属性

何时触发熔断

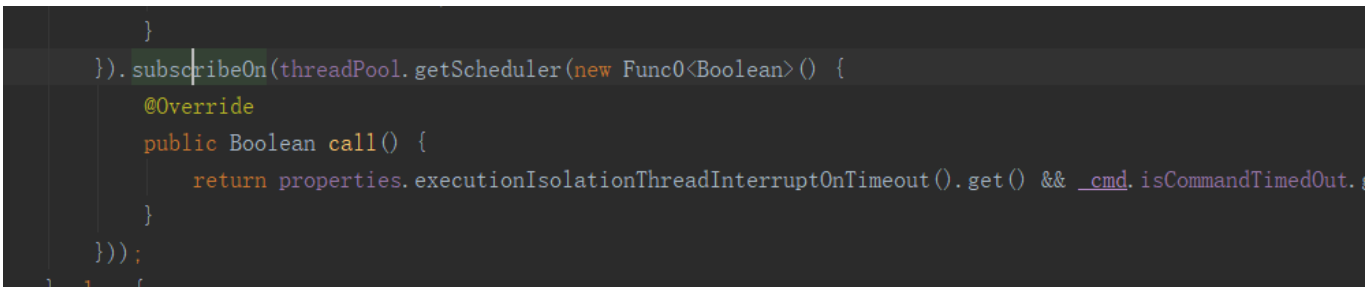
执行模式

https://www.cnblogs.com/cxxjohnson/p/6254967.html

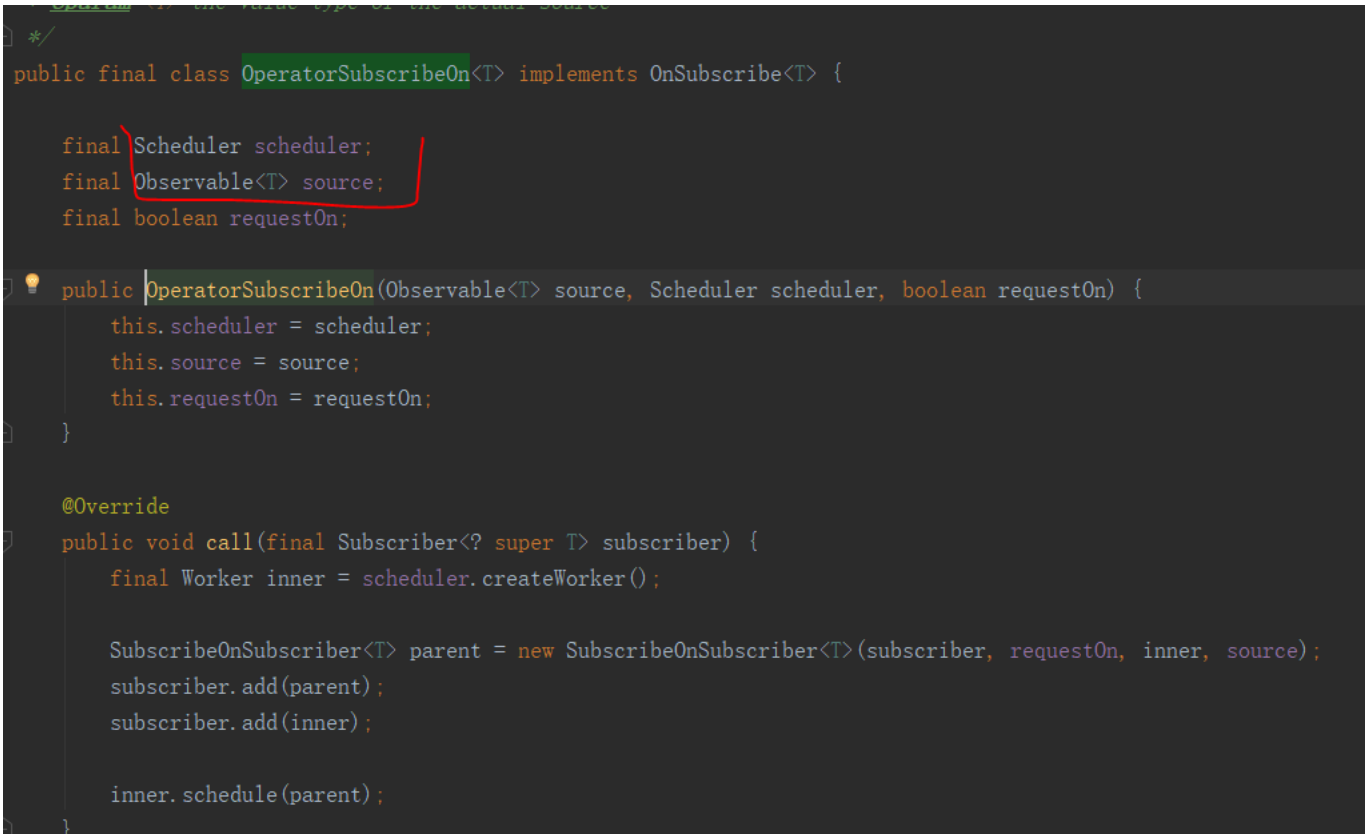
设计模式

执行流程

```
    new Command()
    com.netflix.hystrix.HystrixCommand#queue ->> Future
    com.netflix.hystrix.AbstractCommand#toObservable().toBlocking().toFuture();

    com.netflix.hystrix.AbstractCommand#applyHystrixSemantics
        1. if (circuitBreaker.attemptExecution()) {}
        2. com.netflix.hystrix.AbstractCommand#executeCommandAndObserve
        3.
com.netflix.hystrix.AbstractCommand#executeCommandWithSpecifiedIsolation
        4.
com.netflix.hystrix.AbstractCommand#getUserExecutionObservable().subscribeOn()
                ->>      rx.internal.operators.OperatorSubscribeOn


相关接口
rx.functions.Action0
rx.functions.Func0        ->>   applyHystrixSemantics
```




```
上面截图是提交定时任务的 关键流程
