NotRxJava懒人专用指南

泡在网上的日子 / 文 发表于2015-05-25 18:21 第1806次阅读 RxJava (/tags.php?/RxJava/)

(http://www.jiathis.com/share)

编辑推荐：稀土掘金 (https://juejin.im/)，这是一个针对技术开发者的一个应用，你可以在掘金上获取最新最优质的技术干货，不仅仅是Android知识、前端、后端以至于产品和设计都有涉猎，想成为全栈工程师的朋友不要错过！

这篇文章不是教你如何使用RxJava，而是解释RxJava的演进过程。

- 原文链接 : NotRxJava guide for lazy folks (http://yarikx.github.io/NotRxJava/)
- 原文作者 : Yaroslav Heriatovych (http://yarikx.github.io/)
- 译文出自： 开发技术前线 www.devtf.cn (http://www.devtf.cn)
- 译者 : Rocko (https://github.com/Rocko)
- 校对者: Mr.Simple (https://github.com/bboyfeiyu)
- 状态： 完成校对

如果你是一位 Android 开发者，那么这些天你可能已经听到或看到一些关于 RxJava 满天飞的宣传了。RxJava 是一个能让你摆脱编写一些复杂繁琐的代码去处理异步事件的库。一旦开始在你的项目中使用，你会对它爱不释手的。

然而，RxJava 有个缺陷，它需要一个陡峭的学习过程。对于一个从未接触使用过 RxJava 的人来说，是很难一次就领会到它的精髓所在的，对于它的一些使用方法你也可能会很迷惑。在项目中使用它意味着你需要稍微地改变一下你的代码编写思路，另 外，这样的学习曲线会使得在项目中因为大规模的使用RxJava而引发一些问题。

当然，关于如何去使用 RxJava 已经有许多的教程和代码范例了。感兴趣的开发者可以访问 RxJava 的官方 Wiki (https://github.com/ReactiveX/RxJava/wiki)，里面有关于什么是 Observable 以及它和 Iterable、Future 之间关系的很好的解释。Wiki 里有一篇很有用的文章：How To Use RxJava (https://github.com/ReactiveX/RxJava/wiki/How-To-Use-RxJava)，这篇文章包含怎么去发送事件流并且打印出它们的介绍以及它的样例代码。

但我们要明确的是在还没有学习什么是 Observable 的前提下了解 RxJava 用来解决什么问题以及它是怎么帮助我们组织起异步代码的。

我这篇文章的定位就是 RxJava 官方文档的“前篇”，读完这篇文章能更好地去理解 RxJava 所解决的问题。文章中也有一个小 Demo，就是自己怎么去整理那些凌乱的代码，然后看看我们在没有使用 RxJava 的情况下是怎么去遵循 RxJava 基本原则的。

所以，如果你仍有足够的好奇的话就让我们开始吧！

Cat 应用程序

让我们来创建一个真实世界的例子。我们都知道猫是我们技术发展的引擎，所以就让我们也来创建这么一个用来下载猫图片的典型应用吧。

任务描述

> 我们有个 Web API，能根据给定的查询请求搜索到整个互联网上猫的图片。每个图片包含可爱指数的参数（描述图片可爱度的整型值）。我们的任务将会下载到一个猫列表的集合，选择最可爱的那个，然后把它保存到本地。

我们只关心下载、处理和保存猫的数据。

我们开始吧~

模型和 API

下面是描述“猫”的简单数据结构：

```
1.  public class Cat implements Comparable<Cat>{
2.      Bitmap image;
3.      int cuteness;
4.
5.      @Override
6.      public int compareTo(Cat another) {
7.          return Integer.compare(cuteness, another.cuteness);
8.      }
9.  }
```

还有我们传统阻塞式风格的 API，它被打包进 cat-sdk.jar 中了：

```
1.  public interface Api {
2.      List<Cat> queryCats(String query);
3.      Uri store(Cat cat);
4.  }
```

这足够清楚了吗？当然！那就让我们开始编写业务逻辑吧：

```
1.  public class CatsHelper {
2.
3.      Api api;
4.
5.      public Uri saveTheCutestCat(String query){
6.          List<Cat> cats = api.queryCats(query);
7.          Cat cutest = findCutest(cats);
8.          Uri savedUri = api.store(cutest);
9.          return savedUri;
10.     }
11.
12.     private Cat findCutest(List<Cat> cats) {
13.         return Collections.max(cats);
14.     }
15. }
```

