Hystrix 源码解析

枕边书

简介

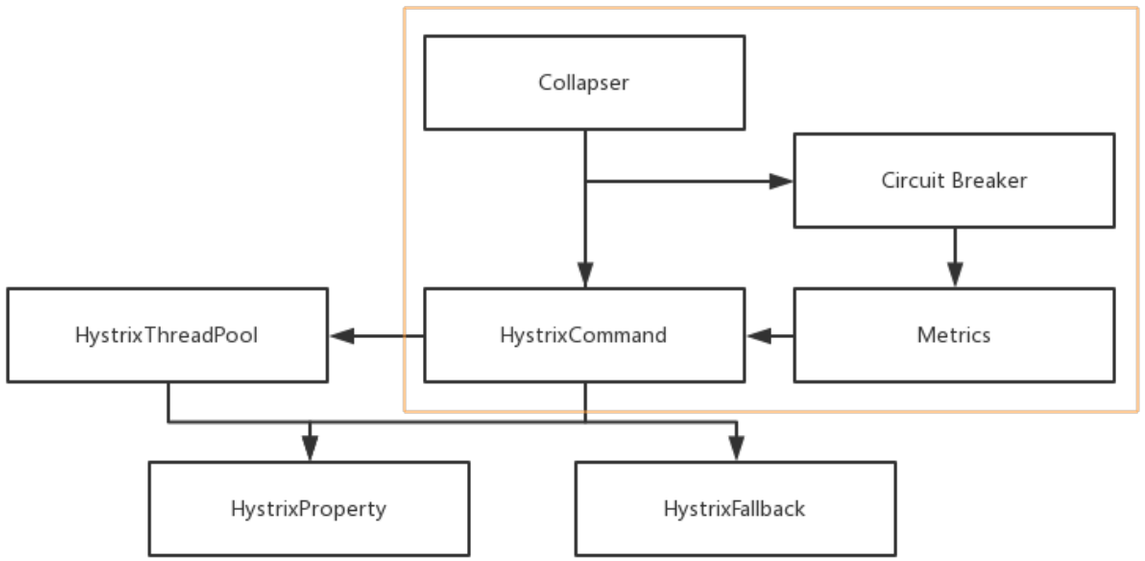
Hystrix 是 Netflix开源的一款容错系统，能帮助使用者码出具备强大的容错能力和鲁棒性的程序。

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

功能

- 故障熔断，错误率高时熔断后续请求，快速失败
- 服务降级，异常时回退执行
- 请求隔离，用线程池大小和信号量限制并发数
- 请求合并，将单个逻辑合并成批量逻辑
- 结果缓存，缓存同样key的请求结果

