RxJava的观察者模式

**Github**：<https://github.com/ReactiveX/RxJava>

**任务 ：**RxJava 是 ReactiveX 在 Java 上的开源的实现。Observable（观察者） 和 Subscriber（订阅者）是两个主要的类。在 RxJava 上，一个 Observable 是一个发出数据流或者事件的类，Subscriber 是一个对这些发出的 items （数据流或者事件）进行处理（采取行动）的类。一个 Observable 的标准流发出一个或多个 item，然后成功完成或者出错。一个 Observable 可以有多个 Subscribers，并且通过 Observable 发出的每一个 item，该 item 将会被发送到 Subscriber.onNext() 方法来进行处理。一旦 Observable 不再发出 items，它将会调用 Subscriber.onCompleted() 方法，或如果有一个出错的话 Observable 会调用 Subscriber.onError() 方法。

**开发者：**RxJava是由Netflix开发的响应式扩展(Reactive Extensions)的Java实现

**模块：**

Observable--------------------------生产者ob

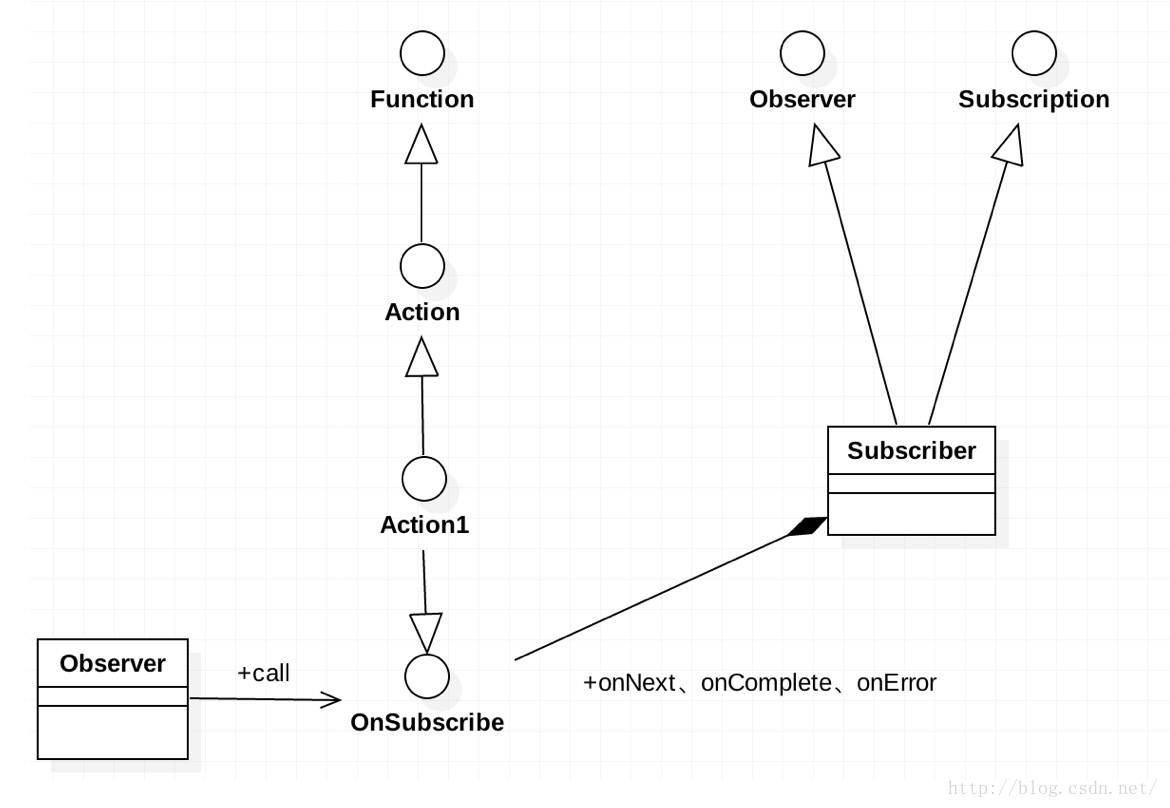
Observable.onSubscribe---------------生产者的任务表onSub

Subscriber--------------------------消费者sub

Observable.subscribe(Subscriber)；------订阅

**模块关系：**Observable和Subject是两个“生产”实体，Observer和Subscriber是两个“消费”实体。Observable 和 Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer。

**代码图**

****

创建Observer：Observer 即观察者，它决定事件触发的时候将有怎样的行为。 RxJava 中的 Observer接口的实现方式：

Observer<String> observer = new Observer<String>() {

@Override

public void onNext(String s) {

Log.d(tag, "Item: " + s);

}

@Override

public void onCompleted() {

Log.d(tag, "Completed!");

}

@Override

public void onError(Throwable e) {

Log.d(tag, "Error!");

}

};

除了 Observer 接口之外，RxJava 还内置了一个实现了 Observer的抽象类：Subscriber。 Subscriber对 Observer接口进行了一些扩展，但他们的基本使用方式是完全一样的：Subscriber<String> subscriber = new Subscriber<String>() {

@Override

public void onNext(String s) {

Log.d(tag, "Item: " + s);

}

@Override

public void onCompleted() {

Log.d(tag, "Completed!");

}

@Override

public void onError(Throwable e) {

Log.d(tag, "Error!");

}

};