唉，这样清晰简单的代码帅到让我窒息啊。来理清一下代码的炫酷之处吧。主方法 saveTheCutestCat 只包含了 3 个其它方法，然后花个几分钟来看看代码和思考这些方法。你给方法提供了输入参数然后就能得到结果返回了，在这个方法工作的时候我们需要等待它的完成。

简洁而有用，让我们再看看组合方法的其它优势：

组合

正如我们看到的，根据其它 3 个方法而新创建了一个方法（`saveTheCutestCat`），因此我们组合了它们。像乐高积木那样，我们把方法之间连接起来组成了乐高积木（实际上可以在之后组合起来）。组合方法是很简单的，从一个方法得到返回结果然后再把它传递给另外的方法做为输入参数，这不简单吗？

错误的传递

另外一个好处就是我们处理错误的方式了。任何一个方法都可能因执行时发生错误而被终止，这个错误能在任何层次上被处理掉，Java 中我们叫它抛出了异常，然后这个错误在 try/catch 代码块中做处理。这里的关键点是我们不需要为组合方法里的每个方法都做异常处理，仅需要对这些组合起来的方法做统一处理，像下面这样：

```
1.  try{
2.      List<Cat> cats = api.queryCats(query);
3.      Cat cutest = findCutest(cats);
4.      Uri savedUri = api.store(cutest);
5.      return savedUri;
6.  } catch (Exception e) {
7.      e.printStackTrace()
8.      return someDefaultValue;
9.  }
```

这个情况下，我们处理了所有执行时的错误，或者说如果我们没有使用 try/catch 代码块我们能够把错误传递到下一个层次上。

走向异步

要知道我们身在一个对等待很敏感的世界里，我们也知道不可能只有阻塞式的调用。在 Android 中我们也总需要处理异步代码。

拿 Android 的 `OnClickListener` 举个例子，当你需要处理一个控件的点击事件时，你必须提供一个监听器（回调）以供在用户点击控件时被调用。这没有理由使用阻塞的方式去接受点击事件的回调，所以对点击来说总是异步的。现在，让我们也使用异步编程吧。

异步的网络调用

开始想象下使用没有阻塞的 HTTP client（例如Ion (https://github.com/koush/ion)），还有就是我们的`cats-sdk.jar`已经更新。它的 API 也换成了异步的方式调用。

新 API 的接口：

```
1.  public interface Api {
2.      interface CatsQueryCallback {
3.          void onCatListReceived(List<Cat> cats);
4.          void onError(Exception e);
5.      }
6.
7.
8.      void queryCats(String query, CatsQueryCallback catsQueryCallback);
9.
10.     Uri store(Cat cat);
11. }
```

所以现在我们能异步的获取猫的信息集合列表了，返回正确或错误的结果时都会通过 `CatsQueryCallback`回调接口。

因为 API 改变了所以我们也不得不改变我们的`CatsHelper`:

```
public class CatsHelper {

    public interface CutestCatCallback {
        void onCutestCatSaved(Uri uri);
        void onQueryFailed(Exception e);
    }

    Api api;

    public void saveTheCutestCat(String query, CutestCatCallback cutestCatCallback){
        api.queryCats(query, new Api.CatsQueryCallback() {
            @Override
            public void onCatListReceived(List<Cat> cats) {
                Cat cutest = findCutest(cats);
                Uri savedUri = api.store(cutest);
                cutestCatCallback.onCutestCatSaved(savedUri);
            }

            @Override
            public void onQueryFailed(Exception e) {
                cutestCatCallback.onError(e);
            }
        });
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

现在我们已经不能使用阻塞的 API 了，我们也不能把我们的客户端上写成阻塞式的调用（实际上是可以的，但需要明确的在线程中使用`synchronized`、`CountdownLatch`、等等，也需要在下层中使用异步处理）。所以我们不能在 `saveTheCutestCat`方法中直接返回一个值，我们需要对它进行异步回调处理。

让我们再深入一点，如果我们 API 的两个方法调用都是异步的，举个例子，我们正在使用非阻塞IO去写进一个文件。