基础组件



自底向上

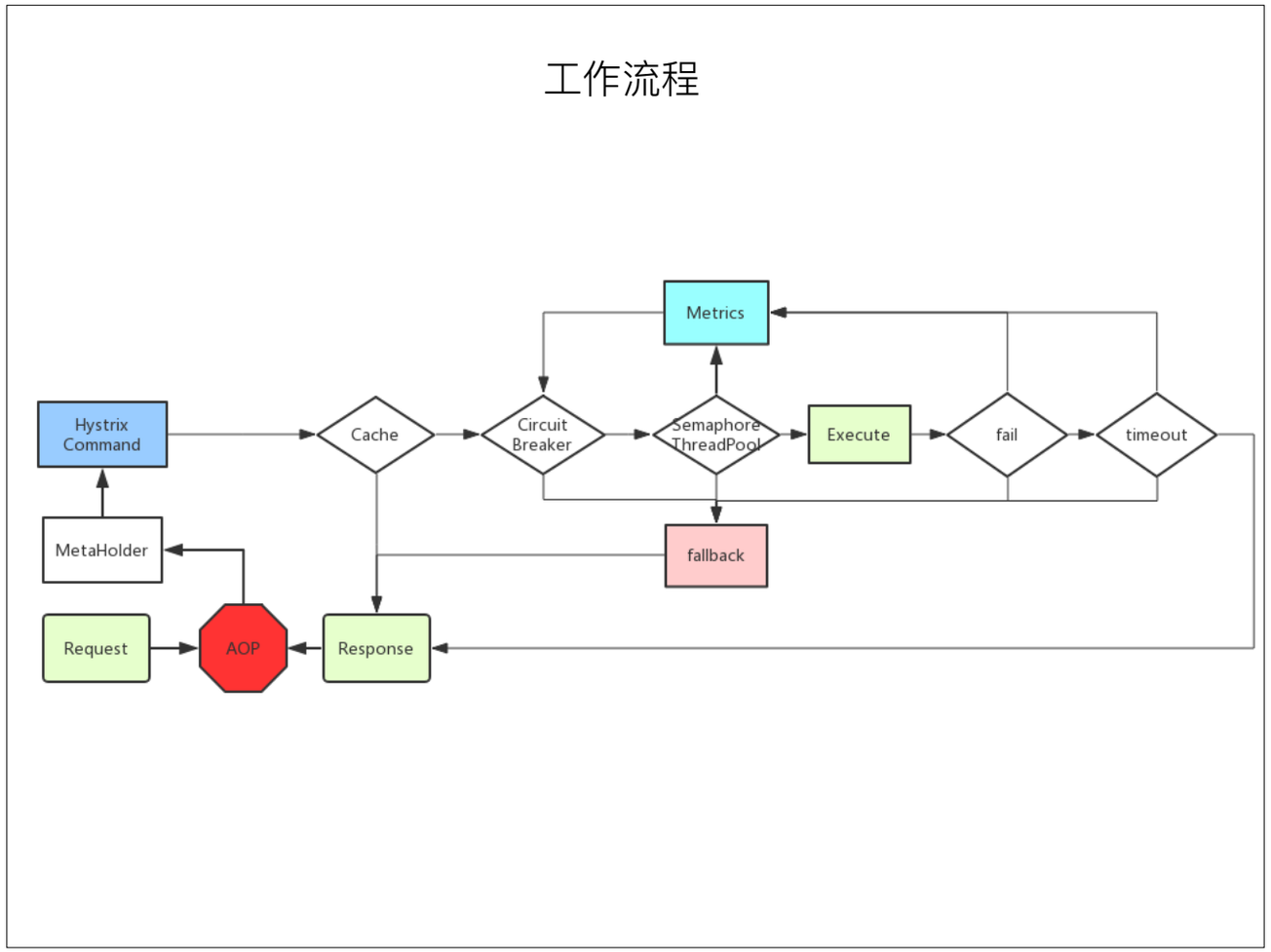
hystrixCommand 、Metric 、CircuitBreaker 形成一个依赖圈

在Spring内实现原理

- AOP
- HystrixCommandAspect 切面
- HystrixCommand 和 HystrixCollasper 两个切点

```
77      @Pointcut("@annotation(com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand)")
78
79      public void hystrixCommandAnnotationPointcut() {
80      }
81
82      @Pointcut("@annotation(com.netflix.hystrix.contrib.javanica.annotation.HystrixCollapser)")
83      public void hystrixCollapserAnnotationPointcut() {
84      }
85
86      @Around("hystrixCommandAnnotationPointcut() || hystrixCollapserAnnotationPointcut()")
87  @   public Object methodsAnnotatedWithHystrixCommand(final ProceedingJoinPoint joinPoint) throws Throwable {
88          Method method = getMethodFromTarget(joinPoint)
```

还可以通过继承实现 hystrix 提供的 command 等方式来实现



业务数据流程

通过上面的流程发现有很多分支，而且有多个事件源相互依赖（metric->command->breaker->metrics），如果使用传统的编程方式，需要有很多 if return。

而且这些流程都是专有的，无法复用。

响应式编程（Reactive Programming）

- 我们一般写的程序(过程或对象) 统称为命令式程序，是以流程为核心的，每一行代码实际上都是机器实际上要执行的指令。
- 而Rxjava的编程风格，称为函数响应式编程。函数响应式编程是以数据流为核心，处理数据的输入，处理以及输出的。

