**Hystrix框架介绍**

**提供服务熔断和服务降级**

**的解决方案**

叶茂

2016年 12 月

# 概述

## Hystrix解决了什么问题

在分布式环境中，很多依赖项中的某些服务将不可避免地失败。 Hystrix是一个开源框架，通过添加延迟容差和逻辑容错来帮助您控制这些分布式服务之间的交互。 Hystrix通过隔离服务之间的访问，停止它们之间的级联失败以及提供降级选项来实现这一点，所有这些都提高了系统的整体弹性。

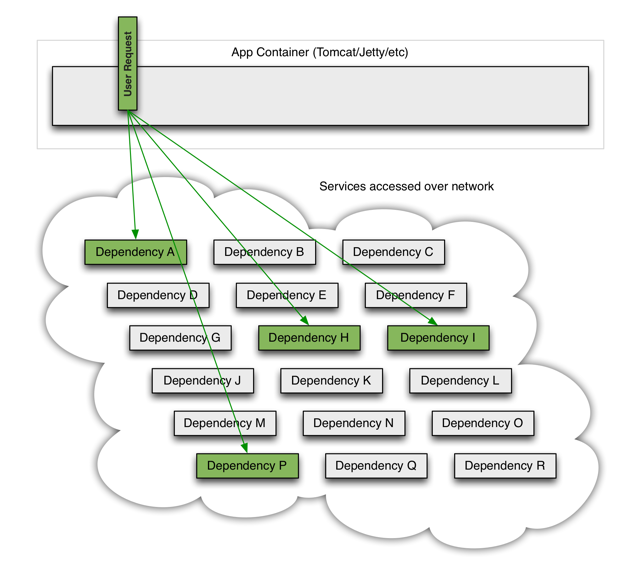
复杂分布式架构中的应用程序有几十个依赖关系，每个依赖关系在某一点都会不可避免地失败。 如果主机应用程序未与这些外部故障隔离，则可能会随之一起不稳定。

例如，对于依赖于30个服务的应用程序，每个服务的正常运行时间为99.99％，您可以估算：

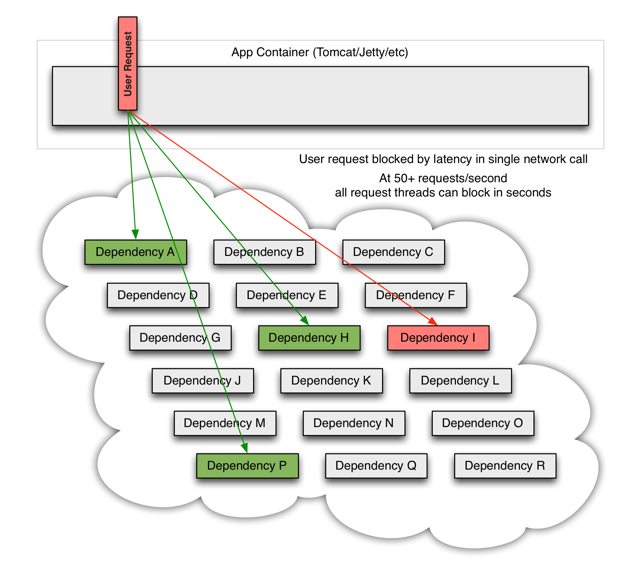
99.99%30 = 99.7% 正常可用  
十亿个请求的0.3% = 3,000,000 个失败

**如果你不为整个系统设计弹性**，即使所有依赖关系都表现良好，即使是在几十个服务中的每个服务器上，甚至0.01％的停机时间的总体影响也会造成相当长的宕机时间。

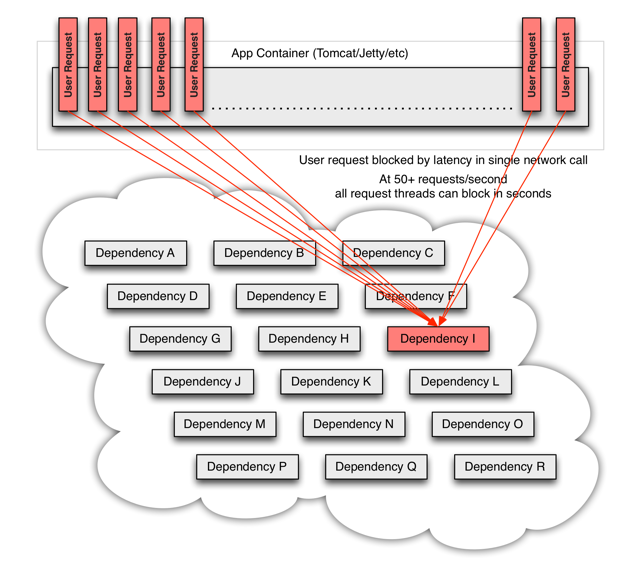
当服务健康时，请求流程可以如下所示：



当其中一个依赖系统出现延迟的时候，它可能阻止整个用户请求：



对于高流量，单个后端依赖出现延迟有可能导致所有服务器上的有资源在几秒钟内饱和。  
应用程序中通过网络或最终通过网络请求的第三方库中的每个点都可能是潜在的失败原因。 比故障更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，这会消耗队列，线程和其他系统资源，从而在整个系统中导致更多的级联故障。