```
public interface Api {
    interface CatsQueryCallback {
        void onCatListReceived(List<Cat> cats);
        void onQueryFailed(Exception e);
    }

    interface StoreCallback{
        void onCatStored(Uri uri);
        void onStoreFailed(Exception e);
    }


    void queryCats(String query, CatsQueryCallback catsQueryCallback);

    void store(Cat cat, StoreCallback storeCallback);
}
```

还有我们的 helper：

```
public class CatsHelper {

    public interface CutestCatCallback {
        void onCutestCatSaved(Uri uri);
        void onError(Exception e);
    }

    Api api;

    public void saveTheCutestCat(String query, CutestCatCallback cutestCatCallback){
        api.queryCats(query, new Api.CatsQueryCallback() {
            @Override
            public void onCatListReceived(List<Cat> cats) {
                Cat cutest = findCutest(cats);
                api.store(cutest, new Api.StoreCallback() {
                    @Override
                    public void onCatStored(Uri uri) {
                        cutestCatCallback.onCutestCatSaved(uri);
                    }

                    @Override
                    public void onStoreFailed(Exception e) {
                        cutestCatCallback.onError(e);
                    }
                });
            }

            @Override
            public void onQueryFailed(Exception e) {
                cutestCatCallback.onError(e);
            }
        });
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

现在再来看看代码，跟之前一样优雅吗？明显不是了，这很糟糕！现在它有了更多无关代码和花括号，但是逻辑是一样的。

那么组合在哪呢？他已经不见了！现在你不能像之前那样组合操作了。对于每一个异步操作你都必须创建出回调接口并在代码中插入它们，每一次都需要手动地加入！

错误传递又在哪？又是一个否定！在这样的代码中错误不会自动地传递，我们需要在更深一层上通过自己手动地再让它传递下去（请看`onStoreFailed`和`onQueryFailed`方法）。

我们很难对这样的代码进行阅读和找出潜在的 bugs。

结束了？

结束了又怎样？我们能拿它来干嘛？我们被困在这个没有组合回调的地狱里了吗？前方高能，请抓紧你的安全带哦，我们将努力的去把这些干掉！

更美好的世界！

泛型回调

可以从我们的回调接口中找到共同的模式：

- 都有一个分发结果的方法（`onCutestCatSaved`,`onCatListReceived`,`onCatStored`）
- 它们中大多数（在我们的例子中是全部）有一个用于错误处理的方法（`onError`,`onQueryFailed`,`onStoreFailed`）

所以我们可以创建一个泛型回调接口去替代原来所有的接口。但是我们不能去改变 API 的调用方法的签名，我们必须创建包装类来间接调用。

所以我们的泛型回调接口看起来是这样的：

```
public interface Callback<T> {
    void onResult(T result);
    void onError(Exception e);
}
```

然后我们来创建`ApiWrapper`来改变一下调用方法的签名：

```
1.  public class ApiWrapper {
2.      Api api;
3.
4.      public void queryCats(String query, Callback<List<Cat>> catsCallback){
5.          api.queryCats(query, new Api.CatsQueryCallback() {
6.              @Override
7.              public void onCatListReceived(List<Cat> cats) {
8.                  catsCallback.onResult(cats);
9.              }
10.
11.             @Override
12.             public void onQueryFailed(Exception e) {
13.                 catsCallback.onError(e);
14.             }
15.         });
16.     }
17.
18.     public void store(Cat cat, Callback<Uri> uriCallback){
19.         api.store(cat, new Api.StoreCallback() {
20.             @Override
21.             public void onCatStored(Uri uri) {
22.                 uriCallback.onResult(uri);
23.             }
24.
25.             @Override
26.             public void onStoreFailed(Exception e) {
27.                 uriCallback.onError(e);
28.             }
29.         });
30.     }
31. }
```

