Android设计架构 — 进化

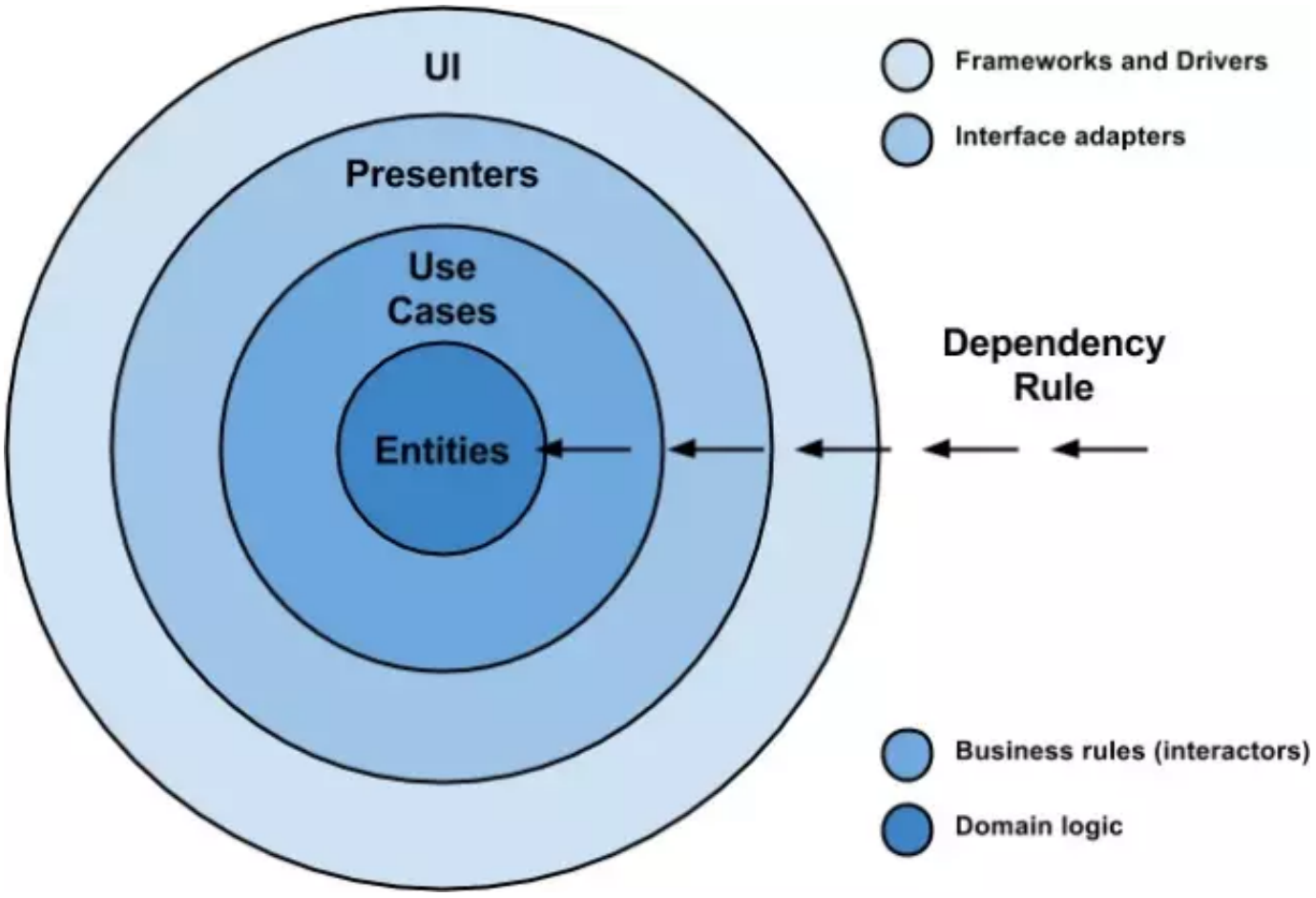
2015-09-21 安卓应用频道

(点击上方公众号，可快速关注)

> 英文：Fernando Cejas
> 译文：enigma
> 链接：http://android.jobbole.com/81541/

嘿！一段时间（收到很多的反馈意见）后，我认为是时候回到这个主题。这篇文章将给你另一种尝试，一种在我看来是设计现代移动应用架构的好方法（这里指的是Android 平台）。