rx.Observable#subscribeOn(rx.Scheduler)
构建生成rx.internal.operators.OperatorSubscribeOn
```

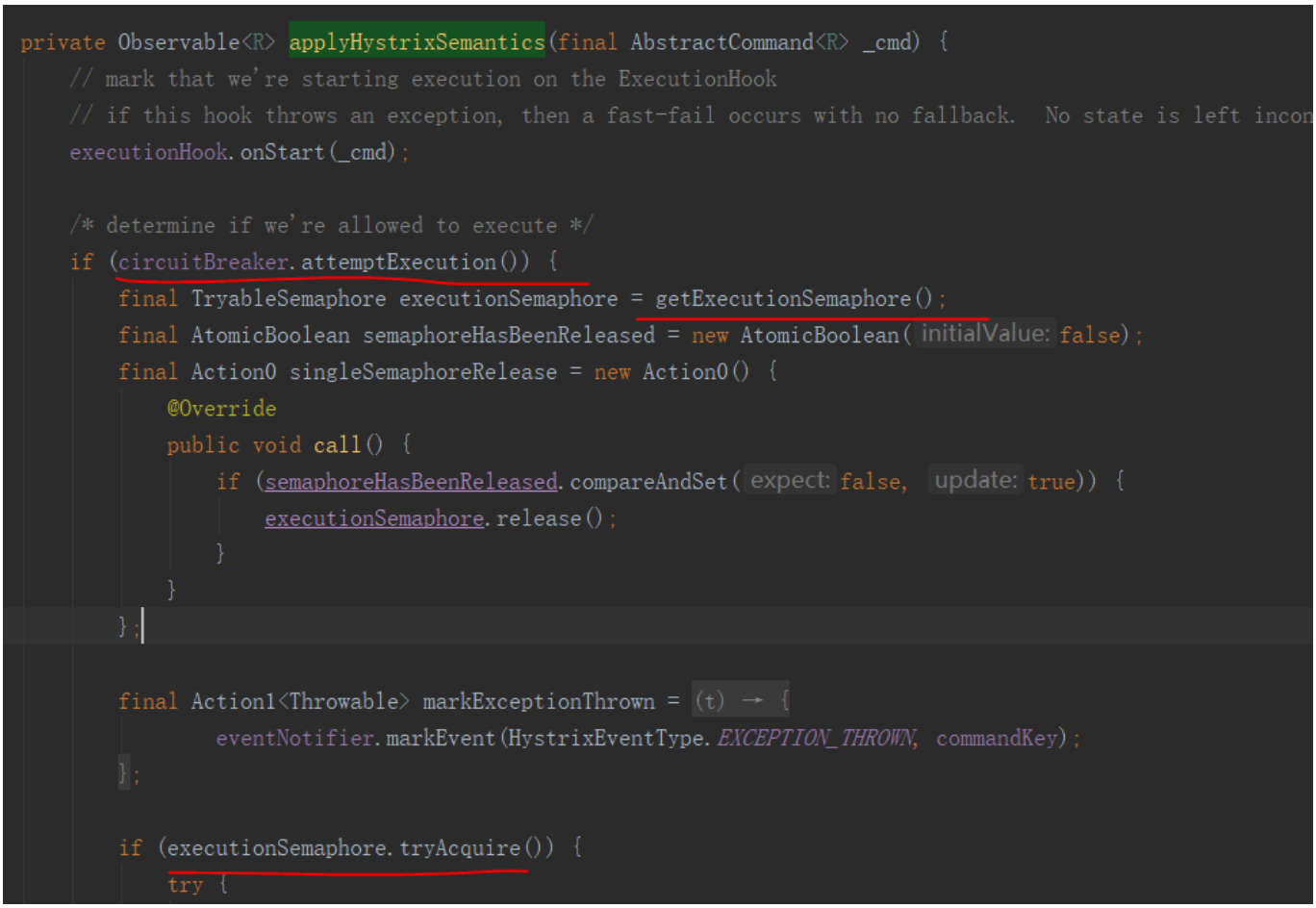


```java
 */
public final class OperatorSubscribeOn<T> implements OnSubscribe<T> {

    final Scheduler scheduler;
    final Observable<T> source;
    final boolean requestOn;

    public OperatorSubscribeOn(Observable<T> source, Scheduler scheduler, boolean requestOn) {
        this.scheduler = scheduler;
        this.source = source;
        this.requestOn = requestOn;
    }

    @Override
    public void call(final Subscriber<? super T> subscriber) {
        final Worker inner = scheduler.createWorker();

        SubscribeOnSubscriber<T> parent = new SubscribeOnSubscriber<T>(subscriber, requestOn, inner, source);
        subscriber.add(parent);
        subscriber.add(inner);

        inner.schedule(parent);
    }
```

