Hystrix方法执行与降级方法执行源码分析

主要对加了**@HystrixCommand**注解的方法用AOP拦截实现类HystrixCommandAspect去拦截，主要拦截执行方法如下

```java
public Object methodsAnnotatedWithHystrixCommand(final ProceedingJoinPoint joinPoint) throws Throwable {
 // 被@HystrixCommand标记的hello()方法
 Method method = getMethodFromTarget(joinPoint);
 MetaHolderFactory metaHolderFactory = ...get(HystrixPointcutType.of(method));
 MetaHolder metaHolder = metaHolderFactory.create(joinPoint);
 // 准备各种材料后，创建HystrixInvokable
 HystrixInvokable invokable = HystrixCommandFactory.getInstance().create(metaHolder);
 Object result;
 try {
  if (!metaHolder.isObservable()) {
  // 利用工具CommandExecutor来执行具体的方法
  result = CommandExecutor.execute(invokable, executionType, metaHolder);
  }
  }
  return result;
 }
```

HystrixInvokable 只是一个空接口，没有任何方法，只是用来标记具备可执行的能力。

那 **HystrixInvokable** 又是如何创建的？它具体的实现类又是什么？先看看 *HystrixCommandFactory.getInstance().create*() 的代码。

```java
public HystrixInvokable create(MetaHolder metaHolder) {
 return new GenericCommand(...create(metaHolder));
}
```

GenericCommand 负责执行具体的方法和fallback时的方法

```java
// 执行具体的方法，如：OrderController的findById()
protected Object run() throws Exception {
 return process(new Action() {
 @Override
 Object execute() {
 return getCommandAction().execute(getExecutionType());
 }
 });
}
 // 执行fallback方法，如：OrderController的findByIdFallback()
 protected Object getFallback() {
  final CommandAction commandAction = getFallbackAction();
  return process(new Action() {
  @Override
  Object execute() {
  MetaHolder metaHolder = commandAction.getMetaHolder();
  Object[] args = createArgsForFallback(...);
  return commandAction.executeWithArgs(..., args);
  }
  });
 }
```

toObservable()核心源码解析

```java
public Observable<R> toObservable() {
 final AbstractCommand<R> _cmd = this;
 // 命令执行结束后的清理者
 final Action0 terminateCommandCleanup = new Action0() {...};
 // 取消订阅时处理者
 final Action0 unsubscribeCommandCleanup = new Action0() {...};
 // 重点：Hystrix 核心逻辑：断路器、隔离
 final Func0<Observable<R>> applyHystrixSemantics = new Func0<Observable<R>>() {...};
 // 发射数据(OnNext表示发射数据)时的Hook
  final Func1<R, R> wrapWithAllOnNextHooks = new Func1<R, R>() {...};
  // 命令执行完成的Hook
  final Action0 fireOnCompletedHook = new Action0() {...};
  // 通过Observable.defer()创建一个Observable
  return Observable.defer(new Func0<Observable<R>>() {
  @Override
  public Observable<R> call() {
  final boolean requestCacheEnabled = isRequestCachingEnabled();
  final String cacheKey = getCacheKey();
  // 首先尝试从请求缓存中获取结果
  if (requestCacheEnabled) {
  HystrixCommandResponseFromCache<R> fromCache = (HystrixCommandResponseFromCache<R>) requestCache.get(cacheKey);
  if (fromCache != null) {
  isResponseFromCache = true;
  return handleRequestCacheHitAndEmitValues(fromCache, _cmd);
  }
  }
  // 使用上面的Func0：applyHystrixSemantics 来创建Observable
  Observable<R> hystrixObservable =
  Observable.defer(applyHystrixSemantics)
  .map(wrapWithAllOnNextHooks);
  Observable<R> afterCache;
  // 如果启用请求缓存，将Observable包装成HystrixCachedObservable并进行相关处理
  if (requestCacheEnabled && cacheKey != null) {
  HystrixCachedObservable<R> toCache = HystrixCachedObservable.from(hystrixObservable, _cmd);
  ...
  } else {
  afterCache = hystrixObservable;
  }
  // 返回Observable
  return afterCache
  .doOnTerminate(terminateCommandCleanup)
  .doOnUnsubscribe(unsubscribeCommandCleanup)
  .doOnCompleted(fireOnCompletedHook);
  }
  });
```

```
}
```