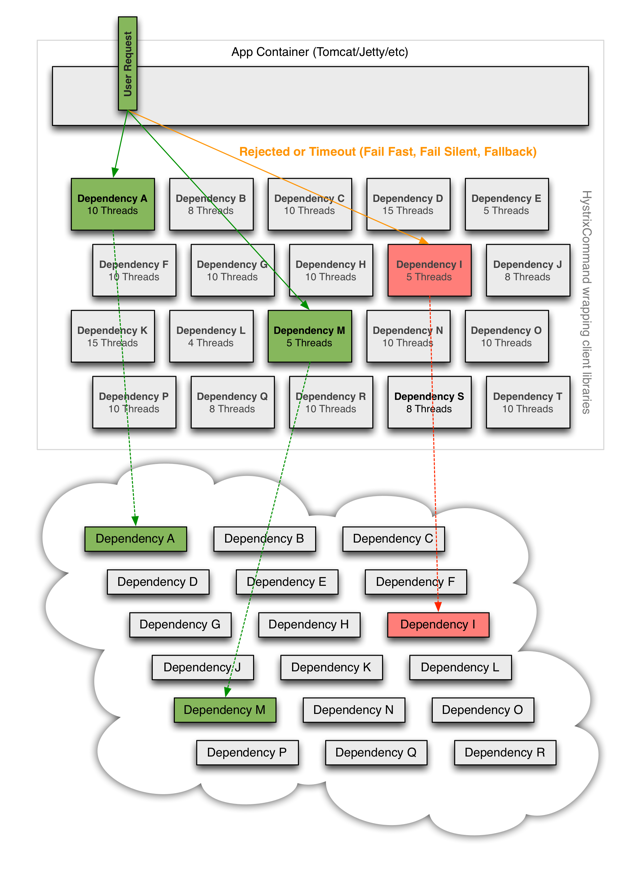
对所有的外部依赖加入基本的超时时间设定（http、dubbo）就可以避免资源的阻塞，但仍然会留有两个问题：一是当前系统仍然会消耗资源对一个基本已确定不可用的外部依赖发起请求，造成不必要的开销；二是对于下游系统而言，在本来就已经出现状况的情况下，继续收到大量的请求，更加难以及时恢复，只会让情况更糟。

## Hystrix的解决方案

Hystrix的做法是：

* 将所有对外部系统的调用（或“依赖关系”）包含在*HystrixCommand*或*HystrixObservableCommand*中并在分开的线程中执行（采用了命令模式）
* 超时调用的时间超过您定义的阈值。 它有一个默认值，但对于大多数依赖项，您可以通过“属性”自定义这些超时时间，以使它们略高于每个依赖项的测量的99.5百分位数性能。
* 为每个依赖关系维护一个小线程池（或信号量）;如果它变满，则发往该依赖的请求将被立即拒绝，而不是排队等候。
* 监控成功，失败（由客户端抛出的异常），超时和线程拒绝。
* 如果服务的错误百分比超过阈值，则熔断（手动或自动）在一段时间内停止对特定服务的所有请求。
* 当请求失败，被拒绝，超时或短路时执行降级逻辑。
* 准实时的监控和配置更改。

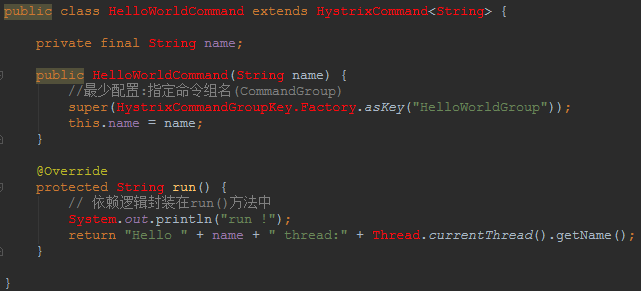
当您使用Hystrix来包装每个底层依赖项时，上图中所示的体系结构将更改为类似于以下图的方式。每个依赖关系彼此隔离，在发生延迟或故障时只会使有限的资源，并且在降级逻辑中覆盖，该降级逻辑决定当依赖关系中发生任何类型的故障时要做出什么响应：



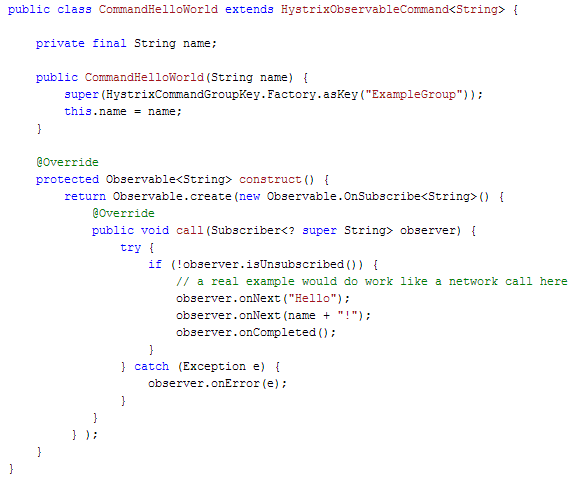
# 功能介绍

## Hello World

一个继承了HystrixCommand的基本Hello World实现如下：



**HystrixObservableCommand**方式的等价实现：  
使用HystrixObservableCommand等效替代HystrixCommand的解决方案将涉及覆盖构造方法，如下所示：



## Command执行

你可以使用execute()方法提交一个同步请求，示例如下：

String s = new CommandHelloWorld("World").execute();

你也可以使用queue()方法提交一个同步请求，示例如下：

Future<String> fs = new CommandHelloWorld("World").queue();