所以这仅仅是对于`Callback`的一些传递 resuts/errors 的调用转发逻辑。

最后我们的`CatsHelper`：

```
1.  public class CatsHelper{
2.
3.      ApiWrapper apiWrapper;
4.
5.      public void saveTheCutestCat(String query, Callback<Uri> cutestCatCallback){
6.          apiWrapper.queryCats(query, new Callback<List<Cat>>() {
7.              @Override
8.              public void onResult(List<Cat> cats) {
9.                  Cat cutest = findCutest(cats);
10.                 apiWrapper.store(cutest, cutestCatCallback);
11.             }
12.
13.             @Override
14.             public void onError(Exception e) {
15.                 cutestCatCallback.onError(e);
16.             }
17.         });
18.     }
19.
20.     private Cat findCutest(List<Cat> cats) {
21.         return Collections.max(cats);
22.     }
23. }
```

可以看到比之前的简明了一些。我们可以通过直接传递一个顶级的`cutestCatCallback`回调接口给`apiWrapper.store`来减少回调间的层级调用，此外作为回调方法的签名是一样的。

但是我们可以做的更好！

你必须把它分开

让我们来看看我们的异步操作（`queryCats`，`queryCats`，还有`saveTheCutestCat`），它们都遵循了相同的模式。调用它们的方法有一些参数（`query`、`cat`）也包括一个回调对象。再次说明：任何异步操作需要携带所需的常规参数和一个回调实例对象。如果我们试图去分开这几个阶段，每个异步操作仅仅将会携带一个参数对象，然后返回一些携带着回调（信息）的临时对象。

我们来应用下这样的模式，看看是否对我们有所帮助。

如果在异步操作中返回一些临时对象，我们需要定义一个出来。这样的一个对象需要包括常见的行为（以回调为单一参数），我们将定义这样的类给所有的异步操作使用，这个类就叫它`AsyncJob`。

> P.S. 称之为`AsyncTask`更合适一点，但是我不希望你混淆了异步操作跟另外一个存在的抽象概念之间的关系（这是不好的一点）。

所以：

```
1.  public abstract class AsyncJob<T> {
2.      public abstract void start(Callback<T> callback);
3.  }
```

非常的简单，Callback传进方法后就会开始它的工作任务。

然后更改包装 API 的调用：

```
1.  public class ApiWrapper {
2.      Api api;
3.
4.      public AsyncJob<List<Cat>> queryCats(String query) {
5.          return new AsyncJob<List<Cat>>() {
6.              @Override
7.              public void start(Callback<List<Cat>> catsCallback) {
8.                  api.queryCats(query, new Api.CatsQueryCallback() {
9.                      @Override
10.                     public void onCatListReceived(List<Cat> cats) {
11.                         catsCallback.onResult(cats);
12.                     }
13.
14.                     @Override
15.                     public void onQueryFailed(Exception e) {
16.                         catsCallback.onError(e);
17.                     }
18.                 });
19.             }
20.         };
21.     }
22.
23.     public AsyncJob<Uri> store(Cat cat) {
24.         return new AsyncJob<Uri>() {
25.             @Override
26.             public void start(Callback<Uri> uriCallback) {
27.                 api.store(cat, new Api.StoreCallback() {
28.                     @Override
29.                     public void onCatStored(Uri uri) {
30.                         uriCallback.onResult(uri);
31.                     }
32.
33.                     @Override
34.                     public void onStoreFailed(Exception e) {
35.                         uriCallback.onError(e);
36.                     }
37.                 });
38.             }
39.         };
40.     }
41. }
```

目前一切顺利，我们只是部分运用我们的包装过的 API 调用在程序之中。现在我们可以开始使用AsyncJob去做我们想要的了，当然也到更改CatsHelper的时间了。

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        return new AsyncJob<Uri>() {
            @Override
            public void start(Callback<Uri> cutestCatCallback) {
                apiWrapper.queryCats(query)
                        .start(new Callback<List<Cat>>() {
                            @Override
                            public void onResult(List<Cat> cats) {
                                Cat cutest = findCutest(cats);
                                apiWrapper.store(cutest)
                                        .start(new Callback<Uri>() {
                                            @Override
                                            public void onResult(Uri result) {
                                                cutestCatCallback.onResult(result);
                                            }

                                            @Override
                                            public void onError(Exception e) {
                                                cutestCatCallback.onError(e);
                                            }
                                        });
                            }

                            @Override
                            public void onError(Exception e) {
                                cutestCatCallback.onError(e);
                            }
                        });
            }
        };
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