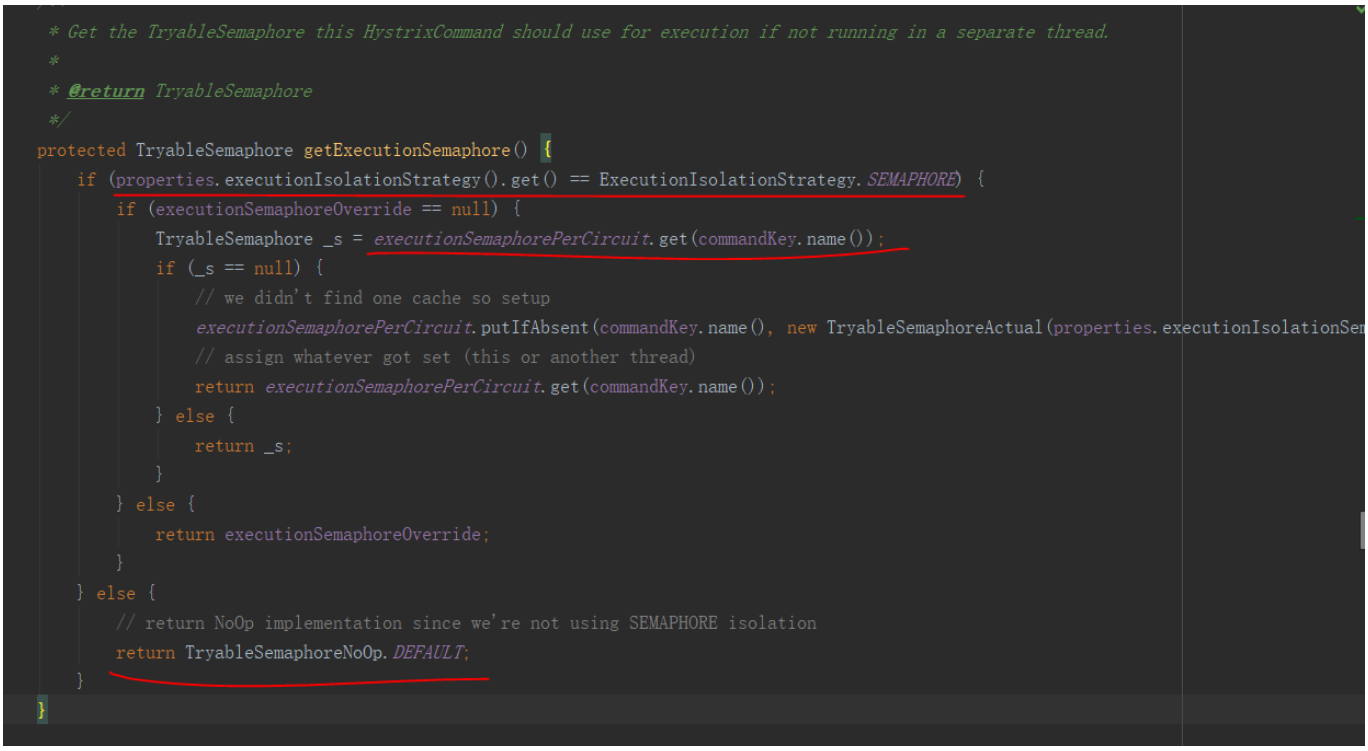
最终会调用 call 提交 Future 任务
https://blog.csdn.net/qpfjalzm123/article/details/80682278

