更上一层楼－Android研发工程师高级进阶

asce1885

Published
with GitBook

目錄

什么是函数响应式编程（Java&Android版本）

原文链接：http://www.bignerdranch.com/blog/what-is-functional-reactive-programming/

函数响应式编程（FRP）为解决现代编程问题提供了全新的视角。一旦理解它，可以极大地简化你的项目，特别是处理嵌套回调的异步事件，复杂的列表过滤和变换，或者时间相关问题。

我将尽量跳过对函数响应式编程学院式的解释（网络上已经有很多），并重点从实用的角度帮你理解什么是函数响应式编程，以及工作中怎么应用它。本文将围绕函数响应式编程的一个具体实现RxJava， 它可用于Java和Android。

开始

我们以一个真实的例子来开始讲解函数响应式编程怎么提高我们代码的可读性。我们的任务是通过查询GitHub的API， 首先获取用户列表，然后请求每个用户的详细信息。这个过程包括两个web 服务端点：https://api.github.com/users - 获取用户列表； https://api.github.com/users/{username} －获取特定用户的详细信息，例如https://api.github.com/users/mutexkid。

旧的风格

下面例子你可能已经很熟悉了：它调用web service，使用回调接口将成功的结果传递给下一个web service请求，同时定义另一个成功回调，然后发起下一个web service请求。你可以看到，这会导致两层嵌套的回调：

```
//The "Nested Callbacks" Way
    public void fetchUserDetails() {
        //first, request the users...
        mService.requestUsers(new Callback<GithubUsersResponse>() {
            @Override
            public void success(final GithubUsersResponse githubUsersResponse,
                                final Response response) {
                Timber.i(TAG, "Request Users request completed");
                final synchronized List<GithubUserDetail> githubUserDetails = new ArrayLi
                //next, loop over each item in the response
                for (GithubUserDetail githubUserDetail : githubUsersResponse) {
                    //request a detail object for that user
                    mService.requestUserDetails(githubUserDetail.mLogin,
                                                new Callback<GithubUserDetail>() {
                        @Override
                        public void success(GithubUserDetail githubUserDetail,
                                            Response response) {
                            Log.i("User Detail request completed for user : " + githubUse
                            githubUserDetails.add(githubUserDetail);
                            if (githubUserDetails.size() == githubUsersResponse.mGithubUs
                                //we've downloaded'em all - notify all who are interested
                                mBus.post(new UserDetailsLoadedCompleteEvent(githubUserDe
                            }
                        }

                        @Override
                        public void failure(RetrofitError error) {
```

```
                            Log.e(TAG, "Request User Detail Failed!!!!", error);
                        }
                    });
                }
            }

            @Override
            public void failure(RetrofitError error) {
                Log.e(TAG, "Request User Failed!!!!", error);
            }
        });
    }
```

尽管这不是最差的代码－至少它是异步的，因此在等待每个请求完成的时候不会阻塞－但由于代码复杂（增加更多层次的回调代码复杂度将呈指数级增长）因此远非理想的代码。当我们不可避免要修改代码时（在前面的web service调用中，我们依赖前一次的回调状态，因此它不适用于模块化或者修改要传递给下一个回调的数据）也远非容易的工作。我们亲切的称这种情况为“回调地狱”。

RxJava的方式

下面让我们看看使用RxJava如何实现相同的功能：

```
public void rxFetchUserDetails() {
        //request the users
        mService.rxRequestUsers().concatMap(Observable::from)
        .concatMap((GithubUser githubUser) ->
                        //request the details for each user
                        mService.rxRequestUserDetails(githubUser.mLogin)
        )
        //accumulate them as a list
        .toList()
        //define which threads information will be passed on
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        //post them on an eventbus
        .subscribe(githubUserDetails -> {
            EventBus.getDefault().post(new UserDetailsLoadedCompleteEvent(githubUserDetai
        });
    }
```

如你所见，使用函数响应式编程模型我们完全摆脱了回调，并最终得到了更短小的程序。让我们从函数响应式编程的基本定义开始慢慢解释到底发生了什么，并逐渐理解上面的代码，这些代码托管在GitHub上面。

从根本上讲，函数响应式编程是在观察者模式的基础上，增加对Observables发送的数据流进行操纵和变换的功能。在上面的例子中，Observables是我们的数据流通所在的管道。

回顾一下，观察者模式包含两个角色：一个Observable和一个或者多个Observers。Observable发送事件，而Observer订阅和接收这些事件。在上面的例子中，.subscribe()函数用于给Observable添加一个Observer，并创建一个请求。

构建一个Observable管道

对Observable管道的每个操作都将返回一个新的Observable，这个新的Observable的内容要么和操作前相同，要么就是经过转换的。这种方式使得我们可以对任务进行分解，并把事件流分解成小的操作，接着把这些Observables拼接起来从而完成更复杂的行为或者重用管道中的每个独立的单元。我们对Observable的每个方法调用会被加入到总的管道中以便我们的数据在其中流动。

下面首先让我们从搭建一个Observable开始，来看一个具体的例子：

```
Observable<String> sentenceObservable = Observable.from(“this”, “is”, “a”, “sentence”);
```

这样我们就定义好了管道的第一个部分：Observable。在其中流通的数据是一个字符串序列。首先要认识到的是这是没有实现任何功能的非阻塞代码，仅仅定义了我们想要完成什么事情。Observable只有在我们“订阅”它之后才会开始工作，也就是说给它注册一个Observer之后。

```
Observable.subscribe(new Action1<String>() {
          @Override
          public void call(String s) {
                System.out.println(s);
          }
 });
```

到这一步Observable才会发送由每个独立的Observable的from()函数添加的数据块。管道会持续发送Observables直到所有Observables都被处理完成。

变换数据流

现在我们得到正在发送的字符串流，我们可以按照需要对这些数据流进行变换，并建立更复杂的行为。

```
Observable<String> sentenceObservable = Observable.from(“this”, “is”, “a”, “sentence”);

sentenceObservable.map(new Func1<String, String>() {
            @Override
            public String call(String s) {
                return s.toUpperCase() + " ";
            }
        })
.toList()
.map(new Func1<List<String>, String>() {
            @Override
            public String call(List<String> strings) {
                Collections.reverse(strings);
                return strings.toString();
            }
        })
//subscribe to the stream of Observables
.subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
```

```
                System.out.println(s);
            }
        });
```

一旦Observable被订阅了，我们会得到“SENTENCE A IS THIS”。上面调用的.map函数接受Func1类的对象，该类有两个范型类型参数：一个是输入类型（前一个Observable的内容），另一个是输出类型（在这个例子中，是一个经过大写转换，格式化并用一个新的Observable实例包装的字符串，最终被传递给下一个函数）。如你所见，我们通过可重用的管道组合实现更复杂的功能。

上面的例子中，我们还可以使用Java8的lambda表达式来进一步简化代码：

```
Observable.just("this", "is", "a", "sentence").map(s -> s.toUpperCase() + " ").toList().m
            Collections.reverse(strings);
            return strings.toString();
        });
```

在subscribe函数中，我们传递Action1类对象作为参数，并以String类型作为范型参数。这定义了订阅者的行为，当被观察者发送最后一个事件后，处理后的字符串就被接收到了。这是.subscribe()函数最简单的重载形式（参见https://github.com/ReactiveX/RxJava/wiki/Observable#establishing-subscribers 可以看到更复杂的函数重载签名）。

这个例子展示了变换函数.map()和聚合函数.toList()，在操纵数据流的能力方面这仅仅是冰山一角（所有可用的数据流操作函数可见https://github.com/ReactiveX/RxJava/wiki），但它显示了基本概念：在函数响应式编程中，我们可以通过实现了数据转换或者数据操作功能的管道中独立的单元来转换数据流。根据需要我们可以在其他由Observables组成的管道复用这些独立的单元。通过把这些Observable单元拼接在一起，我们可以组成更复杂的特性，但同时保持它们作为易于理解和可修改的可组合逻辑小单元。

使用Scheduler管理线程

在web service例子中，我们展示了如何使用RxJava发起网络请求。我们谈及转换，聚合和订阅Observable数据流，但我们没有谈及Observable数据流的web请求是怎样实现异步的。

这就属于FRP编程模型如何调用Scheduler的范畴了－该策略定义了Observable流事件在哪个线程中发生，以及订阅者在哪个线程消费Observable的处理结果。在web service例子中，我们希望请求在后台线程中进行，而订阅行为发生在主线程中，因此我们如下定义：

```
.subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        //post them on an eventbus
        .subscribe(githubUserDetails -> {
            EventBus.getDefault().post(new UserDetailsLoadedCompleteEvent(githubUserDetai
        });
```

Observable.subscribeOn(Scheduler scheduler)函数指定Observable的工作需要在指定的Scheduler线程中执行。Observable.observeOn(Scheduler scheduler)指定Observable在哪个Scheduler线程触发订阅者们的

onNext()，onCompleted()，和onError()函数，并调用Observable的observeOn()函数，传递正确的Scheduler给它。

下面是可能会用到Scheduler：

- Schedulers.computation()：用于计算型工作例如事件循环和回调处理，不要在I/O中使用这个函数（应该使用Schedulers.io()函数）；
- Schedulers.from(executor)：使用指定的Executor作为Scheduler；
- Schedulers.immediate()：在当前线程中立即开始执行任务；
- Schedulers.io()：用于I/O密集型工作例如阻塞I/O的异步操作，这个调度器由一个会随需增长的线程池支持；对于一般的计算工作，使用Schedulers.computation()；
- Schedulers.newThread()：为每个工作单元创建一个新的线程；
- Schedulers.test()：用于测试目的，支持单元测试的高级事件；
- Schedulers.trampoline()：在当前线程中的工作放入队列中排队，并依次操作。

通过设置observeOn和subscribeOn调度器，我们定义了网络请求使用哪个线程（Schedulers.newThread()）。

下一步

我们已经在本文中涵盖了很多基础内容，到这里你应该对函数响应式编程如何工作有了很好的认识。请查看并理解本文介绍的工程，它托管在GitHub上面，阅读RxJava文档并检出rxjava-koans工程，以测试驱动的方式掌握函数响应式编程范型。

深入浅出Android Support Annotations

原文链接：http://anupcowkur.com/posts/a-look-at-android-support-annotations/

在Android Support Library19.1版本中，Android工具小组引入了几个很酷的注解类型，供开发者在工程中使用。Support Library自身也使用这些注解，这是一个好兆头。就让我们好好研究下。 通过gradle可以很容易的把这些注解添加到我们的工程中：

```
compile 'com.android.support:support-annotations:20.0.0'
```

有三种类型的注解可供我们使用：

- Nullness注解；
- 资源类型注解；
- IntDef和StringDef注解；

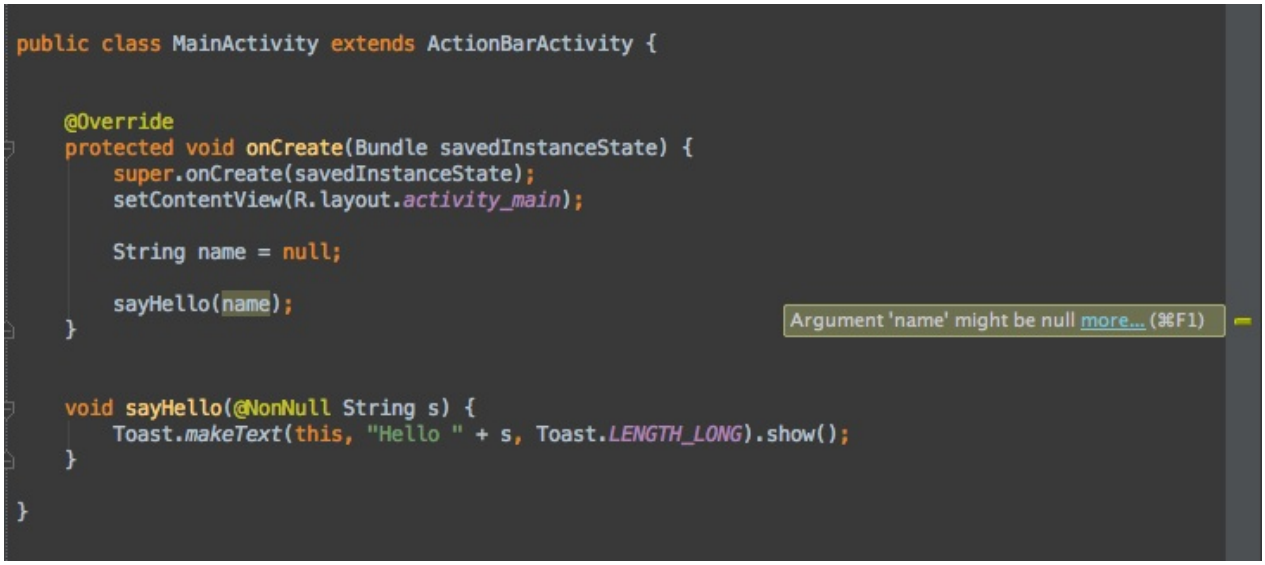
我们将通过代码例子来讲解每一种类型的作用以及在工程中如何使用它们。

Nullness注解

使用@NonNull注解修饰的参数不能为null。在下面的代码例子中，我们有一个取值为null的name变量，它被作为参数传递给sayHello函数，而该函数要求这个参数是非null的String类型：

```
public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        String name = null;

        sayHello(name);
    }


    void sayHello(@NonNull String s) {
        Toast.makeText(this, "Hello " + s, Toast.LENGTH_LONG).show();
    }

}
```

