSize(10) //配置队列大小
 .withCoreSize(2) // 配置线程池里的线程数
)
);
 }

 @Override
 protected String run() throws Exception {
 Random rand = new Random();
 //模拟错误百分比(方式比较粗鲁但可以证明问题)
 if(1==rand.nextInt(2)){
 //
 System.out.println("make exception");
 throw new Exception("make exception");
 }
 return "running: ";
 }

 @Override
 protected String getFallback() {
 //
 System.out.println("FAILBACK");
 return "fallback: ";
 }

 public static class UnitTest{

 @Test
 public void testCircuitBreaker() throws Exception{
 for(int i=0;i<25;i++){
 Thread.sleep(500);
 HystrixCommand<String> command = new GetOrderCircuitBreakerCommand("test");
 String result = command.execute();
 //本例子中从第11次, 熔断器开始打开
 System.out.println("call times:"+i+" result:"+result+" isCircuitBreakerOpen:"+command.isCircuitBreakerOpen());
 //本例子中5s以后, 熔断器尝试关闭, 放开新的请求进来
 }
 }
 }
}
```

测试结果：



(/apps/redi  
utm\_sourc  
banner-clc

call times:1 result:fallback: isCircuitBreakerOpen: false  
call times:2 result:running: isCircuitBreakerOpen: false  
call times:3 result:running: isCircuitBreakerOpen: false  
call times:4 result:fallback: isCircuitBreakerOpen: false  
call times:5 result:running: isCircuitBreakerOpen: false  
call times:6 result:fallback: isCircuitBreakerOpen: false  
call times:7 result:fallback: isCircuitBreakerOpen: false  
call times:8 result:fallback: isCircuitBreakerOpen: false  
call times:9 result:fallback: isCircuitBreakerOpen: false  
call times:10 result:fallback: isCircuitBreakerOpen: false

*熔断器打开*

call times:11 result:fallback: isCircuitBreakerOpen: true  
call times:12 result:fallback: isCircuitBreakerOpen: true  
call times:13 result:fallback: isCircuitBreakerOpen: true  
call times:14 result:fallback: isCircuitBreakerOpen: true  
call times:15 result:fallback: isCircuitBreakerOpen: true  
call times:16 result:fallback: isCircuitBreakerOpen: true  
call times:17 result:fallback: isCircuitBreakerOpen: true  
call times:18 result:fallback: isCircuitBreakerOpen: true  
call times:19 result:fallback: isCircuitBreakerOpen: true  
call times:20 result:fallback: isCircuitBreakerOpen: true

*5s后熔断器关闭*

call times:21 result:running: isCircuitBreakerOpen: false  
call times:22 result:running: isCircuitBreakerOpen: false  
call times:23 result:fallback: isCircuitBreakerOpen: false  
call times:24 result:running: isCircuitBreakerOpen: false  
call times:25 result:running: isCircuitBreakerOpen: false

### 3.3、熔断器(Circuit Breaker)源代码HystrixCircuitBreaker.java分析

```
<<接口>>
HystrixCircuitBreaker.java

public boolean allowRequest();
public boolean isOpen();
void markSuccess();
public static class Factory{}
static class HystrixCircuitBreakerImpl implements HystrixCircuitBreaker {}
```

HystrixCircuitBreaker.java.png

**Factory 是一个工厂类，提供HystrixCircuitBreaker实例**



```
public static class Factory {
 //用一个ConcurrentHashMap来保存HystrixCircuitBreaker对象
 private static ConcurrentHashMap<String, HystrixCircuitBreaker> circuitBreakersByCommand;

 //Hystrix首先会检查ConcurrentHashMap中有没有对应的缓存的断路器，如果有的话直接返回。如果没有的话，就创建一个新的断路器。
 public static HystrixCircuitBreaker getInstance(HystrixCommandKey key, HystrixCommandProperties properties) {
 HystrixCircuitBreaker previouslyCached = circuitBreakersByCommand.get(key.name());
 if (previouslyCached != null) {
 return previouslyCached;
 }

 HystrixCircuitBreaker cbForCommand = circuitBreakersByCommand.putIfAbsent(key.name(), new HystrixCircuitBreaker(key, properties));
 if (cbForCommand == null) {
 return circuitBreakersByCommand.get(key.name());
 } else {
 return cbForCommand;
 }
 }

 public static HystrixCircuitBreaker getInstance(HystrixCommandKey key) {
 return circuitBreakersByCommand.get(key.name());
 }

 static void reset() {
 circuitBreakersByCommand.clear();
 }
}
```

(/apps/redi  
utm\_sourc  
banner-clic

**HystrixCircuitBreakerImpl是HystrixCircuitBreaker的实现，allowRequest()、isOpen()、markSuccess()都会在HystrixCircuitBreakerImpl有默认的实现。**



(/apps/redi  
utm\_sourc  
banner-clip