在开始之前，假定你已经读过我的前面推送的文章Android设计架构 — 简洁之道 。如果没有读过，这是一个阅读的好机会，有助于更好地理解接下来的文章内容。

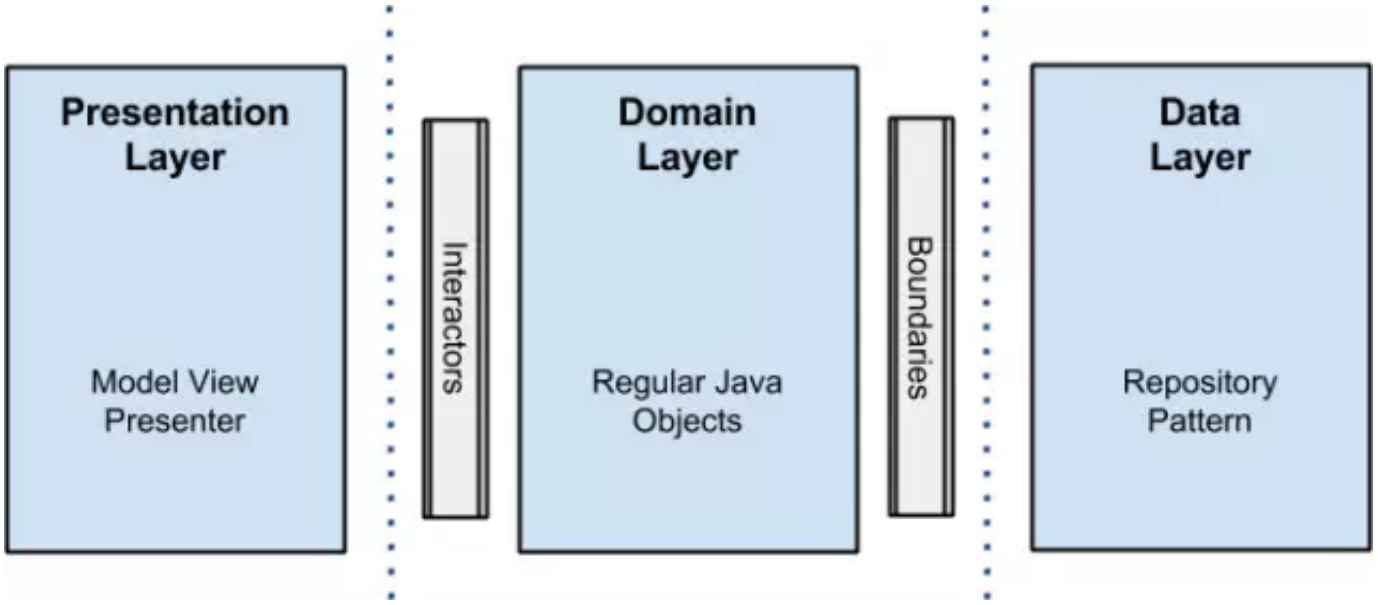


架构进化

进化（Evolution）代表一个渐进的过程，事物在这个过程演变成不同的形式，通常是更复杂或更好的形式。

如此说来，软件会随着时间改变，最终进化成为架构。其实一个好的软件设计必须保持其健壮性来帮助我们成长，拓展我们的解决方案，而不是重写代码（虽然有些情况下，重写代码是更好的方式，但是这应该是另一篇文章的主题。所以相信我，我们应该更多关注前面指出的问题）。

在这篇文章中，我将会带你检阅我认为必要和重要的关键点，以保持我们 Android 代码库的健壮性。记住这张图，我们开始吧。



响应式方法：RxJava

我不准备在此讨论 RxJava 的好处（我想你早已经尝试过它了， RxJava）。该技术已经有很多文章和大牛，他们做了出色的工作！而我将指出它在 Android 应用开发方面的有趣之处，以及它是怎样帮助我在简洁的架构上踏出第一步。