哇，之前的版本更简单些啊，我们现在的优势是什么？答案就是现在我们可以给客户端返回“组合”操作的`AsyncJob<Uri>`。所以一个客户端（在 activity 或者 fragment 处）可以用组合起来的工作来操作。

Breaking things

这是我们的逻辑数据流：

```
          (async)                (sync)          (async)
query ============> List<Cat> -------------> Cat ==========> Uri
        queryCats              findCutest           store
```

为了让我们的代码拥有之前的可读性，我们从这个事件流中强行进入到里面的操作里。但有件事需要注意，如果某些操作（方法）是异步的，然后调用它的操 作（方法）又是异步的，就比如，查询猫的操作是异步的，然后寻找出最可爱的猫（即使有一个阻塞调用）也是一个异步操作（客户端希望接收的结果）。

所以我们可以使用 AsyncJobs 把我们的方法分解成更小的操作：

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        AsyncJob<List<Cat>> catsListAsyncJob = apiWrapper.queryCats(query);
        AsyncJob<Cat> cutestCatAsyncJob = new AsyncJob<Cat>() {
            @Override
            public void start(Callback<Cat> callback) {
                catsListAsyncJob.start(new Callback<List<Cat>>() {
                    @Override
                    public void onResult(List<Cat> result) {
                        callback.onResult(findCutest(result));
                    }

                    @Override
                    public void onError(Exception e) {
                        callback.onError(e);
                    }
                });
            }
        };

        AsyncJob<Uri> storedUriAsyncJob = new AsyncJob<Uri>() {
            @Override
            public void start(Callback<Uri> cutestCatCallback) {
                cutestCatAsyncJob.start(new Callback<Cat>() {
                    @Override
                    public void onResult(Cat cutest) {
                        apiWrapper.store(cutest)
                                .start(new Callback<Uri>() {
                                    @Override
                                    public void onResult(Uri result) {
                                        cutestCatCallback.onResult(result);
                                    }

                                    @Override
                                    public void onError(Exception e) {
                                        cutestCatCallback.onError(e);
                                    }
                                });
                    }

                    @Override
                    public void onError(Exception e) {
                        cutestCatCallback.onError(e);
                    }
                });
            }
        };
        return storedUriAsyncJob;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

代码量多了许多，但是更加清晰了。低层次嵌套的回调，利于理解的变量名（`catsListAsyncJob`、`cutestCatAsyncJob`、`storedUriAsyncJob`）。

看起来好了许多，但我们要再做一些事情：

简单映射

现在看看 `AsyncJob<Cat> cutestCatAsyncJob`的部分：

```
AsyncJob<Cat> cutestCatAsyncJob = new AsyncJob<Cat>() {
            @Override
            public void start(Callback<Cat> callback) {
                catsListAsyncJob.start(new Callback<List<Cat>>() {
                    @Override
                    public void onResult(List<Cat> result) {
                        callback.onResult(findCutest(result));
                    }

                    @Override
                    public void onError(Exception e) {
                        callback.onError(e);
                    }
                });
            }
        };
```

这 16 行代码只有一行是对我们有用（对于逻辑来说）的操作：

```
findCutest(result)
```

剩下的仅仅是开启另外一个AsyncJob和传递结果与错误的样板代码。此外，这些代码并不用于特定的任务，我们可以把其移动到其它地方而不影响编写我们真正需要的业务代码。

我们该怎么写呢？我们必须做下面的两件事情：

- AsyncJob是我们转换的结果
- 转换方法

这又有另外一个问题，因为在 Java 中不能直接传递方法（函数）所以我们需要通过类（和接口）来间接实现这样的功能，然后我们就来定义这个 “方法”：

```
1.  public interface Func<T, R> {
2.      R call(T t);
3.  }
```