```

static class HystrixCircuitBreakerImpl implements HystrixCircuitBreaker {
 private final HystrixCommandProperties properties;
 private final HystrixCommandMetrics metrics;

 /* 变量circuitOpen来代表断路器的状态，默认是关闭 */
 private AtomicBoolean circuitOpen = new AtomicBoolean(false);

 /* 变量circuitOpenedOrLastTestedTime记录着断路恢复计时器的初始时间，用于Open状态时
 private AtomicLong circuitOpenedOrLastTestedTime = new AtomicLong();

 protected HystrixCircuitBreakerImpl(HystrixCommandKey key, HystrixCommandGroupKey groupKey) {
 this.properties = properties;
 this.metrics = metrics;
 }

 /*用于关闭熔断器并重置统计数据*/
 public void markSuccess() {
 if (circuitOpen.get()) {
 if (circuitOpen.compareAndSet(true, false)) {
 //win the thread race to reset metrics
 //Unsubscribe from the current stream to reset the health counts
 //and all other metric consumers are unaffected by the reset
 metrics.resetStream();
 }
 }
 }

 @Override
 public boolean allowRequest() {
 //是否设置强制开启
 if (properties.circuitBreakerForceOpen().get()) {
 return false;
 }
 if (properties.circuitBreakerForceClosed().get()) { //是否设置强制关闭
 isOpen();
 // properties have asked us to ignore errors so we will ignore the re
 return true;
 }
 return !isOpen() || allowSingleTest();
 }

 public boolean allowSingleTest() {
 long timeCircuitOpenedOrWasLastTested = circuitOpenedOrLastTestedTime.get()
 //获取熔断恢复计时器记录的初始时间circuitOpenedOrLastTestedTime，然后判断以下
 // 1) 熔断器的状态为开启状态(circuitOpen.get() == true)
 // 2) 当前时间与计时器初始时间之差大于计时器阈值circuitBreakerSleepWindowInMs
 //如果同时满足的话，表示可以从Open状态向Close状态转换。Hystrix会通过CAS操作将c
 if (circuitOpen.get() && System.currentTimeMillis() > timeCircuitOpenedOrWasLastTested +
 // We push the 'circuitOpenedTime' ahead by 'sleepWindow' since we ha
 // If it succeeds the circuit will be closed, otherwise another singl
 if (circuitOpenedOrLastTestedTime.compareAndSet(timeCircuitOpenedOrWasLastTested,
 // if this returns true that means we set the time so we'll retur
 // if it returned false it means another thread raced us and allo
 return true;
 }
 }
 return false;
 }

 @Override
 public boolean isOpen() {
 if (circuitOpen.get()) { //获取断路器的状态
 // if we're open we immediately return true and don't bother attempti
 return true;
 }

 // Metrics数据中获取HealthCounts对象
 HealthCounts health = metrics.getHealthCounts();

 // 检查对应的请求总数(totalCount)是否小于属性中的请求容量阈值circuitBreakerRequestVolumeThreshold
 if (health.getTotalRequests() < properties.circuitBreakerRequestVolumeThreshold) {
 return false;
 }
 }
}

```





```
@Override
protected String run() {
 if (throwException) {
 throw new RuntimeException("failure from CommandThatFailsFast");
 } else {
 return "success";
 }
}
```

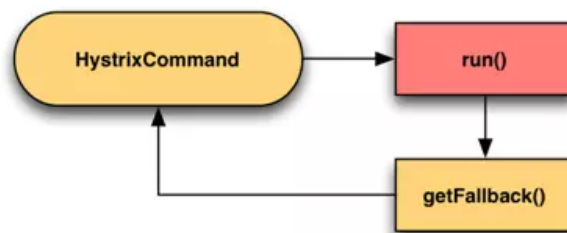
(/apps/redi  
utm\_sourc  
banner-clc

如果我们实现的是HystrixObservableCommand.java则 重写 resumeWithFallback方法

```
@Override
protected Observable<String> resumeWithFallback() {
 if (throwException) {
 return Observable.error(new Throwable("failure from CommandThatFailsFast"));
 } else {
 return Observable.just("success");
 }
}
```

#### 4.2.2、Fail Silent 无声失败

返回null，空Map，空List



fail silent.png