断路器、隔离核心代码applyHystrixSemantics()

```
// Semantics 译为语义, 应用Hystrix语义很拗口，其实就是应用Hystrix的断路器、隔离特性
private Observable<R> applyHystrixSemantics(final AbstractCommand<R> _cmd) {
 // 源码中有很多executionHook、eventNotifier的操作，这是Hystrix拓展性的一种体现。这里面啥事也没做，留了个口子，开发人员可以拓展
 executionHook.onStart(_cmd);
 // 判断断路器是否开启
 if (circuitBreaker.attemptExecution()) {
 // 获取执行信号
 final TryableSemaphore executionSemaphore = getExecutionSemaphore();
 final AtomicBoolean semaphoreHasBeenReleased = new AtomicBoolean(false);
 final Action0 singleSemaphoreRelease = new Action0() {...};
 final Action1<Throwable> markExceptionThrown = new Action1<Throwable>() {...};
 // 判断是否信号量拒绝
 if (executionSemaphore.tryAcquire()) {
 try {
 // 重点: 处理隔离策略和Fallback策略
 return executeCommandAndObserve(_cmd)
 .doOnError(markExceptionThrown)
 .doOnTerminate(singleSemaphoreRelease)
 .doOnUnsubscribe(singleSemaphoreRelease);
 } catch (RuntimeException e) {
 return Observable.error(e);
 }
 } else {
 return handleSemaphoreRejectionViaFallback();
 }
 }
 // 开启了断路器,执行Fallback
 else {
 return handleShortCircuitViaFallback();
 }
}
```

executeCommandAndObserve()处理隔离策略和各种Fallback

```
private Observable<R> executeCommandAndObserve(final AbstractCommand<R> _cmd) {
 final HystrixRequestContext currentRequestContext = HystrixRequestContext.getContextForCurrentThread();
 final Action1<R> markEmits = new Action1<R>() {...};
 final Action0 markOnCompleted = new Action0() {...};
 // 利用Func1获取处理Fallback的 Observable
 final Func1<Throwable, Observable<R>> handleFallback = new Func1<Throwable, Observable<R>>() {
 @Override
 public Observable<R> call(Throwable t) {
 circuitBreaker.markNonSuccess();
```

```java
Exception e = getExceptionFromThrowable(t);
executionResult = executionResult.setExecutionException(e);
// 拒绝处理
if (e instanceof RejectedExecutionException) {
return handleThreadPoolRejectionViaFallback(e);
// 超时处理
} else if (t instanceof HystrixTimeoutException) {
return handleTimeoutViaFallback();
} else if (t instanceof HystrixBadRequestException) {
return handleBadRequestByEmittingError(e);
} else {
...
return handleFailureViaFallback(e);
}
}
};
final Action1<Notification<? super R>> setRequestContext ...
Observable<R> execution;
// 利用特定的隔离策略来处理
if (properties.executionTimeoutEnabled().get()) {
execution = executeCommandWithSpecifiedIsolation(_cmd)
.lift(new HystrixObservableTimeoutOperator<R>(_cmd));
} else {
execution = executeCommandWithSpecifiedIsolation(_cmd);
}
return execution.doOnNext(markEmits)
.doOnCompleted(markOnCompleted)
// 绑定Fallback的处理者
.onErrorResumeNext(handleFallback)
.doOnEach(setRequestContext);
}
```