相当简单，Func接口有两个类型成员，T对应于参数类型而R对应于返回类型。

当我们从一个AsyncJob中装换处结果后我们就需要做一些值之间的映射，这样的方法我们就叫它map。定义这个方法实例（Func 类型）最好的地方就在AsyncJob类中，所以AsyncJob代码里看起来就是这样了：

```
1.  public abstract class AsyncJob<T> {
2.      public abstract void start(Callback<T> callback);
3.
4.      public <R> AsyncJob<R> map(Func<T, R> func){
5.          final AsyncJob<T> source = this;
6.          return new AsyncJob<R>() {
7.              @Override
8.              public void start(Callback<R> callback) {
9.                  source.start(new Callback<T>() {
10.                     @Override
11.                     public void onResult(T result) {
12.                         R mapped = func.call(result);
13.                         callback.onResult(mapped);
14.                     }
15.
16.                     @Override
17.                     public void onError(Exception e) {
18.                         callback.onError(e);
19.                     }
20.                 });
21.             }
22.         };
23.     }
24. }
```

赞，这时CatsHelper就是下面这样了：

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        AsyncJob<List<Cat>> catsListAsyncJob = apiWrapper.queryCats(query);
        AsyncJob<Cat> cutestCatAsyncJob = catsListAsyncJob.map(new Func<List<Cat>, Cat>() {
            @Override
            public Cat call(List<Cat> cats) {
                return findCutest(cats);
            }
        });

        AsyncJob<Uri> storedUriAsyncJob = new AsyncJob<Uri>() {
            @Override
            public void start(Callback<Uri> cutestCatCallback) {
                cutestCatAsyncJob.start(new Callback<Cat>() {
                    @Override
                    public void onResult(Cat cutest) {
                        apiWrapper.store(cutest)
                                .start(new Callback<Uri>() {
                                    @Override
                                    public void onResult(Uri result) {
                                        cutestCatCallback.onResult(result);
                                    }

                                    @Override
                                    public void onError(Exception e) {
                                        cutestCatCallback.onError(e);
                                    }
                                });
                    }

                    @Override
                    public void onError(Exception e) {
                        cutestCatCallback.onError(e);
                    }
                });
            }
        };
        return storedUriAsyncJob;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

现在好多了，创建`AsyncJob<Cat> cutestCatAsyncJob`只需要 6 行代码而回调也只有一个层级了。

高级映射

前面的那些已经很赞了，但是创建`AsyncJob<Uri> storedUriAsyncJob`的部分还有些不忍直视。能在这里创建映射吗？我们来试试吧：

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        AsyncJob<List<Cat>> catsListAsyncJob = apiWrapper.queryCats(query);
        AsyncJob<Cat> cutestCatAsyncJob = catsListAsyncJob.map(new Func<List<Cat>, Cat>() {
            @Override
            public Cat call(List<Cat> cats) {
                return findCutest(cats);
            }
        });

        AsyncJob<Uri> storedUriAsyncJob = cutestCatAsyncJob.map(new Func<Cat, Uri>() {
            @Override
            public Uri call(Cat cat) {
                return apiWrapper.store(cat);
        //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ will not compile
        //      Incompatible types:
        //      Required: Uri
        //      Found: AsyncJob<Uri>
            }
        });
        return storedUriAsyncJob;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

呃呃。。。并不容易哦，我们来修改下结果的类型变量，试下其它方法：

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        AsyncJob<List<Cat>> catsListAsyncJob = apiWrapper.queryCats(query);
        AsyncJob<Cat> cutestCatAsyncJob = catsListAsyncJob.map(new Func<List<Cat>, Cat>() {
            @Override
            public Cat call(List<Cat> cats) {
                return findCutest(cats);
            }
        });

        AsyncJob<AsyncJob<Uri>> storedUriAsyncJob = cutestCatAsyncJob.map(new Func<Cat, AsyncJob<Uri>>() {
            @Override
            public AsyncJob<Uri> call(Cat cat) {
                return apiWrapper.store(cat);
            }
        });
        return storedUriAsyncJob;
        //^^^^^^^^^^^^^^^^^^^^^^^^ will not compile
        //      Incompatible types:
        //      Required: AsyncJob<Uri>
        //      Found: AsyncJob<AsyncJob<Uri>>
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

在目前这点上我们只能有`AsyncJob<AsyncJob<Uri>>`。我们需要往更深处挖吗？我们希望的是，去把`AsyncJob`在一个级别上的两个异步操作扁平化成一个单一的异步操作。