脱胎于观察者模式：跟观察者模式不一样的是，如果没有观察者，不会发射事件，有些像拉模式，在订阅后开始获取事件。
观察者模式下，我们让观察者在被观察者处注册，一旦有事件发生即通知被观察者。
Rp 即一开始我们定义一个方法计划表，表里预定义几个事件函数，一般有 `onNext onComplete onError`, 一旦某些事件被触发，就会按照方法表查找并调用对应的方法。

优点：模块解耦事件通知，解决嵌套回调，代码有逼格
缺点：入门难，代码不好追

RxJava

根据RP编程思想在 JVM 实现的框架，它通过观察者队列的做法，将消息的异步处理与基于事件的编程进行很好地结合。

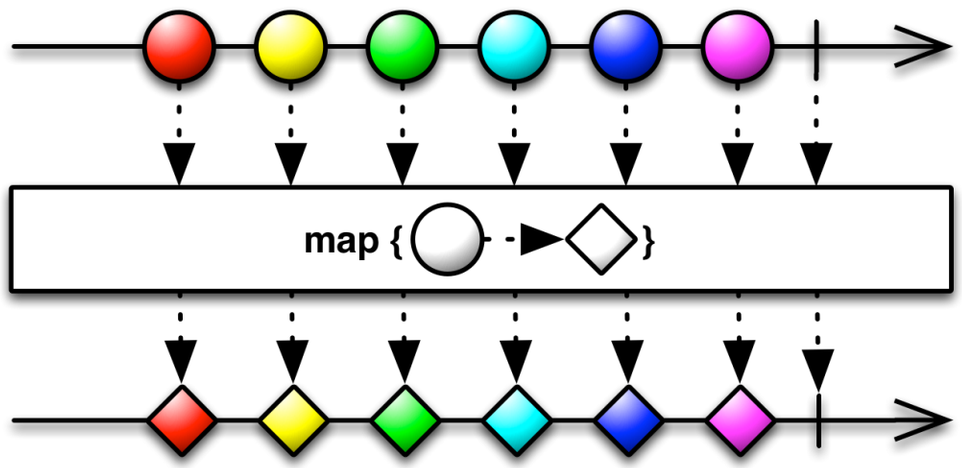
GitHub: https://github.com/ReactiveX

安卓用得比较多，主要处理很多的用户事件

一个在Java虚拟机上实现的响应式扩展库：提供了基于observable序列实现的异步调用及基于事件编程。 它扩展了观察者模式，支持数据、事件序列并允许你合并序列，无需关心底层的线程处理、同步、线程安全、并发数据结构和非阻塞I/O处理。

RxJava 响应式编程

Observable 事件发射源、或触发器



Operator:
事件加工器

Observer/ Subscribe 事件接收处理器

Observable 传入 OnSubscribe 对象创建，对象有 call 方法来定义被订阅后要做的事

有 interval() defer() just()等方法，defer 传入一个 observable 工厂在订阅时 创建 Observable

Observer Subscriber 是接收器 一般有 onNext onComplete onError 方法分别用来接收处理对应的消息，Subscriber 多了个 unsubscribe() 方法

Operator 有 map()、window()、buffer() 、filter()分别处理单个事件或合并时间容器内的事件，跟 Java 的 stream 编程类似

Subject 事件中转站，即可作为发射源也可作为接收器

如果不清楚，可以对比观察者模式

Code: RxJava

```java
public static void main(String[] args) {
    Subscriber<String> mySubscriber = new Subscriber<String>() {
        @Override
        public void onCompleted() { }
        @Override
        public void onError(Throwable e) { }
        @Override
        public void onNext(String s) { System.out.println(s); }
    };

    Observable<String> myObservable = Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            subscriber.onNext( t: "hello world!"); // hello world!
        }
    });

    myObservable.subscribe(mySubscriber);
}
```

```java
public static void main(String[] args) {
    Observable.just("Hello, world!")
            .map(new Func1<String, String>() {
                @Override
                public String call(String s) { return s + " test"; }
            })
            .subscribe(new Action1<String>() {
                @Override
                public void call(String s) { System.out.println(s); // Hello, world! test }
            });
}
```