hystrix 隔离

隔离分为线程池隔离和信号量隔离



```
private Observable<R> applyHystrixSemantics(final AbstractCommand<R> _cmd) {
    // mark that we're starting execution on the ExecutionHook
    // if this hook throws an exception, then a fast-fail occurs with no fallback.  No state is left incon
    executionHook.onStart(_cmd);

    /* determine if we're allowed to execute */
    if (circuitBreaker.attemptExecution()) {
        final TryableSemaphore executionSemaphore = getExecutionSemaphore();
        final AtomicBoolean semaphoreHasBeenReleased = new AtomicBoolean( initialValue: false);
        final Action0 singleSemaphoreRelease = new Action0() {
            @Override
            public void call() {
                if (semaphoreHasBeenReleased.compareAndSet( expect: false,  update: true)) {
                    executionSemaphore.release();
                }
            }
        };

        final Action1<Throwable> markExceptionThrown = (t) -> {
                eventNotifier.markEvent(HystrixEventType.EXCEPTION_THROWN, commandKey);
        };

        if (executionSemaphore.tryAcquire()) {
            try {
```



```
 * Get the TryableSemaphore this HystrixCommand should use for execution if not running in a separate thread.
 *
 * @return TryableSemaphore
 */
protected TryableSemaphore getExecutionSemaphore() {
    if (properties.executionIsolationStrategy().get() == ExecutionIsolationStrategy.SEMAPHORE) {
        if (executionSemaphoreOverride == null) {
            TryableSemaphore _s = executionSemaphorePerCircuit.get(commandKey.name());
            if (_s == null) {
                // we didn't find one cache so setup
                executionSemaphorePerCircuit.putIfAbsent(commandKey.name(), new TryableSemaphoreActual(properties.executionIsolationSem
                // assign whatever got set (this or another thread)
                return executionSemaphorePerCircuit.get(commandKey.name());
            } else {
                return _s;
            }
        } else {
            return executionSemaphoreOverride;
        }
    } else {
        // return NoOp implementation since we're not using SEMAPHORE isolation
        return TryableSemaphoreNoOp.DEFAULT;
    }
}
```

hystrix 的失败与成功统计

```java
private Observable<R> applyHystrixSemantics(final AbstractCommand<R> _cmd) {
    executionHook.onStart(_cmd);
    if (circuitBreaker.attemptExecution()) {
        final TryableSemaphore executionSemaphore = getExecutionSemaphore();
        final AtomicBoolean semaphoreHasBeenReleased = new AtomicBoolean( initialValue: false);
        final Action0 singleSemaphoreRelease = () -> {
            if (semaphoreHasBeenReleased.compareAndSet( expect: false,  update: true)) {
                executionSemaphore.release();
            }
        };
        final Action1<Throwable> markExceptionThrown = t -> eventNotifier.markEvent(HystrixEventType.EXCEPTION_THROWN, commandKey)

        if (executionSemaphore.tryAcquire()) {
            try {
                /* used to track userThreadExecutionTime */
                executionResult = executionResult.setInvocationStartTime(System.currentTimeMillis());
                return executeCommandAndObserve(_cmd)
                        .doOnError(markExceptionThrown)
                        .doOnTerminate(singleSemaphoreRelease)
                        .doOnUnsubscribe(singleSemaphoreRelease);
            } catch (RuntimeException e) {
                return Observable.error(e);
            }
        } else {
            return handleSemaphoreRejectionViaFallback();
        }
    } else {
        return handleShortCircuitViaFallback();
    }
}
```

```java
final Func1<Throwable, Observable<R>> handleFallback = new Func1<Throwable, Observable<R>>() {
    @Override
    public Observable<R> call(Throwable t) {
        circuitBreaker.markNonSuccess();
        Exception e = getExceptionFromThrowable(t);
        executionResult = executionResult.setExecutionException(e);
        if (e instanceof RejectedExecutionException) {
            return handleFailureViaFallback(e);
        }
    }
};
final Action1<Notification<? super R>> setRequestContext = 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);
}
```

