中山大学移动信息工程学院本科生实验报告

年级	1501	专业（方向）	移动（互联网）
学号	15352015	姓名	曹广杰
电话	13727022190	Email	1553118845@qq.com

Content

一、实验题目

`Retrofit+RxJava+OkHttp`实现网络请求

二、实现内容

本次实验模拟实现一个`github`盒子。

- 学习使用 Retrofit 实现网络请求
- 学习`RxJava`中 Observable 的使用

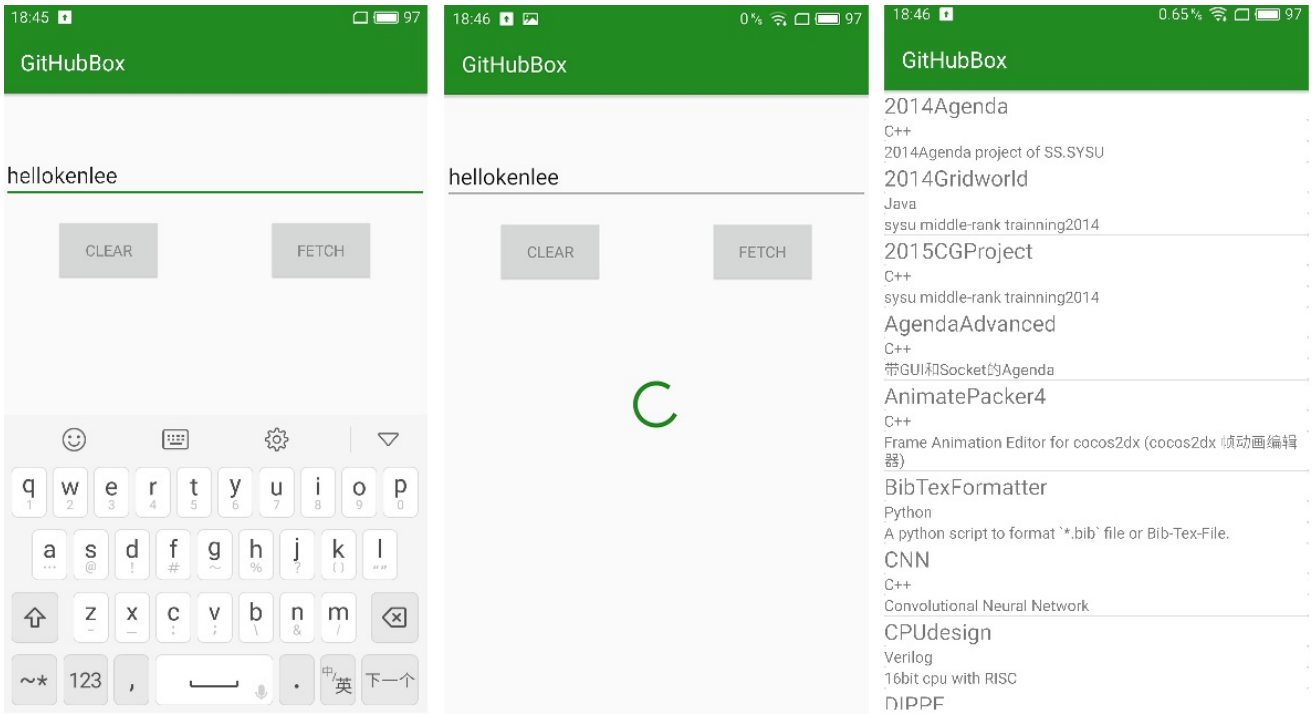
- 复习同步异步概念

实现方式要求:

- 对于 User Model, 显示 id, login, blog
- 对于 Repository Model, 显示 name, description, language

三、课堂实验结果

实验截图



实验步骤以及关键代码

本次实验需要使用网络请求的函数，从特定的接口获得查询用户的信息，并从中提取我们需要的信息，将其显示在我们当前的界面上。而在加载的过程中，则需要使用ProgressBar表明当前的加载进展，提醒用户耐心等待。

`ProgressBar`的使用

`ProgressBar`是一个显示控件，在本次实验中只需要控制其出现和隐藏，就可以很好地完成当前的UI设计。

`ProgressBar`使用前需要先在布局文件中声明:

```
<ProgressBar
        android:id="@+id/progress_bar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

并在调用其控件的位置使用函数设置其可见性:

```
        progressBar.setVisibility(View.INVISIBLE);
```

`OKHttpClient`与`Retrofit`

笔者使用`retrofit`对`OkHttpClient`进行了封装，封装为了类`ServiceFactory`，以方便调用：

```
public class ServiceFactory {
    private static OkHttpClient createOkHttp(){}
    public static GithubService getApi(String baseUrl){
         createOkHttp();
         getRetrofit(baseUrl);
        /*...*/
    }
    private static Retrofit getRetrofit(String baseUrl){}
}
```

函数`createOkHttp`与函数`getRetrofit`都是在当前的文件中使用的，分别用于创建`OkHttpClient`的对象以及`Retrofit`的实例化对象。实例化的对象将会用于网络消息的请求，并根据网络中返回的信息，返回一定的结果。

函数`getApi`则是真正可以在外部调用的接口，在使用前需要调用另外两个函数申请网络消息，并对消息进行处理，返回一个`GithubService`的值——这个类是笔者自己定义的。

`OkHttpClient`

`OkHttpClient`是真正实现用户访问网络的接口以及获取信息的部分。在使用前需要获取其实例化对象，这里创建一个接口，用于返回创建的实例化：

```
    public final static int CONNECT_TIMEOUT = 10;
    public final static int READ_TIMEOUT = 30;
    public final static int WRITE_TIMEOUT = 10;

    private static OkHttpClient createOkHttp(){
        return new OkHttpClient.Builder()
                .retryOnConnectionFailure(true)
                .connectTimeout(CONNECT_TIMEOUT, TimeUnit.SECONDS)
                .readTimeout(READ_TIMEOUT, TimeUnit.SECONDS)
                .writeTimeout(WRITE_TIMEOUT, TimeUnit.SECONDS)
                .build();
    }
```

该组件实现网络请求的时候，需要为其设置一些参数，保证使用者的用户体验。

`connectTimeout`：连接超时的限定，如果超时，则网络请求失败，会回调对应的函数；

`readTimeout`：读取超时的限定；

`writeTimeout`：写入超时的限定；

`Retrofit`

可以使用该接口为Retrofit添加客户端的申请功能，所以说，Retrofit更像是对于`OkHttpClient`的封装，是在使用其实现网络请求的内容：

```
OkHttpClient okHttpClient = null;
Retrofit retrofit = null;
private static Retrofit getRetrofit(String baseUrl){

        if(okHttpClient == null)
```

```
            okHttpClient = createOkHttp();

        retrofit = new Retrofit.Builder()
                .baseUrl(baseUrl)
                .addConverterFactory(GsonConverterFactory.create())
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
                .client(okHttpClient)
                .build();

        return retrofit;
    }
```

既然Retrofit是对于`OkHttpClient`的封装，则Retrofit就需要将客户端的部分传入`OkHttpClient`作为参数。

此外还有一些其他的参数需要设置，以保证Retrofit的正常运作：

`baseUrl`：基础的`Url`信息，之后获取的用户信息，仓库信息都需要从该网址中获得，直接作用在接口`getApi`中。

`addConverterFactory`：添加解码器，由于从网络接口获得的信息并不是直接可用的字符串，所以需要使用该函数将`JSon`类型的信息转化为可用的字符串信息。

`addCallAdapterFactory`：添加回调的适配器信息，使用`retrofit`请求，有可能会失败，在失败之后，应该有一个返回值，作为失败的通知，以便系统或者用户采取下一步的操作。所以需要添加回调函数的适配器。

API接口的设置

在笔者封装`OkHttpClient`与`Retrofit`的类`ServiceFactory`中，有一个外部可以调用的接口`getApi`，返回一个`GithubService`的实例化对象。

`GithubService`

对于类`GithubService`，该类是一个处理`Url`的接口（可以看到在声明中使用了关键词`Interface`）：

```
public interface GithubService {
    @GET("/users/{user}")
    Observable<Github> getUser(@Path("user") String user);
    @GET("/users/{user}/repos")
    Observable<List<Repos>> getRepos(@Path("user") String user);
}
```

之前有传入的`Url`网络地址信息可以在这个接口中进行补全：