由于代码中参数String s使用@NonNull注解修饰，因此IDE将会以警告的形式提醒我们这个地方有问题：



如果我们给name赋值，例如String name = “Our Lord Duarte”，那么警告将消失。使用@Nullable注解修饰的函数参数或者返回值可以为null。假设User类有一个名为name的变量，使用User.getName()访问，那么我们可以编写如下代码：

```
public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        User user = new User("Our Lord Duarte");

        Toast.makeText(this, "Hello " + getName(user), Toast.LENGTH_LONG).show();
    }

    @Nullable
    String getName(@NonNull User user) {
        return user.getName();
    }

}
```

因为getName函数的返回值使用@Nullable修饰，所以调用：

```
Toast.makeText(this, "Hello " + getName(user), Toast.LENGTH_LONG).show();
```

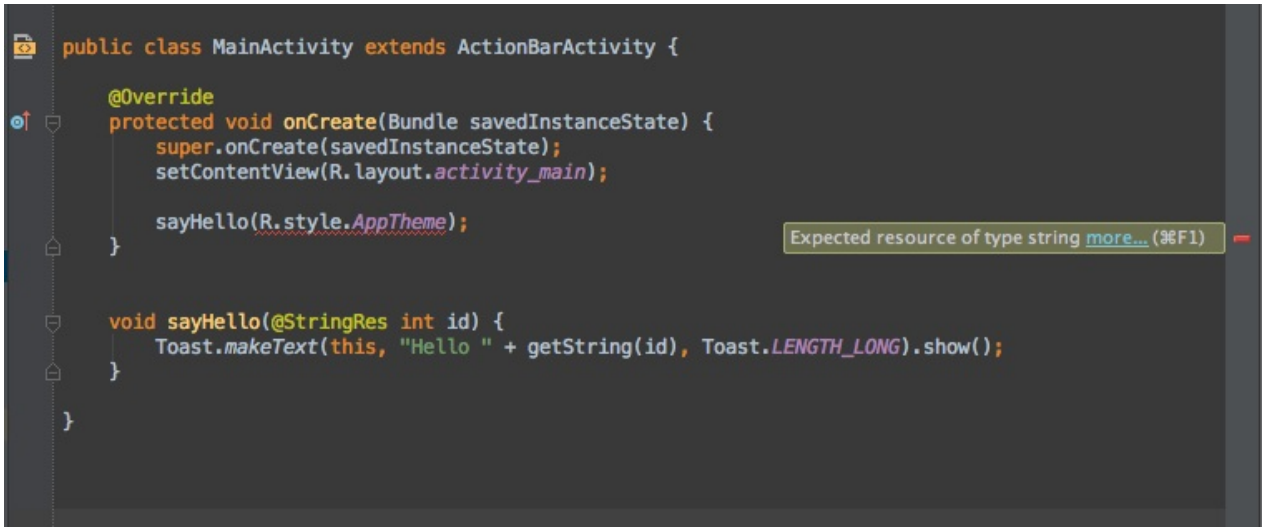
没有检查getName的返回值是否为空，将可能导致crash。

资源类型注解

是否曾经传递了错误的资源整型值给函数，还能够愉快的得到本来想要的整型值吗？资源类型注解可以帮助我们准确实现这一点。在下面的代码中，我们的sayHello函数预期接受一个字符串类型的id，并使用@StringRes注解修饰：

```
public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        sayHello(R.style.AppTheme);
    }


    void sayHello(@StringRes int id) {
        Toast.makeText(this, "Hello " + getString(id), Toast.LENGTH_LONG).show();
    }

}
```

而我们传递了一个样式资源id给它，这时IDE将提示警告如下：



类似的，我们把警告的地方使用一个字符串资源id代替警告就消失了：

```
sayHello(R.string.name);
```

IntDef和StringDef注解

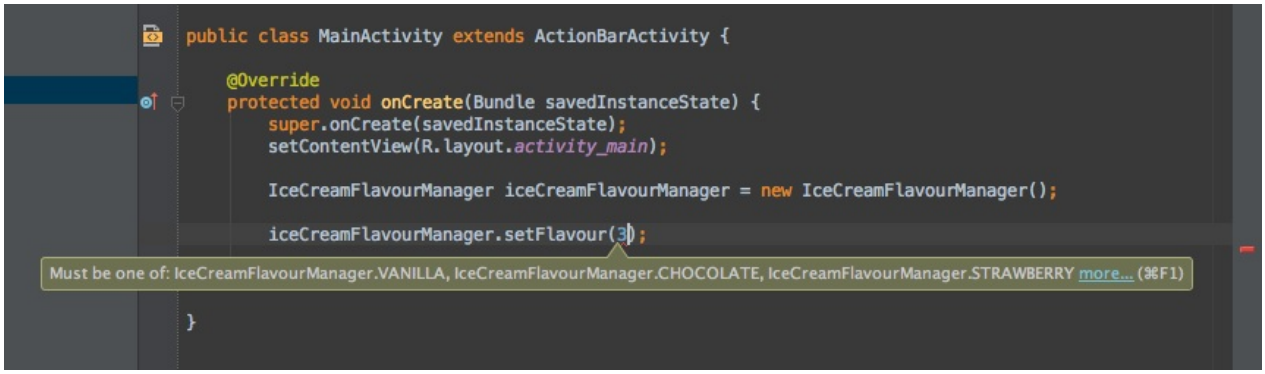
我们要介绍的最后一种类型的注解是基于Intellij的“魔术常量”检查机制（http://blog.jetbrains.com/idea/2012/02/new-magic-constant-inspection/）（我们不需要详细了解这个机制具体是如何实现的，想了解的话可以点击链接）。

很多时候，我们使用整型常量代替枚举类型（性能考虑），例如我们有一个IceCreamFlavourManager类，它具有三种模式的操作：VANILLA，CHOCOLATE和STRAWBERRY。我们可以定义一个名为@Flavour的新注解，并使用@IntDef指定它可以接受的值类型。

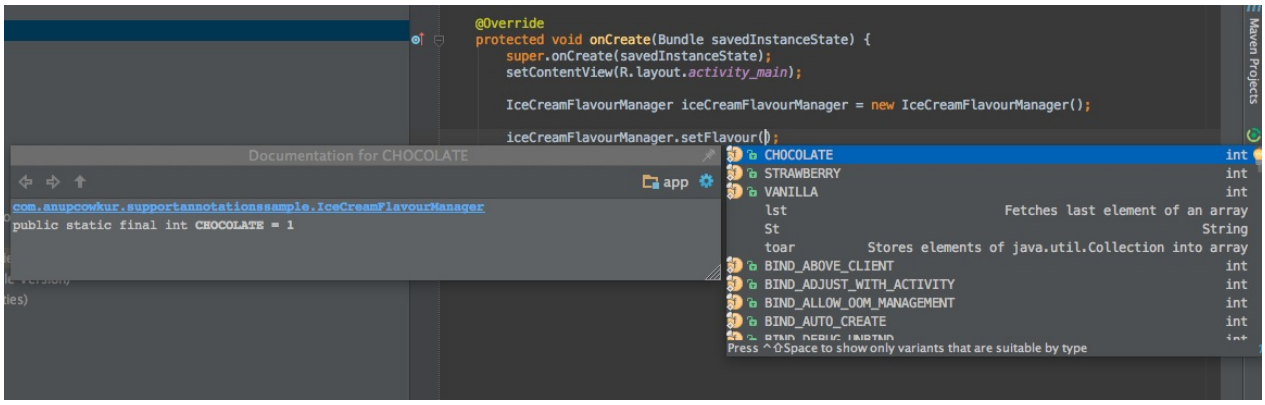
```
public class IceCreamFlavourManager {
```

```
    private int flavour;

    public static final int VANILLA = 0;
    public static final int CHOCOLATE = 1;
    public static final int STRAWBERRY = 2;

    @IntDef({VANILLA, CHOCOLATE, STRAWBERRY})
    public @interface Flavour {
    }

    @Flavour
    public int getFlavour() {
        return flavour;
    }

    public void setFlavour(@Flavour int flavour) {
        this.flavour = flavour;
    }

}
```

这时如果我们使用错误的整型值调用IceCreamFlavourManager.setFlavour时，IDE将报错如下：



IDE甚至会提示我们可以使用的有效的取值：



我们也可以指定整型值作为标志位，也就是说这些整型值可以使用'|'或者'&'进行与或等操作。如果我们把@Flavour定义为如下标志位：

```
@IntDef(flag = true, value = {VANILLA, CHOCOLATE, STRAWBERRY})
    public @interface Flavour {
    }
```

那么可以如下调用：

```
iceCreamFlavourManager.setFlavour(IceCreamFlavourManager.VANILLA & IceCreamFlavourManager
                .CHOCOLATE);
```

@StringDef用法和@IntDef基本差不多，只不过是针对String类型而已。

关于将来计划增加哪些新的注解类型或者这些注解的依赖以及和Intellij自身的注解如何交互等等问题，可以查看网址：http://tools.android.com/tech-docs/support-annotations。

彻底了解RxJava（一）基础知识

原文链接：http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/

RxJava是目前在Android开发者中新兴热门的函数库。唯一的问题是刚开始接触时会感到较难理解。函数响应式编程对于“外面世界”来的开发人员而言是很难理解的，但一旦理解了它，你会感觉真是太棒了。

我将介绍RxJava的一些知识，这系列文章（四部分）的目标是把你领进RxJava的大门。我不会解释所有相关的知识点（我也做不到），我只想引起你对RxJava的兴趣并知道它是如何工作的。

基础知识

响应式代码的基本组成部分是Observables和Subscribers（事实上Observer才是最小的构建块，但实践中使用最多的是Subscriber，因为Subscriber才是和Observables的对应的。）。Observable发送消息，而Subscriber则用于消费消息。

消息的发送是有固定模式的。Observable可以发送任意数量的消息（包括空消息），当消息被成功处理或者出错时，流程结束。Observable会调用它的每个Subscriber的Subscriber.onNext()函数，并最终以Subscriber.onComplete()或者Subscriber.onError()结束。

这看起来像标准的观察者模式， 但不同的一个关键点是：Observables一般只有等到有Subscriber订阅它，才会开始发送消息（术语上讲就是热启动Observable和冷启动Observable。热启动Observable任何时候都会发送消息，即使没有任何观察者监听它。冷启动Observable只有在至少有一个订阅者的时候才会发送消息（我的例子中都是只有一个订阅者）。这个区别对于开始学习RxJava来说并不重要。）。换句话说，如果没有订阅者观察它，那么将不会起什么作用。

Hello, World!

让我们以一个具体例子来实际看看这个框架。首先，我们创建一个基本的Observable：

```
Observable<String> myObservable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> sub) {
            sub.onNext("Hello, world!");
            sub.onCompleted();
        }
    }
);
```

我们的Observable发送“Hello,world!”消息然后完成。现在让我们创建Subscriber来消费这个数据：

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) { System.out.println(s); }

    @Override
```

```
    public void onCompleted() { }

    @Override
    public void onError(Throwable e) { }
};
```

上面代码所做的工作就是打印由Observable发送的字符串。现在我们有了myObservable和mySubscriber，就可以通过subscribe()函数把两者关联起来：

```
myObservable.subscribe(mySubscriber);
// Outputs "Hello, world!"
```

当订阅完成，myObservable将调用subscriber的onNext()和onComplete()函数，最终mySubscriber打印“Hello, world!”然后终止。

更简洁的代码

上面为了打印“Hello, world!”写了大量的样板代码，目的是为了让你详细了解到底发生了什么。RxJava提供了很多快捷方式来使编码更简单。

首先让我们简化Observable，RxJava为常见任务提供了很多内建的Observable创建函数。在以下这个例子中，Observable.just()发送一个消息然后完成，功能类似上面的代码（严格来说，Observable.just()函数跟我们原来的代码并不完全一致，但在本系列第三部分之前我不会说明原因）：

```
Observable<String> myObservable =
    Observable.just("Hello, world!");
```

接下来，让我们处理Subscriber不必要的样板代码。如果我们不关心onCompleted()或者onError()的话，那么可以使用一个更简单的类来定义onNext()期间要完成什么功能：

```
Action1<String> onNextAction = new Action1<String>() {
    @Override
    public void call(String s) {
        System.out.println(s);
    }
};
```

Actions可以定义Subscriber的每一个部分，Observable.subscribe()函数能够处理一个，两个或者三个Action参数，分别表示onNext()，onError()和onComplete()函数。上面的Subscriber现在如下所示：

```
myObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);
```

然而，现在我们不需要onError()和onComplete()函数，因此只需要第一个参数：

```
myObservable.subscribe(onNextAction);
// Outputs "Hello, world!"
```

现在，让我们把上面的函数调用链接起来从而去掉临时变量：