com.netflix.hystrix.HystrixCircuitBreaker 核心分析

共三种状态
CLOSE(正常状态), OPEN(完全熔断), HALF_OPEN(半熔断)
熔断器有一个配置，叫做窗口长度(sleepWindowTime, 单位Milliseconds),

```
/**
 * @ExcludeFromJavadoc
 * @ThreadSafe
 */
class Factory {
    // String is HystrixCommandKey.name() (we can't use HystrixCommandKey directly as we can't guarantee it implements hashcode/equals co
    private static ConcurrentHashMap<String, HystrixCircuitBreaker> circuitBreakersByCommand = new ConcurrentHashMap<String, HystrixCircu
```

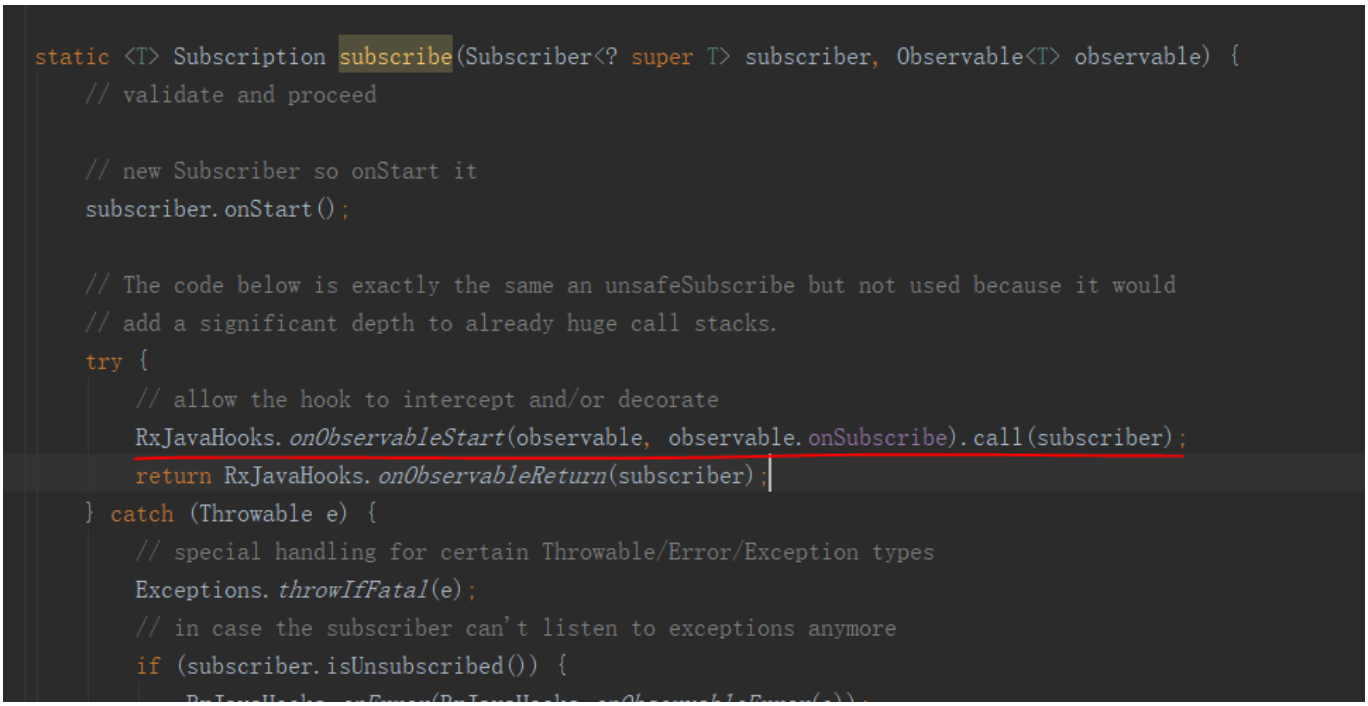
```
/* package */class HystrixCircuitBreakerImpl implements HystrixCircuitBreaker {
    private final HystrixCommandProperties properties;
    private final HystrixCommandMetrics metrics;

    enum Status {
        CLOSED, OPEN, HALF_OPEN;
    }

    private final AtomicReference<Status> status = new AtomicReference<Status>(Status.CLOSED);
    private final AtomicLong circuitOpened = new AtomicLong( initialValue: -1);
    private final AtomicReference<Subscription> activeSubscription = new AtomicReference<>( initialValue: null);
```

```
private boolean isAfterSleepWindow() {
    final long circuitOpenTime = circuitOpened.get();
    final long currentTime = System.currentTimeMillis();
    final long sleepWindowTime = properties.circuitBreakerSleepWindowInMilliseconds().get();
    return currentTime > circuitOpenTime + sleepWindowTime;
}

@Override
public boolean attemptExecution() {
    if (properties.circuitBreakerForceOpen().get()) {
        return false;
    }
    if (properties.circuitBreakerForceClosed().get()) {
        return true;
    }
    if (circuitOpened.get() == -1) {
        return true;
    } else {
        if (isAfterSleepWindow()) {
            if (status.compareAndSet(Status.OPEN, Status.HALF_OPEN)) {
                //only the first request after sleep window should execute
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
}
```