```
@Override
protected String getFallback() {
 return null;
}
@Override
protected List<String> getFallback() {
 return Collections.emptyList();
}
@Override
protected Observable<String> resumeWithFallback() {
 return Observable.empty();
}
```

#### 4.2.3、Fallback: Static 返回默认值

回退的时候返回静态嵌入代码中的默认值，这样就不会导致功能以Fail Silent的方式被清楚，也就是用户看不到任何功能了。而是按照一个默认的方式显示。



```
@Override
protected Boolean getFallback() {
 return true;
}

@Override
protected Observable<Boolean> resumeWithFallback() {
 return Observable.just(true);
}
```

(/apps/redi  
utm\_sourc  
banner-clc

#### 4.2.4、Fallback: Stubbed 自己组装一个值返回

当我们执行返回的结果是一个包含多个字段的对象时，则会以Stubbed 的方式回退。Stubbed 值我们建议在实例化Command的时候就设置好一个值。以countryCodeFromGeoLookup为例，countryCodeFromGeoLookup的值，是在我们调用的时候就注册进来初始化好的。CommandWithStubbedFallback command = new CommandWithStubbedFallback(1234, "china");主要代码如下：

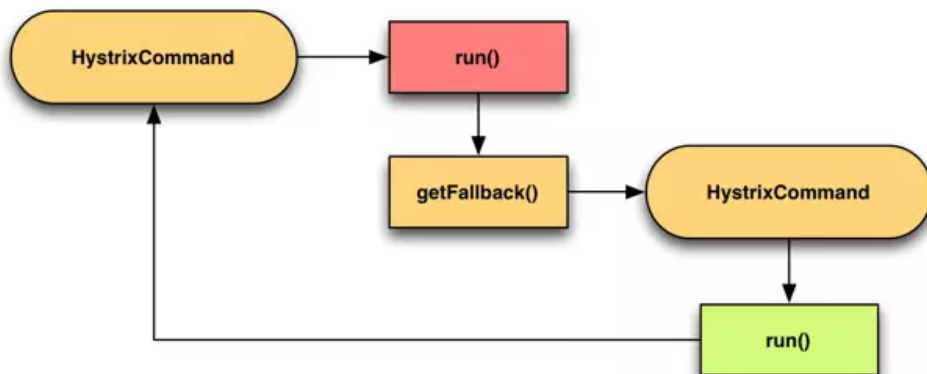
```
public class CommandWithStubbedFallback extends HystrixCommand<UserAccount> {

 protected CommandWithStubbedFallback(int customerId, String countryCodeFromGeoLookup) {
 super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
 this.customerId = customerId;
 this.countryCodeFromGeoLookup = countryCodeFromGeoLookup;
 }

 @Override
 protected UserAccount getFallback() {
 /**
 * Return stubbed fallback with some static defaults, placeholders,
 * and an injected value 'countryCodeFromGeoLookup' that we'll use
 * instead of what we would have retrieved from the remote service.
 */
 return new UserAccount(customerId, "Unknown Name",
 countryCodeFromGeoLookup, true, true, false);
 }
}
```

#### 4.2.5、Fallback: Cache via Network 利用远程缓存

通过远程缓存的方式。在失败的情况下再发起一次remote请求，不过这次请求的是一个缓存比如redis。由于是又发起一起远程调用，所以会重新封装一次Command，这个时候要注意，执行fallback的线程一定要跟主线程区分开，也就是重新命名一个ThreadPoolKey。



Cache via Network.png





```

public class CommandWithFallbackViaNetwork extends HystrixCommand<String> {
 private final int id;

 protected CommandWithFallbackViaNetwork(int id) {
 super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceXClient")
 .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueCommand"))));
 this.id = id;
 }

 @Override
 protected String run() {
 // RemoteServiceXClient.getValue(id);
 throw new RuntimeException("force failure for example");
 }

 @Override
 protected String getFallback() {
 return new FallbackViaNetwork(id).execute();
 }

 private static class FallbackViaNetwork extends HystrixCommand<String> {
 private final int id;

 public FallbackViaNetwork(int id) {
 super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceXClient")
 .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueFallbackViaNetwork"))
 // use a different threadpool for the fallback command
 // so saturating the RemoteServiceX pool won't prevent
 // fallbacks from executing
 .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("RemoteServiceXClientFallbackViaNetwork"))));
 this.id = id;
 }