首先，我通过转化用例（use cases， 在简洁构架命名规则中称作“交互器”， interactors）选择了一个响应式模型返回 Obervables<T>。这意味着所有底层也将随着调用链返回 Obervables<T>。

```
public abstract class UseCase {
  private final ThreadExecutor threadExecutor;
  private final PostExecutionThread postExecutionThread;
  private Subscription subscription = Subscriptions.empty();
  protected UseCase(ThreadExecutor threadExecutor,
      PostExecutionThread postExecutionThread) {
   this.threadExecutor = threadExecutor;
   this.postExecutionThread = postExecutionThread;
  }
```

```
  protected abstract Observable buildUseCaseObservable();
  public void execute(Subscriber UseCaseSubscriber) {
   this.subscription = this.buildUseCaseObservable()
       .subscribeOn(Schedulers.from(threadExecutor))
       .observeOn(postExecutionThread.getScheduler())
       .subscribe(UseCaseSubscriber);
  }
  public void unsubscribe() {
   if (!subscription.isUnsubscribed()) {
     subscription.unsubscribe();
   }
  }
}
```

如你所见，所有的用例都继承自这个抽象类并且实现了抽象方法 buildUseCaseObservable() ，这个方法将设定一个 Observable<T> 用以处理复杂逻辑并返回所需的数据。

值得一提的是 execute() 方法，我们要确保 Observable<T> 在一个单独的线程里执行，因此极大程度上避免了我们 Android 主线程的阻塞。结果由 Android 主线程调度器发送给主线程。

至此，我们的 Observable<T> 已经启动并且运行了。但是如你所知，必须有人关注由此发送的数据序列。为了实现这个功能，我把展示器（presenters， 表示层 presentation Layer 的 MVP 模式中的一部分）发展成 订阅者（Subscribers）。它将“响应（react）”这些用例发出的信息，用以更新用户界面。

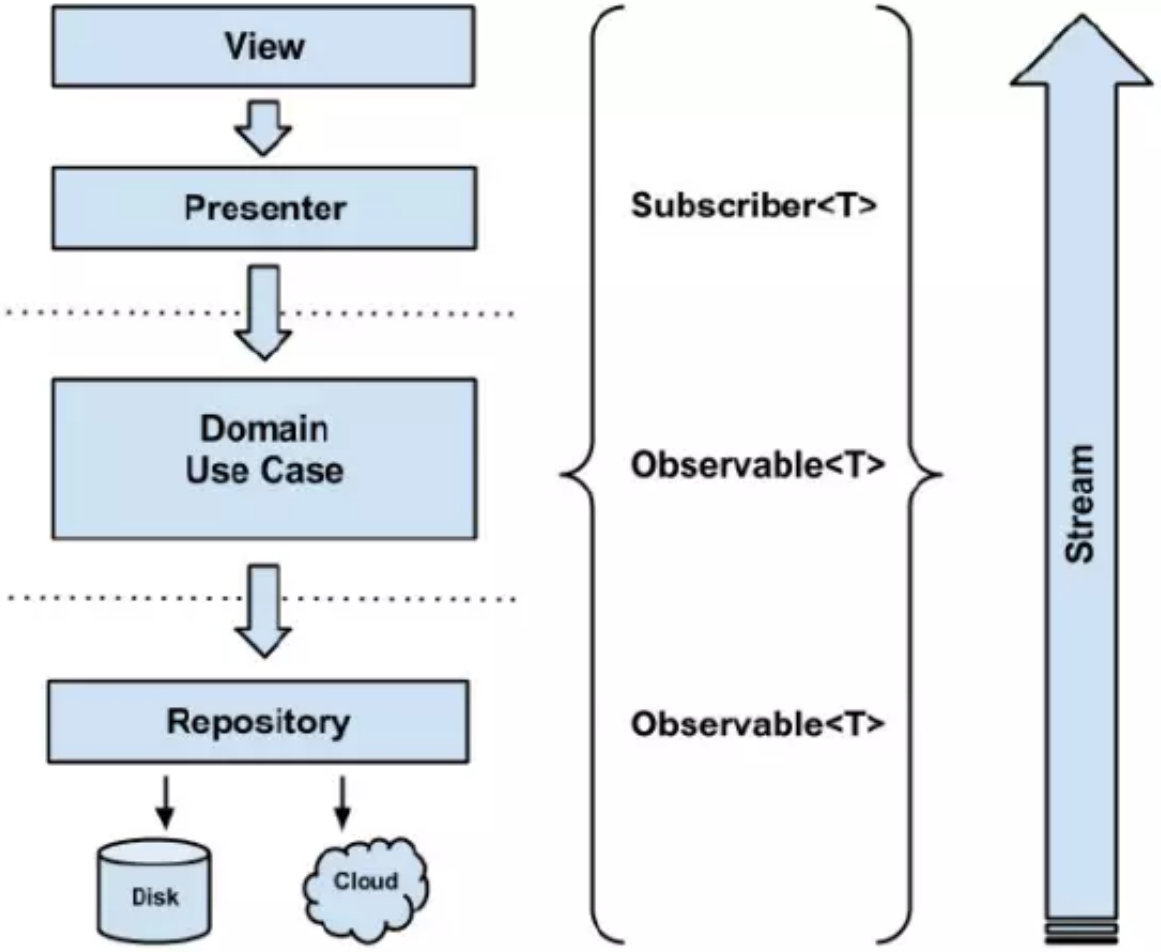
如下代码所示即为订阅者：

```
private final class UserListSubscriber extends DefaultSubscriber<List<User>> {
  @Override public void onCompleted() {
    UserListPresenter.this.hideViewLoading();
  }
  @Override public void onError(Throwable e) {
    UserListPresenter.this.hideViewLoading();
    UserListPresenter.this.showErrorMessage(new DefaultErrorBundle((Exception) e));
    UserListPresenter.this.showViewRetry();
  }
  @Override public void onNext(List<User> users) {
```

```
    UserListPresenter.this.showUsersCollectionInView(users);
  }
}
```