上面是基础使用方式

下面是高级使用方式 subscribe() 有 subscribe(final Action1<? super T> onNext) 形式

Action0-ActionN 分别传 1-N+1 个参数，不返回值

Func0-FuncN 分别传 1-N+1 个参数，返回一个值

Hystrix 里的典型 RxJava

```
483  @  private Observable<R> executeCommandAndObserve(final AbstractCommand<R> _cmd) {
484         final HystrixRequestContext currentRequestContext = HystrixRequestContext.getContextForCurrentThread();
485
486         final Action1<R> markEmits = new Action1<R>() {...};
502
503         final Action0 markOnCompleted = new Action0() {...};
515
516         final Func1<Throwable, Observable<R>> handleFallback = new Func1<Throwable, Observable<R>>() {...};
541
542         final Action1<Notification<? super R>> setRequestContext = new Action1<Notification<? super R>>() {...};
548
549         Observable<R> execution;
550         if (properties.executionTimeoutEnabled().get()) {
551             execution = executeCommandWithSpecifiedIsolation(_cmd)
552                     .lift(new HystrixObservableTimeoutOperator<R>(_cmd));
553         } else {
554             execution = executeCommandWithSpecifiedIsolation(_cmd);
555         }
556
557         return execution.doOnNext(markEmits)
558                 .doOnCompleted(markOnCompleted)
559                 .onErrorResumeNext(handleFallback)
560                 .doOnEach(setRequestContext);
561     }
```

创建了任务之前、任务开始、任务完成、任务出错 四个事件处理方法，注册到事件发射器上

一旦 hystrix 任务执行时到对应的时机，立刻调用对应的方法

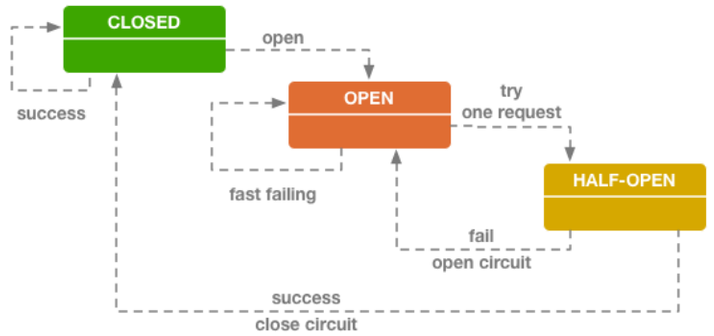
Circuit Breaker

& Metrics

Circuit Breaker

- 熔断策略
  - 强制开/关
  - 总请求数
  - 错误百分比
- 常用配置
  - 熔断器打开时尝试关闭的时间延迟

Circuit Breaker FSM



Circuit Breaker State Diagram

Code: Circuit Breaker

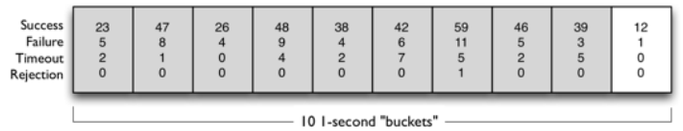
```
159         private Subscription subscribeToStream() {
160             return metrics.getHealthCountsStream().observe()
161                     .subscribe(new Subscriber<HealthCounts>() {  // 订阅处理 getHealthCountsStream 的每个事件
162                         @Override
163                         public void onCompleted() { }
164                         @Override
165                         public void onError(Throwable e) { }
166                         @Override
167                         public void onNext(HealthCounts hc) {
168                             if (hc.getTotalRequests() < properties.circuitBreakerRequestVolumeThreshold().get()) { // 配置 circuitBreaker.requestVolumeThreshold
169                             } else {
170                                 if (hc.getErrorPercentage() < properties.circuitBreakerErrorThresholdPercentage().get()) { // 配置 circuitBreaker
    .errorThresholdPercentage
171                                 } else {
172                                     if (status.compareAndSet(Status.CLOSED, Status.OPEN)) {
173                                         circuitOpened.set(System.currentTimeMillis());
174                                     }
175                                 }
176                             }
177                         }
178                     });
```

```
232         private boolean isAfterSleepWindow() {
233             final long circuitOpenTime = circuitOpened.get();
234             final long currentTime = System.currentTimeMillis();
235             final long sleepWindowTime = properties.circuitBreakerSleepWindowInMilliseconds().get(); // 配置 circuitBreaker.sleepWindowInMilliseconds
236             return currentTime > circuitOpenTime + sleepWindowTime;
237         }
238         @Override
239         public boolean attemptExecution() { // 在熔断器打开时，尝试执行，成功后会将熔断器关闭
240             if (properties.circuitBreakerForceOpen().get()) { return false; } // 配置 circuitBreaker.forceOpen
241             if (properties.circuitBreakerForceClosed().get()) { return true; } // 配置 circuitBreaker.forceClosed
242             if (circuitOpened.get() == -1) {
243                 return true;
244             } else {
245                 if (isAfterSleepWindow()) {
246                     if (status.compareAndSet(Status.OPEN, Status.HALF_OPEN)) { // 如果过了熔断时间，尝试将状态置为半开，只有一个线程能够成功
247                         return true;
248                     } else {
249                         return false;
250                     }
251                 } else {
252                     return false;
253                 }
254             }
255         }
256     }
```