隔离特性的处理：executeCommandWithSpecifiedIsolation()

```java
private Observable<R> executeCommandWithSpecifiedIsolation(final AbstractCommand<R> _cmd) {
// 线程池隔离
if (properties.executionIsolationStrategy().get() == ExecutionIsolationStrategy.THREAD) {
// 再次使用 Observable.defer(), 通过执行Func0来得到Observable
return Observable.defer(new Func0<Observable<R>>() {
@Override
public Observable<R> call() {
// 收集metric信息
metrics.markCommandStart(commandKey, threadPoolKey, ExecutionIsolationStrategy.THREAD);
...
try {
... // 获取真正的用户Task
return getUserExecutionObservable(_cmd);
} catch (Throwable ex) {
return Observable.error(ex);
}
```

```java
...
}
// 绑定各种处理者
}).doOnTerminate(new Action0() {...})
.doOnUnsubscribe(new Action0() {...})
// 线程隔离，绑定超时处理者
.subscribeOn(threadPool.getScheduler(new Func0<Boolean>() {
@Override
public Boolean call() {
return properties.executionIsolationThreadInterruptOnTimeout().get() && _cmd.isCommandTimedOut.get() == TimedOutStatus.TIMED_OUT;
}
}));
}
// 信号量隔离，和线程池大同小异
else {
return Observable.defer(new Func0<Observable<R>>() {...}
}
}
```

线程隔离threadPool.getScheduler

```java
//隔离线程池初始化
private static HystrixThreadPool initThreadPool(HystrixThreadPool fromConstructor, HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter threadPoolPropertiesDefaults) {
 if (fromConstructor == null) {
 // get the default implementation of HystrixThreadPool
 return HystrixThreadPool.Factory.getInstance(threadPoolKey, threadPoolPropertiesDefaults);
 } else {
 return fromConstructor;
 }
 }


static HystrixThreadPool getInstance(HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter propertiesBuilder) {
 // get the key to use instead of using the object itself so that if people forget to implement equals/hashcode things will still work
 String key = threadPoolKey.name();

 // this should find it for all but the first time
 HystrixThreadPool previouslyCached = threadPools.get(key);
 if (previouslyCached != null) {
 return previouslyCached;
 }

 // if we get here this is the first time so we need to initialize
 synchronized (HystrixThreadPool.class) {
 if (!threadPools.containsKey(key)) {
 threadPools.put(key, new HystrixThreadPoolDefault(threadPoolKey, propertiesBuilder));
 }
```

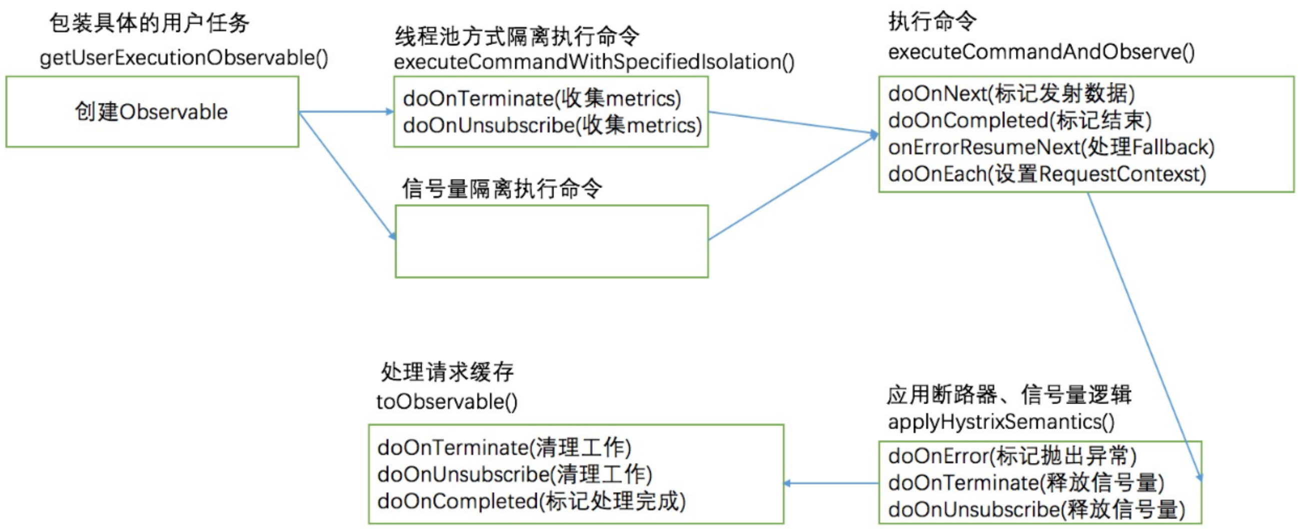
```
}
return threadPools.get(key);
}

public HystrixThreadPoolDefault(HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter propertiesDefaults) {
this.properties = HystrixPropertiesFactory.getThreadPoolProperties(threadPoolKey, propertiesDefaults);
HystrixConcurrencyStrategy concurrencyStrategy = HystrixPlugins.getInstance().getConcurrencyStrategy();
this.queueSize = properties.maxQueueSize().get();

this.metrics = HystrixThreadPoolMetrics.getInstance(threadPoolKey,
concurrencyStrategy.getThreadPool(threadPoolKey, properties),
properties);
this.threadPool = this.metrics.getThreadPool();
this.queue = this.threadPool.getQueue();

/* strategy: HystrixMetricsPublisherThreadPool */
HystrixMetricsPublisherFactory.createOrRetrievePublisherForThreadPool(threadPoolKey, this.metrics, this.properties);
}
```