每个订阅者是都是嵌套在各自的展示器里的内部类，并且实现了 DefaultSubscriber<T>，用于默认错误处理。

在所有的组件都就绪后，你可以通过下图来了解整个想法：



让我们列举一下基于 RxJava方法的一些好处：

- 低耦合的 观察者和 订阅者：提高了【可】维护性，并使测试更简单；
- 简化的异步任务（asynchronous tasks）：在多于单个层面的异步执行是必须的情况下， Java 中的 Thread 和Future 就变得维护复杂、难以同步。通过调度程序我们可以轻松地在后台和主线程之间跳转（不需要额外的功夫），在需要更新 UI 的时候更是格外简单。同时也避免了我们说的“回调地狱（Callback Hell）”，它将使我们的代码无法阅读并且难以跟进。

- 数据转化与组成：在不影响客户端的情况下可以组合多个 Observables<T>，这让我们的方案更具有扩展性。

- 错误处理：当任何 Observable<T> （的实现类）出现错误时，都有一个信号发送给消费者。

我的观点里有一个缺陷，事实上这个方案需要付出的代价——那就是不熟悉这个概念的开发者必须要付出努力完成学习曲线。然而，你将从中获得无价的东西。响应式方法必胜！（译者注：学习曲线 Learning curve，表示了经验与效率之间的关系）

依赖（dependency）注入：Dagger 2

我不准备过多讨论依赖注入的问题，因为我已经写了一篇完整的文章了。我非常推荐你读这篇文章，这样你就能跟上这里讨论的内容。

值得一提的是，通过实现一个像 Dagger 2 这样的依赖注入框架，我们可以得到：

- 组件复用， 因为依赖注入和配置独立于组件之外。

- 得益于对象的初始化驻留在一个孤立且耦合性低的位置。当注入抽象方法的时候，我们只需要修改对象的实现方法，而不用大改代码库。

- 依赖可以注入到一个组件中：完全可以注入这些依赖的模拟实现，这使测试更加简单了。

Lambda 表达式：Retrolambda

没有人会抱怨利用 Java 8 的 Lambad 表达式，尤其当他们简化了代码并且摆脱了很多模式的时候。可以看看以下代码：

```
private final Action1<UserEntity> saveToCacheAction =
  userEntity -> {
   if (userEntity != null) {
    CloudUserDataStore.this.userCache.put(userEntity);
   }
  };
```

然而，我却对此有复杂的感情，我会对此进行解释。原因是我和 @SoundCloud 有过一个关于Retrolambda讨论，主要围绕是否使用Retrolambda，结果如下：

1. 优点：
    - Lambda 表达式和方法的引用；
    - Try with语句可以直接在资源上使用；
    - 开发的业报（Dev karma）。
2. 缺点：
    - Java 8 API使用的频率很低；
    - 第三方库非常具有侵入性；
    - 依赖第三方 Gradle 插件使之工作在 Android 上。