 @Override
 protected String run() {
 MemCacheClient.getValue(id);
 }

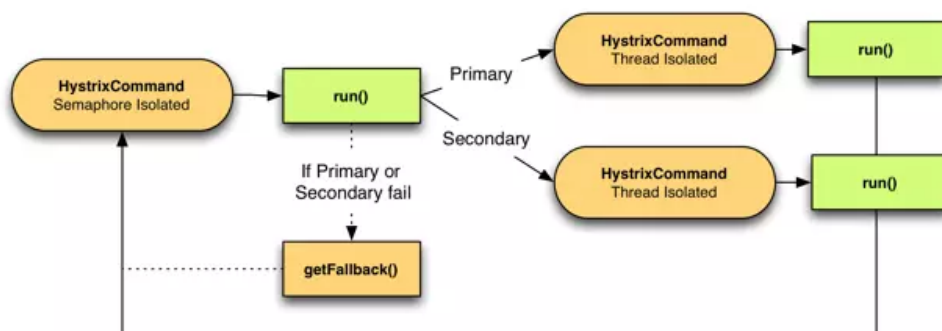
 @Override
 protected String getFallback() {
 // the fallback also failed
 // so this fallback-of-a-fallback will
 // fail silently and return null
 return null;
 }
 }
}

```

(/apps/redi  
utm\_sourc  
banner-clic

#### 4.2.6、Primary + Secondary with Fallback 主次方式回退（主要和次要）

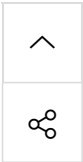
这个有点类似我们日常开发中需要上线一个新功能，但为了防止新功能上线失败可以回退到老的代码，我们会做一个开关比如使用zookeeper做一个配置开关，可以动态切换到老代码功能。那么Hystrix它是使用通过一个配置来在两个command中进行切换。



Primary + Secondary with Fallback.png

---

(/apps/redi  
utm\_sourc  
banner-clic



```

/**
 * Sample {@link HystrixCommand} pattern using a semaphore-isolated command
 * that conditionally invokes thread-isolated commands.
 */
public class CommandFacadeWithPrimarySecondary extends HystrixCommand<String> {

 private final static DynamicBooleanProperty usePrimary = DynamicPropertyFactory.<Boolean>.get("hystrix.command.default.usePrimary");

 private final int id;

 public CommandFacadeWithPrimarySecondary(int id) {
 super(Setter
 .withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
 .andCommandKey(HystrixCommandKey.Factory.asKey("PrimarySecondaryCommand"))
 .andCommandPropertiesDefaults(
 // we want to default to semaphore-isolation since this wraps
 // 2 others commands that are already thread isolated
 // 采用信号量的隔离方式
 HystrixCommandProperties.Setter()
 .withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)
)
);
 this.id = id;
 }

 //通过DynamicPropertyFactory来路由到不同的command
 @Override
 protected String run() {
 if (usePrimary.get()) {
 return new PrimaryCommand(id).execute();
 } else {
 return new SecondaryCommand(id).execute();
 }
 }

 @Override
 protected String getFallback() {
 return "static-fallback-" + id;
 }

 @Override
 protected String getCacheKey() {
 return String.valueOf(id);
 }

 private static class PrimaryCommand extends HystrixCommand<String> {

 private final int id;

 private PrimaryCommand(int id) {
 super(Setter
 .withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
 .andCommandKey(HystrixCommandKey.Factory.asKey("PrimaryCommand"))
 .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("PrimaryCommand"))
 .andCommandPropertiesDefaults(
 // we default to a 600ms timeout for primary
 HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(600)
)
);
 this.id = id;
 }

 @Override
 protected String run() {
 // perform expensive 'primary' service call
 return "responseFromPrimary-" + id;
 }
 }

 private static class SecondaryCommand extends HystrixCommand<String> {

 private final int id;

 private SecondaryCommand(int id) {
 super(Setter
 .withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
);
 this.id = id;
 }

 @Override
 protected String run() {
 // perform expensive 'secondary' service call
 return "responseFromSecondary-" + id;
 }
 }
}

```

(/apps/redi  
utm\_sourc  
banner-clic



```

 .andCommandKey(HystrixCommandKey.Factory.asKey("SecondaryCommand"))
 .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("SecondaryCommand"))
 .andCommandPropertiesDefaults(
 // we default to a 100ms timeout for secondary
 HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(100)
);
 this.id = id;
 }