您可以使用Future得到Command的结果：

String s = fs.get();

这两种方式是等价的：

String s1 = new CommandHelloWorld("World").execute();

String s2 = new CommandHelloWorld("World").queue().get();

## 响应式执行

你还可以通过使用以下方法之一来观察HystrixCommand作为Observable的结果：  
observe() - 返回一个“热”的Observable并立即执行的命令

toObservable（） - 返回一个“冷”的Observable，它不会执行命令和发出结果，直到你订阅了这个Observable：

Observable <String> ho = new CommandHelloWorld(“World”).observe();

//或Observable <String> co = new CommandHelloWorld(“World”).toObservable();

然后，通过订阅Observable来得到命令的值：

ho.subscribe(new Action1<String>() {

@Override

public void call(String s) {

// 返回值在这里获取

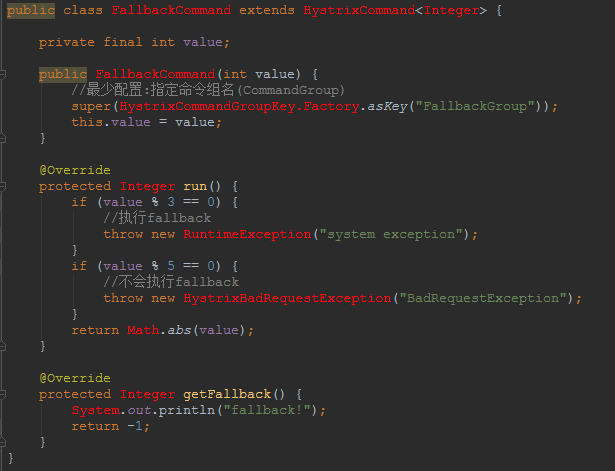
}

});

## 降级

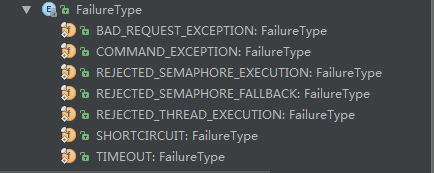
您可以通过添加一个备用方法来支持Hystrix命令的正常降级，Hystrix将调用该方法以获取一个或多个默认值，以防主命令失败。

在普通的HystrixCommand中，通过getFallback()方法实现来实现降级。 Hystrix将对所有类型的故障（如run()失败，超时，线程池或信号量拒绝和熔断器短路）执行此备用。以下示例包含此类降级：



## 错误传播

除了HystrixBadRequestException之外，从run()方法抛出的所有异常都计为失败并触发getFallback()和熔断器逻辑。  
你可以使用HystrixBadRequestException包装你想抛出的异常并通过getCause()来获取它。HystrixBadRequestException旨在用于报告非法参数或非系统故障的用例，这些故障不应计入故障指标，不应触发降级逻辑。



## Command命名和分组

一个Command的名字默认从类名中派生而成：

getClass().getSimpleName();

要显式定义名称，通过HystrixCommand或HystrixObservableCommand构造函数传递它。

Hystrix使用CommandGroupKey将诸如监控或团队/库所有权等命令进行分组。  
默认情况下Hystrix使用它来定义命令线程池，除非你单独定义了一个。

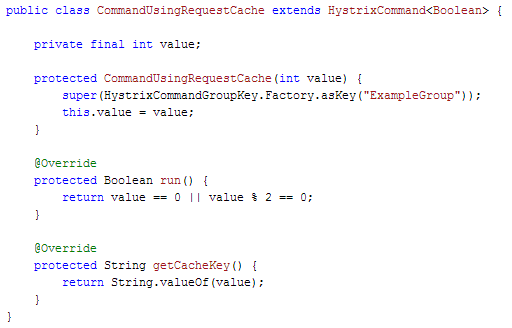
HystrixThreadPoolKey用于标识监控，缓存和其他此类用途的HystrixThreadPool。 HystrixCommand与由HystrixThreadPoolKey注入的单个HystrixThreadPool相关联，或者它默认为使用创建的HystrixCommandGroupKey创建的HystrixThreadPool。

要显式定义名称，通过构造函数传递它：



## 请求缓存

通过在HystrixCommand或HystrixObservableCommand对象上实现getCacheKey()方法来启用高速缓存，如下所示：



因为这取决于请求上下文，我们必须初始化HystrixRequestContext。通常，此上下文将通过包装用户请求或其他生命周期钩子的ServletFilter进行初始化和关闭。

## 请求上下文设置

要使用请求范围的功能（请求缓存，请求崩溃，请求日志），您必须管理HystrixRequestContext生命周期（或实现一个替代HystrixConcurrencyStrategy）。

这意味着您必须在请求之前执行以下操作：

HystrixRequestContext context = HystrixRequestContext.initializeContext();