```
Observable.just("Hello, world!")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
              System.out.println(s);
        }
    });
```

最后，我们使用Java 8的lambdas表达式来去掉丑陋的Action1代码：

```
Observable.just("Hello, world!")
    .subscribe(s -> System.out.println(s));
```

如果你在Android平台上（因此不能使用Java 8），那么我严重推荐使用retrolambda，它将极大的减少代码的冗余度。

变换

让我们来点刺激的。假如我想把我的签名拼接到“Hello, world!的输出中。一个可行的办法是改变Observable：

```
Observable.just("Hello, world! -Dan")
    .subscribe(s -> System.out.println(s));
```

当你有权限控制Observable时这么做是可行的，但不能保证每次都可以这样，如果你使用的是别人的函数库呢？另外一个可能的问题是：如果项目中在多个地方使用Observable，但只在某个地方需要增加签名，这时怎么办？

我们尝试修改Subscriber如何呢？

```
Observable.just("Hello, world!")
    .subscribe(s -> System.out.println(s + " -Dan"));
```

这个解决方案也不尽如人意，不过原因不同于上面：我想Subscribers尽可能的轻量级，因为我可能要在主线程中运行它。从更概念化的层面上讲，Subscribers是用于被动响应的，而不是主动发送消息使其他对象发生变化。 如果可以通过某些中间步骤来对上面的“Hello, world!”进行转换，那岂不是很酷？

Operators简介

接下来我们将介绍如何解决消息转换的难题：使用Operators。Operators在消息发送者Observable和消息消费者Subscriber之间起到操纵消息的作用。RxJava拥有大量的opetators，但刚开始最好还是从一小部分开始熟悉。 这种情况下，map() operator可以被用于将已被发送的消息转换成另外一种形式：

```
Observable.just("Hello, world!")
    .map(new Func1<String, String>() {
        @Override
        public String call(String s) {
            return s + " -Dan";
        }
    })
    .subscribe(s -> System.out.println(s));
```

同样的，我们可以使用Java 8的lambdas表达式来简化代码：

```
Observable.just("Hello, world!")
    .map(s -> s + " -Dan")
    .subscribe(s -> System.out.println(s));
```

很酷吧？我们的map() operator本质上是一个用于转换消息对象的Observable。我们可以级联调用任意多个的map()函数，一层一层地将初始消息转换成Subscriber需要的数据形式。

map()更多的解释

map()函数有趣的一点是：它不需要发送和原始的Observable一样的数据类型。假如我的Subscriber不想直接输出原始的字符串，而是想输出原始字符串的hash值：

```
Observable.just("Hello, world!")
    .map(new Func1<String, Integer>() {
        @Override
        public Integer call(String s) {
            return s.hashCode();
        }
    })
    .subscribe(i -> System.out.println(Integer.toString(i)));
```

有趣的是，我们的原始输入是字符串，但Subscriber最终收到的是Integer类型。同样的，我们可以使用lambdas简化代码如下：

```
Observable.just("Hello, world!")
    .map(s -> s.hashCode())
    .subscribe(i -> System.out.println(Integer.toString(i)));
```

正如我前面说过的，我希望Subscriber做尽量少的工作，我们可以把hash值转换成字符串的操作移动到一个新的map()函数中：

```
Observable.just("Hello, world!")
    .map(s -> s.hashCode())
    .map(i -> Integer.toString(i))
    .subscribe(s -> System.out.println(s));
```

你不想瞧瞧这个吗？我们的Observable和Subscriber变回了原来的代码。我们只是在中间添加了一些变换的步骤，我甚至可以把我的签名转换添加回去：

```
Observable.just("Hello, world!")
    .map(s -> s + " -Dan")
    .map(s -> s.hashCode())
    .map(i -> Integer.toString(i))
    .subscribe(s -> System.out.println(s));
```

那又怎样？

到这里你可能会想“为了得到简单的代码有很多聪明的花招”。确实，这是一个简单的例子，但有两个概念你需要理解：

关键概念＃**1**：**Observable**和**Subscriber**能完成任何事情。

放飞你的想象，任何事情都是可能的。 你的Observable可以是一个数据库查询，Subscriber获得查询结果然后将其显示在屏幕上。你的Observable可以是屏幕上的一个点击，Subscriber响应该事件。你的Observable可以从网络上读取一个字节流，Subscriber将其写入本地磁盘中。 这是一个可以处理任何事情的通用框架。

关键概念＃**2**：**Observable**和**Subscriber**与它们之间的一系列转换步骤是相互独立的。

我们可以在消息发送者Observable和消息消费者Subscriber之间加入任意多个想要的map()函数。这个系统是高度可组合的：它很容易对数据进行操纵。只要operators符合输入输出的数据类型，那么我可以得到一个无穷尽的调用链（好吧，并不是无穷尽的，因为总会到达物理机器的极限的，但你知道我想表达的意思）。

结合两个关键概念，你可以看到一个有极大潜能的系统。然而到这里我们只介绍了一个operator：map()，这严重地限制了我们的可能性。在本系列的第二部分，我们将详细介绍RxJava中大量可用的operators。

Android事件驱动编程（一）

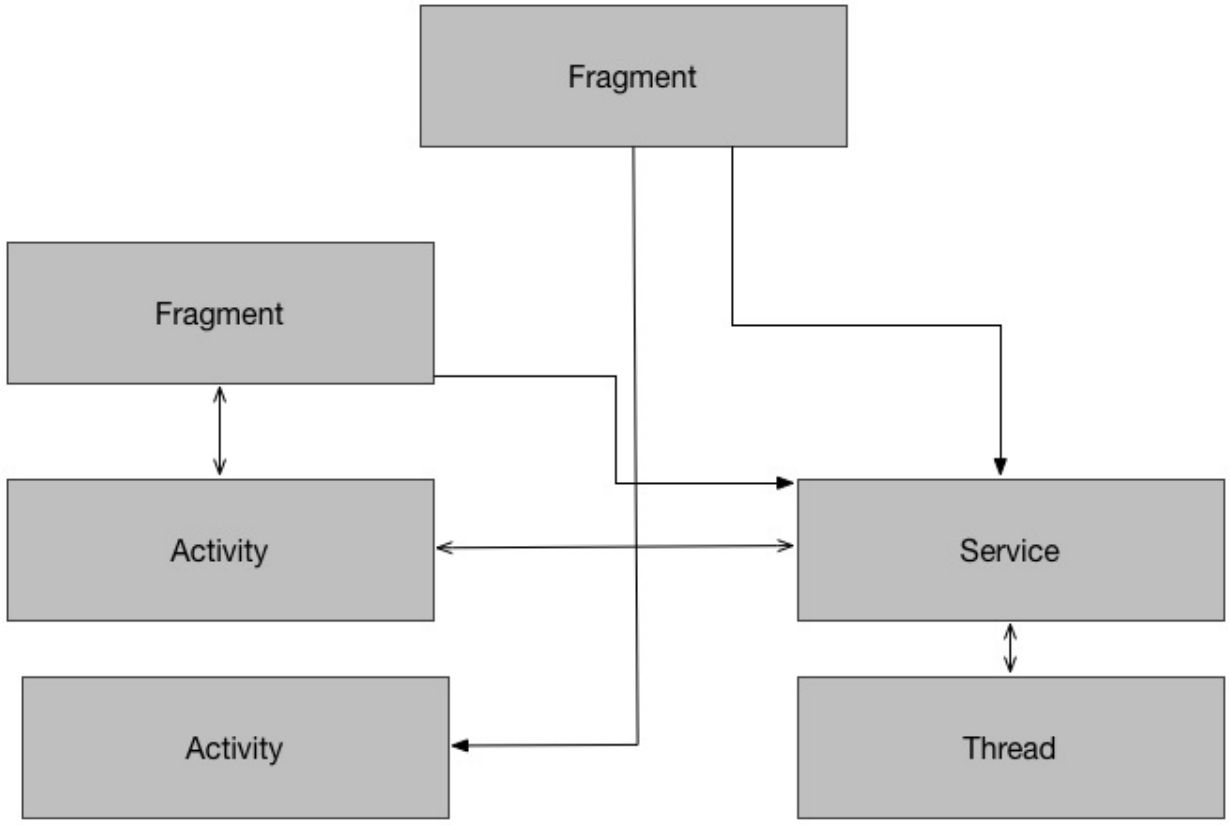
原文链接：https://medium.com/google-developer-experts/event-driven-programming-for-android-part-i-f5ea4a3c4eab

虽然在Android开发具有某些事件驱动的特性，但它还远不是纯粹的事件驱动架构。这算是好事还是坏事呢？正如在软件开发领域中任何事情一样，想回答它并不容易：这取决于具体情况。

首先我们来给事件驱动编程下一个定义。事件驱动编程是一种编程范式，程序的执行流程是由动作（actions，例如用户交互，其他线程发送的消息等等）触发的事件（event）决定的。在这个意义上，Android是部分事件驱动：我们都知道的onClick监听器或者Activity的生命周期，都是应用中由动作触发的事件。我为什么说它不是纯粹的事件驱动系统呢？默认情况下，每个事件被绑定在特定的controller上面，因此很难在其他地方使用该事件（例如onClick事件是定义为View服务的，因此它的使用范围很有限）。

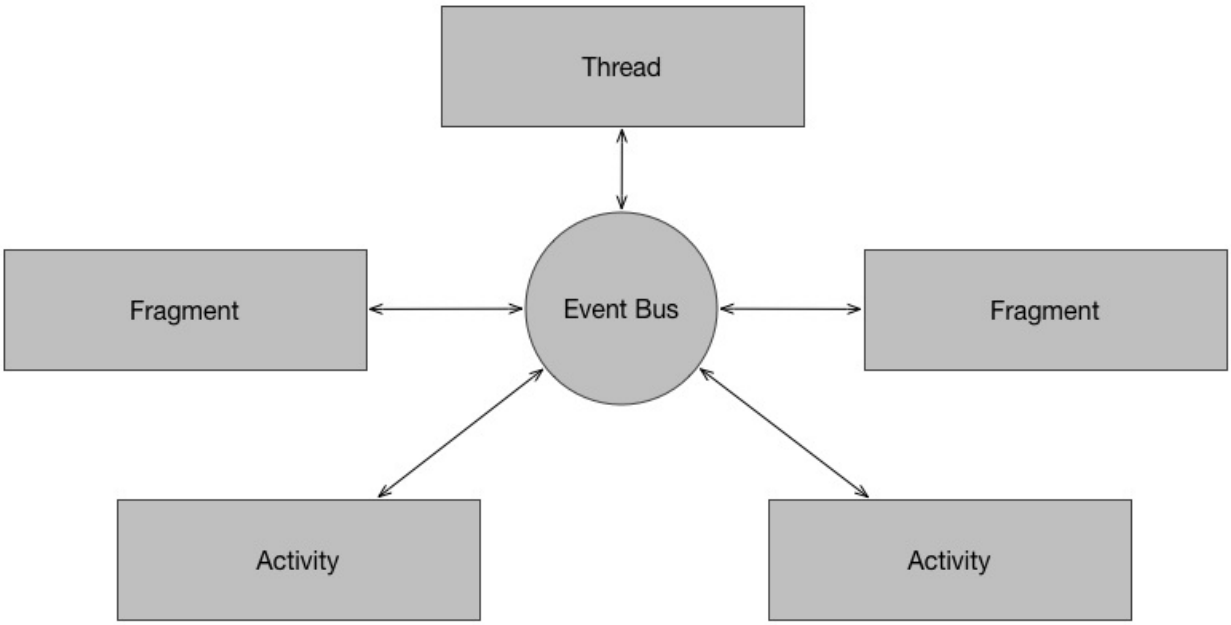
等一下，你在讨论的是一个全新的编程范式，使用这个新框架或者方法总会带来成本的，那么它能带来什么好处呢？答案是肯定的，为了说明它，我将首先说一说传统Android开发存在的一些限制。

很多情况下我们工程代码会很容易演变成下图所示的结构：



Activities可以和Fragments通信，Fragments可以给其他的Fragments和Services发送消息。各个组件之间耦合严重，修改代码的时间代价昂贵。这经常导致产生样板代码，例如需要在不同层之间传播实现了回调函数的接口，你应该知道我想表达的意思。由于代码量的增加，可维护性和良好的软件工程实践随之降低。

那么事件驱动编程如何使用呢？下面让我们来看看另一种系统架构设计：



概念上讲，上面的系统具有一个Event Bus，不同的实体订阅这个Event Bus，要么发送事件，要么监听事件－分别作为生产者或者消费者。任何订阅者可以在不知道业务逻辑的情况下执行动作。想象一下，想象某种具体的可能性：一个Fragment可以在不知道任何操作背后业务逻辑的情况下，只需要收到一个事件，就可以重新执行渲染并更新屏幕显示。想象一下代码解耦并得到一个简洁的，划分良好的架构的可能性。

Android支持这种编程范式吗？好吧，部分支持。就如上面提到的，Android SDK本身提供了一个简化的事件处理技术，但我们想要更多的支持。在这里有一些名字我想提一下：

- EventBus，出自greenrobot。这个函数库专为Android而优化过，并具有某些高级特性例如线程传递和订阅者优先级。
- Otto，出自Square。起初是从Guava fork过来的，后面经过发展，已经在Android平台上进行了完善。

两个函数库都试用过，我更喜欢EventBus。Greenrobot声称EventBus在性能上比Otto更好，并提供了很多额外的特性。

	EventBus	Otto
Posting 1000 events, Android 2.3 emulator	~70% faster	
Posting 1000 events, S3 Android 4.0	~110% faster	
Register 1000 subscribers, Android 2.3 emulator	~10% faster	
Register 1000 subscribers, S3 Android 4.0	~70% faster	
Register subscribers cold start, Android 2.3 emulator	~350% faster	
Register subscribers cold start, S3 Android 4.0	About the same	

	EventBus	Otto
Declare event handling methods	Name conventions	Annotations
Event inheritance	Yes	Yes
Subscriber inheritance	Yes	No
Cache most recent events	Yes, sticky events	No
Event producers (e.g. for coding cached events)	No	Yes
Event delivery in posting thread	Yes (Default)	Yes
Event delivery in main thread	Yes	No
Event delivery in background thread	Yes	No
Aynchronous event delivery	Yes	No

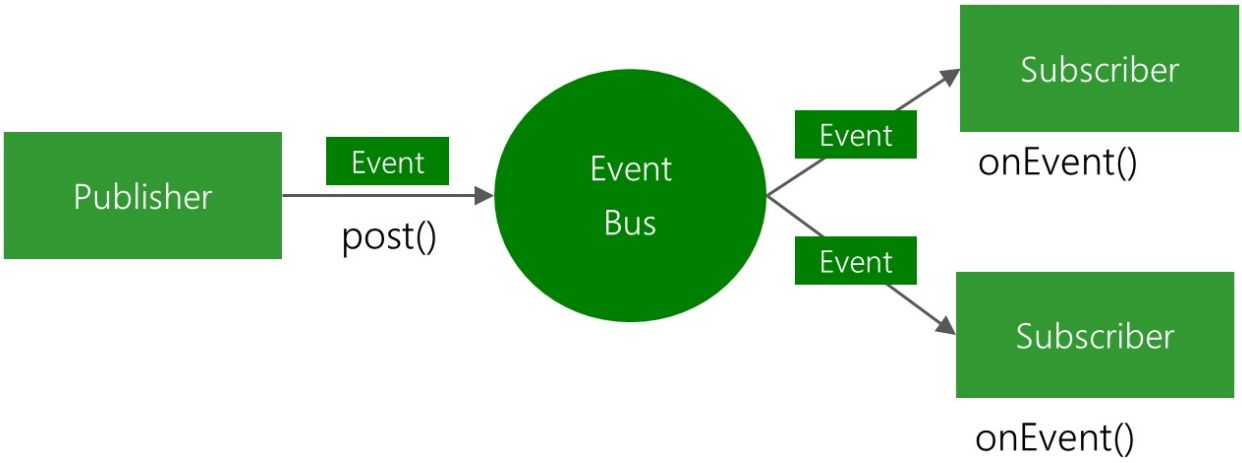
下一篇文章将讲解在EventBus中如何实现基本的功能。

Android事件驱动编程（二）

原文链接：https://medium.com/google-developer-experts/event-driven-programming-for-android-part-ii-b1e05698e440

在前面的文章中我们简单介绍了事件驱动编程，现在让我们看看真实的代码并介绍EventBus的基础用法。

首先我会参考下图（从EventBus仓库截取的），介绍在事件驱动编程中扮演中心角色的实体们。



- 事件总线EventBus：连接所有其他实体的中央通信通道；
- 事件Event：发生的动作，几乎可以是任何事情（应用启动，收到某些数据，用户交互等等）；
- 订阅者Subscriber：订阅者监听事件总线，当总线中有事件在流通时，订阅者将被触发；
- 发布者Publisher：往事件总线发送事件；

唯有亲身实践才能有清晰的认知，下面就让我们来看一个简单的例子：

- 一个加载了两个fragments的应用；
- 第二个fragment包含一个TextView，当单击按钮时，TextView将被刷新；
- 当新的fragment显示在当前屏幕上时，ActionBar标题栏将随之改变；

宿主Activity

宿主Activity需要在它的onCreate函数中注册EventBus：