设计模式

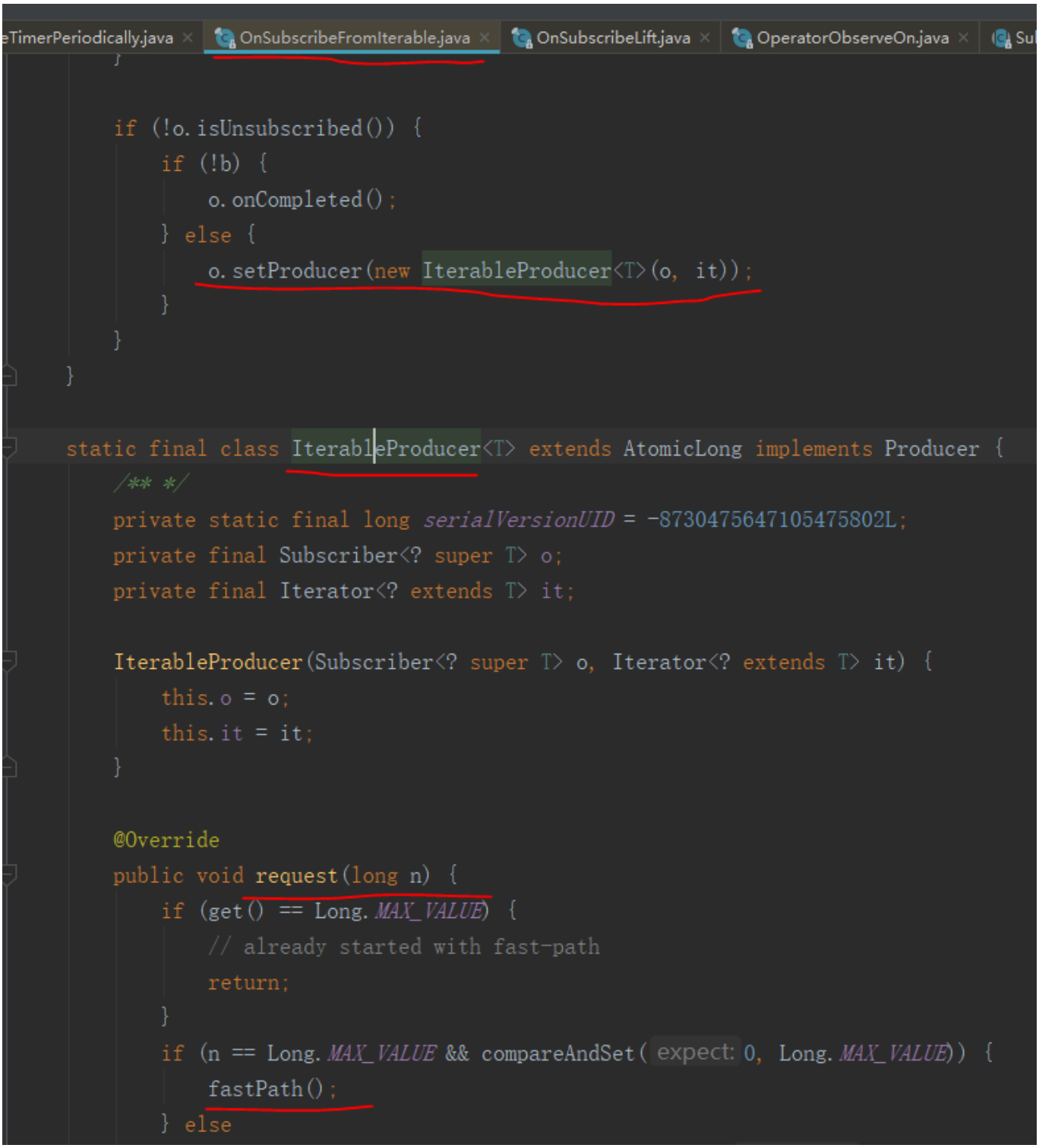
rxjava 示例于源码解读

```
核心类：
    Observer 观察者
        desc: 事件的实际处理方，提供实际的事件处理的方法
        method: onCompleted()
        method: onError()
        method: onNext()

    Subscriber  extend Observer  订阅者
        desc: 配合OnSubscribe 类，提供serProducer方法， 调用OnSubscribe#request(),
启动事件处理，
        method: rx.Subscriber#setProducer  连接


    OnSubscribe     此类是核核心类(负责连接和流程控制，下面会有详细分析两个demo)
        desc: 常用来适配Observable和Observer(常提供Observer的自定义实现), 被
Subscribe#setProducer 调用  request() 方法进行事件传递
        example: OnSubscribeFromIterable   使用迭代器管理事件
        example: OnSubscribeCreate         使用Emitter事件
        example: OnSubscribeConcatMap
        example: OnSubscribeFilter         增加了 FilterSubscriber.predicate, 过滤
消息，然后才真正调用 subscribe
        example: OnSubscribeTimerPeriodically  周期性发送事件
        example: OperatorObserveOn
        example: OnSubscribeFilter



    Observable
        desc: 被观察者，核心是引用了一个OnSubscribe类，并且提供各种静态方法创建不同的
OnSubscribe
        var: OnSubscribe
        method: rx.Observable#subscribe()  核心方法, 使用 OnSubscribe 调用
Subscriber(继承自Observer)的方法
```