最后我们认定它不能为我们解决问题：虽然你的代码看起来更好而且可读性更高，但这不是必需品。因为现今所有的大型 IDE 都包含代码折叠的选项，这至少在可以接受的方式下满足了这一需求。

说实话，虽然很有可能在业余项目上再次使用，但是我在这里使用是抱着玩耍的心态在 Android 上尝试 Lambda 表达式。在这里只是我的看法，你可以自己决定。当然，这个库的作者的出色工作值得称赞。

测试方法

在测试方面第一版的例子差距不大：

- 表示层：使用Android instrumentation或espresso进行 UI 测试；
- 领域层：因为是常规 Java 模块，使用 JUnit 和 Mockito 进行测试；
- 数据层： 迁移数据测试使用 Robolectric 3 + JUnit + Mockito。这一层的测试曾经使用了一个独立的 Android 模型，因为当时（在第一版的示例的时候）还没有内建测试支持，也没有一个像 Robolectric 的框架，所以复杂度很高。需要一系列的暴力操作来使之正常工作。

幸运的是，这是过去的事情了。现在所有的东西都是开箱即用了，这样我就可以将它们迁移到数据模块里，指定它的默认测试位置： src/test/java 文件夹。

包结构

我认为代码和包的结构是好架构的一个关键因素：包结构是程序员浏览源代码时第一个看到的东西。一切始之与它，一切依赖于它。

我们可以通过两种途径把应用划分成不同的包：

- 通过层次来分包：每个包包含的项目往往和其他包没有紧密关系。这导致了包之间存在低内聚和低模块化，而且有很高的耦合。结果是，编辑一个功能涉及了不同包的文件。此外，删除一个特性也几乎不可能在一个操作中执行。

- 通过功能来分包：这种方式使用包来反映功能集合。它尝试把涉及单个功能（且只有该功能的）所有的项目放到一个包里。这导致了包之间存在高内聚和模块化，而且耦合度极低。紧密相关的项目被相邻放置，而并不是散布在应用程序中。

