介绍

Hystrix 是 Netflix 开源的一款容错框架。

具有**隔离、熔断、降级、恢复、请求缓存、请求合并、监控**等特性。

注：Hystrix 用的是 rxjava

官方文档

https://github.com/Netflix/Hystrix/wiki

版本

hystrix 版本：1.5.12

例子

要想使用 hystrix，只需要继承 `HystrixCommand` 或 `HystrixObservableCommand` 。

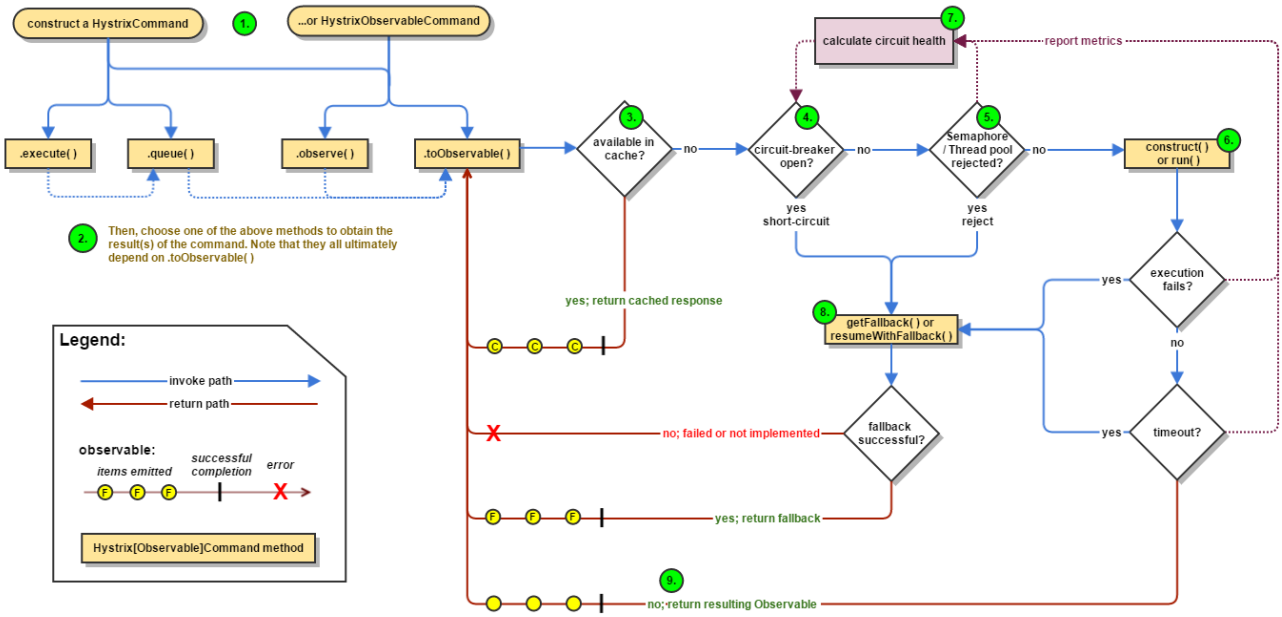
```
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import org.junit.Test;

public class HystrixDemo {

    @Test
    public void test1() {
        // execute()执行的就是run()方法
        String result = new HelloWorldHystrixCommand("world").execute();
        System.out.println(result); // 会输出Hello world
    }

    public class HelloWorldHystrixCommand extends HystrixCommand<String> {
        private final String name;

        public HelloWorldHystrixCommand(String name) {
            super(HystrixCommandGroupKey.Factory.asKey("DemoGroup")); // 设置groupKey
            this.name = name;
        }

        // 如果继承的是HystrixObservableCommand，要重写Observable construct()
        @Override
        protected String run() {
            return "Hello " + name;
        }
    }
}
```