```
EventBus.getDefault().register(this);
```

注册之后，宿主Activity就可以从总线上读取数据了，我们同时需要在Activity的onDestroy函数中反注册EventBus：

```
EventBus.getDefault().unregister(this);
```

Activity会捕获到两个不同的事件：一个用于更新ActionBar，一个用于加载第一个fragment。我们会编写两个onEvent函数来处理这两个事件：

```
public void onEvent(ShowFragmentEvent event) {
    getFragmentManager().beginTransaction().replace(R.id.container, event.getFragment()).
}
public void onEvent(UpdateActionBarTitleEvent e) {
    getActionBar().setTitle(e.getTitle());
}
```

事件

每个事件需要在类中声明，事件中可以包含变量：

```
public final class ShowFragmentEvent {
 private Fragment fragment;
  public ShowFragmentEvent(Fragment fragment) {
    this.fragment = fragment;
  }
  public Fragment getFragment() {
    return fragment;
  }
}
```

The Fragments

接下来我们要来创建fragments。第一个fragment包含一个用来打开第二个fragment的按钮，第二个fragment包含一个按钮，当点击按钮时，将刷新TextView。Fragments也需要注册和反注册EventBus，为了得到一个简洁的结构，我们将定义一个BaseFragment来封装这些公共的操作。

现在让我们创建更多一些动作，第一个fragment将通过下面的函数来打开第二个fragment：

```
@OnClick(R.id.first_button)
 public void firstButtonClick() {
    EventBus.getDefault().post(new ShowFragmentEvent(new SecondFragment()));
 }
```

需要注意的是这里我使用了ButterKnife中定义的注解，它可以产生更简单和整洁的代码。如果你还没有使用过它，那么现在应该开始使用了。

第二个fragment的按钮会向事件总线发送一个事件，用来更新TextView：

```
EventBus.getDefault().post(new UpdateTextEvent(getString(R.string.text_updated)));
```

第二个fragment同时需要监听这个事件，这样当它接收到这个事件时，可以相应的改变文字显示：

```
public void onEvent(UpdateTextEvent event) {
    textView.setText(event.getTitle());
}
```

我们的简单应用具有两个fragments，通过事件实现两个fragments之间的通信，一个fragment通过事件获得更新。我已经把代码上传到GitHub上面，你可以检出并看一下。

一个关键的问题是如何逐步升级一个事件驱动的架构。在下一篇文章中我将介绍一个简洁的，可理解的架构，来支持Android中的事件驱动编程。

我为什么主张反对使用Android Fragment

原文链接：https://corner.squareup.com/2014/10/advocating-against-android-fragments.html

最近我在Droidcon Paris举办了一场技术讲座，我讲述了Square公司在使用Android fragments时遇到的问题，以及其他人如何避免使用fragments。

在2011年，基于以下原因我们决定在项目中使用fragments：

- 在那个时候，我们还没有支持平板设备－但是我们知道最终将会支持的，Fragments有助于构建响应式UI；
- Fragments是view controllers，它们包含可测试的，解耦的业务逻辑块；
- Fragments API提供了返回堆栈管理功能（即把activity堆栈的行为映射到单独一个activity中）；
- 由于fragments是构建在views之上的，而views很容易实现动画效果，因此fragments在屏幕切换时具有更好的控制；
- Google推荐使用fragments，而我们想要我们的代码标准化；

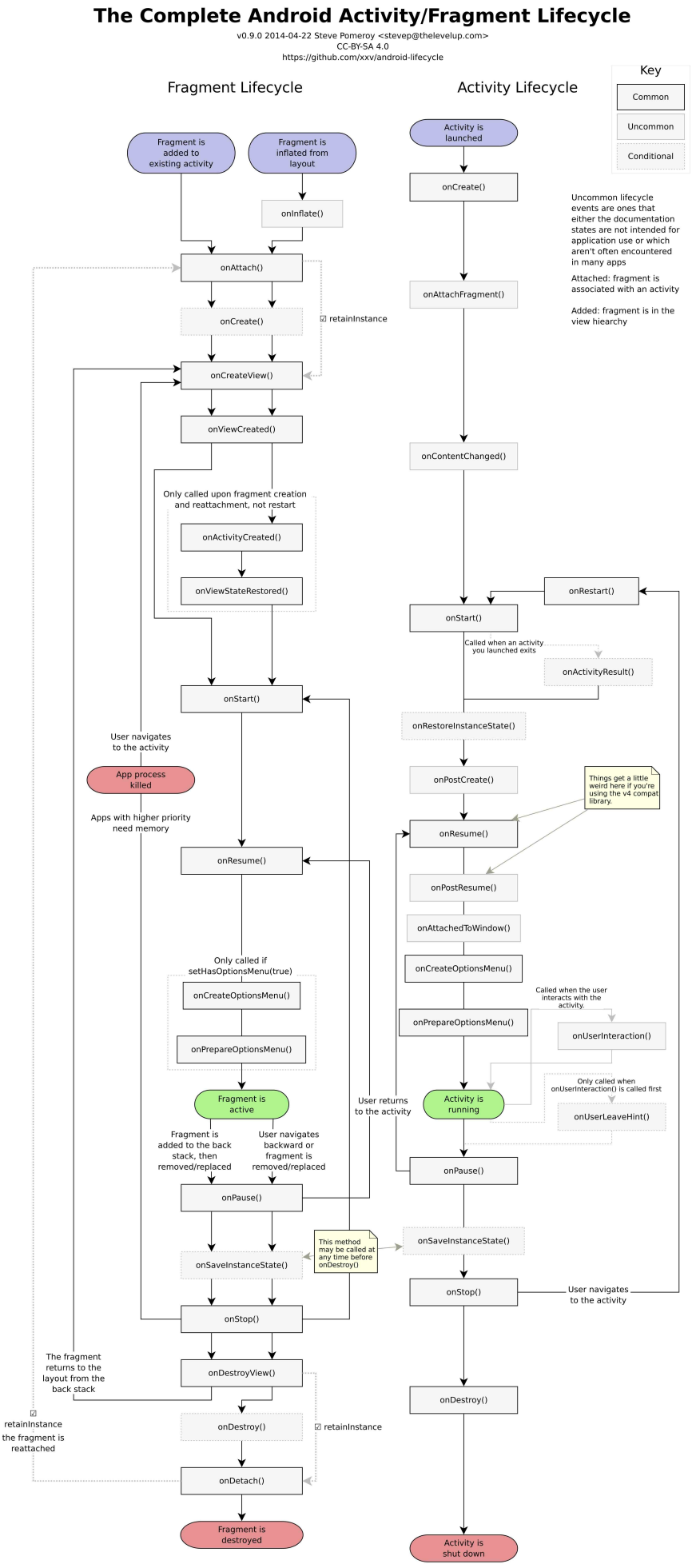
自从2011年以来，我们为Square找到了更好的选择。

关于fragments你所不知道的

复杂的生命周期

Android中，Context是一个上帝对象（god object），而Activity是具有附加生命周期的context。具有生命周期的上帝对象？有点讽刺的意味。Fragments不是上帝对象，但它们为了弥补这一点，实现了及其复杂的生命周期。

Steve Pomeroy为Fragments复杂的生命周期制作了一张图表看起来并不可爱：


The Complete Android Activity/Fragment Lifecycle
v0.9.0 2014-04-22 Steve Pomeroy <stevep@thelevelup.com>
CC-BY-SA 4.0
https://github.com/xxv/android-lifecycle
Fragment Lifecycle
Activity Lifecycle
Key
Common
Uncommon
Conditional
Fragment is added to existing activity
Fragment is inflated from layout
onInflate()
onAttach()
onCreate()
retainInstance
onCreateView()
onViewCreated()
Only called upon fragment creation and reattachment, not restart
onActivityCreated()
onViewStateRestored()
onStart()
User navigates to the activity
App process killed
Apps with higher priority need memory
onResume()
Only called if setHasOptionsMenu(true)
onCreateOptionsMenu()
onPrepareOptionsMenu()
Fragment is active
Fragment is added to the back stack, then removed/replaced
User navigates backward or fragment is removed/replaced
onPause()
onSaveInstanceState()
onStop()
The fragment returns to the layout from the back stack
onDestroyView()
onDestroy()
retainInstance
the fragment is reattached
retainInstance
onDetach()
Fragment is destroyed
Activity is launched
onCreate()
onAttachFragment()
onContentChanged()
onRestart()
onStart()
Called when an activity you launched exits
onActivityResult()
onRestoreInstanceState()
onPostCreate()
Things get a little weird here if you're using the v4 compat library.
onResume()
onPostResume()
onAttachedToWindow()
onCreateOptionsMenu()
Called when the user interacts with the activity.
onPrepareOptionsMenu()
onUserInteraction()
Only called when onUserInteraction() is called first
onUserLeaveHint()
User returns to the activity
Activity is running
onPause()
This method may be called at any time before onDestroy()
onSaveInstanceState()
onStop()
User navigates to the activity
onDestroy()
Activity is shut down
Uncommon lifecycle events are ones that either the documentation states are not intended for application use or which aren't often encountered in many apps
Attached: fragment is associated with an activity
Added: fragment is in the view hierarchy

上面Fragments的生命周期使得开发者很难弄清楚在每个回调处要做什么，这些回调是同步的还是异步的？顺序如何？

难以调试

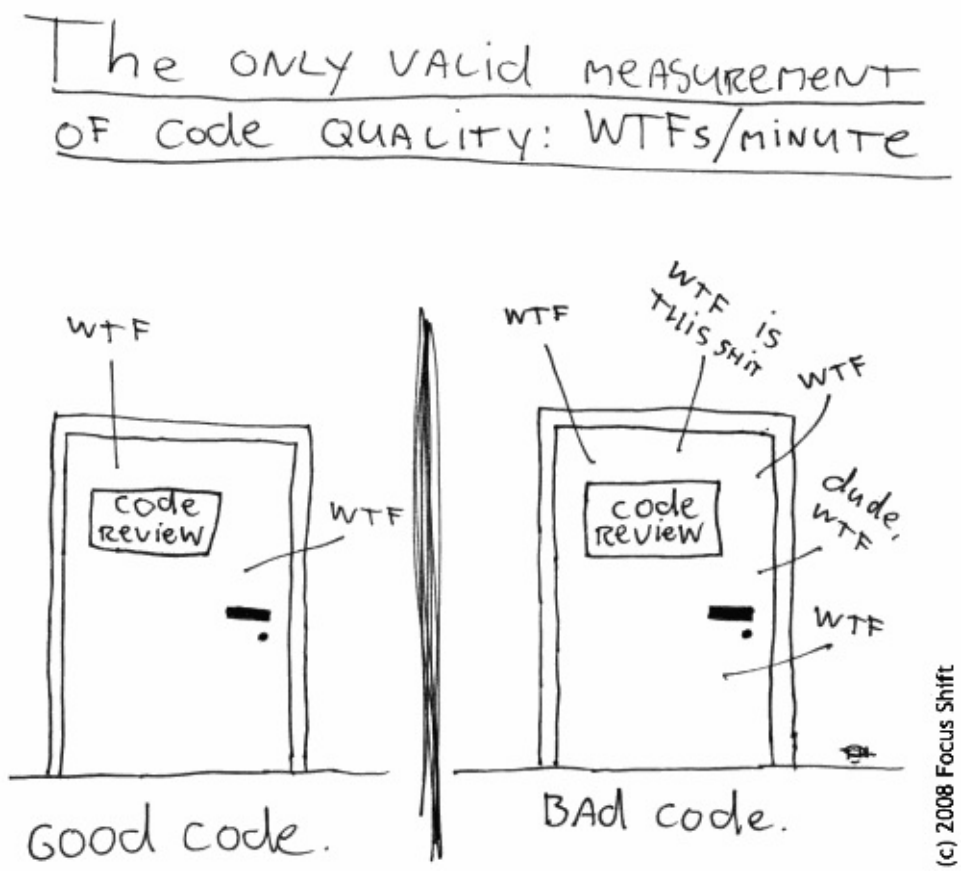
当你的app出现bug，你使用调试器并一步一步执行代码以便了解到底发生了什么，这通常能很好地工作，直到你遇到了FragmentManagerImpl：它是地雷。

下面这段代码很难跟踪和调试，这使得很难正确的修复app中的bug：