然后这个在请求结束时：

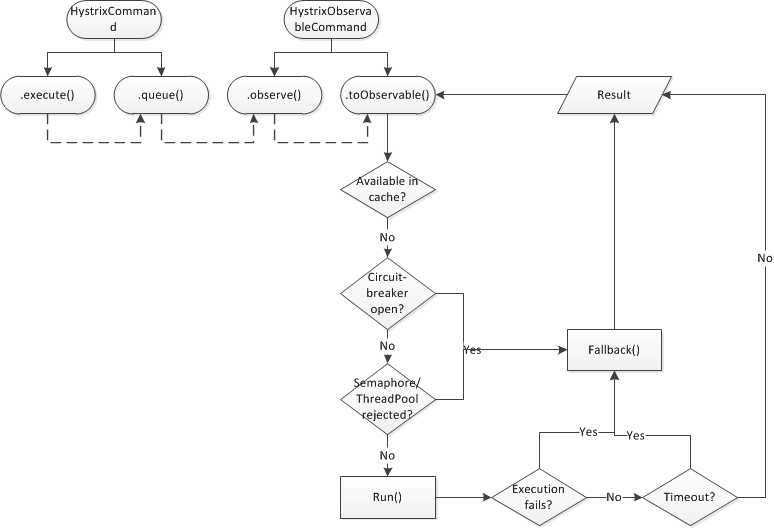
context.shutdown();

在标准的Java Web应用程序中，您可以使用ServletFilter通过实现类似于以下的过滤器来初始化此生命周期。

# 实现原理

## 处理流程

下图显示了通过Hystrix向服务依赖关系发出请求时的处理流程：



下面将更详细地解释此流程：

1.构造HystrixCommand或HystrixObservableCommand对象  
第一步是构造一个HystrixCommand或HystrixObservableCommand对象来表示你对外部依赖的请求。传递在发出请求时需要的参数通过构造函数传递到Command。

如果希望依赖项返回单个的响应，请构造HystrixCommand对象。例如：

HystrixCommand command = new HystrixCommand(arg1, arg2);

如果希望依赖项返回带有多个响应的Observable，请构造HystrixCommand对象。例如：

HystrixObservableCommand command = new HystrixObservableCommand(arg1, arg2);

2. 执行Command  
有四种方法可以执行命令，通过使用Hystrix命令对象的以下四种方法之一（前两种方法仅适用于简单的HystrixCommand对象，但不适用于HystrixObservableCommand）：

K value = command.execute();

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

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

Observable<K> ocValue = command.toObservable(); //cold observable

同步调用execute()会调用queue().get()方法。 queue()依次调用toObservable().toBlocking().toFuture()。 这就是说，最终每个HystrixCommand都由一个Observable的实现来返回，甚至那些旨在返回单个简单值的命令。

3. 返回是否已缓存？  
如果为此命令启用了请求高速缓存，并且如果请求的响应在高速缓存中可用，则此高速缓存的响应将立即以Observable的形式返回。 （请参阅下面的“请求缓存”。）

4. 熔断器是否断开？  
当您执行该命令时，Hystrix与熔断器检查电路是否断开。

如果电路断开开（或“跳闸”），Hystrix将不执行命令，但将流程路由到（8）获取降级响应。

如果电路闭合，则流程进行到（5）以检查是否有可用于运行命令的容量。

5. 线程池/队列/信号量是否满载？  
如果与命令相关联线程池和队列（或信号量，如果不在一个线程中运行）已经满负载，则Hystrix不会执行命令，但会立即将流程路由到（8）获取降级。

6. HystrixObservableCommand.construct()或HystrixCommand.run()  
这里，Hystrix通过您为此命令编写的方法调用对依赖关系的请求，执行下面两种方法的其中一种：

HystrixCommand.run() - 返回单个响应或抛出异常

HystrixObservableCommand.construct() - 返回一个发出响应的Observable或发送onError通知  
如果run()或construct()方法超出了命令的超时值，线程将抛出一个TimeoutException异常（如果命令本身没有运行在自己的线程中，将由一个独立的定时器线程抛出）。在这种情况下，Hystrix将响应路由到（8）获取降级响应，如果该方法不取消/中断，它会丢弃run()或construct()方法最终的返回值。

如果命令没有抛出任何异常并返回了响应结果，Hystrix在执行一些日志记录和指标报告后返回此响应。在run()的情况下，Hystrix返回一个Observable，它发出单个响应，然后发出onCompleted通知；在construct()的情况下Hystrix返回由construct()返回的相同的Observable。

7. 计算熔断器健康指标  
Hystrix向断路器报告成功，失败，拒绝和超时，它保持动态数据的统计计数。

它使用这些统计信息来确定电路何时应该“跳闸”，此时它会将任何后续请求短路，直到恢复周期过去，然后在首次检查某些健康状况后再次闭合电路。

8. 获取降级  
Hystrix试图在命令执行失败时返回降级结果：当（6）construct()或run()发出异常时；当（4）命令被短路导致电路断开时；当（5）命令的线程池和队列或信号量容量已满时；或者当命令超过其超时时间时。

建议编写您的降级方案以提供一个通用的响应，它最好没有任何网络依赖，来自内存中缓存或通过其他静态逻辑得到。如果必须在降级中使用网络调用，则应通过另一个HystrixCommand或HystrixObservableCommand来执行此操作。

在HystrixCommand的情况下，要提供降级逻辑，你需要实现HystrixCommand.getFallback（），它返回单个降级后的结果值。

在HystrixObservableCommand的情况下，为了提供降级逻辑，您实现HystrixObservableCommand.resumeWithFallback()，它返回一个可能发出一个或多个返回值的Observable。

如果降级方法返回一个响应，那么Hystrix会将这个响应返回给调用者。在HystrixCommand.getFallback（）的情况下，它将返回一个Observable，它发出从方法返回的值。在HystrixObservableCommand.resumeWithFallback（）的情况下，它将返回从该方法返回的相同的Observable。