降级：只需重写 `getFallback()` 方法。

```
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;
import org.junit.Test;

public class HystrixDemo {

    @Test
    public void test1() {
        String result = new HelloWorldHystrixCommand("world").execute();
        System.out.println(result); // 因为超时会输出Fallback world
    }

    public class HelloWorldHystrixCommand extends HystrixCommand<String> {
        private final String name;

        public HelloWorldHystrixCommand(String name) {
            super(HystrixCommandGroupKey.Factory.asKey("DemoGroup"));
            this.name = name;
        }

        @Override
        protected String run() throws Exception {
            Thread.sleep(1200); // hystrix默认超时时间为1000毫秒。
            return "Hello " + name;
        }

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

总体流程

总体流程图：



步骤：

1、创建 HystrixCommand 或 HystrixObservableCommand 对象

区别：

1. 前者的命令封装在 run()，后者在 contruct()
2. 前者的降级处理封装在 getFallBack()，后者在 resumeWithFallback()
3. 前者用新线程执行 run()，后者用主线程来执行 contruct()
4. 前者只能在 run() 中返回一个值，后者可以在 contruct() 中顺序定义多个 onNext 而返回多个值（当调用 subscribe() 注册成功后将依次执行这些 onNext ）

2、执行命令（execute()、queue()、observe()、toObservable()）

除了 execute()，还可以通过其他方法执行命令（ run() 或 contruct() ）。

注：HystrixObservableCommand 只有其中 observe()、toObservable() 方法。

区别：

- execute()：同步阻塞方式执行 run()。
- queue()：异步非阻塞方式执行 run()。会返回 Future 对象。
- observe()：事件注册（ subscribe() ）前执行 run() / construct()。
- toObservable()：返回 `Observable` 对象，通过 subscribe() 事件注册后才会执行 run() / construct()。