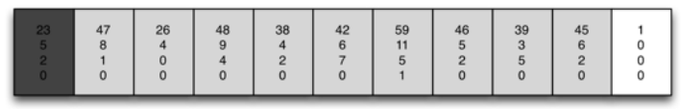
此外，方法执行成功后尝试将状态从“半开” 切换到 “关闭”
方法执行失败后尝试将状态从 “半开” 切换到 “打开”

Metric Entity

- Stream



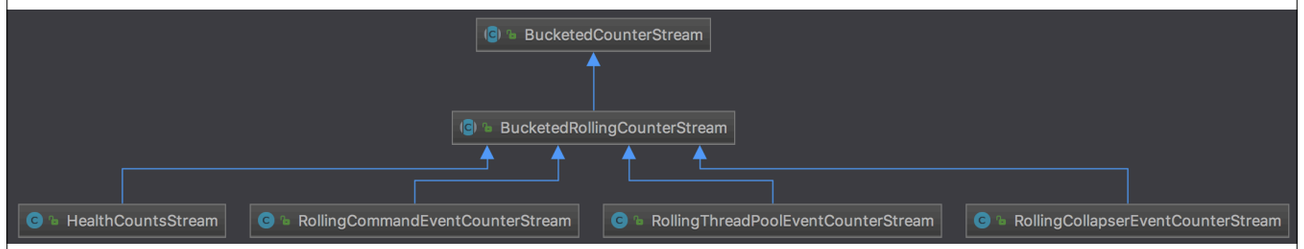
- Bucket

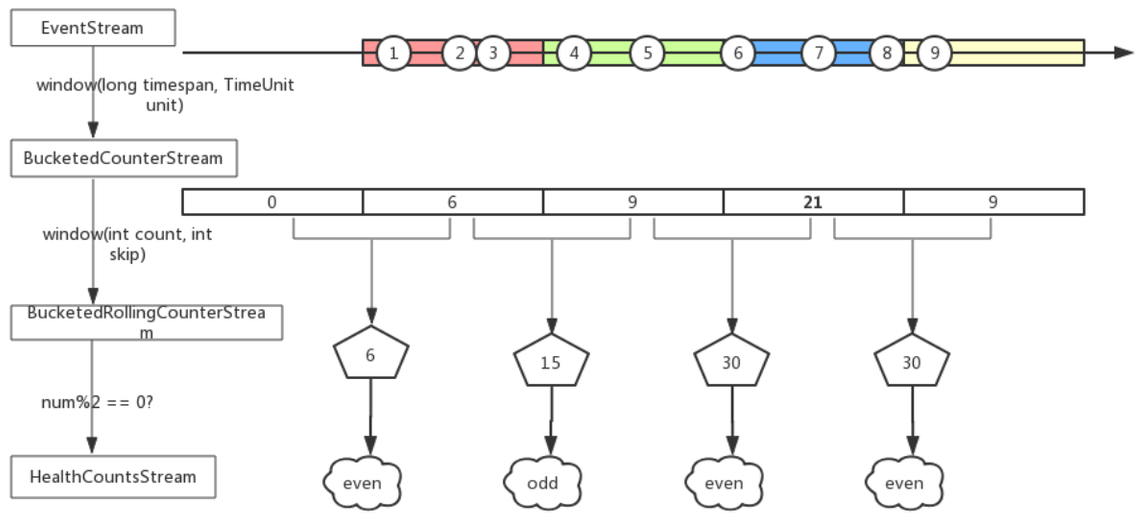


On "getLatestBucket" if the 1-second window is passed a new bucket is created, the rest slid over and the oldest one dropped.