如果你没有为你的Hystrix命令实现一个降级方法，或者如果降级本身抛出一个异常，Hystrix仍然返回一个Observable，但是一个没有任何东西，并立即以onError通知终止。正是通过这个onError通知，导致命令失败的异常被传回给调用者。 （实现可能失败的降级是一个糟糕的做法，您应该实现您的降级，使其不执行任何可能失败的逻辑。）

异常或不存在的降级的结果将根据您调用Hystrix命令的方式而有所不同：

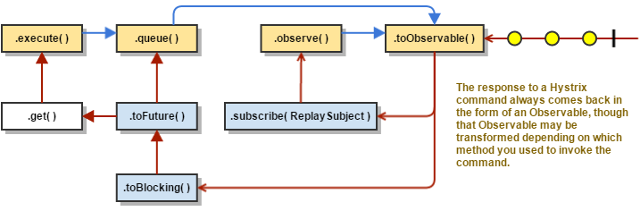
execute() — 抛出异常

queue() — 成功返回一个Future, 但是在调用get()方法的时候会抛出异常

observe() — 返回一个Observable，当你订阅它的时候会立刻调用订阅者的onError方法并终止

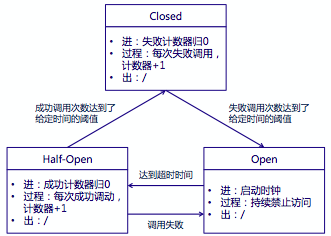
toObservable() — 返回一个Observable，当你订阅它的时候会立刻调用订阅者的onError方法并终止

9. 返回成功响应  
如果Hystrix命令成功，它将以Observable的形式返回对调用者的一个或多个响应。 根据您在上述步骤2中调用命令的方式的不同，此Observable可能会在返回给您之前进行转换：



## 熔断器circuit-breaker

先说说熔断器的概念：

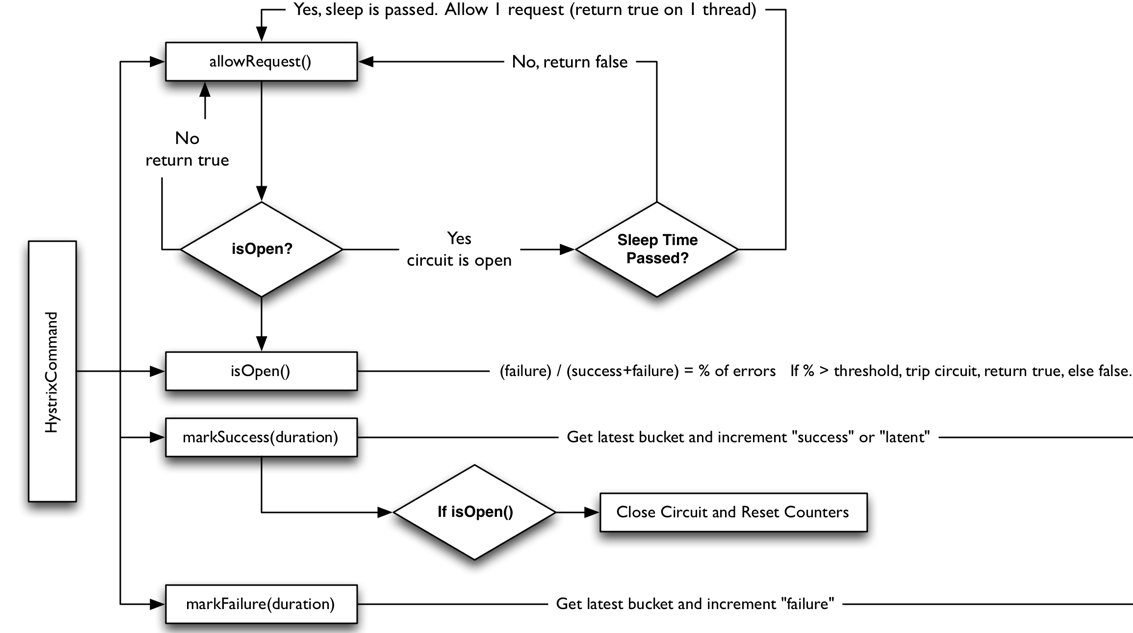


Closed：熔断器关闭状态，调用失败次数积累，到了阈值（或一定比例）则启动熔断机制；

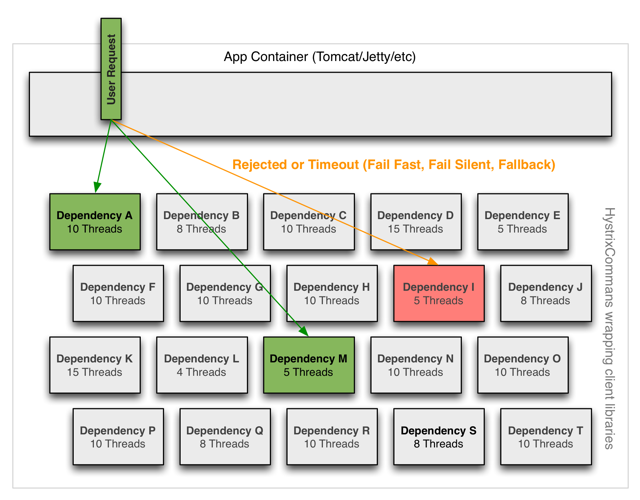
Open：熔断器打开状态，此时对下游的调用都内部直接返回错误，不走网络，但设计了一个时钟选项，默认的时钟达到了一定时间（这个时间一般设置成平均故障处理时间，也就是MTTR），到了这个时间，进入半熔断状态；