```
K             value   = command.execute();
Future<K>     fValue  = command.queue();
Observable<K> ohValue = command.observe();         // hot observable
Observable<K> ocValue = command.toObservable();    // cold observable
```

3、判断请求缓存是否开启

如果开启则直接从缓存取出结果并返回。

缓存 key 可以通过重写 getCacheKey() 方法进行设置（并且必须有 HystrixRequestContext ）。

4、检查其断路器状态

如果是开路状态，则直接快速失败，进行降级处理。

各 Command 的熔断器可通过 commandKey 来区分，相同的 commandKey 使用同一个熔断器。commandKey 默认值为类名，如果用的是注解形式，则默认值为方法名。可以在构建 HystrixCommand / HystrixObservableCommand 时通过 Setter 对象的 andCommandKey() 方法设置。

5、判断线程池 / 队列 / 信号量是否已满

如果隔离策略是线程池方式，则判断线程池和队列是否已满；如果是信号量隔离方式，则判断信号量是否已满。

如果已满，则直接快速失败，进行降级处理。

6、执行命令

执行 HystrixCommand.run() 或 HystrixObservableCommand.construct()。

如果执行超时或异常，会进行降级。

7、计算断路器各状态值

将请求成功，失败，被拒绝或超时信息报告给熔断器。熔断器维护一些用于统计数据用的计数器。

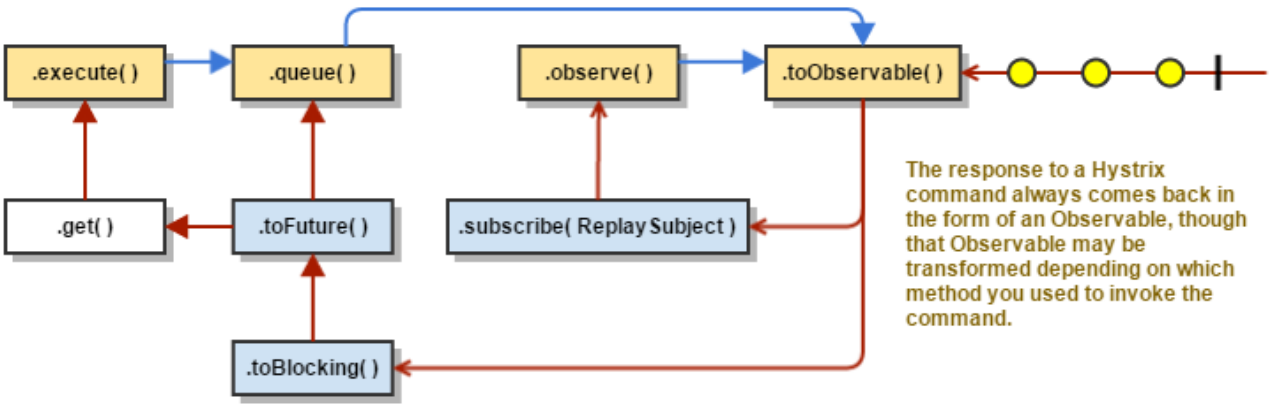
8、执行失败降级

若失败回退方法执行失败，或者用户未提供失败回退方法，Hystrix 会根据调用执行命令的方法的不同而产生不同的行为：

- execute()：抛出异常。
- queue()：成功返回 `Future` 对象，但其 `get()` 方法被调用时，会抛出异常。
- observe()：返回 `Observable` 对象，当订阅它的时候，会立即调用 subscriber 的 `onError` 方法中止请求。
- toObservable()：返回 `Observable` 对象，当订阅它的时候，会立即调用 subscriber 的 `onError` 方法中止请求。

9、返回正常

若命令成功执行，Hystrix 将响应返回给调用方，或者通过 `Observable` 的形式返回。根据上述调用命令方式的不同（如第 2 条），`Observable` 对象会进行一些转换：

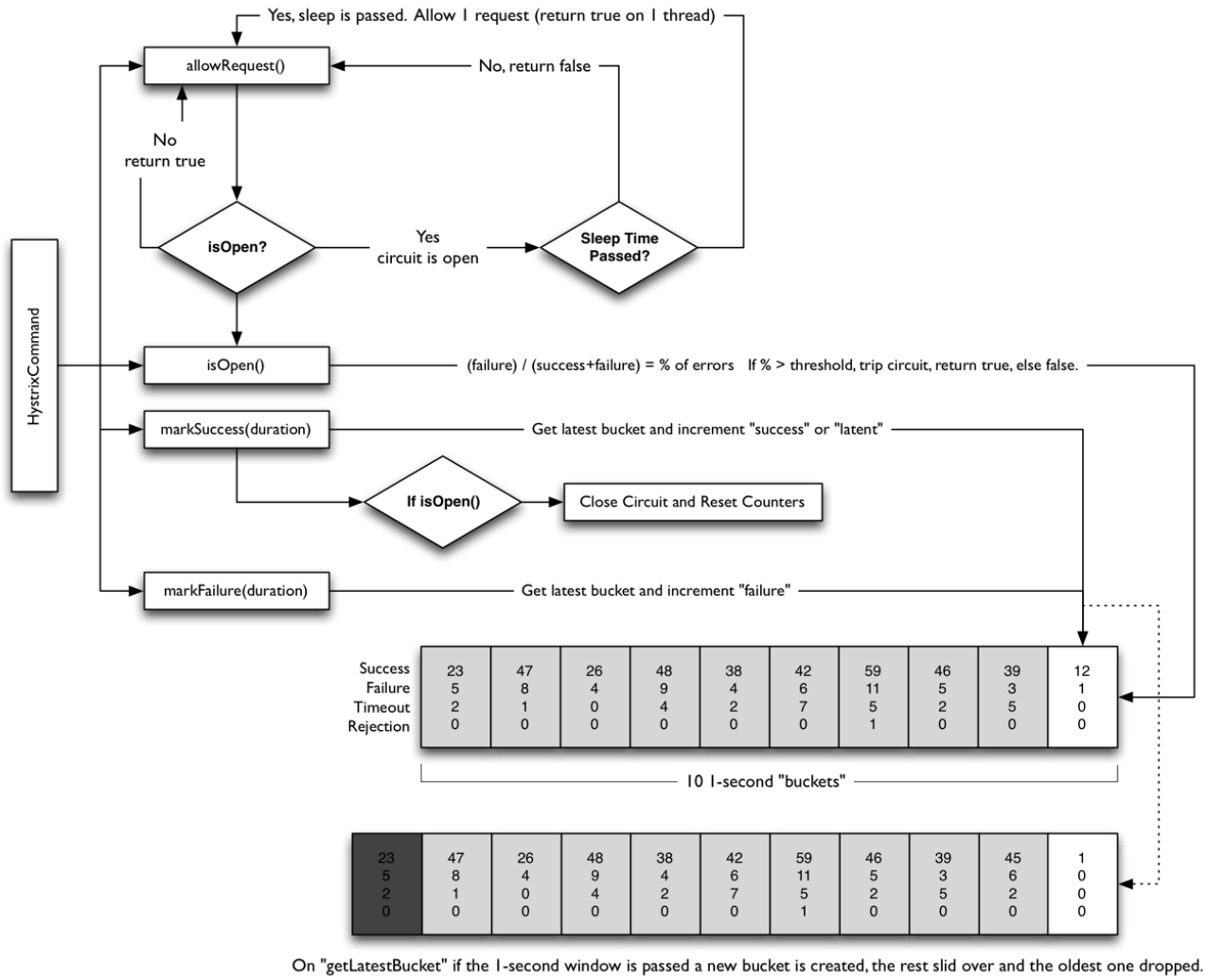


- execute()：由 `queue()` 产生 `Future` 对象，接着调用其 `get()` 方法，生成 `Observable` 对象。
- queue()：将 `Observable` 对象转换成 `BlockingObservable` 对象，并生成 `Future` 对象后返回。
- observe()：产生 `Observable` 对象后，进行订阅，然后返回该 `Observable` 对象。当再调用其 `subscribe` 方法时，会重放产生的响应信息给订阅者。