Stream 就是 Subject， 即接收 completionEvent， 又提供新的事件

继承体系





BucketedCounterStream 将事件聚合成桶，提供桶流

BucketedRollingCounterStream 将桶聚合成滑动窗口，提供滑动窗口流

HealthCountsStream 将滑动窗口聚合成健康数据，提供健康数据流

从抽象到具体

Code: Slide Window

```
        this.numBuckets = numBuckets;
        this.reduceBucketToSummary = new Func1<Observable<Event>, Observable<Bucket>>() {
            @Override
            public Observable<Bucket> call(Observable<Event> eventBucket) {
                return eventBucket.reduce(getEmptyBucketSummary(), appendRawEventToBucket);
            }
        };

        final List<Bucket> emptyEventCountsToStart = new ArrayList<Bucket>();
        for (int i = 0; i < numBuckets; i++) {
            emptyEventCountsToStart.add(getEmptyBucketSummary());
        }

        this.bucketedStream = Observable.defer(new Func0<Observable<Bucket>>() {
            @Override
            public Observable<Bucket> call() {
                return inputEventStream
                        .observe()
                        .window(bucketSizeInMs, TimeUnit.MILLISECONDS) // 按单元窗口长度来将某个时间段内的调用事件聚集起来
                        .flatMap(reduceBucketToSummary)                // 将每个单元窗口内聚集起来的事件集合聚合成桶
                        .startWith(emptyEventCountsToStart);           // 保证逻辑统一，初始化一定数量的空桶
            }
        });
    }
```

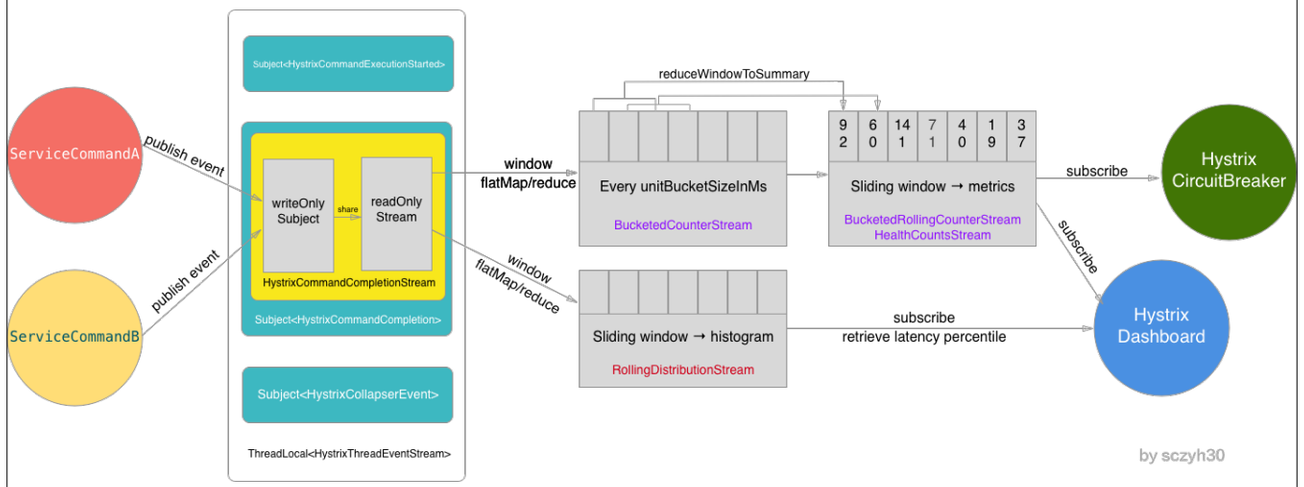
```
        Func1<Observable<Bucket>, Observable<Output>> reduceWindowToSummary = new Func1<Observable<Bucket>, Observable<Output>>() {
            @Override
            public Observable<Output> call(Observable<Bucket> window) {
                return window.scan(getEmptyOutputValue(), reduceBucket).skip(numBuckets);
            }
        };
        this.sourceStream = bucketedStream      // 事件流换成桶流
                .window(numBuckets, skip: 1)          // 以步长为1，将 numBuckets 个 bucket集合成一个桶集合
                .flatMap(reduceWindowToSummary) // 将桶多个桶聚合成数据对象，如 健康数据
                .doOnSubscribe(new Action0() {
                    @Override
                    public void call() { isSourceCurrentlySubscribed.set(true); }
                })
                .doOnUnsubscribe(new Action0() {
                    @Override
                    public void call() { isSourceCurrentlySubscribed.set(false); }
                })
                .share()                        // 各个订阅者会读到相同数据
                .onBackpressureDrop();          // 消费慢的话数据不会累积
    }
```