Half-Open：半开状态，允许定量的服务请求，如果调用都成功（或一定比例）则认为恢复了，关闭熔断器，否则认为还没好，又回到熔断器打开状态

HystrixCircuitBreaker实现逻辑：



## 隔离



## 线程池

线程池的优点：

线程在自己的线程池中隔离的好处是：

该应用程序完全保护避免客户端库失控。给定依赖库的池可以填满，但不会影响应用程序的其它部分。

如果客户端库配置错误，线程池的运行状况将快速显示此错误（延迟，超时，拒绝等），您可以处理它（通常通过实时动态属性配置），而不会影响应用程序功能。

如果客户端服务改变性能特征，继而导致需要调整属性（增加/减少超时，改变重试等），这可以通过线程池指标（错误，延迟，超时，拒绝）看到，并且可以在不影响其他客户端请求或用户请求的情况下进行处理。

除了隔离优势之外，拥有专用线程池提供了内置并发性，可以利用它来在同步客户端库之上构建异步外观（类似于Netflix API在Hystrix命令之上构建响应式变成，完全异步的Java API） 。

简而言之，线程池提供的隔离允许客户端库和子系统性能特征的始终变化和动态组合被优雅地处理而不引起中断。

注意：尽管单独的线程提供了隔离，您的底层客户端代码也应该有超时和/或响应线程中断，因此它不能无限期地阻塞和饱和Hystrix线程池。

线程池的缺点：

线程池的主要缺点是它们增加了计算开销。每个命令执行涉及在单独的线程上运行命令涉及的排队，调度和上下文切换。

Netflix在设计此系统时，决定接受此开销的成本，以换取其提供的好处，并认为该成本小到不会对成本或性能产生重大影响。

# 核心源码分析

## execute()相关逻辑

/\*\*

\* This decorates "Hystrix" functionality around the run() Observable.

\*

\* @return R

\*/

private Observable<R> executeCommandAndObserve(final AbstractCommand<R> \_cmd) {

final HystrixRequestContext currentRequestContext = HystrixRequestContext.getContextForCurrentThread();

final Action1<R> markEmits = ...

final Action0 markOnCompleted = ...

final Func1<Throwable, Observable<R>> handleFallback = new Func1<Throwable, Observable<R>>() {

@Override

public Observable<R> call(Throwable t) {

Exception e = getExceptionFromThrowable(t);

executionResult = executionResult.setExecutionException(e);

if (e instanceof RejectedExecutionException) {

return handleThreadPoolRejectionViaFallback(e);

} else if (t instanceof HystrixTimeoutException) {

return handleTimeoutViaFallback();

} else if (t instanceof HystrixBadRequestException) {

return handleBadRequestByEmittingError(e);

} else {

if (e instanceof HystrixBadRequestException) {

eventNotifier.markEvent(HystrixEventType.BAD\_REQUEST, commandKey);

return Observable.error(e);

}

return handleFailureViaFallback(e);

}

}

};

final Action1<Notification<? super R>> setRequestContext = new Action1<Notification<? super R>>() {

@Override

public void call(Notification<? super R> rNotification) {

setRequestContextIfNeeded(currentRequestContext);

}

};

Observable<R> execution;

if (properties.executionTimeoutEnabled().get()) {

execution = executeCommandWithSpecifiedIsolation(\_cmd)

.lift(new HystrixObservableTimeoutOperator<R>(\_cmd));

} else {

execution = executeCommandWithSpecifiedIsolation(\_cmd);

}

return execution.doOnNext(markEmits)

.doOnCompleted(markOnCompleted)

.onErrorResumeNext(handleFallback)

.doOnEach(setRequestContext);

}

public Observable<R> call() {

executionResult = executionResult.setExecutionOccurred();

if (!commandState.compareAndSet(CommandState.OBSERVABLE\_CHAIN\_CREATED, CommandState.USER\_CODE\_EXECUTED)) {

return Observable.error(new IllegalStateException("execution attempted while in state : " + commandState.get().name()));

}

metrics.markCommandStart(commandKey, threadPoolKey, ExecutionIsolationStrategy.THREAD);

if (isCommandTimedOut.get() == TimedOutStatus.TIMED\_OUT) {

return Observable.error(new RuntimeException("timed out before executing run()"));

}

if (threadState.compareAndSet(ThreadState.NOT\_USING\_THREAD, ThreadState.STARTED)) {

HystrixCounters.incrementGlobalConcurrentThreads();

threadPool.markThreadExecution();

endCurrentThreadExecutingCommand = Hystrix.startCurrentThreadExecutingCommand(getCommandKey());

executionResult = executionResult.setExecutedInThread();

try {

executionHook.onThreadStart(\_cmd);

executionHook.onRunStart(\_cmd);

executionHook.onExecutionStart(\_cmd);

return getUserExecutionObservable(\_cmd);

} catch (Throwable ex) {

return Observable.error(ex);

}

} else {

return Observable.error(new RuntimeException("unsubscribed before executing run()"));

}

}

@Override

final protected Observable<R> getExecutionObservable() {

return Observable.defer(new Func0<Observable<R>>() {

@Override

public Observable<R> call() {

try {

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());