- toObservable()：返回 `Observable` 对象。必须调用其 `subscribe` 方法，才能执行命令。

断路器 Circuit Breaker

流程图：



断路条件

如果在一个时间窗口内，请求量达到阈值（`HystrixCommandProperties.circuitBreakerRequestVolumeThreshold()`），且失败率超过配置的百分比（`HystrixCommandProperties.circuitBreakerErrorThresholdPercentage()`），则会开路（从 CLOSED 转成 OPEN）。

恢复条件

一定时间（`HystrixCommandProperties.circuitBreakerSleepWindowInMilliseconds()`）后，会变成半开路状态（HALF-OPEN），并重新尝试请求。如果请求失败，会返回到 OPEN 状态；反之则进入 CLOSED 状态。

隔离策略 ExecutionIsolationStrategy

有线程、信号量两种策略（默认是线程隔离）。可以根据参数随时进行切换。

每个 Command 有它的 groupKey，groupKey 相同的 Command 会使用同一个线程池（或信号量），且该线程池（或信号量）不会管其他线程池（或信号量）里的。所以，某线程池（或信号量）负荷而降级了，不会影响到其他 Command。（如果有设置 threadPoolKey，就按 threadPoolKey 来划分）

线程隔离

通过线程池进行线程隔离。

请求线程和执行依赖的服务的线程不是同一个线程。

允许超时。

缺点：增加开销（排队、调度、上下文切换等）。

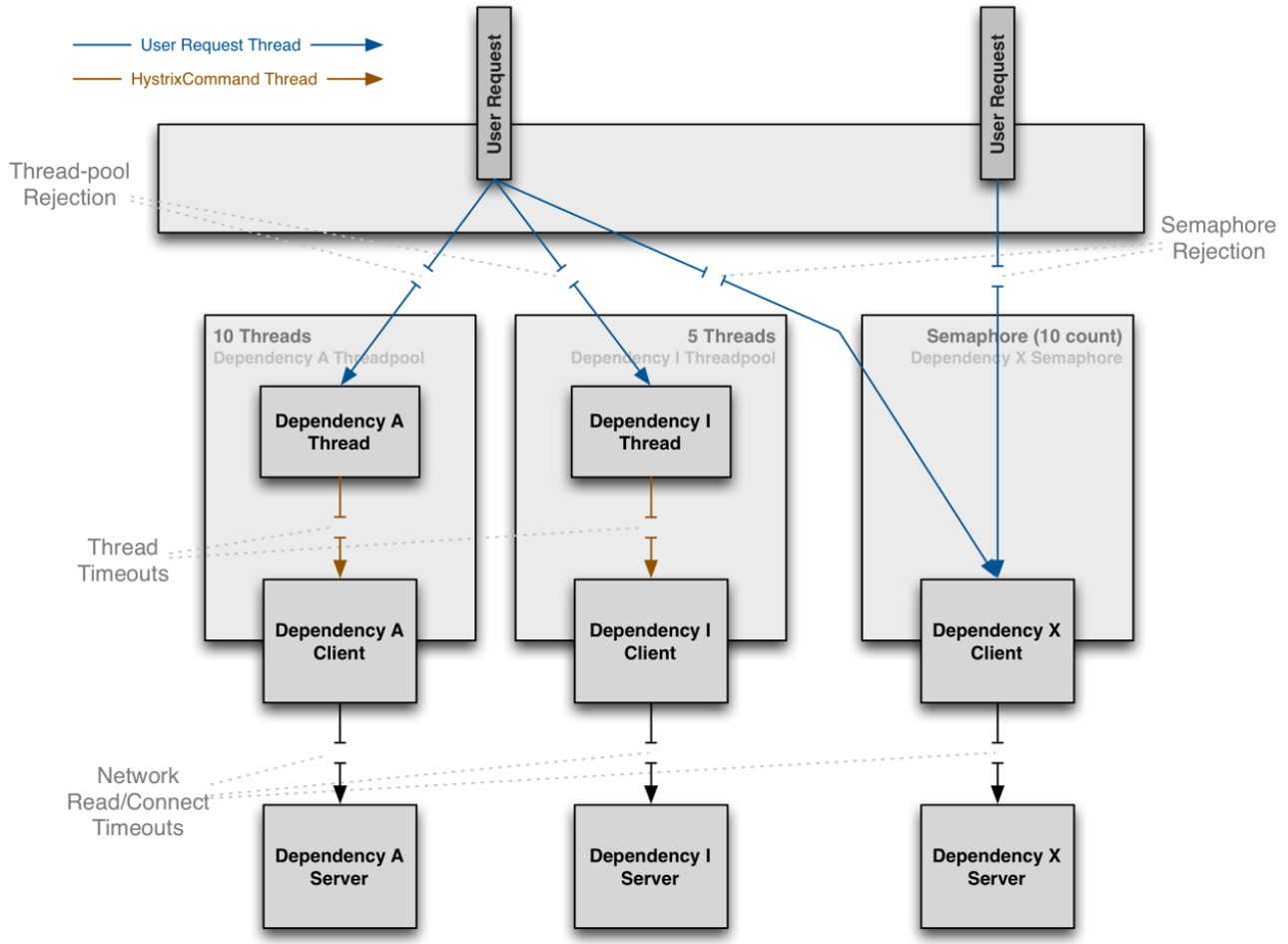
信号量隔离

通过信号量（计数器）限制各最大并发量。

业务请求线程和执行依赖服务的线程是同一个线程。

缺点：不支持异步；无法配置断路，即每次都会尝试获取信号量；一旦达到阈值，就会开始拒绝，但线程还是不能离开，所以不允许超时执行。

注：HystrixObservableCommand 默认使用信号量隔离。



源码

核心代码在 `AbstractCommand` 类里。

如 `toObservable()` - `applyHystrixSemantics()` - `executeCommandAndObserve()` - `executeCommandWithSpecifiedIsolation()` 方法。

其中，`applyHystrixSemantics()` 方法里，通过 `circuitBreaker.attemptExecution()` 判断没有熔断后，在内部通过信号量进行了限流（判断并发量）。通过的请求才会继续调用 `executeCommandAndObserve()` 。

`executeCommandAndObserve()` 里，如果判断参数 `executionTimeoutEnabled` 有开启，则每次请求都会提交一个任务到线程池中延迟执行。比如，配置的超时时间为 10ms，那么会提交一个延迟 10ms 执行的任务来判断是否执行超时。

之后，`executeCommandWithSpecifiedIsolation()` 会根据隔离策略进行处理，并执行用户定义的 `run()` 方法。

Bug

`HystrixCircuitBreaker.HystrixCircuitBreakerImpl.attemptExecution()` ：