命令真正的调用逻辑入口getUserExecutionObservable

```
private Observable<R> getUserExecutionObservable(final AbstractCommand<R> _cmd) {
Observable<R> userObservable;

try {
userObservable = getExecutionObservable();
} catch (Throwable ex) {
// the run() method is a user provided implementation so can throw instead of using Observable.onError
// so we catch it here and turn it into Observable.error
userObservable = Observable.error(ex);
}

return userObservable
.lift(new ExecutionHookApplication(_cmd))
.lift(new DeprecatedOnRunHookApplication(_cmd));
}

final protected Observable<R> getExecutionObservable() {
return Observable.defer(new Func0<Observable<R>>() {
@Override
public Observable<R> call() {
try {
//调用命令的真正方法run()入口
return Observable.just(run());
} catch (Throwable ex) {
```

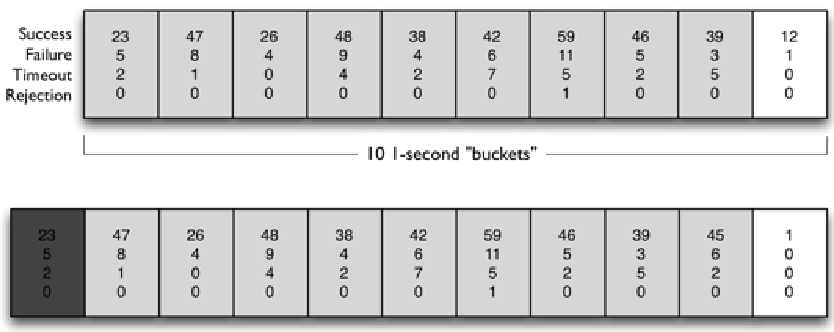
```
return Observable.error(ex);
}
}
}).doOnSubscribe(new Action0() {
@Override
public void call() {
// Save thread on which we get subscribed so that we can interrupt it later if needed
executionThread.set(Thread.currentThread());
}
});
}
```