}

});

}}

## 熔断器相关逻辑

@Override

public boolean allowRequest() {

...

return !isOpen() || allowSingleTest();

}

public boolean allowSingleTest() {

long timeCircuitOpenedOrWasLastTested = circuitOpenedOrLastTestedTime.get();

...

if (circuitOpen.get() && System.currentTimeMillis() > timeCircuitOpenedOrWasLastTested + properties.circuitBreakerSleepWindowInMilliseconds().get()) {

...

if (circuitOpenedOrLastTestedTime.compareAndSet(timeCircuitOpenedOrWasLastTested, System.currentTimeMillis())) {

return true;

}

}

return false;

}

@Override

public boolean isOpen() {

...

if (health.getTotalRequests() < properties.circuitBreakerRequestVolumeThreshold().get()) {

// we are not past the minimum volume threshold for the statisticalWindow so we'll return false immediately and not calculate anything

return false;

}

if (health.getErrorPercentage() < properties.circuitBreakerErrorThresholdPercentage().get()) {

return false;

} else {

if (circuitOpen.compareAndSet(false, true)) {

circuitOpenedOrLastTestedTime.set(System.currentTimeMillis());

return true;

} else {

return true;

}

}

}

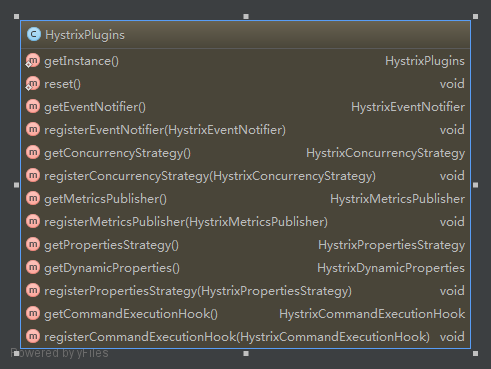
# Hystrix对扩展的支持

## HystrixPlugins简介

我们可以通过实现Hystrix插件的方式来修改或者扩展其功能，并使用HystrixPlugins类在你的应用中注册你自己的实现类，然后Hystrix会在所有的HystrixCommand、HystrixObservableCommand中应用这些拓展插件，以达到扩展开发的目的。

HystrixPlugins插件主要包括EventNotifier、MetricsPublisher、PropertiesStrategy、ConcurrencyStrategy、CommandExceutionHook五大类。每个类型的插件都在Hystrix中有默认方式的实现，当没有声明注册指定插件实现时，将使用其默认的实现。

HystrixPlugins类图如下：



## 使用扩展插件

当您第一次调用HystrixCommand时，它开始访问由插件管理的功能， 由于您无法在运行时更换插件，因此需要在第一次调用HystrixCommand（或在环境初始化时初始化时）加载自定义插件。

可以使用如下代码注册一个自定义的插件：

HystrixPlugins.getInstance().registerCommandExecutionHook(new HystrixPluginsImpl.ExecutionHook());

下面对这几个扩展插件类的功能作简要介绍：

EventNotifier：

对HystrixCommand和HystrixObservableCommand执行过程中触发的所有事件进行扩展，可以发送到报警或数据收集系统。

MetricsPublisher：

对每个给定的HystrixCommandKey的所有HystrixCommands，都会有HystrixMetrics指标的快照，可以通过MetricsPublisher将快照数据发送出去，在其他的进程/系统中来处理某些事情，例如持久化、统计、报警。

PropertiesStrategy：

实现自定义的HystrixPropertiesStrategy，则可以完全控制系统属性的定义，包括Command属性和线程池属性等，这些将直接影响Hyxtrix对并发控制、容错、熔断的行为表现。

ConcurrencyStrategy：

Hystrix使用ThreadLocal，Callable，Runnable，ThreadPoolExecutor和BlockingQueue的实现作为其线程隔离和请求范围功能的一部分。默认情况下Hystrix有实现“开箱即用”，但许多环境更希望使用自定义的替代品，因此这个插件允许注入自定义实现或装饰行为。

您可以使用以下代码实现HystrixConcurrencyStrategy类：

getThreadPool（）和getBlockingQueue（）方法是注入所选实现的简单选项，或者只是一个带有额外日志和指标的修饰版本。

wrapCallable（）方法允许你装饰每个由Hystrix执行的Callable。这对于依赖于ThreadLocal状态用于应用程序功能的系统来说是必要的。包装Callable可以根据需要从父线程捕获和复制状态和参数到子线程。

getRequestVariable（）方法期望HystrixRequestVariable <T>的实现，它的功能类似于ThreadLocal，除了作用范围内的请求 - 在请求中的所有线程上可用。通常，使用HystrixRequestContext和它自己的HystrixRequestVariable的默认实现已经足够。

CommandExceutionHook：

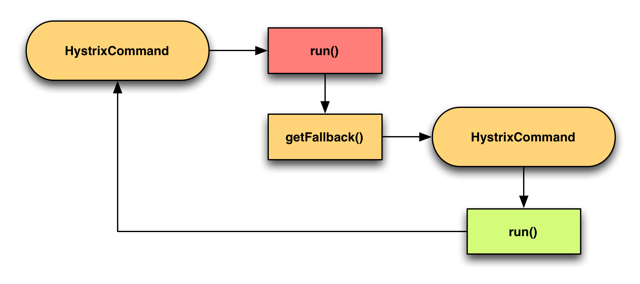
实现HystrixCommandExecutionHook可以访问HystrixCommand或HystrixObservableCommand的执行生命周期，以便可以注入行为，记录，覆盖响应，更改线程状态等，包括onStart、onEmit、onSuccess、onFallbackSuccess等等。