现在我们需要的是得到能使方法返回映射成`R`类型也是`AsyncJob<R>`类型的操作。这个操作应该像`map`，但在最后应该`flatten`我们嵌套的`AsyncJob`。我们叫它为`flatMap`吧，然后就是来实现它：

```
public abstract class AsyncJob<T> {
    public abstract void start(Callback<T> callback);

    public <R> AsyncJob<R> map(Func<T, R> func){
        final AsyncJob<T> source = this;
        return new AsyncJob<R>() {
            @Override
            public void start(Callback<R> callback) {
                source.start(new Callback<T>() {
                    @Override
                    public void onResult(T result) {
                        R mapped = func.call(result);
                        callback.onResult(mapped);
                    }

                    @Override
                    public void onError(Exception e) {
                        callback.onError(e);
                    }
                });
            }
        };
    }

    public <R> AsyncJob<R> flatMap(Func<T, AsyncJob<R>> func){
        final AsyncJob<T> source = this;
        return new AsyncJob<R>() {
            @Override
            public void start(Callback<R> callback) {
                source.start(new Callback<T>() {
                    @Override
                    public void onResult(T result) {
                        AsyncJob<R> mapped = func.call(result);
                        mapped.start(new Callback<R>() {
                            @Override
                            public void onResult(R result) {
                                callback.onResult(result);
                            }

                            @Override
                            public void onError(Exception e) {
                                callback.onError(e);
                            }
                        });
                    }

                    @Override
                    public void onError(Exception e) {
                        callback.onError(e);
                    }
                });
            }
        };
    }
}
```

FlatMap 的粗略实现，但这些东西的实现都在一个地方了，在客户端的业务代码中不会再见到它。接下来我们修复下`CatsHelper`:

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        AsyncJob<List<Cat>> catsListAsyncJob = apiWrapper.queryCats(query);
        AsyncJob<Cat> cutestCatAsyncJob = catsListAsyncJob.map(new Func<List<Cat>, Cat>() {
            @Override
            public Cat call(List<Cat> cats) {
                return findCutest(cats);
            }
        });

        AsyncJob<Uri> storedUriAsyncJob = cutestCatAsyncJob.flatMap(new Func<Cat, AsyncJob<Uri>>() {
            @Override
            public AsyncJob<Uri> call(Cat cat) {
                return apiWrapper.store(cat);
            }
        });
        return storedUriAsyncJob;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

哈哈！它能用了，读和写也简单了不少。

最后的要点

再来看看我们编写的代码，眼熟吗？如果我们使用 Java 8 的 lambdas（逻辑是一样的但是看起来更爽一些）代码会更加地简洁。

```
public class CatsHelper {

    ApiWrapper apiWrapper;

    public AsyncJob<Uri> saveTheCutestCat(String query) {
        AsyncJob<List<Cat>> catsListAsyncJob = apiWrapper.queryCats(query);
        AsyncJob<Cat> cutestCatAsyncJob = catsListAsyncJob.map(cats -> findCutest(cats));
        AsyncJob<Uri> storedUriAsyncJob = cutestCatAsyncJob.flatMap(cat -> apiWrapper.store(cat));
        return storedUriAsyncJob;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

它看起来会更好吗？我认为这样的代码跟我们第一次阻塞的版本差不多：

```
public class CatsHelper {

    Api api;

    public Uri saveTheCutestCat(String query){
        List<Cat> cats = api.queryCats(query);
        Cat cutest = findCutest(cats);
        Uri savedUri = api.store(cutest);
        return savedUri;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

是的，就是这样，逻辑是相似的！也有可能会复杂些（语义是一样的）。
我们这样的代码有组合性吗？请大声的说有！我们组合了所有的异步操作然后作为返回结果我们仅需一个组合后的结果对象而已。
错误传递呢？当然也有！所有的错误都会传递到最后的回调中。
接下来的最后呢。。。

RxJava (https://github.com/ReactiveX/RxJava)

嘿，你不需要把那些代码拷到你的项目中，因为我们还是实现地不够完全的，仅仅算是非线程安全的 RxJava 的一小部分而已。

它们之间只有一些差异：

- `AsyncJob<T>` 就是实际上的 Observable (http://reactivex.io/documentation/observable.html)，它不仅可以只分发一个单一的结果也可以是一个序列（可以为空）。