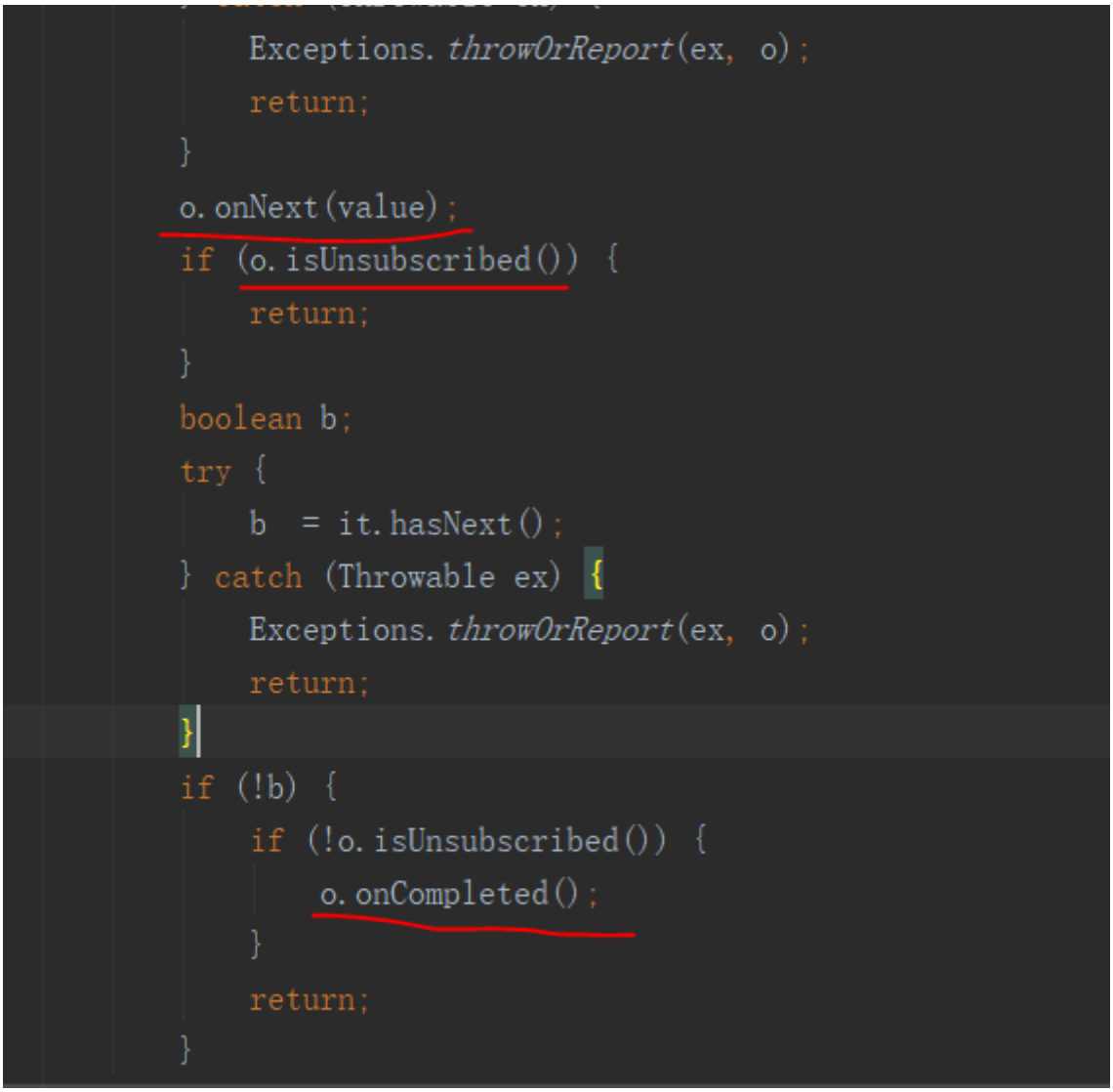
```
static <T> Subscription subscribe(Subscriber<? super T> subscriber, Observable<T> observable) {
    // validate and proceed

    // new Subscriber so onStart it
    subscriber.onStart();

    // The code below is exactly the same an unsafeSubscribe but not used because it would
    // add a significant depth to already huge call stacks.
    try {
        // allow the hook to intercept and/or decorate
        RxJavaHooks.onObservableStart(observable, observable.onSubscribe).call(subscriber);
        return RxJavaHooks.onObservableReturn(subscriber);
    } catch (Throwable e) {
        // special handling for certain Throwable/Error/Exception types
        Exceptions.throwIfFatal(e);
        // in case the subscriber can't listen to exceptions anymore
        if (subscriber.isUnsubscribed()) {
```