# 当前系统接入Hystrix的使用场景

## 可降级的外部依赖

如epay对风控、积分、活动、urs、短信等系统的依赖，都可以使用HystrixCommand命令封装一层，以获得在外部系统不可用的时候的降级结果并做统一的处理，而不必每次每次关心try catch逻辑。代码示例见[2.4 降级](#_降级)。

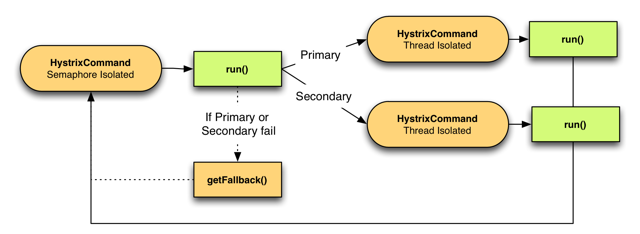
这里需要关注一种情况，即在降级逻辑中，需要再次发起一个外部依赖请求时，由于降级将通过网络，这是另一个可能的故障点，因此它还需要由HystrixCommand或HystrixObservableCommand封装。



## 具有首选和备用通道的依赖

有时次要或故障转移被认为是故障状态，它仅用于降级；在这些情况下，它将适合与上述“通过网络缓存”相同的模式。

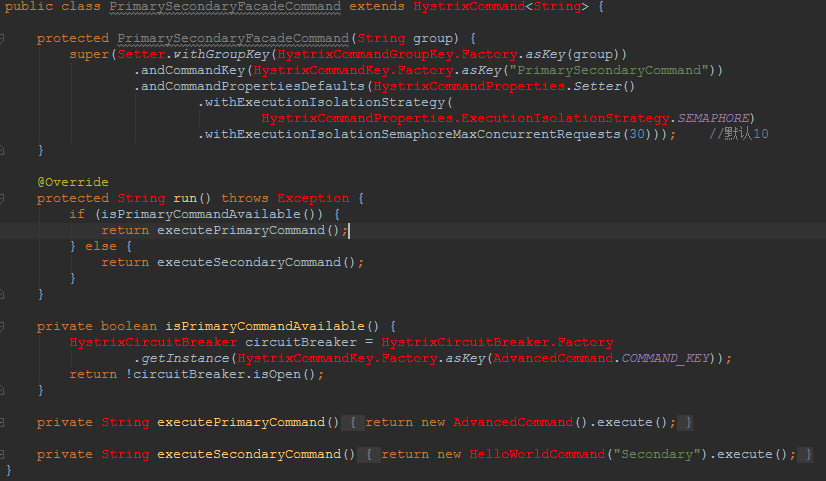
然而，如果翻转到辅助系统是常见的，则每次主节点将处于故障状态，脱扣断路器和点火警报时，都将使用辅助系统，如发送短信功能，集团短信通道和云信短信通道都用于发送短信，他们的对epay的作用是等价的 ，并根据他们自身的状态需要我们随时进行通道切换。因此，在这种情况下，策略反而是将初级和次级之间的切换视为正常，健康的模式，并在其前面放置一个Facade。



主和辅助HystrixCommand实现是线程隔离的，因为它们正在做网络流量和业务逻辑。可以不公开这两个命令，而是将它们隐藏在信号量隔离的另一个HystrixCommand之后，并实现关于是调用主命令还是辅助命令的条件逻辑。如果主通道失败和辅助失败，则控制切换到Facade命令本身的fallback。

FacadeHystrixCommand可以使用信号量隔离，因为它正在进行的所有工作都通过已经线程隔离的另外两个HystrixCommands。只要外观的run()方法不进行任何其他网络调用，重试逻辑或其他“容易出错”的事情，就不必再有一层线程。

核心示例代码如下：



目前系统里有的是根据msgprop配置通道的优先级T1|T2|T3，优先发送到T1通道，重发时使用T2通道。而当T1通道不可用时，用户发送短信验证码时仍然会优先使用T1通道，然后出现报错；目前的做法，出现这种情况的时候往往都是通过人工干预刷新msgprop配置，改为T2|T3|T1,来解决问题，直到T1通道恢复稳定后，再人工把配置改回来。而当使用上述主备通道切换的方式，可以实现当指定通道不可用时，自动切换到下一个通道，而不是在一个已经出现故障的通道上，“明知山有虎偏向虎山行”。

那么现在有另一个问题，如果使用通道切换的方式，msgprop的配置还需要保留吗？个人觉得可以保留，原因如下：1.此配置可以指定，在所有通道都正常的情况下，系统优先该使用哪个通道（即谁是Primary谁是Secondary）2.不同通道后端的表现行为不一定完全相同（如接入的运营商不同等），可能存在对某一用户而言，T1通道走不通但T2通道能走通的情况，故而当用户重试时（短信重发）即使T1通道是可用状态的，也应保留路由到T2通道的逻辑。也就是说，主备通道切换是解决可用率的问题，msgprop是解决优先级的问题。

即先根据msgprop判断T1通道，然后检查T1通道的熔断状态，如果是断开的，则再根据msgprop选取T2通道，再判断T2通道状态，依次类推。