- `Callback<T>` 就是 Observer (http://reactivex.io/documentation/operators/subscribe.html)，除了 Callback 少了`onNext(T t)`方法。Observer 中在`onError(Throwable t)`方法被调用后，会继而调用`onCompleted()`，然后 Observer 会包装好并发送出事件流（因为它能发送一个序列）。
- `abstract void start(Callback<T> callback)`对应 Subscription subscribe(final Observer<? super T> observer) (http://reactivex.io/RxJava/javadoc/rx/Observable.html#subscribe%28rx.Observer%29)，这个方法也返回 Subscription (http://reactivex.io/RxJava/javadoc/rx/Subscription.html)，在不需要它时你可以决定取消接收事件流。
- 除了`map`和`flatMap`方法，`Observable`在 Observalbes 之上也有一些其它 (http://reactivex.io/documentation/operators.html)有用的操作。

<

p>下面的代码是使用 RxJava 来完成我们前面自己写的代码的功能：

```
public class ApiWrapper {
    Api api;

    public Observable<List<Cat>> queryCats(final String query) {
        return Observable.create(new Observable.OnSubscribe<List<Cat>>() {
            @Override
            public void call(final Subscriber<? super List<Cat>> subscriber) {
                api.queryCats(query, new Api.CatsQueryCallback() {
                    @Override
                    public void onCatListReceived(List<Cat> cats) {
                        subscriber.onNext(cats);
                    }

                    @Override
                    public void onQueryFailed(Exception e) {
                        subscriber.onError(e);
                    }
                });
            }
        });
    }

    public Observable<Uri> store(final Cat cat) {
        return Observable.create(new Observable.OnSubscribe<Uri>() {
            @Override
            public void call(final Subscriber<? super Uri> subscriber) {
                api.store(cat, new Api.StoreCallback() {
                    @Override
                    public void onCatStored(Uri uri) {
                        subscriber.onNext(uri);
                    }

                    @Override
                    public void onStoreFailed(Exception e) {
                        subscriber.onError(e);
                    }
                });
            }
        });
    }
}

public class CatsHelper {

    ApiWrapper apiWrapper;

    public Observable<Uri> saveTheCutestCat(String query) {
        Observable<List<Cat>> catsListObservable = apiWrapper.queryCats(query);
        Observable<Cat> cutestCatObservable = catsListObservable.map(new Func1<List<Cat>, Cat>() {
            @Override
            public Cat call(List<Cat> cats) {
                return CatsHelper.this.findCutest(cats);
            }
        });
        Observable<Uri> storedUriObservable = cutestCatObservable.flatMap(new Func1<Cat, Observable<? extends Uri>>() {
            @Override
            public Observable<? extends Uri> call(Cat cat) {
                return apiWrapper.store(cat);
            }
        });
        return storedUriObservable;
    }

    private Cat findCutest(List<Cat> cats) {
        return Collections.max(cats);
    }
}
```

你可以看到代码是相同的，除了使用`Observable`来替代`AsyncJob`。

总结

我们看到，通过简单的转化我们可以把异步操作给抽象出来。这个抽象出来的东西可以被用来操作和组合异步操作就像简单的方法那样。通过这种方法我们可以摆脱嵌套的回调，在处理异步结果时也能手动处理错误的传递。

如果你看到了这里的话建议你放松下，思考下 sync/async 之间的二元关系，然后看看这个很棒的来自Erik Meijer (http://en.wikipedia.org/wiki/Erik_Meijer_%28computer_scientist%29)的视频 (https://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2014/Keynote-Duality)。

一些有用的链接

- http://reactivex.io (http://reactivex.io)
- https://github.com/ReactiveX/RxJava (https://github.com/ReactiveX/RxJava)
- https://github.com/ReactiveX/RxJava/wiki (https://github.com/ReactiveX/RxJava/wiki)
- http://queue.acm.org/detail.cfm?id=2169076 (http://queue.acm.org/detail.cfm?id=2169076)
- https://www.coursera.org/course/reactive (https://www.coursera.org/course/reactive)
- http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1 (http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/)

感谢

感谢我的朋友 Alexander Yakushev 帮忙翻译。