上面方法层层调用，倒过来看，就是先创建一个Observable，然后绑定各种事件对应的处理者，如下图



断路器源码分析

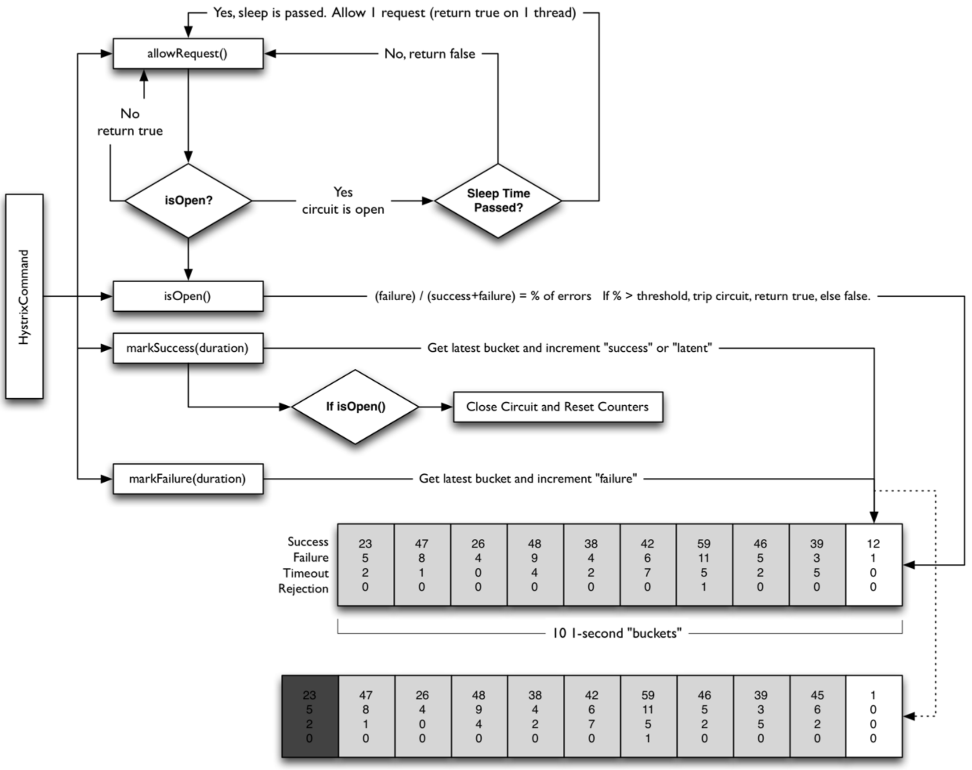
Hystrix 有点类似，例如：**以秒为单位来统计请求的处理情况(成功请求数量、失败请求数、超时请求数、被拒绝的请求数)，然后每次取最近10秒的数据来进行计算，如果失败率超过50%，就进行熔断，不再处理任何请求。**
这是Hystrix官网的一张图：



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

它演示了 Hystrix **滑动窗口** 策略，假定以秒为单位来统计请求处理情况，上面每个格子代表1秒，格子中的数据就是1秒内各处理结果的请求数量，格子称为 Bucket(译为桶)。
若每次的决策都以10个Bucket的数据为依据，计算10个Bucket的请求处理情况，当失败率超过50%时就熔断。
10个Bucket就是10秒，这个10秒就是一个 **滑动窗口(Rolling window)**。
为什么叫滑动窗口？因为在没有熔断时，每当收集好一个新的Bucket后，就会丢弃掉最旧的一个Bucket。上图中的深色的(23 5 2 0)就是被丢弃的桶。

下面是官方完整的流程图，策略是：不断收集数据，达到条件就熔断；熔断后拒绝所有请求一段时间(sleepWindow)；然后放一个请求过去，如果请求成功，则关闭熔断器，否则继续打开熔断器。



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

相关配置

默认配置都在HystrixCommandProperties类中。

先看两个metrics收集的配置。

- **metrics.rollingStats.timeInMilliseconds**

表示滑动窗口的时间(the duration of the statistical rolling window)，默认10000(10s)，也是熔断器计算的基本单位。

- **metrics.rollingStats.numBuckets**

滑动窗口的Bucket数量(the number of buckets the rolling statistical window is divided into)，默认10. 通过timeInMilliseconds和numBuckets可以计算出每个Bucket的时长。

metrics.rollingStats.timeInMilliseconds % metrics.rollingStats.numBuckets 必须等于 0，否则将抛异常。

