Hystrix使用入门手册（中文）

> 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均对官网How-To-Use进行了二次修改，见https://github.com/star2478/java-hystrix

Hystrix是Netflix开源的一款容错系统，能帮助使用者码出具备强大的容错能力和鲁棒性的程序。如果某程序或class要使用Hystrix，只需简单继承 `HystrixCommand/HystrixObservableCommand` 并重写 `run()/construct()` ，然后调用程序实例化此class并执行 `execute()/queue()/observe()/toObservable()` 。

```
public class HelloWorldHystrixCommand extends HystrixCommand {
    private final String name;
    public HelloWorldHystrixCommand(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        return "Hello " + name;
    }
}


String result = new HelloWorldHystrixCommand("HLX").execute();
System.out.println(result);
```

pom.xml加上以下依赖。spring cloud也集成了hystrix，不过本文只介绍原生hystrix。

```
<dependency>
  <groupId>com.netflix.hystrix</groupId>
  <artifactId>hystrix-core</artifactId>
  <version>1.5.8</version>
</dependency>
```

> 本文重点介绍的是Hystrix各项基础能力的用法及其效果，不从零介绍hystrix，要了解基础知识推荐官网wiki或民间blog

1、HystrixCommand vs HystrixObservableCommand

要想使用hystrix，只需要继承 `HystrixCommand` 或 `HystrixObservableCommand` ，简单用法见上面例子。两者主要区别是：

- 前者的命令逻辑写在 `run()` ；后者的命令逻辑写在 `construct()`
- 前者的 `run()` 是由新创建的线程执行；后者的 `construct()` 是由调用程序线程执行

- 前者一个实例只能向调用程序发送（emit）单条数据，比如上面例子中`run()`只能返回一个String结果；后者一个实例可以顺序发送多条数据，比如demo中顺序调用多个`onNext()`，便实现了向调用程序发送多条数据，甚至还能发送一个范围的数据集

2、4个命令执行方法

`execute()`、`queue()`、`observe()`、`toObservable()`这4个方法用来触发执行`run()/construct()`，一个实例只能执行一次这4个方法，特别说明的是`HystrixObservableCommand`没有`execute()`和`queue()`。

4个方法的主要区别是：

- `execute()`：以同步堵塞方式执行`run()`。以demo为例，调用`execute()`后，hystrix先创建一个新线程运行`run()`，接着调用程序要在`execute()`调用处一直堵塞着，直到`run()`运行完成
- `queue()`：以异步非堵塞方式执行`run()`。以demo为例，一调用`queue()`就直接返回一个Future对象，同时hystrix创建一个新线程运行`run()`，调用程序通过`Future.get()`拿到`run()`的返回结果，而`Future.get()`是堵塞执行的
- `observe()`：事件注册前执行`run()/construct()`。以demo为例，第一步是事件注册前，先调用`observe()`自动触发执行`run()/construct()`（如果继承的是`HystrixCommand`，hystrix将创建新线程非堵塞执行`run()`；如果继承的是`HystrixObservableCommand`，将以调用程序线程堵塞执行`construct()`），第二步是从`observe()`返回后调用程序调用`subscribe()`完成事件注册，如果`run()/construct()`执行成功则触发`onNext()`和`onCompleted()`，如果执行异常则触发`onError()`
- `toObservable()`：事件注册后执行`run()/construct()`。以demo为例，第一步是事件注册前，一调用`toObservable()`就直接返回一个`Observable<String>`对象，第二步调用`subscribe()`完成事件注册后自动触发执行`run()/construct()`（如果继承的是`HystrixCommand`，hystrix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`HystrixObservableCommand`，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()`