 @Override
 protected String run() {
 // perform fast 'secondary' service call
 return "responseFromSecondary-" + id;
 }
}

public static class UnitTest {

 @Test
 public void testPrimary() {
 HystrixRequestContext context = HystrixRequestContext.initializeContext();
 try {
 //将属性"primarySecondary.usePrimary"设置为true, 则走PrimaryCommand; 设置
 ConfigurationManager.getConfigInstance().setProperty("primarySecondary.usePrimary", true);
 assertEquals("responseFromPrimary-20", new CommandFacadeWithPrimarySecondary().execute());
 } finally {
 context.shutdown();
 ConfigurationManager.getConfigInstance().clear();
 }
 }

 @Test
 public void testSecondary() {
 HystrixRequestContext context = HystrixRequestContext.initializeContext();
 try {
 //将属性"primarySecondary.usePrimary"设置为true, 则走PrimaryCommand; 设置
 ConfigurationManager.getConfigInstance().setProperty("primarySecondary.usePrimary", true);
 assertEquals("responseFromSecondary-20", new CommandFacadeWithPrimarySecondary().execute());
 } finally {
 context.shutdown();
 ConfigurationManager.getConfigInstance().clear();
 }
 }
}
}

```

(/apps/redi  
utm\_sourc  
banner-clic

### 4.3、回退降级小结

降级的处理方式，返回默认值，返回缓存里面的值（包括远程缓存比如redis和本地缓存比如jvmcache）。

但回退的处理方式也有不适合的场景：

- 1、写操作
- 2、批处理
- 3、计算

以上几种情况如果失败，则程序就要将错误返回给调用者。

## 总结

Hystrix为我们提供了一套线上系统容错的技术实践方法，我们通过在系统中引入Hystrix的jar包可以很方便的使用线程隔离、熔断、回退等技术。同时它还提供了监控页面配置，方便我们管理查看每个接口的调用情况。像spring cloud这种微服务构建模式中也引入了Hystrix，我们可以放心使用Hystrix的线程隔离技术，来防止雪崩这种可怕的致命性线上故障。



转载请注明出处，并附上链接  
<http://www.jianshu.com/p/3e11ac385c73>  
(<https://www.jianshu.com/p/3e11ac385c73>)

(/apps/redi  
utm\_sourc  
banner-clic

参考资料：  
<https://github.com/Netflix/Hystrix/wiki> (<https://link.jianshu.com?t=https://github.com/Netflix/Hystrix/wiki>)  
《亿级流量网站架构核心技术》一书

 Hystrix (/nb/15707608) 举报文章 © 著作权归作者所有



新栋BOOK (/u/f2fa1bce6780) ♂

写了 113116 字，被 438 人关注，获得了 387 个喜欢

(/u/f2fa1bce6780)


+ 关注

写文字是一种爱好，我的书籍作品：《架构修炼之道》，参与著作《决战618：探秘京东技术取胜之道》添...

喜欢 | 21



更多分享



登录后发表评论 (/sign\_in?utm\_source=desktop&utm\_medium=not-signed-in-comment)

评论


智慧如你，不想发表一点想法 (/sign\_in?utm\_source=desktop&utm\_medium=not-signed-in-nocomments-text)咩~


被以下专题收入，发现更多相似内容


-  程序员 (/c/NEt52a?utm\_source=desktop&utm\_medium=notes-included-collection)
-  @IT·互联网 (/c/V2CqjW?utm\_source=desktop&utm\_medium=notes-included-collection)
-  首页投稿（暂停... (/c/bDHhpK?utm\_source=desktop&utm\_medium=notes-included-collection)


^



 好文章 (/c/fbdc49549a37?utm\_source=desktop&utm\_medium=notes-included-collection)

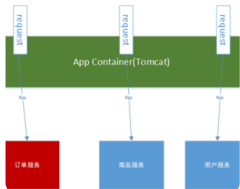
 Spring学习教程 (/c/06ecbc5c3b41?utm\_source=desktop&utm\_medium=notes-included-collection)

 污力\_Java (/c/b661c20f74f6?utm\_source=desktop&utm\_medium=notes-included-collection)

 Spring ... (/c/807a00bf454b?utm\_source=desktop&utm\_medium=notes-included-collection)

(/apps/redi  
utm\_sourc  
banner-lic

(/p/b6e8d91b2a96?



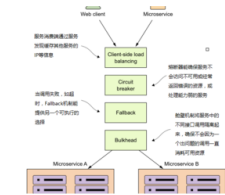
utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio  