OnSubscribe详解OnSubscribeFromIterable



```
eTimerPeriodically.java × OnSubscribeFromIterable.java × OnSubscribeLift.java × OperatorObserveOn.java × Su

        }

        if (!o.isUnsubscribed()) {
            if (!b) {
                o.onCompleted();
            } else {
                o.setProducer(new IterableProducer<T>(o, it));
            }
        }
    }

    static final class IterableProducer<T> extends AtomicLong implements Producer {
        /** */
        private static final long serialVersionUID = -8730475647105475802L;
        private final Subscriber<? super T> o;
        private final Iterator<? extends T> it;

        IterableProducer(Subscriber<? super T> o, Iterator<? extends T> it) {
            this.o = o;
            this.it = it;
        }

        @Override
        public void request(long n) {
            if (get() == Long.MAX_VALUE) {
                // already started with fast-path
                return;
            }
            if (n == Long.MAX_VALUE && compareAndSet( expect: 0, Long.MAX_VALUE)) {
                fastPath();
            } else
```

```
void fastPath() {
    // fast-path without backpressure
    final Subscriber<? super T> o = this.o;
    final Iterator<? extends T> it = this.it;
    for (;;) {

        T value;
        try {
            value = it.next();
        } catch (Throwable ex) {
```



```
            Exceptions.throwOrReport(ex, o);
            return;
        }
        o.onNext(value);
        if (o.isUnsubscribed()) {
            return;
        }
        boolean b;
        try {
            b = it.hasNext();
        } catch (Throwable ex) {
            Exceptions.throwOrReport(ex, o);
            return;
        }
        if (!b) {
            if (!o.isUnsubscribed()) {
                o.onCompleted();
            }
            return;
        }
```

OnSubscribe详解OnSubscribeTimerPeriodically

```
public final class OnSubscribeTimerPeriodically implements OnSubscribe<Long> {
    final long initialDelay;
    final long period;
    final TimeUnit unit;
    final Scheduler scheduler;

    public OnSubscribeTimerPeriodically(long initialDelay, long period, TimeUnit unit, Scheduler scheduler) {
        this.initialDelay = initialDelay;
        this.period = period;
        this.unit = unit;
        this.scheduler = scheduler;
    }

    @Override
    public void call(final Subscriber<? super Long> child) {
        final Worker worker = scheduler.createWorker();
        child.add(worker);
        worker.schedulePeriodically(new Action0() {
            long counter;
            @Override
            public void call() {
                try {
                    child.onNext(counter++);
                } catch (Throwable e) {
                    try {
                        worker.unsubscribe();
                    } finally {
                        Exceptions.throwOrReport(e, child);
                    }
```

future