```
switch (f.mState) {
    case Fragment.INITIALIZING:
        if (f.mSavedFragmentState != null) {
            f.mSavedViewState = f.mSavedFragmentState.getSparseParcelableArray(
                    FragmentManagerImpl.VIEW_STATE_TAG);
            f.mTarget = getFragment(f.mSavedFragmentState,
                    FragmentManagerImpl.TARGET_STATE_TAG);
            if (f.mTarget != null) {
                f.mTargetRequestCode = f.mSavedFragmentState.getInt(
                        FragmentManagerImpl.TARGET_REQUEST_CODE_STATE_TAG, 0);
            }
            f.mUserVisibleHint = f.mSavedFragmentState.getBoolean(
                    FragmentManagerImpl.USER_VISIBLE_HINT_TAG, true);
            if (!f.mUserVisibleHint) {
                f.mDeferStart = true;
                if (newState > Fragment.STOPPED) {
                    newState = Fragment.STOPPED;
                }
            }
        }
// ...
}
```

如果你曾经遇到屏幕旋转时旧的unattached的fragment重新创建，那么你应该知道我在谈论什么（不要让我从嵌套fragments讲起）。

正如Coding Horror所说，根据法律要求我需要附上这个动画的链接。



经过多年深入的分析，我得到的结论是WTFs/min = 2^fragment的个数。

View controllers？没这么快

由于fragments创建，绑定和配置views，它们包含了大量的视图相关的代码。这实际上意味着业务逻辑没有和视图代码解耦－这使得很难针对fragments编写单元测试。

Fragment事务

Fragment事务使得你可以执行一系列fragment操作，不幸的是，提交事务是异步的，而且是附加在主线程handler队列尾部的。当你的app接收到多个点击事件或者配置发生变化时，将处于不可知的状态。

```
class BackStackRecord extends FragmentTransaction {
    int commitInternal(boolean allowStateLoss) {
        if (mCommitted)
            throw new IllegalStateException("commit already called");
        mCommitted = true;
        if (mAddToBackStack) {
            mIndex = mManager.allocBackStackIndex(this);
        } else {
            mIndex = -1;
        }
        mManager.enqueueAction(this, allowStateLoss);
        return mIndex;
    }
}
```

Fragment创建魔法

Fragment实例可以由你或者fragment manager创建。下面代码似乎很合理：

```
DialogFragment dialogFragment = new DialogFragment() {
  @Override public Dialog onCreateDialog(Bundle savedInstanceState) { ... }
};
dialogFragment.show(fragmentManager, tag);
```

然而，当恢复activity实例的状态时，fragment manager可能会尝试通过反射机制重新创建这个fragment类的实例。由于这是一个匿名内部类，它的构造函数有一个隐藏的参数，持有外部类的引用。

```
android.support.v4.app.Fragment$InstantiationException:
    Unable to instantiate fragment com.squareup.MyActivity$1:
    make sure class name exists, is public, and has an empty
    constructor that is public
```

Fragments的经验教训

尽管存在缺点，fragments教给我们宝贵的教训，让我们在编写app的时候可以重用：

- 单Activity界面：没有必要为每个界面使用一个activity。我们可以分割我们的app为解耦的组件然后根据需要进行组合。这使得动画和生命周期变得简单。我们可以把组件代码分割成视图代码和控制器代码。
- 返回栈不是activity特性的概念；我们可以在一个activity中实现返回栈。
- 没有必要使用新的API；我们所需要的一切都是早就存在的：activities，views和layout inflaters。

响应式UI：fragments vs 自定义views

Fragments

让我们看一个fragment的简单例子，一个列表和详情UI。

HeadlinesFragment是一个简单的列表：

```
public class HeadlinesFragment extends ListFragment {
  OnHeadlineSelectedListener mCallback;

  public interface OnHeadlineSelectedListener {
    void onArticleSelected(int position);
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setListAdapter(
        new ArrayAdapter<String>(getActivity(),
            R.layout.fragment_list,
```

```
            Ipsum.Headlines));
  }

  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);
    mCallback = (OnHeadlineSelectedListener) activity;
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    mCallback.onArticleSelected(position);
    getListView().setItemChecked(position, true);
  }
}
```

接下来比较有趣：ListFragmentActivity到底需要处理相同界面上的细节还是不需要呢？

```
public class ListFragmentActivity extends Activity
    implements HeadlinesFragment.OnHeadlineSelectedListener {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.news_articles);
    if (findViewById(R.id.fragment_container) != null) {
      if (savedInstanceState != null) {
        return;
      }
      HeadlinesFragment firstFragment = new HeadlinesFragment();
      firstFragment.setArguments(getIntent().getExtras());
      getFragmentManager()
          .beginTransaction()
          .add(R.id.fragment_container, firstFragment)
          .commit();
    }
  }
  public void onArticleSelected(int position) {
    ArticleFragment articleFrag =
        (ArticleFragment) getFragmentManager()
            .findFragmentById(R.id.article_fragment);
    if (articleFrag != null) {
      articleFrag.updateArticleView(position);
    } else {
      ArticleFragment newFragment = new ArticleFragment();
      Bundle args = new Bundle();
      args.putInt(ArticleFragment.ARG_POSITION, position);
      newFragment.setArguments(args);
      getFragmentManager()
          .beginTransaction()
          .replace(R.id.fragment_container, newFragment)
          .addToBackStack(null)
          .commit();
    }
  }
}
```

自定义views

让我们只使用views来重新实现上面代码的相似版本。

首先，我们定义Container的概念，它可以显示一个item，也可以处理返回键。

```
public interface Container {
  void showItem(String item);

  boolean onBackPressed();
}
```

Activity假设总会存在一个container，并把工作委托给它。

```
public class MainActivity extends Activity {
  private Container container;

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
    container = (Container) findViewById(R.id.container);
  }

  public Container getContainer() {
    return container;
  }

  @Override public void onBackPressed() {
    boolean handled = container.onBackPressed();
    if (!handled) {
      finish();
    }
  }
}
```

列表的代码也类似如下：

```
public class ItemListView extends ListView {
  public ItemListView(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override protected void onFinishInflate() {
    super.onFinishInflate();
    final MyListAdapter adapter = new MyListAdapter();
    setAdapter(adapter);
    setOnItemClickListener(new OnItemClickListener() {
      @Override public void onItemClick(AdapterView<?> parent, View view,
            int position, long id) {
        String item = adapter.getItem(position);
        MainActivity activity = (MainActivity) getContext();
        Container container = activity.getContainer();
        container.showItem(item);
```

```
      }
    });
  }
}
```

接着任务是：基于资源限定符加载不同的XML布局文件。

res/layout/main_activity.xml：

```
<com.squareup.view.SinglePaneContainer
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/container"
    >
  <com.squareup.view.ItemListView
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      />
</com.squareup.view.SinglePaneContainer>
```

res/layout-land/main_activity.xml：

```
<com.squareup.view.DualPaneContainer
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:id="@+id/container"
    >
  <com.squareup.view.ItemListView
      android:layout_width="0dp"
      android:layout_height="match_parent"
      android:layout_weight="0.2"
      />
  <include layout="@layout/detail"
      android:layout_width="0dp"
      android:layout_height="match_parent"
      android:layout_weight="0.8"
      />
</com.squareup.view.DualPaneContainer>
```

下面是这些containers的简单实现：

```
public class DualPaneContainer extends LinearLayout implements Container {
  private MyDetailView detailView;

  public DualPaneContainer(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override protected void onFinishInflate() {
    super.onFinishInflate();
```

```
    detailView = (MyDetailView) getChildAt(1);
  }

  public boolean onBackPressed() {
    return false;
  }

  @Override public void showItem(String item) {
    detailView.setItem(item);
  }
}
```

```
public class SinglePaneContainer extends FrameLayout implements Container {
  private ItemListView listView;

  public SinglePaneContainer(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override protected void onFinishInflate() {
    super.onFinishInflate();
    listView = (ItemListView) getChildAt(0);
  }

  public boolean onBackPressed() {
    if (!listViewAttached()) {
      removeViewAt(0);
      addView(listView);
      return true;
    }
    return false;
  }