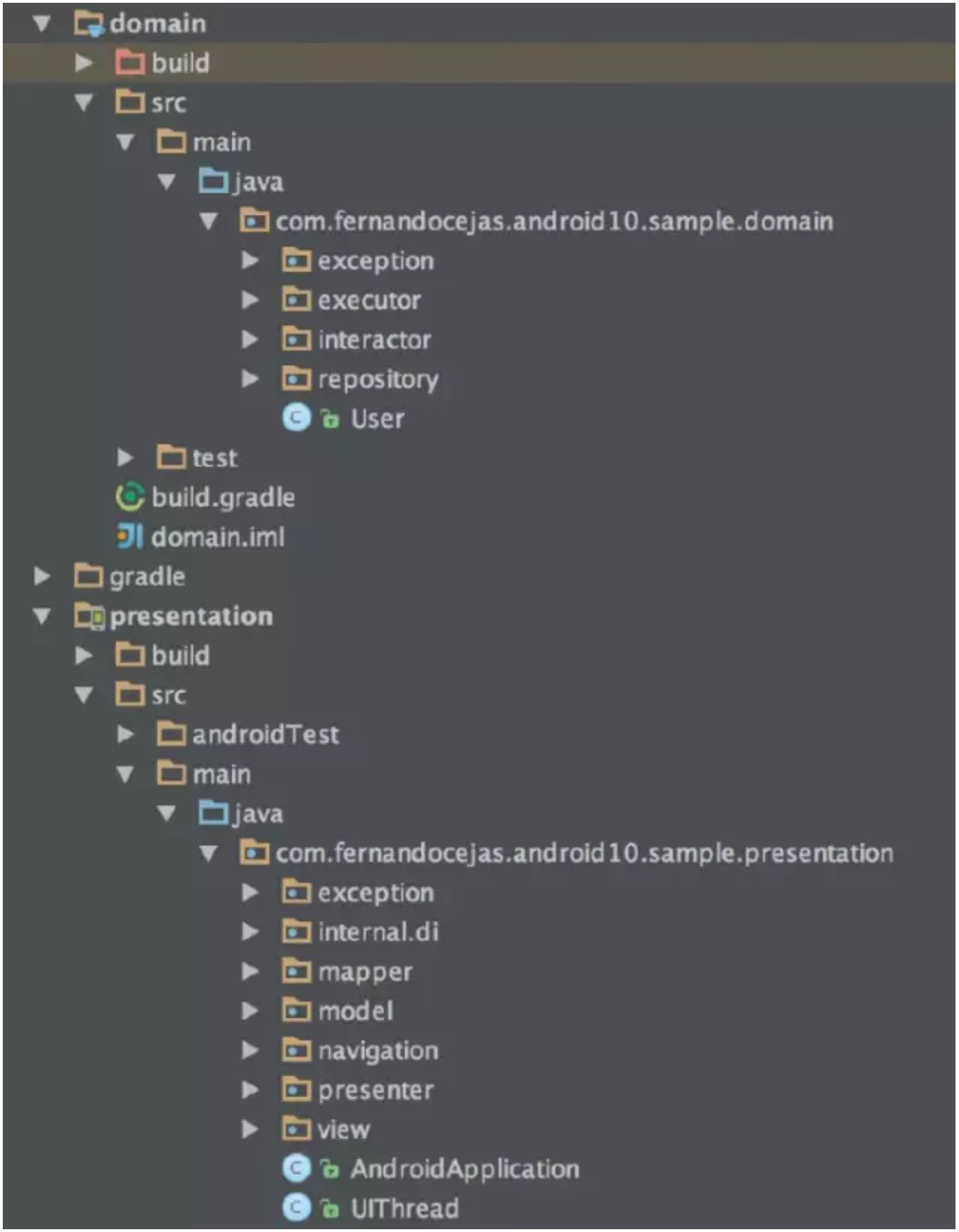
我推荐通过功能来分包，主要有以下的好处：

- 高度模块化

- 代码导航更容易

- 规模最小化

- 补充一点比较有趣的事情，你如何与功能团队（feature team）合作（就好像我们和@SoundCloud合作 一样），组织代码所有权会更容易，也更加模块化，这对于那些许多开发者在同一代码库中工作的成长型团队来说是非常有益的。



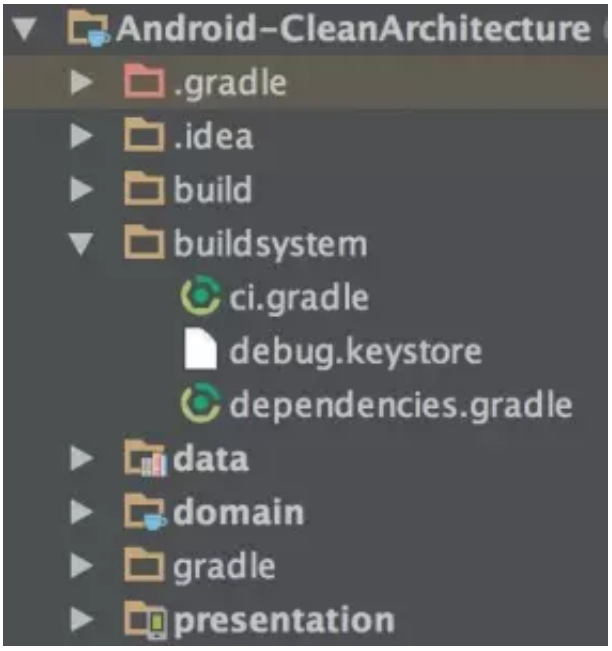
正如你所在图中看到的，我的方法看起来就是通过层次来分包的方式：我可能错了（比如把所有的东西都放在“users”下面），但在这种情况下我会原谅自己，这个例子是拿来作为学习的反例。这里我想展示整洁架构方法的关键概念。照我说的做，不要像我一样做 :)。

奖励球：组织你的构建逻辑

众所周知，建造一所房子要从地基开始。软件开发也是一样。我想说的是，在我看来，构建系统（以及其组织方式）是软件架构非常重要的一部分。

在 Android 开发中，我们使用Gradle 来构建。事实上它是一个非常强大的无关平台的构建系统。我准备在这里提供一堆提示和技巧来简化你的生活。当谈及组织应用的构建，有一条准则：