**Hystrix技术解析(图片版) (/p/b6e8d91b2a96?utm\_campaign=maleskine...**

一、认识Hystrix Hystrix是Netflix开源的一款容错框架，包含常用的容错方法：线程池隔离、信号量隔离、熔断、降级回退。在高并发访问下，系统所依赖的服务的稳定性对系统的影响非常大，依赖有很多不可控的...

 新栋BOOK (/u/f2fa1bce6780?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio


(/p/6c574abe50c1?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio  
**服务容错保护——Spring Cloud Hystrix (/p/6c574abe50c1?utm\_campa...**

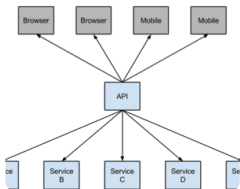
(git上的源码：https://gitee.com/rain7564/spring\_microservices\_study/tree/master/forth-spring-cloud-hystrix)

几乎每一个系统，特别是分布式系统，都会有调用失败的情况，最有效的办法...

 sprinkle\_liz (/u/c13f3c6ada04?)


utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio

(/p/46fd0faecac1?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio  
**Spring Cloud (/p/46fd0faecac1?utm\_campaign=maleskine&utm\_conte...**

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线）。分布式系统的协调导致了样板模式，使用Spring Cloud开发人员可...

 卡卡罗2017 (/u/d90908cb0d85?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio

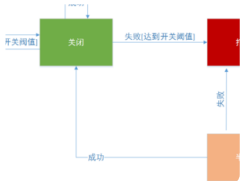


(/p/14958039fd15?

utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio

Hystrix熔断器技术解析-HystrixCircuitBreaker (/p/14...

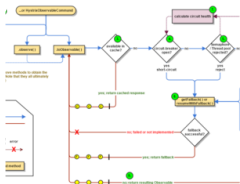
一、熔断器(Circuit Breaker)介绍 熔断器，现实生活中有一个很好的类比，就是家庭电路中都会安装一个保险盒，当电流过大的时候保险盒里面的保险丝会自...



 新栋BOOK (/u/f2fa1bce6780?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio (/apps/redi  
utm\_sourc  
banner-clic


(/p/b9af028efebb?)



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio

Hystrix使用入门手册（中文） (/p/b9af028efebb?utm\_campaign=malesk...


导语：网上资料（尤其中文文档）对hystrix基础功能的解释比较笼统，看了往往一头雾水。为此，本文将通过若干demo，加入对官网How-it-Works的理解和翻译，力求更清晰解释hystrix的基础功能。所用demo均...

 star24 (/u/0d0c633a5494?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio

我所理解的《死亡诗社》 (/p/9ec588257fdc?utm\_campaign=maleskine&...

观看电影《死亡诗社》，感慨良多。基丁老师是一个充满人文主义关怀的老师，鼓励和教授一个班里的男孩子们念诗写诗，用慈父般的态度去关怀自己的学生，与严厉顽固的校长形成鲜明的对比，这对我对当下的...

 伯鱼 (/u/ac5e8b9e93dd?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio


(/p/73bfb23bf1d4?)



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio

【101】反思整理 (/p/73bfb23bf1d4?utm\_campaign=maleskine&utm\_co...

男友说人是需要经常反思自己的，我深以为然。我觉得自己有一些习惯很奇怪。比如我愿意早起自己做早餐、愿意在家吃早餐，周末我常常去菜市买菜，洗切煮的花上两三个小时做三五道菜，比如厨房里用脏...

 何雨桐 (/u/121cfbaaf855?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio


(/p/e2c018b3cd5c?)



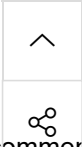
utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendatio

赵薇亲身经验告诉你：“只有脸是不行的” (/p/e2c018b3cd5c?utm\_campai...

最近赵薇因参加综艺节目“中餐厅”备受关注，不少人发现女神重回颜值巅峰，嗯~在节目中的赵薇又美回来了~要知道，以前参加活动时，一身粉色系套装却被称为“乡村女企业家”。不仅仅胖了一圈，穿衣搭配简直是...


 潮流一起说 (/u/3c370eaef652?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendatio



Animation's Brother ---- Animator (/p/16c05faa5e15?utm\_campaign=m...

动画进阶之 Animator 上篇文章学习了Animation的原理,也许细心的人发现了,Animation只是在画布(Canvas)上不断的重绘,那么这样就会出现一个问题, 我们动画结束后,原来的view属性没有变化,假如一个button经过...

 UniGenius\_Mx (/u/7a775fd771fd?utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation\_banner-click)

(/apps/redi  
utm\_sourc  
banner-clic