  @Override public void showItem(String item) {
    if (listViewAttached()) {
      removeViewAt(0);
      View.inflate(getContext(), R.layout.detail, this);
    }
    MyDetailView detailView = (MyDetailView) getChildAt(0);
    detailView.setItem(item);
  }

  private boolean listViewAttached() {
    return listView.getParent() != null;
  }
}
```

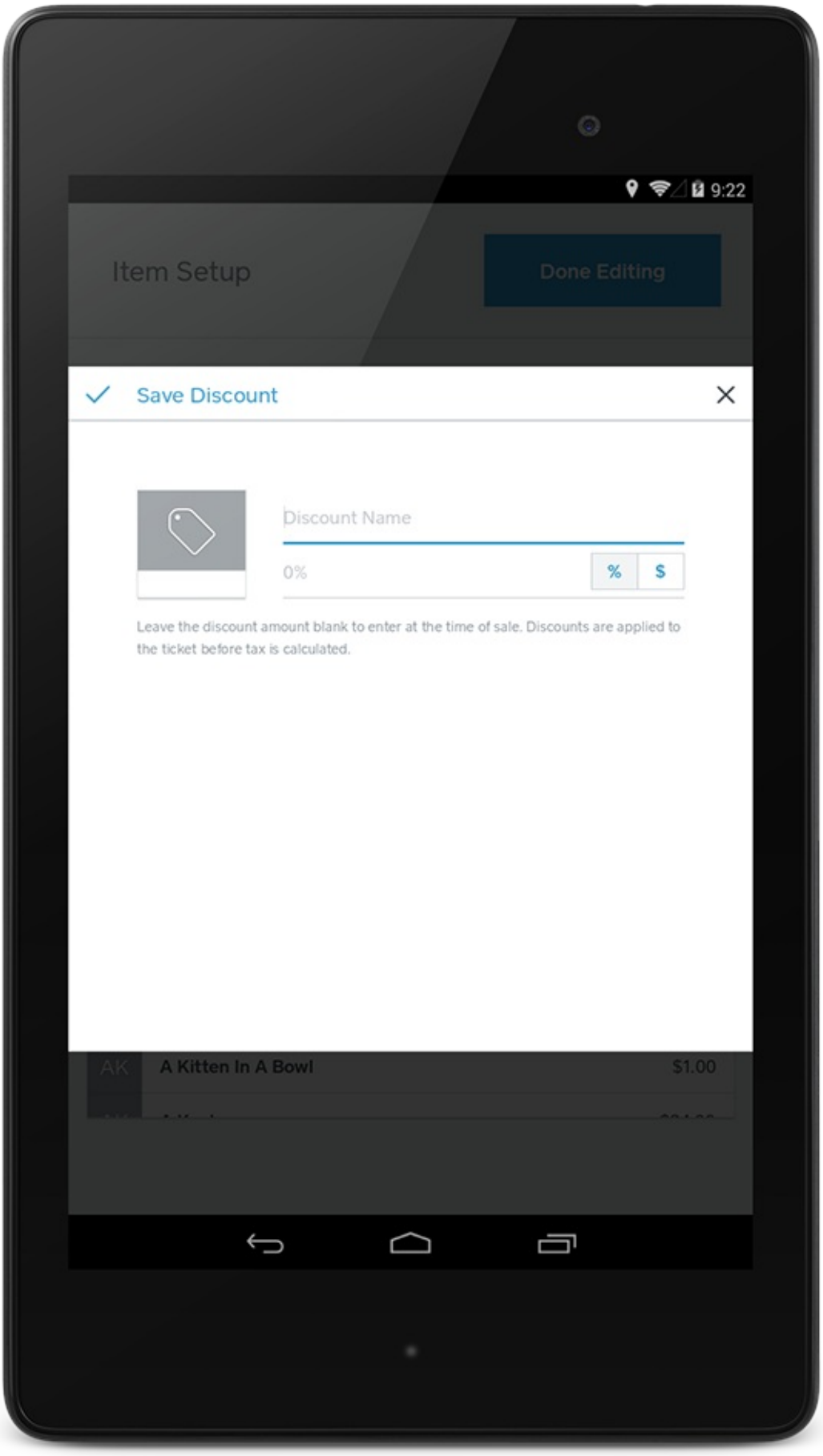
抽象出这些container并以这种方式来构建app并不难－我们不仅不需要fragments，而且代码将是易于理解的。

Views & presenters

使用自定义views是很棒的，但我们想把业务逻辑分离到专门的controllers中。我们把这些controller称为presenters。这样一来，代码将更加可读，测试更加容易。上面例子中的MyDetailView如下所示：

```
public class MyDetailView extends LinearLayout {
  TextView textView;
  DetailPresenter presenter;

  public MyDetailView(Context context, AttributeSet attrs) {
    super(context, attrs);
    presenter = new DetailPresenter();
  }

  @Override protected void onFinishInflate() {
    super.onFinishInflate();
    presenter.setView(this);
    textView = (TextView) findViewById(R.id.text);
    findViewById(R.id.button).setOnClickListener(new OnClickListener() {
      @Override public void onClick(View v) {
        presenter.buttonClicked();
      }
    });
  }

  public void setItem(String item) {
    textView.setText(item);
  }
}
```

让我们看一下从Square Register中抽取的代码，编辑账号信息的界面如下：



presenter在高层级操作view：

```
class EditDiscountPresenter {
  // ...
  public void saveDiscount() {
    EditDiscountView view = getView();
    String name = view.getName();
    if (isBlank(name)) {
      view.showNameRequiredWarning();
      return;
    }
    if (isNewDiscount()) {
      createNewDiscountAsync(name, view.getAmount(), view.isPercentage());
    } else {
      updateNewDiscountAsync(discountId, name, view.getAmount(),
        view.isPercentage());
    }
    close();
  }
}
```

为这个presenter编写测试是轻而易举的事：

```
@Test public void cannot_save_discount_with_empty_name() {
  startEditingLoadedPercentageDiscount();
  when(view.getName()).thenReturn("");
  presenter.saveDiscount();
  verify(view).showNameRequiredWarning();
  assertThat(isSavingInBackground()).isFalse();
}
```

返回栈管理

管理返回栈不需要异步事务，我们发布了一个小的函数库Flow来实现这个功能。Ray Ryan写了一篇很赞的博文介绍Flow。

我已经深陷在**fragment**的泥沼中，我如何逃离呢？

把fragments做成空壳，把view相关的代码写到自定义view类中，把业务逻辑代码写到presenter中，由presenter和自定义views进行交互。这样一来，你的fragment几乎就是空的了，只需要在其中inflate自定义views，并把views和presenters关联起来。

```
public class DetailFragment extends Fragment {
  @Override public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
    return inflater.inflate(R.layout.my_detail_view, container, false);
  }
}
```

到这里，你可以消除fragment了。

从fragments模式移植过来并不容易，但我们做到了－感谢Dimitris Koutsogiorgas 和 Ray Ryan的杰出工

作。

Dagger&Mortar如何呢？

Dagger&Mortar和fragments是正交的，它们可以和fragments一起工作，也可以脱离fragments而工作。

Dagger帮助我们把app模块化成一个解耦的组件图。他处理所有的绑定，使得可以很容易的提取依赖并编写自相关对象。

Mortar工作于Dagger之上，它具有两大优点：

- 它为被注入组件提供简单的生命周期回调。这使你可以编写在屏幕旋转时不会被销毁的presenters单例，而且可以保存状态到bundle中从而在进程死亡中存活下来。
- 它为你管理Dagger子图，并帮你把它绑定到activity的生命周期中。这让你有效的实现范围的概念：一个views生成的时候，它的presenter和依赖会作为子图创建；当views销毁的时候，你可以很容易的销毁这个范围，并让垃圾回收起作用。

结论

我们曾经大量的使用fragments，但最终改变了我们的想法：

- 我们很多疑难的crashes都和fragment生命周期相关；
- 我们只需要views来构建响应式的UI，一个返回栈和屏幕转场功能。

Android中的依赖注入：Dagger函数库的使用（一）

原文链接：http://antonioleiva.com/dependency-injection-android-dagger-part-1/

在这个新系列中，我将解释什么是依赖注入，它的主要目的是什么，以及在Android工程中如何使用Dagger函数库实现它，Dagger是目前最流行的专为Android设计的依赖注入函数库。

本文是之前的文章《Android中MVP的实现》的后续之作，因为我相信读者中有一部分人会很乐意看到这两个特性在同一个工程中实现，而且我认为它们可以很好的协同工作。

本文将仅介绍基本的理论知识来奠定基础。理解依赖注入的概念以及它存在的理由是很重要的，否者我们会认为得到的好处不如付出的努力。

什么是依赖注入

如果我们想要注入依赖，首先要理解依赖是什么。简单的说，依赖是我们代码中两个模块之间的耦合（在面向对象语言中，指的是两个类），通常是其中一个模块使用另外一个提供的功能。

为什么依赖是危险的？

从上层到底层依赖都是危险的，因为我们在某种程度上把两个模块进行耦合，这样当需要修改其中一个模块时，我们必须修改与其耦合的模块的代码。这对于创建一个可测试的app来说是很不利的，因为单元测试要求测试一个模块时，要保证它和app中其他模块是隔离的。为了做到这一点，我们需要使用mocks代替依赖，想象一下这样的代码：

```
public class Module1{
   private Module2 module2;

   public Module1(){
      module2 = new Module2();
   }

   public void doSomething(){
      ...
      module2.doSomethingElse();
      ...
   }
}
```

如何在不测试doSomethingElse函数的前提下测试doSomething函数呢？如果测试失败，是哪个函数导致的呢？我们不得而知。如果doSomethingElse函数在数据库中保存数据或者向服务器端发起API请求，那么事情将变得更加糟糕。

每当敲下new关键字我们都应该意识到这可能是需要避免的强依赖。编写更少的模块当然不是解决方案，不要忘记单一职责原则。

如何解决呢？依赖反转

如果不能在一个模块内部初始化另外的模块，那么需要以其他的形式初始化这些模块。你能想象如何实现吗？没错，通过构造函数。这基本上就是依赖反转原则的涵义了。你不应该依赖具体的模块对象，应该依赖抽象。

前面的代码应该修改为：

```
public class Module1{
   private Module2 module2;

   public Module1(Module2 module2){
      this.module2 = module2
   }

   public void doSomething(){
      ...
      module2.doSomethingElse();
      ...
   }
}
```

那么什么是依赖注入呢？

你已经知道了！它通过构造函数传递依赖（注入），从而把创建模块的任务从另一个模块内部抽离出来。对象在其他地方创建，并以构造函数参数的形式传递给另一个对象。

但新问题出现了。如果我们不能在模块内部创建其他的模块，那么必须有个地方对这些模块进行初始化。另外，如果我们需要创建的模块的构造函数包含大量的依赖参数，代码将变得丑陋和难以阅读，app中将存在大量传递的对象。依赖注入正是为解决这类问题而诞生的。

什么是依赖注入器？

我们可以把它理解成app中的另一个模块，专门负责提供其他模块的实例并注入他们的依赖。这是它的基本义务。模块的创建集中于app中的一个统一的入口，我们对它有完全的控制权。

最后，什么是**Dagger**？

Dagger是专为低端设备设计的依赖注入器。大部分依赖注入器通过反射来创建和注入依赖。反射机制是很棒的，但在低端设备上面耗时严重，特别是在老的android版本上面。然而，Dagger使用预编译器创建工作所需的所有类。这样一来，就不需要用到反射了。Dagger相比其他依赖注入器功能稍弱，但却是效率最高的。

Dagger只是用于测试吗？

当然不是！它使得在其他app中重用你的模块变得容易，或者在相同的app中改变这些模块。想象这样一个例子：你的app在debug模式下需要从本地文件中读取数据，而在release模式下需要从服务器端API请求中获取这些数据。完全有可能通过在不同的模式下注入不同的模块来实现。

结论

我知道这篇文章有点难度，但我认为在继续下一篇文章之前，有必要把基本概念先明确好。我们已经知道什么是依赖，通过依赖反转改进了什么以及我们如何通过依赖注入器实现它。

在下一篇文章中我们会开始真正的实践操作，敬请关注！

Android中的依赖注入：Dagger函数库的使用（二）

原文链接：http://antonioleiva.com/dagger-android-part-2/

如果你阅读了关于依赖注入的第一篇文章，你可能期待看到真实的代码。在Dagger网页上面可以找到coffee makers优美的例子，以及由Jake Wharton创建的，更适用于有经验读者的很棒的示例工程。但我们需要更简单的例子，coffee makers并不是我们主要的业务模式。因此，本文会提供简单的例子，展示注入简单的组件，让我们理解基本知识。

本文介绍的例子源码可以在这个页面找到https://github.com/antoniolg/DaggerExample；

在工程中引入Dagger

如果想使用Dagger的话，需要添加两个函数库：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.squareup.dagger:dagger:1.2.+'
    provided 'com.squareup.dagger:dagger-compiler:1.2.+'
}
```

第一个是Dagger函数库，第二个是Dagger编译器函数库，它会创建注入依赖所需的类。通过创建预编译的类可以避免大部分的反射操作。由于我们只需要Dagger编译器函数库来编译工程，在应用中不会使用到它，因此我们把它标记为provided，这样在最终的apk中就不会包含这个函数库了。

创建第一个module

Modules是使用Dagger经常碰到的概念，所以你需要习惯它们。Modules是提供依赖注入时所需对象实例的类，它们通过@Module注解来修饰类。还有其他一些额外参数可能需要配置，但我们会在使用到的时候再进行介绍。

创建一个名为AppModule的类，假设用于提供Application Context。为Application Context提供简单的访问入口通常是有趣的。我创建继承自Application的类App，并添加到AndroidManifest文件中。

```
@Module(
        injects = {
                App.class
        }
)
public class AppModule {

    private App app;

    public AppModule(App app) {
        this.app = app;
    }

    @Provides @Singleton public Context provideApplicationContext() {
```

```
        return app;
    }
}
```

上面涉及到哪些新知识呢？

- @Module：把这个类标识为Dagger module。 injects：标识module将要注入这个类的任何依赖。我们需要明确指定将直接注入到对象图中的那些类，这将很快会讲到。
- @Providers：标识函数作为注入提供者，函数名并不重要，它只依赖于所提供的类类型。
- @Singleton：如果标识为Singleton，那这个函数会一直返回相同的对象实例，这比常规的单例好很多。否则，每次注入类型都会得到一个新的实例。在这个例子中，由于我们没有创建新实例，而是返回已经存在的实例，因此即使不把函数标识为Singleton，每次调用还是会返回相同的实例的，但这样能够更好地说明提供者到底做了什么。Application实例是唯一的。