- 通过功能性分组，放入独立的 Gradle 构建文件中。




```
ci.gradle
def ciServer = 'TRAVIS'
def executingOnCI = "true".equals(System.getenv(ciServer))
// Since for CI we always do full clean builds, we don't want to pre-dex
// See http://tools.android.com/tech-docs/new-build-system/tips
subprojects {
  project.plugins.whenPluginAdded { plugin ->
    if ('com.android.build.gradle.AppPlugin'.equals(plugin.class.name) ||
        'com.android.build.gradle.LibraryPlugin'.equals(plugin.class.name)) {
      project.android.dexOptions.preDexLibraries = !executingOnCI
    }
  }
}
```

```
apply from: 'buildsystem/ci.gradle'
apply from: 'buildsystem/dependencies.gradle'
buildscript {
 repositories {
  jcenter()
 }
 dependencies {
  classpath 'com.android.tools.build:gradle:1.2.3'
  classpath 'com.neenbedankt.gradle.plugins:android-apt:1.4'
 }
}
allprojects {
 ext {
 ...
 }
}
...
```

这样的话，你就可以使用 “apply from: ‘buildsystem/ci.gradle’” 把配置放入任何 Gradle 构建文件。不要把所有配置都放在一个 build.gradle 文件里，否则你将开始创造一个怪物。经验之谈啊！

- 创建依赖关系图

```
...
ext {
 //Libraries
 daggerVersion = '2.0'
 butterKnifeVersion = '7.0.1'
 recyclerViewVersion = '21.0.3'
 rxJavaVersion = '1.0.12'
 //Testing
 robolectricVersion = '3.0'
 jUnitVersion = '4.12'
 assertJVersion = '1.7.1'
 mockitoVersion = '1.9.5'
 dexmakerVersion = '1.0'
 espressoVersion = '2.0'
 testingSupportLibVersion = '0.1'
```

```
  ...
  domainDependencies = [
    daggerCompiler:    "com.google.dagger:dagger-compiler:${daggerVersion}",
    dagger:            "com.google.dagger:dagger:${daggerVersion}",
    javaxAnnotation:   "org.glassfish:javax.annotation:${javaxAnnotationVersion}",
    rxJava:            "io.reactivex:rxjava:${rxJavaVersion}",
  ]
  domainTestDependencies = [
    junit:             "junit:junit:${jUnitVersion}",
    mockito:           "org.mockito:mockito-core:${mockitoVersion}",
  ]
  ...
  dataTestDependencies = [
    junit:             "junit:junit:${jUnitVersion}",
    assertj:           "org.assertj:assertj-core:${assertJVersion}",
    mockito:           "org.mockito:mockito-core:${mockitoVersion}",
    robolectric:       "org.robolectric:robolectric:${robolectricVersion}",
  ]
}
```

```
apply plugin: 'java'
sourceCompatibility = 1.7
targetCompatibility = 1.7
...
dependencies {
  def domainDependencies = rootProject.ext.domainDependencies
  def domainTestDependencies = rootProject.ext.domainTestDependencies
  provided domainDependencies.daggerCompiler
  provided domainDependencies.javaxAnnotation
  compile domainDependencies.dagger
  compile domainDependencies.rxJava
  testCompile domainTestDependencies.junit
  testCompile domainTestDependencies.mockito
}
```

如果想要在你的项目不同模块间重复利用一个工件（artifact）版本，或是必须给不同的模块应用不同的依赖版本，那么创建依赖关系图是非常有用的。另一个好处是，你还可以在一个地方控制依赖，例如增进工件版本变得非常简单。

总结

这基本上是我现在做的事情，作为一个总结，千万记住这世上没有银弹。然而，一个好软件的架构将帮助我们把代码变得简洁而又健壮，同时易于扩展和维护。（译者注：银弹，silver bullets，据说可以杀死狼人和吸血鬼）
还有几件事情我想要指出，当遇到一个软件问题的时候，你必须要采取的态度：

- 遵循 SOLID 原则；（译者注：S.O.L.I.D是面向对象设计和编程中几个重要编码原则的首字母缩写，分别是单一、开放、里氏、接口、依赖）
- 不要想太多（不把过度设计）；
- 遵循实用主义；
- 尽可能地在项目中减少框架依赖。

源代码

1. Clean architecture github repository – master branch
2. Clean architecture github repository – releases

拓展阅读

1. Architecting Android..the clean way
2. Tasting Dagger 2 on Android
3. The Mayans Lost Guide to RxJava on Android
4. It is about philosophy: Culture of a good programmer

参考资料

1. RxJava wiki by Netflix
2. Framework bound by Uncle Bob

3. Gradle user guide

4. Package by feature, not layer