　　不仅基本使用方式一样，实质上，在 RxJava 的 subscribe 过程中，Observer也总是会先被转换成一个 Subscriber再使用。所以如果你只想使用基本功能，选择 Observer和 Subscriber是完全一样的。它们的区别对于使用者来说主要有两点：

　　onStart(): 这是 Subscriber增加的方法。它会在 subscribe 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作，例如数据的清零或重置。这是一个可选方法，默认情况下它的实现为空。需要注意的是，如果对准备工作的线程有要求（例如弹出一个显示进度的对话框，这必须在主线程执行）， onStart()就不适用了，因为它总是在 subscribe 所发生的线程被调用，而不能指定线程。要在指定的线程来做准备工作，可以使用 doOnSubscribe()方法，具体可以在后面的文中看到。

unsubscribe(): 这是 Subscriber所实现的另一个接口 Subscription的方法，用于取消订阅。在这个方法被调用后，Subscriber将不再接收事件。一般在这个方法调用前，可以使用 isUnsubscribed()先判断一下状态。 unsubscribe()这个方法很重要，因为在 subscribe()之后， Observable会持有 Subscriber的引用，这个引用如果不能及时被释放，将有内存泄露的风险。所以最好保持一个原则：要在不再使用的时候尽快在合适的地方（例如 onPause()onStop()等方法中）调用unsubscribe()来解除引用关系，以避免内存泄露的发生。

创建 Observable

Observable 即被观察者，它决定什么时候触发事件以及触发怎样的事件。 RxJava 使用 create()方法来创建一个 Observable ，并为它定义事件触发规则：

Observable observable = Observable.create(new Observable.OnSubscribe<String>() {

@Override

public void call(Subscriber<? super String> subscriber) {

subscriber.onNext("Hello");

subscriber.onNext("Hi");

subscriber.onNext("Aloha");

subscriber.onCompleted();

}

});

　　可以看到，这里传入了一个 OnSubscribe对象作为参数。OnSubscribe会被存储在返回的 Observable对象中，它的作用相当于一个计划表，当 Observable被订阅的时候，OnSubscribe的 call()方法会自动被调用，事件序列就会依照设定依次触发（对于上面的代码，就是观察者Subscriber将会被调用三次 onNext()和一次 onCompleted()）。这样，由被观察者调用了观察者的回调方法，就实现了由被观察者向观察者的事件传递，即观察者模式。

　　create()方法是 RxJava 最基本的创造事件序列的方法。基于这个方法， RxJava 还提供了一些方法用来快捷创建事件队列，例如：

　　just(T...): 将传入的参数依次发送出来。

Observable observable = Observable.just("Hello", "Hi", "Aloha");

// 将会依次调用：

// onNext("Hello");

// onNext("Hi");

// onNext("Aloha");

// onCompleted();

　　from(T[])/ from(Iterable<? extends T>): 将传入的数组或 Iterable拆分成具体对象后，依次发送出来。

String[] words = {"Hello", "Hi", "Aloha"};

Observable observable = Observable.from(words);

// 将会依次调用：

// onNext("Hello");

// onNext("Hi");

// onNext("Aloha");

// onCompleted();

上面 just(T...)的例子和 from(T[])的例子，都和之前的 create(OnSubscribe)的例子是等价的。

3、Subscribe (订阅)

创建了 Observable和 Observer之后，再用 subscribe()方法将它们联结起来，整条链子就可以工作了。代码形式很简单：

observable.subscribe(observer);

// 或者：

observable.subscribe(subscriber);

除了 subscribe(Observer)和 subscribe(Subscriber)，subscribe()还支持不完整定义的回调，RxJava 会自动根据定义创建出Subscriber 。形式如下：

Action1<String> onNextAction = new Action1<String>() {

// onNext()

@Override

public void call(String s) {

Log.d(tag, s);

}

};

Action1<Throwable> onErrorAction = new Action1<Throwable>() {

// onError()

@Override

public void call(Throwable throwable) {

// Error handling

}

};

Action0 onCompletedAction = new Action0() {

// onCompleted()

@Override

public void call() {

Log.d(tag, "completed");

}

};

// 自动创建 Subscriber ，并使用 onNextAction 来定义 onNext()

observable.subscribe(onNextAction);

// 自动创建 Subscriber ，并使用 onNextAction 和 onErrorAction 来定义 onNext() 和 onError()

observable.subscribe(onNextAction, onErrorAction);

// 自动创建 Subscriber ，并使用 onNextAction、 onErrorAction 和 onCompletedAction 来定义 onNext()、 onError() 和 onCompleted()

observable.subscribe(onNextAction, onErrorAction, onCompletedAction);

　　简单解释一下这段代码中出现的 Action1和 Action0。 Action0是 RxJava 的一个接口，它只有一个方法 call()，这个方法是无参无返回值的；由于 onCompleted()方法也是无参无返回值的，因此 Action0可以被当成一个包装对象，将 onCompleted()的内容打包起来将自己作为一个参数传入 subscribe()以实现不完整定义的回调。这样其实也可以看做将 onCompleted() 方法作为参数传进了subscribe()，相当于其他某些语言中的『闭包』。

　　Action1也是一个接口，它同样只有一个方法 call(T param)，这个方法也无返回值，但有一个参数；与 Action0同理，由于 onNext(T obj)和 onError(Throwable error)也是单参数无返回值的，因此 Action1可以将 onNext(obj)和 onError(error)打包起来传入 subscribe()以实现不完整定义的回调。事实上，虽然 Action0和 Action1在 API 中使用最广泛，但 RxJava 是提供了多个 ActionX形式的接口 (例如 Action2, Action3) 的，它们可以被用以包装不同的无返回值的方法。