图一： 将事件流（完成事件）聚合成多个桶

图二：将多个桶聚合成滑动窗口

ErrorCount = (failure + timeout + threadPoolRejected + semaphoreRejectedCount)
updatedTotalCount = ErrorCount + succ

滑动窗口整体实现



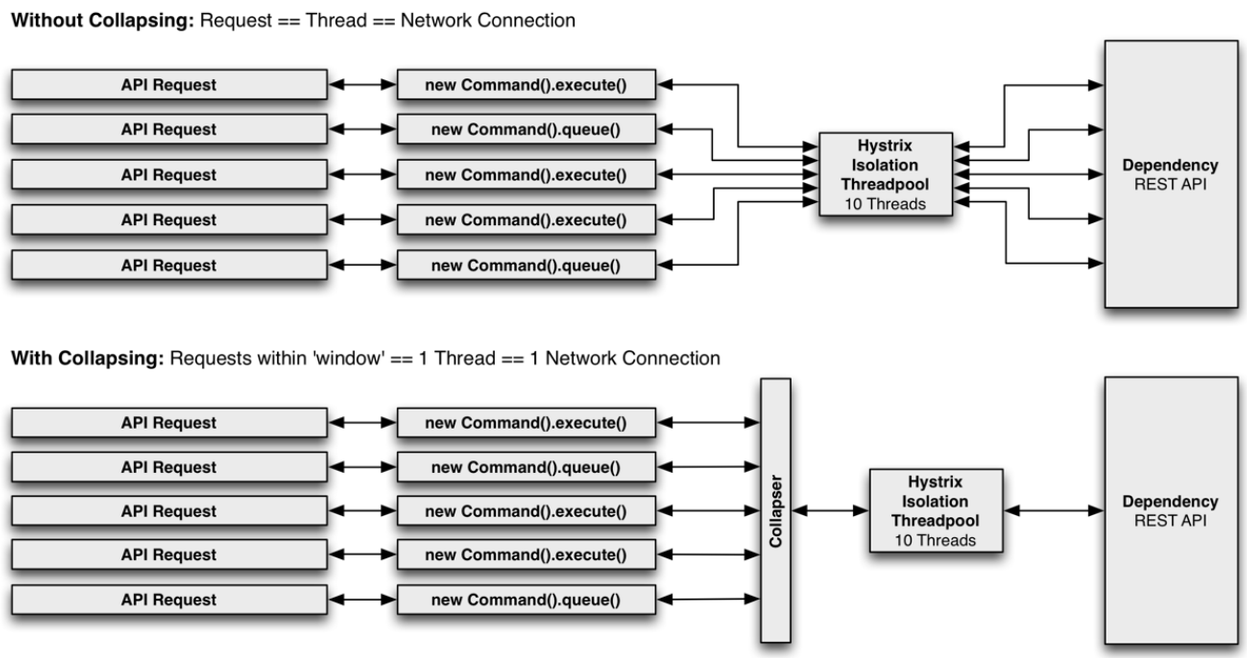
1. HystricCommand 结束后调用 handleCommandEnd 方法将 HystrixCommandCompletion 事件发布到事件流中
2. 事件流通过 window() 方法将事件分组，并通过 flatMap() 方法将事件聚合成桶
3. 再将各个桶使用 window 聚合成滑动窗口
4. 将滑动容口聚合成数据对象（如健康数据流）
5. CircuitBreaker 订阅健康数据流，修改熔断器的开关