再看看熔断器的配置。

- **circuitBreaker.requestVolumeThreshold**

滑动窗口触发熔断的最小请求数。如果值是20，但滑动窗口的时间内请求数只有19，那即使19个请求全部失败，也不会熔断，必须达到这个值才行，否则样本太少，没有意义。

- **circuitBreaker.sleepWindowInMilliseconds**

这个和熔断器自动恢复有关，为了检测后端服务是否恢复，可以放一个请求过去试探一下。sleepWindow指的发生熔断后，必须隔sleepWindow这么长的时间，才能放请求过去试探下服务是否恢复。默认是5s

- **circuitBreaker.errorThresholdPercentage**

错误率阈值，表示达到熔断的条件。比如默认的50%，当一个滑动窗口内，失败率达到50%时就会触发熔断。

断路器的初始化是在AbstractCommand构造器中做的初始化

```
private static HystrixCircuitBreaker initCircuitBreaker(boolean enabled, HystrixCircuitBreaker fromConstructor, HystrixCommandKey commandKey...) {
  // 如果启用了熔断器
```

```java
if (enabled) {
// 若commandKey没有对应的CircuitBreaker,则创建
if (fromConstructor == null) {
return HystrixCircuitBreaker.Factory.getInstance(commandKey, groupKey, properties, metrics);
} else {
// 如果有则返回现有的
return fromConstructor;
 }
 } else {
 return new NoOpCircuitBreaker();
 }
}
```

再看看 **HystrixCircuitBreaker.Factory.getInstance(commandKey, groupKey, properties, metrics)** 如何创建circuit-breakder?

circuitBreaker以commandKey为维度，每个commandKey都会有对应的circuitBreaker

```java
public static HystrixCircuitBreaker getInstance(HystrixCommandKey key, HystrixCommandGroupKey group, HystrixCommandProperties properties, HystrixCommandMetrics metrics) {
// 如果有则返回现有的, key.name()即command的name作为检索条件
HystrixCircuitBreaker previouslyCached = circuitBreakersByCommand.get(key.name());
if (previouslyCached != null) {
return previouslyCached;
}
// 如果没有则创建并cache
HystrixCircuitBreaker cbForCommand = circuitBreakersByCommand.putIfAbsent(key.name(), new HystrixCircuitBreakerImpl(key, group, properties, metrics));
if (cbForCommand == null) {
 return circuitBreakersByCommand.get(key.name());
 } else {
 return cbForCommand;
 }
}


//初始化断路器
protected HystrixCircuitBreakerImpl(HystrixCommandKey key, HystrixCommandGroupKey commandGroup, final HystrixCommandProperties properties, HystrixCommandMetrics metrics) {
 this.properties = properties;
 // 这是Command中的metrics对象,metrics对象也是commandKey维度的
 this.metrics = metrics;
 // 重点:订阅事件流
 Subscription s = subscribeToStream();
 activeSubscription.set(s);
}
// 订阅事件流, 各事件以结构化数据汇入了Stream中
private Subscription subscribeToStream() {
 // HealthCountsStream是重点
 return metrics.getHealthCountsStream()
 .observe()
 // 利用数据统计的结果HealthCounts, 实现熔断器
 .subscribe(new Subscriber<HealthCounts>() {
```

```
@Override
public void onCompleted() {}
@Override
public void onError(Throwable e) {}
@Override
public void onNext(HealthCounts hc) {
// 检查是否达到最小请求数,默认20个; 未达到的话即使请求全部失败也不会熔断
if (hc.getTotalRequests() < properties.circuitBreakerRequestVolumeThreshold().get()) {
// 啥也不做
} else {
// 错误百分比未达到设定的阀值
if (hc.getErrorPercentage() < properties.circuitBreakerErrorThresholdPercentage().get()) {
} else {
// 错误率过高, 进行熔断
if (status.compareAndSet(Status.CLOSED, Status.OPEN)) {
circuitOpened.set(System.currentTimeMillis());
}
}
}
}
});
}
```