为什么常规的单例是不好的

单例可能是一个工程能够具有的最危险的依赖了。首先，由于我们没有创建实例，因此很难知道要在什么地方使用它，因此具有“隐藏依赖”。另外，我们没有办法通过mock来测试它，或者以另外的模块替换它，因此代码变得难以维护，测试和进化。被注入的单例，相反，具有单例的优势（唯一的实例），同时，由于我们可以在任何时候创建新的实例，因此很容易mock或者通过子类化或者使其实现一个公有接口来使用另一个代码片段替换它。

我们将在一个新的名为domain的包里面创建另外一个module。在每个架构层拥有至少一个module是很有用的。这个module将提供一个统计管理器，在app启动时通过显示一个Toast来抛出一个事件。在实际的工程中，这个管理器可能调用任何统计服务例如Google Analytics。

```
@Module(
        complete = false,
        library = true
)
public class DomainModule {

    @Provides @Singleton public AnalyticsManager provideAnalyticsManager(Application app)
        return new AnalyticsManager(app);
    }

}
```

通过把这个module指明为未完成，表示这个module的某些依赖需要另外一个module来提供。也就是AppModule里面的Application。当我们从一个依赖注入请求这个AnalyticsManager时，dagger将会使用这个函数，并会检测到它需要另外一个依赖：Application，这会通过向对象图发起请求来获得。我们同时需要把module指明为library，因为dagger编译器会检测到AnalyticsMananger没有被它自身或者它的注入类所使用。对于AppModule来说，DomainModule就是一个库模块。

我们将指定AppModule会包含DomainModule，因此返回AppModule类，修改代码如下：

```
@Module(
        injects = {
                App.class
        },
        includes = {
                DomainModule.class
        }
)
public class AppModule {
...
}
```

includes属性正是用来实现这个目的的。

创建对象图

对象图是所有依赖存在的地方。对象图包含被创建的实例，并能够把这些实例注入到相应的对象中。

在前面的例子中（AnalyticsManager）我们看到了经典的依赖注入，注入是通过构造函数参数传递的。但在Android中有的类（Application，Activity）我们对构造函数没有控制权，因此我们需要另外一种方式来注入依赖。

ObjectGraph创建和直接注入的组合方式在App类中可以看到。主对象图在Application类中创建并被注入以获得依赖。

```
public class App extends Application {

    private ObjectGraph objectGraph;
    @Inject AnalyticsManager analyticsManager;

    @Override public void onCreate() {
        super.onCreate();
        objectGraph = ObjectGraph.create(getModules().toArray());
        objectGraph.inject(this);
        analyticsManager.registerAppEnter();
    }

    private List<Object> getModules() {
        return Arrays.<Object>asList(new AppModule(this));
    }
}
```

我们通过@Inject注解来指明谁是依赖，这些依赖项必须是public或者default范围的，以便dagger可以给它们赋值。我们创建了一个modules数组（我们只有一个module，DomainModule是包含在AppModule里面的），并通过它来创建一个ObjectGraph实例。然后，我们手动注入App实例，这样之后，依赖就被注入了，因此可以直接调用AnalyticsManager的函数了。

结论

现在你掌握了Dagger的基础用法。ObjectGraph和Modules是有效的使用Dagger必须掌握的最有意思的组

件。还有其他更多的工具例如懒注入或者provider注入，在Dagger网页上面有介绍，但在你熟练掌握本文所介绍的知识之前，我不建议你深入研究它们。

不要忘记托管在Github上面的源代码哦。

下一篇（很可能是最后一篇）关于Dagger的文章将重点讨论scoped object graphs。它主要包括创建只存活在创建者范围内的新对象图。为activities创建scoped graphs是很常见的。

ASCE1885的移动开发技术周报（第一期）

> 毫不费力就到嘴的食物，不是毒药，就是诱饵。---《狼图腾》

精彩博文

1）炫丽的进度条Loading动效

前两天我们这边的头儿给我说，有个 gif 动效很不错，可以考虑用来做项目里的loading，问我能不能实现，看了下效果确实不错，也还比较有新意，复杂度也不是非常高，所以就花时间给做了

2）跨平台开发时代的 (再次) 到来？

这篇文章主要想谈谈最近又刮起的移动开发跨平台之风，并着重介绍和对比一下像是 Xamarin，NativeScript 和 React Native 之类的东西。不会有特别深入的技术讨论，大家可以当作一篇科普类的文章来看。

3）OKHttp源码解析

Android为我们提供了两种HTTP交互的方式：HttpURLConnection 和 Apache HTTP Client，虽然两者都支持HTTPS，流的上传和下载，配置超时，IPv6和连接池，已足够满足我们各种HTTP请求的需求。但更高效的使用HTTP可以让您的应用运行更快、更节省流量。而OkHttp库就是为此而生。

4）Google+ 团队的 Android UI 测试

Google+ 团队总结了一些 UI 测试时的经验和策略。

5）Android中保存和恢复Fragment状态的最好方法

经过这几年使用Fragment之后，我想说，Fragment的确是一种充满智慧的设计，但是使用Fragment时有太多需要我们逐一解决的问题，尤其是在处理数据保持的时候。

6）Android 编程下 Touch 事件的分发和消费机制

Android 中与 Touch 事件相关的方法包括：dispatchTouchEvent(MotionEvent ev)、onInterceptTouchEvent(MotionEvent ev)、onTouchEvent(MotionEvent ev)；能够响应这些方法的控件包括：ViewGroup、View、Activity。

7）Android NDK开发Crash错误定位

在Android开发中，程序Crash分三种情况：未捕获的异常、ANR（Application Not Responding）和闪退（NDK引发错误）。其中未捕获的异常根据logcat打印的堆栈信息很容易定位错误。ANR错误也好查，Android规定，应用与用户进行交互时，如果5秒内没有响应用户的操作，则会引发ANR错误，并弹出一个系统提示框，让用户选择继续等待或立即关闭程序。并会在/data/anr目录下生成一个traces.txt文件，记录

系统产生anr异常的堆栈和线程信息。如果是闪退，这问题比较难查，通常是项目中用到了NDK引发某类致命的错误导致闪退。

8）Android&Gradle集成NDK开发

介绍了如何在Android&Gradle开发环境中集成NDK，并使用自定义的android.mk进行编译。

9）iOS图片加载速度极限优化—FastImageCache解析

FastImageCache是Path团队开发的一个开源库，用于提升图片的加载和渲染速度，让基于图片的列表滑动起来更顺畅，来看看它是怎么做的。

开源函数库

1）Fresco

Facebook最近开源的功能强大的Android平台图片请求和加载函数库，主要特性如下：

- 为了节省流量和CPU，提供了三级缓存：两个内存缓存和一个内部存储缓存
- 支持类似网页JPEG图片的渐进式加载
- 支持GIF和WebP格式
- 支持多样式显示，如圆角，加载进度条，点击重试等

2）Sweet Alert Dialog

漂亮的Android对话框实现，交互及动画效果很赞。

3）AndroidDevTools

超级棒的Android开发工具网站，收集整理Android开发所需的Android SDK、开发中用到的工具、Android开发教程、Android设计规范，免费的设计素材等。

Android中判断app何时启动和关闭的技术研究

> 只有两种东西能让一个团队团结，恐惧或忠诚。---《速度与激情7》

原文链接：http://engineering.meetme.com/2015/04/android-determine-when-app-is-opened-or-closed/

存在的问题

Android开发中不可避免的会遇到需要检查app何时进入前台，何时被用户关闭。奇怪的是，要达到这个目的并不容易。检查app第一次启动并不难，但要判断它何时重新打开和关闭就没有那么简单了。

这篇文章将介绍一种判断app打开，重新打开和关闭的技术。

让我们开始吧

判断一个app打开和关闭的关键在于判断它的activities是否正在前台显示。让我们先从简单的例子开始，一个只有一个activity的app，而且不支持水平模式。这样想要判断app是打开还是关闭只需要检查activity的onStart和onStop方法即可：

```
@Override
protected void onStart() {
    super.onStart();
    // The Application has been opened!
}

@Override
protected void onStop() {
    super.onStop();
    // The Application has been closed!
}
```

上面例子的问题在于当需要支持水平模式时该方法就失效了。当我们旋转设备时activity将会重建，onStart方法将被再次调用，这时将会错误的判断为app第二次被打开。

为了处理设备旋转的情况，我们需要增加一个校验步骤。当activity退出时启动一个定时器，用于判断短时间内app的这个activity是否又被启动，如果没有，说明用户真的退出了这个app，如果重新启动了这个activity，说明用户还逗留在这个app中。

这种校验方式也适用于拥有多个activities的app，因为从app的一个activity跳转到另一个activity也可以用这种校验方式来处理。

使用这个技术我创建了一个管理类，所有的activities在可见和不可见时都会通知这个管理类。这个管理类为每个activity处理上述的校验步骤，从而避免错误的检测。它也提供了发布订阅（观察者）模式，任何对app启动和关闭感兴趣的模块都可以通过它来得到对应的通知。

这个管理类的使用分为三个步骤：

1）把它添加到你的工程中

```
import android.app.Activity;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.support.annotation.NonNull;
import android.text.format.DateUtils;

import java.lang.ref.Reference;
import java.lang.ref.WeakReference;
import java.util.HashSet;
import java.util.Set;

/**
 * This class is responsible for tracking all currently open activities.
 * By doing so this class can detect when the application is in the foreground
 * and when it is running in the background.
 */
public class AppForegroundStateManager {
    private static final String TAG = AppForegroundStateManager.class.getSimpleName();
    private static final int MESSAGE_NOTIFY_LISTENERS = 1;
    public static final long APP_CLOSED_VALIDATION_TIME_IN_MS = 30 * DateUtils.SECOND_IN_
    private Reference<Activity> mForegroundActivity;
    private Set<OnAppForegroundStateChangeListener> mListeners = new HashSet<>();
    private AppForegroundState mAppForegroundState = AppForegroundState.NOT_IN_FOREGROUND
    private NotifyListenersHandler mHandler;

    // Make this class a thread safe singleton
    private static class SingletonHolder {
        public static final AppForegroundStateManager INSTANCE = new AppForegroundStateMa
    }

    public static AppForegroundStateManager getInstance() {
        return SingletonHolder.INSTANCE;
    }

    private AppForegroundStateManager() {
        // Create the handler on the main thread
        mHandler = new NotifyListenersHandler(Looper.getMainLooper());
    }

    public enum AppForegroundState {
        IN_FOREGROUND,
        NOT_IN_FOREGROUND
    }

    public interface OnAppForegroundStateChangeListener {
        /** Called when the foreground state of the app changes */
        public void onAppForegroundStateChange(AppForegroundState newState);
    }

    /** An activity should call this when it becomes visible */
    public void onActivityVisible(Activity activity) {
        if (mForegroundActivity != null) mForegroundActivity.clear();
        mForegroundActivity = new WeakReference<>(activity);
        determineAppForegroundState();
    }
```

```
    /** An activity should call this when it is no longer visible */
    public void onActivityNotVisible(Activity activity) {
        /*
         * The foreground activity may have been replaced with a new foreground activity
         * So only clear the foregroundActivity if the new activity matches the foregroun
         */
        if (mForegroundActivity != null) {
            Activity ref = mForegroundActivity.get();

            if (activity == ref) {
                // This is the activity that is going away, clear the reference
                mForegroundActivity.clear();
                mForegroundActivity = null;
            }
        }

        determineAppForegroundState();
    }

    /** Use to determine if this app is in the foreground */
    public Boolean isAppInForeground() {
        return mAppForegroundState == AppForegroundState.IN_FOREGROUND;
    }

    /**
     * Call to determine the current state, update the tracking global, and notify subscr
     */
    private void determineAppForegroundState() {
        /* Get the current state */
        AppForegroundState oldState = mAppForegroundState;

        /* Determine what the new state should be */
        final boolean isInForeground = mForegroundActivity != null && mForegroundActivity
        mAppForegroundState = isInForeground ? AppForegroundState.IN_FOREGROUND : AppFore

        /* If the new state is different then the old state the notify subscribers of the
        if (mAppForegroundState != oldState) {
            validateThenNotifyListeners();
        }
    }

    /**
     * Add a listener to be notified of app foreground state change events.
     *
     * @param listener
     */
    public void addListener(@NonNull OnAppForegroundStateChangeListener listener) {
        mListeners.add(listener);
    }

    /**
     * Remove a listener from being notified of app foreground state change events.
     *
     * @param listener
     */
    public void removeListener(OnAppForegroundStateChangeListener listener) {
        mListeners.remove(listener);
    }
```

```
    /** Notify all listeners the app foreground state has changed */
    private void notifyListeners(AppForegroundState newState) {
        android.util.Log.i(TAG, "Notifying subscribers that app just entered state: " + n

        for (OnAppForegroundStateChangeListener listener : mListeners) {
            listener.onAppForegroundStateChange(newState);
        }
    }

    /**
     * This method will notify subscribes that the foreground state has changed when and
     * <br><br>
     * We do not want to just notify listeners right away when the app enters of leaves t
     * closing the app quickly we briefly pass through a NOT_IN_FOREGROUND state that mus
     * Sent when we detect a change. We will not notify that a foreground change happened
     * foreground change is detected during the delay period then the notification will b
     */
    private void validateThenNotifyListeners() {
        // If the app has any pending notifications then throw out the event as the state
        if (mHandler.hasMessages(MESSAGE_NOTIFY_LISTENERS)) {
            android.util.Log.v(TAG, "Validation Failed: Throwing out app foreground state
            mHandler.removeMessages(MESSAGE_NOTIFY_LISTENERS);
        } else {
            if (mAppForegroundState == AppForegroundState.IN_FOREGROUND) {
                // If the app entered the foreground then notify listeners right away; th
                mHandler.sendEmptyMessage(MESSAGE_NOTIFY_LISTENERS);
            } else {
                // We need to validate that the app entered the background. A delay is us
                // background but we do not want to consider the app being backgrounded s
                mHandler.sendEmptyMessageDelayed(MESSAGE_NOTIFY_LISTENERS, APP_CLOSED_VAL
            }
        }
    }

    private class NotifyListenersHandler extends Handler {
        private NotifyListenersHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message inputMessage) {
            switch (inputMessage.what) {
                // The decoding is done
                case MESSAGE_NOTIFY_LISTENERS:
                    /* Notify subscribers of the state change */
                    android.util.Log.v(TAG, "App just changed foreground state to: " + mA
                    notifyListeners(mAppForegroundState);
                    break;
                default:
                    super.handleMessage(inputMessage);
            }
        }
    }
}
```