HystrixCollasper

请求合并

请求中处理一次系统 I/O 的消耗是非常大的，如果有非常多的请求都进行同一类 I/O 操作，可以将这些 I/O 操作都合并到一起，进行一次 I/O 操作，可以大大降低下游资源服务器的负担。
混到 hystrix 里代码上有点不那么整洁了。

Hystrix collapser



从图看出：

会保留Future 线程

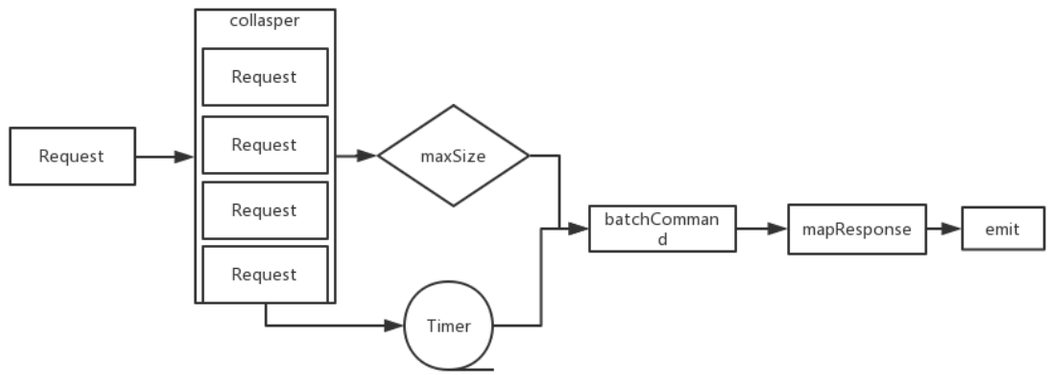
整体会延迟 1/2 整合延时

Sample

```
263  public class HystrixCollapserSample {
264
265      @HystrixCollapser(
266              batchMethod = "batch",
267              collapserKey = "single",
268              scope = com.netflix.hystrix.HystrixCollapser.Scope.GLOBAL,
269              collapserProperties = {
270                      @HystrixProperty(name = "maxRequestsInBatch", value = "100"),
271                      @HystrixProperty(name = "timerDelayInMilliseconds", value = "1000"),
272                      @HystrixProperty(name = "requestCache.enabled", value = "true")
273              })
274      public Future<Boolean> single(String input) {
275          return null; // single方法不会被执行到
276      }
277
278      public List<Boolean> batch(List<String> inputs) {
279          return inputs.stream().map(it -> Boolean.TRUE).collect(Collectors.toList());
280      }
281  }
```

Collapser Workflow

- Collasper 合并器，存储Request
- BatchCommand
- CollapsedTask 消费Request的Timer



1. 在 AOP 内通过切点创建一个 metaHolder , 使用它创建一个 collapser
2. 在切点时，先找到 requestCollapser, 如果 scope 是全局的，从 globalScopedCollapsers(ConcurrentHashMap<String, RequestCollapser>) 里获取 collapser
3. 向 collapser 内提交请求
4. 判断此 collapser 是否已注册 timer, 未注册就注册一个，这个 timer 会定时扫描 batch 中的请求，创建一个合并请求发起
5. timer 通过多个 request 构造一个 batchCommand 然后执行
6. 将执行结果映射到各个请求，交触发各个请求的 onComplete 事件通知完成

总结

- HttpClient
- 数据窗口操作
- 操作转异步

RxJava，非常适合回调场景，可以用在：

- HttpClient 应用回调
- 数据窗口聚合 就用 stream window，如转码的集群状态等
- 做一些异步操作，使用通知机制
- Zip等待全完成 contact() 合并

Thanks!