```
public boolean attemptExecution() {
```

```
        if (properties.circuitBreakerForceOpen().get()) {
            return false;
        }
        if (properties.circuitBreakerForceClosed().get()) {
            return true;
        }
        if (circuitOpened.get() == -1) { // circuitOpened永远都是-1
            return true;
        } else {
            if (isAfterSleepWindow()) {
                if (status.compareAndSet(Status.OPEN, Status.HALF_OPEN)) {
                    return true;
                } else {
                    return false;
                }
            } else {
                return false;
            }
        }
    }
```

circuitOpened 指上一次发生熔断时的时间戳，初始值为 -1。

但这里调试会发现不管断路器有没有发生降级，circuitOpened 都会是 -1。

主要原因就是标记断路器状态时的方法：

```
    public void markSuccess() {
        if (status.compareAndSet(Status.HALF_OPEN, Status.CLOSED)) {
            //This thread wins the race to close the circuit - it resets the stream to
    start it over from 0
            metrics.resetStream();
            Subscription previousSubscription = activeSubscription.get();
            if (previousSubscription != null) {
                previousSubscription.unsubscribe();
            }
            Subscription newSubscription = subscribeToStream();
            activeSubscription.set(newSubscription);
            circuitOpened.set(-1L);
        }
    }

    @Override
    public void markNonSuccess() {
        if (status.compareAndSet(Status.HALF_OPEN, Status.OPEN)) {
            //This thread wins the race to re-open the circuit - it resets the start time
    for the sleep window
            circuitOpened.set(System.currentTimeMillis());
        }
    }
```

如上面的 `markNonSuccess()`，status 初始值是 CLOSED，所以就算降级了也不会变成 OPEN 状态，circuitOpened 也还会是 -1。

也就是说，断路器会一直为 CLOSED 状态。