*2）Activities*在可见性改变的需要发送通知

app中所有activities都要增加下面的代码，用于可见性改变时通知管理类。最好的实现方式是把这段代码加到工程的BaseActivity中。

```
@Override
protected void onStart() {
    super.onStart();
    AppForegroundStateManager.getInstance().onActivityVisible(this);
}

@Override
protected void onStop() {
    AppForegroundStateManager.getInstance().onActivityNotVisible(this);
    super.onStop();
}
```

3）订阅*app*的前台可见性改变事件

在感兴趣的模块中订阅app前台可见性改变事件，application类的onCreate函数是一个不错的地方，它可以保证每次app启动和关闭，你都能得到通知。

```
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        AppForegroundStateManager.getInstance().addListener(this);
    }

    @Override
    public void onAppForegroundStateChange(AppForegroundStateManager.AppForegroundState n
        if (AppForegroundStateManager.AppForegroundState.IN_FOREGROUND == newState) {
            // App just entered the foreground. Do something here!
        } else {
            // App just entered the background. Do something here!
        }
    }

}
```

进一步的思考

有一些细节需要进一步讨论，下面讨论的几点针对具体的应用可以做微调。

校验时间

校验定时器检查app是否真的进入后台的时间间隔是多少合适呢？上面的代码设置为30秒，原因如下。

当你的app在运行时，可能存在第三方的activities会覆盖全屏幕，一些常见的例子是Google应用内购买和Facebook登录注册页面。这些情况下你的app都会被迫进入后台，前台用于显示这些第三方页面。如果把这种情况当做用户离开了你的app，显然是不对的。30秒超时设置就是用来避免这种情况的。例如当用户在30秒内完成应用内购买，大部分用户都可以做得到，那么就不会当做用户突然离开app了。

如果你的app不存在上述这种情况，我建议可以把你的校验时间设置为4秒，这样对于低配设备当屏幕旋转重新创建activity的时间间隔是合适的。

CPU休眠

可能存在的问题是当用户关闭app或者app仍处于前台时用户锁屏了，这时CPU可能不会等到定时器检测就休眠了。为了保证这种情况下定时器能够正常检测用户退出app，我们需要持有wakelock防止CPU休眠直到app关闭事件被确认。实践中相比使用wakelock，这种情况并不算问题。

判断app是如何启动的

现在我们已经知道如何检测app何时启动和关闭，但我们不知道app是如何启动的。是用户点击通知栏消息？还是点击一个链接？亦或是他们直接通过桌面图标或最近使用启动？

跟踪启动机制

首先我们需要知道在哪里检测app是如何启动的。基于前面一个例子我们可以打印出app何时启动，以及如何启动。

```
public class MyApplication extends Application {
    public final String TAG = MyApplication.class.getSimpleName();

    public enum LaunchMechanism {
        DIRECT,
        NOTIFICATION,
        URL;
    }

    private LaunchMechanism mLaunchMechanism = LaunchMechanism.DIRECT;

    public void setLaunchMechanism(LaunchMechanism launchMechanism) {
        mLaunchMechanism = launchMechanism;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        AppForegroundStateManager.getInstance().addListener(this);
    }

    @Override
    public void onAppForegroundStateChange(AppForegroundStateManager.AppForegroundState n
        if (AppForegroundStateManager.AppForegroundState.IN_FOREGROUND.equals(newState))
            // App just entered the foreground.
            Log.i(TAG, "App Just Entered the Foreground with launch mechanism of: " + mLa
        } else {
            // App just entered the background. Set our launch mode back to the default o
            mLaunchMechanism = LaunchMechanism.DIRECT;
        }
    }
}
```

设置启动机制

现在我们可以打印app何时启动的机制，但我们没有设置它。因此下一步就是在用户通过链接或者通知启动app时我们记下它。如果没有通过这两种方式设置过，说明用户是通过点击app图标启动的。

跟踪链接点击事件

为了跟踪用户点击链接打开app，你需要找到代码中处理链接的地方，并加入下面的代码来跟踪启动机制。要确保这些代码在activity的onStart()函数之前调用。在哪些地方加入下面的代码取决于你的app架构了。

```
getApplication().setLaunchMechanism(LaunchMechanism.URL);
```

跟踪通知事件

不幸的是跟踪通知点击需要更多技巧，通知显示后，点击它将会打开之前绑定好的一个PendingIntent，这里的技巧是为通知的所有PendingIntents添加一个标识表明是由通知发出的。

例如当为通知创建PendingIntent时为每个intent添加如下代码：

```
public static final String EXTRA_HANDLING_NOTIFICATION = "Notification.EXTRA_HANDLING_NOT

// Put an extra so we know when an activity launches if it is a from a notification
intent.putExtra(EXTRA_HANDLING_NOTIFICATION, true);
```

到这一步我们需要做的就是在每个activity(统一在BaseActivity中添加)中检查这个标识。当识别到这个标识时说明是从通知启动的，这时可以把启动机制设置为通过通知。这一步应该在onCreate中处理，这样在app启动到前台之前就设置好了(会触发启动机制的打印)。

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Intent intent = getIntent();
    if (intent != null && intent.getExtras() != null) {
        // Detect if the activity was launched by the user clicking on a notification
        if (intent.getExtras().getBoolean(EXTRA_HANDLING_NOTIFICATION, false)) {
            // Notify that the activity was opened by the user clicking on a notification
            getApplication().setLaunchMechanism(LaunchMechanism.NOTIFICATION);
        }
    }
}
```

本文接近尾声了，到这里你应该已经掌握了如何检测app何时启动和关闭，以及它是如何启动的。

ASCE1885的移动开发技术周报（第二期）

精彩博文

1）iOS最佳实践

跳进了 iOS 的坑真是麻烦。无论是 Swift 还是 Objective-C， 都没有在其他地方广泛使用，而且这个平台对每个东西都几乎有它自己的命名方式，并且连要在真机上调试都充满了坎坷。无论你是刚刚入门 Cocoa 还是想纠正自己开发习惯的开发者，都能从本文档获益。不过下面写的仅仅是建议，所以如果你有一个更好的方案，那就试试吧！

2）JNI/NDK开发指南系列

相信很多做过Java或Android开发的朋友经常会接触到JNI方面的技术，由其做过Android的朋友，为了应用的安全性，会将一些复杂的逻辑和算法通过本地代码（C或C++）来实现，然后打包成so动态库文件，并提供Java接口供应用层调用，这么做的目的主要就是为了提供应用的安全性，防止被反编译后被不法分子分析应用的逻辑。当然打包成so也不能说完全安全了，只是相对反编译Java的class字节码文件来说，反汇编so动态库来分析程序的逻辑要复杂得多，没那么容易被破解。

3）Android字体渲染器

任何一个有几年的客户端应用开发经验的开发者都会知道文本渲染有多复杂。至少我在2010年开始写libhwui(基于OpenGL的安卓2D绘制API)之前是这么认为的。在开始写libhwui后，我意识到如果试图用GPU来渲染文本会使文本渲染变得更复杂。

4）怎样尊重一个程序员

得知一位久违的同学来到了旧金山湾区，然而我见到他时，这人正处于一生中最痛苦的时期。他告诉我，自己任职的公司在他加入之前和之后，判若两人。录取的时候公司对他说，我们对你在实习期间的表现和学术背景非常满意，你不用面试，甚至不用毕业拿学位，直接就可以加入我们公司成为正式员工。

5）Google推荐的图片加载库Glide介绍

在泰国举行的谷歌开发者论坛上，谷歌为我们介绍了一个名叫 Glide 的图片加载库，作者是bumptech。这个库被广泛的运用在google的开源项目中，包括2014年google I/O大会上发布的官方app。 它的成功让我非常感兴趣。我花了一整晚的时间把玩，决定分享一些自己的经验。在开始之前我想说，Glide和Picasso有90%的相似度，准确的说，就是Picasso的克隆版本。但是在细节上还是有不少区别的。

6）facebook的Android调试工具Stetho介绍

Stetho是Facebook出品的一个强大的Android调试工具，使用该工具你可以在Chrome Developer Tools查看App的布局，网络请求，sqlite，preference，一切都是可视化的操作，无须自己在去使用adb，也不需要root你的设备。使用的方式很简单，配置好之后，在Chrome地址栏输入chrome://inspect （哈哈，和webview 远程调试的方式一样）。

7）什么是函数响应式编程（Java&Android版本）

函数响应式编程（FRP）为解决现代编程问题提供了全新的视角。一旦理解它，可以极大地简化你的项目，特别是处理嵌套回调的异步事件，复杂的列表过滤和变换，或者时间相关问题。 我将尽量跳过对函数响应式编程学院式的解释（网络上已经有很多），并重点从实用的角度帮你理解什么是函数响应式编程，以及工作中怎么应用它。本文将围绕函数响应式编程的一个具体实现RxJava， 它可用于Java和Android。

开源函数库

1）中科院开源协会镜像站 Android SDK镜像测试发布

科技网最大的镜像站，中科院开源协会镜像站项目正式启动。 目前先行发布Android SDK镜像。支持IPV6，享受飞一般的速度。

2）pugnotification

Android平台上创建通知的强大的函数库

3）Android APK Decompiler

一个在线的APK反编译工具

ASCE1885的移动开发技术周报（第三期）

> Life isn't about how to survive the storm,it's about learning to dance in the rain -- Taylor Swift

精彩博文

1） Android中判断app何时启动和关闭的技术研究

Android开发中不可避免的会遇到需要检查app何时进入前台，何时被用户关闭。奇怪的是，要达到这个目的并不容易。检查app第一次启动并不难，但要判断它何时重新打开和关闭就没有那么简单了。 这篇文章将介绍一种判断app打开，重新打开和关闭的技术。

2）Android性能优化系列 渲染篇 运算篇 内存篇 电量篇

Google近期在Udacity上发布了Android性能优化的在线课程，分别从渲染，运算与内存，电量几个方面介绍了如何去优化性能，这些课程是Google之前在Youtube上发布的Android性能优化典范专题课程的细化与补充。

3）iOS应用架构谈系列 开篇 view层的组织和调用方案

安居客iOS app开发者的iOS架构经验分享，目前出了两篇文章。

4）唐巧的QCon参会笔记

QCon全球软件开发大会一线参会者的感受。

5）Android Studio入门指南

总结的比较全面的适用于国内开发者的Android Studio安装和基本使用教程，献给终于想从Eclipse＋Ant转向Android Studio＋Gradle的开发者们。

6）Android最佳性能实践系列 合理管理内存 分析内存的使用情况 高性能编码优化 布局优化技巧

Android性能优化的一些常规实践，虽然有点老调重弹，但本系列总结的不错，还是值得一读的。

7）Android Support Library 22.1

最新发布的Android支持库一如既往地添加了许多实用的组件，并对Support V4、AppCompat、Leanback、RecyclerView、Palette和Renderscript库的内部实现逻辑作出改变。从新的AppCompatActivity和AppCompatDialog 到Android TV全新的引导流程我们可以发现，新的库确实带来许多让我们耳目一新的惊喜。

8）我为什么主张反对使用Android Fragment

最近我在Droidcon Paris举办了一场技术讲座，我讲述了Square公司在使用Android fragments时遇到的问题，以及其他人如何避免使用fragments。

开源函数库和工具

1）adb-idea

一个Android Studio和Intellij IDEA插件，用于加速日常Android的开发，主要提供的功能有：

1. 卸载app
2. 杀掉app
3. 启动app
4. 重新启动app
5. 清除app数据
6. 清除app数据并重新启动

2）QCon2015的嘉宾演讲资料

刚刚结束的QCon 2015的嘉宾演讲资料汇总，感兴趣的筒靴们可以翻阅一下。

3）Cmd Markdown客户端

作业部落的Markdown编辑器全平台（Windows/Mac/Linux/浏览器）客户端发布，很好用的一款MarkDown编辑器，支持作业部落的云端同步和发布。