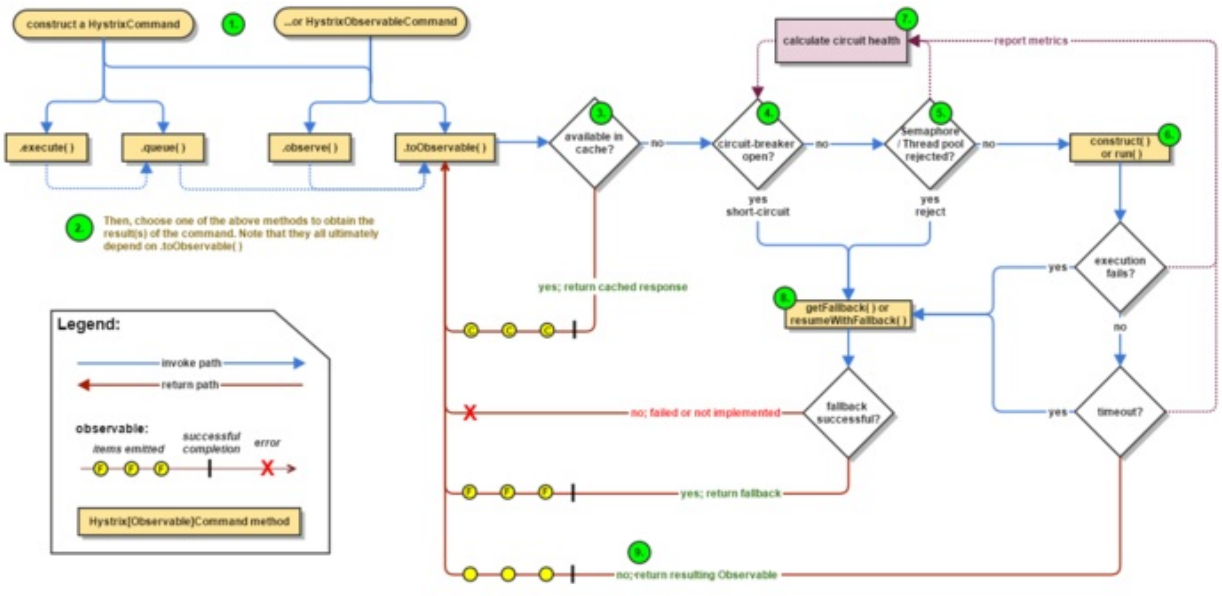
3、fallback（降级）

使用fallback机制很简单，继承`HystrixCommand`只需重写`getFallback()`，继承`HystrixObservableCommand`只需重写`resumeWithFallback()`，比如`HelloWorldHystrixCommand`加上下面代码片段：

```
@Override
protected String getFallback() {
    return "fallback: " + name;
}
```

fallback实际流程是当`run()/construct()`被触发执行时或执行中发生错误时，将转向执行`getFallback()/resumeWithFallback()`。结合下图，4种错误情况将触发fallback：

- 非HystrixBadRequestException异常：以demo为例，当抛出HystrixBadRequestException时，调用程序可以捕获异常，没有触发`getFallback()`，而其他异常则会触发`getFallback()`，调用程序将获得`getFallback()`的返回
- `run()/construct()`运行超时：以demo为例，使用无限while循环或sleep模拟超时，触发了`getFallback()`
- 熔断器启动：以demo为例，我们配置10s内请求数大于3个时就启动熔断器，请求错误率大于80%时就熔断，然后for循环发起请求，当请求符合熔断条件时将触发`getFallback()`。更多熔断策略见下文
- 线程池/信号量已满：以demo为例，我们配置线程池数目为3，然后先用一个for循环执行`queue()`，触发的`run()`sleep 2s，然后再用第2个for循环执行`execute()`，发现所有`execute()`都触发了fallback，这是因为第1个for的线程还在sleep，占用着线程池所有线程，导致第2个for的所有命令都无法获取到线程



来自hystrix github wiki

> 调用程序可以通过`isResponseFromFallback()`查询结果是由`run()/construct()`还是`getFallback()/resumeWithFallback()`返回的

4、隔离策略

hystrix提供了两种隔离策略：线程池隔离和信号量隔离。hystrix默认采用线程池隔离。

- 线程池隔离：不同服务通过使用不同线程池，彼此间将不受影响，达到隔离效果。以demo为例，我们通过andThreadPoolKey配置使用命名为`ThreadPoolTest`的线程池，实现与其他命名的线程池天然隔离，如果不配置andThreadPoolKey则使用withGroupKey配置来命名线程池

- 信号量隔离：线程隔离会带来线程开销，有些场景（比如无网络请求场景）可能会因为用开销换隔离得不偿失，为此hystrix提供了信号量隔离，当服务的并发数大于信号量阈值时将进入fallback。以demo为例，通过 `withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)` 配置为信号量隔离，通过 `withExecutionIsolationSemaphoreMaxConcurrentRequests` 配置执行并发数不能大于3，由于信号量隔离下无论调用哪种命令执行方法，hystrix都不会创建新线程执行 `run()/construct()` ，所以调用程序需要自己创建多个线程来模拟并发调用 `execute()` ，最后看到一旦并发线程>3，后续请求都进入fallback

5、熔断机制

熔断机制相当于电路的跳闸功能，举个栗子，我们可以配置熔断策略为当请求错误比例在10s内>50%时，该服务将进入熔断状态，后续请求都会进入fallback。

以demo为例，我们通过 `withCircuitBreakerRequestVolumeThreshold` 配置10s内请求数超过3个时熔断器开始生效，通过 `withCircuitBreakerErrorThresholdPercentage` 配置错误比例>80%时开始熔断，然后for循环执行 `execute()` 触发 `run()` ，在 `run()` 里，如果 `name` 是小于10的偶数则正常返回，否则超时，通过多次循环后，超时请求占所有请求的比例将大于80%，就会看到后续请求都不进入 `run()` 而是进入 `getFallback()` ，因为不再打印 `"running run():" + name` 了。

除此之外，hystrix还支持多长时间从熔断状态自动恢复等功能，见下文附录。

6、结果cache

hystrix支持将一个请求结果缓存起来，下一个具有相同key的请求将直接从缓存中取出结果，减少请求开销。要使用hystrix cache功能，第一个要求是重写 `getCacheKey()` ，用来构造cache key；第二个要求是构建context，如果请求B要用到请求A的结果缓存，A和B必须同处一个context。通过 `HystrixRequestContext.initializeContext()` 和 `context.shutdown()` 可以构建一个context，这两条语句间的所有请求都处于同一个context。

以demo的 `testWithCacheHits()` 为例，*command2a*、*command2b*、*command2c*同处一个context，前两者的cache key都是 `2HLX` （见 `getCacheKey()` ），所以*command2a*执行完后把结果缓存，*command2b*执行时就不走 `run()` 而是直接从缓存中取结果了，而*command2c*的cache key是 `2HLX1` ，无法从缓存中取结果。此外，通过 `isResponseFromCache()` 可检查返回结果是否来自缓存。

7、合并请求collapsing

hystrix支持N个请求自动合并为一个请求，这个功能在有网络交互的场景下尤其有用，比如每个请求都要网络访问远程资源，如果把请求合并为一个，将使多次网络交互变成一次，极大节省开销。重要一点，两个请求能自动合并的前提是两者足够“近”，即两者启动执行的间隔时长要足够小，默认为10ms，即超过10ms将不自动合并。

以demo为例，我们连续发起多个queue请求，依次返回*f1~f6*共6个Future对象，根据打印结果可知*f1~f5*同处一个线程，说明这5个请求被合并了，而*f6*由另一个线程执行，这是因为*f5*和*f6*中间隔了一个sleep，超过了合并要求的最大间隔时长。

附录：各种策略配置

根据http://hot66hot.iteye.com/blog/2155036 整理而得。

HystrixCommandProperties

```
private final HystrixProperty metricsRollingStatisticalWindowInMilliseconds;

private final HystrixProperty metricsRollingStatisticalWindowBuckets;

private final HystrixProperty metricsRollingPercentileEnabled;

private final HystrixProperty circuitBreakerRequestVolumeThreshold;

private final HystrixProperty circuitBreakerSleepWindowInMilliseconds;

private final HystrixProperty circuitBreakerEnabled;

private final HystrixProperty circuitBreakerErrorThresholdPercentage;

private final HystrixProperty circuitBreakerForceOpen;

private final HystrixProperty circuitBreakerForceClosed;

private final HystrixProperty executionIsolationSemaphoreMaxConcurrentRequests;

private final HystrixProperty fallbackIsolationSemaphoreMaxConcurrentRequests;

private final HystrixProperty executionIsolationStrategy;

private final HystrixProperty executionIsolationThreadTimeoutInMilliseconds;

private final HystrixProperty executionIsolationThreadPoolKeyOverride;

private final HystrixProperty fallbackEnabled;

private final HystrixProperty executionIsolationThreadInterruptOnTimeout;

private final HystrixProperty requestLogEnabled;

private final HystrixProperty requestCacheEnabled;
```

HystrixCollapserProperties

```
private final HystrixProperty maxRequestsInBatch;

private final HystrixProperty timerDelayInMilliseconds;

private final HystrixProperty requestCacheEnabled;
```

HystrixThreadPoolProperties

```
private final HystrixProperty corePoolSize;

private final HystrixProperty maxQueueSize;
```

参考文献

https://github.com/Netflix/Hystrix
https://github.com/Netflix/Hystrix/wiki/How-To-Use
http://hot66hot.iteye.com/blog/2155036