```
    GithubService github = null;
    public static GithubService getApi(String baseUrl){
        github = getRetrofit(baseUrl).create(GithubService.class);
        return github;
    }
```

由于函数`getRetrofit`返回一个`Retrofit`类型的实体化变量，对于`Retrofit`的实体化变量是可以使用`create`创建一个接口的实例化变量的。这里就使用了`GithubService`作为其创建的接口实例化类。

此时两个类实现了拼接：

1. `Retrofit`获得了`baseUrl`的地址，并创建了`GithubService`的实例化对象；
2. `GithubService`所以就继承了来自于`Retrofit`的`Url`信息；
3. 在内部使用`@GET`实现了`Url`的补全，并通过已经封装好的类`Retrofit`发送网络请求；
4. 网络请求的反馈则使用下面的接口`getUser`或者`getRepos`获取返回值；
5. 返回值为接口之前的`Observable<Github>`或者`Observable<List<Repos>>`；
6. 由于`Github`和`Repos`都是笔者自定义实现的类，可以使用其内部接口获得对应的字符串。

`@GET`与`@Path`

`@GET`用于补全来自于`Retrofit`的`Url`，其中的`/`之后的内容放在`Url`后面真是毫无违和感。但是在其参数中出现的`{}`让人难以理解。

是这样的：

```
    @GET("/users/{user}")
    Observable<Github> getUser(@Path("user") String user);
```

在下面的接口中，函数`getUser`是有参数传入的。这个参数就是`String`类型的`user`。这个参数将会根据`@Path`中字符串的指引，找到`Url`中缺失的位置——正是`{}`标注出的位置。并使用该参数替代整个大括号标注的内容，此时的`Url`就是笔者期望的`Url`了。

`CardView`的使用

直接在布局文件中声明了，当做其他控件一样，使用`findViewById`获得对应的控件信息，感觉没有什么不一样的：

```
    <android.support.v7.widget.CardView
        android:layout_weight="1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/title2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </android.support.v7.widget.CardView>
```

实验遇到困难以及解决思路

1. 没有找到关于Retrofit的信息：

   先按照课件上的代码实现一部分，有了轮廓之后再根据一些博客的记载将内部的各个部分串联起来。

2. 不知道网络服务的API接口作用；

   困境现象：

   - 在`Retrofit`的服务中对接口进行了封装；
   - 接口的实现不能使用Call，而需要使用Observable
     - 接口定义信息`Type 'java.util.Observable' does not have type parameter`

       一开始猜测是版本问题，在当前版本下，Observable是不需要参数的；

       后来经过询问得知，引入的`import`不对，应该是`import rx.Observable;`

通过询问得知使用的结构。

3. 对回传的`json`数据解码：

   解码类使用的系统自带的解码类，已经不必自定义实现解码类了。

4. adapter的数据应该如何导入

   使用convert函数，用于指定某一行的信息同步修改。

   但是在实现的过程中，如果使用`convert`函数（该函数是抽象的，需要在特定的文件下特定地使用，因为不同的文件可能使用不同的数据结构），则该类必须要声明为`abstract`，否则就不能使用该函数。

   因为convert函数是在adapter中实现的，而且一定要在adapter中实现，是因为需要对某一行的数据进行绑定修改，只有在adapter内部才可以完成。如果convert函数不是abstract的，则需要知道当前的主函数文件使用的类——这种实现方法，实现后的adapter复用能力很低。

5. adapter 添加项，界面没有更新

   `RecyclerView`没有设置显示的管理系统，应该设置Manager；

6. 自定义adapter设置监听器

   需要先添加包含OnClick和onLongClick的函数的接口，该接口其实就是一个监听器，独立的监听器。

7. adapter刷新会闪退

   删除for循环则可以使用，针对一个条目可以使用，多个条目更新之后一起刷新则会闪退； 循环内部分节奏刷新也会闪退； 逐行排查，发现系统无法分辨null与空字符串，导致闪退。

四、课后实验结果

使用RecyclerView为各个项目之间添加了分割线。

五、实验思考及感想

有时候blog上的代码无法分清是自定义的还是必须使用的；无法区分使用的组件的结构（好像这么多次实验了，每一次实验都无法区分），可能是因为忽略掉了一些信息：

- 有一些函数的调用。

  如果是在两份代码中有不同的信息参量的传入，信息参量的传入就可以表明各个结构之间的关系。而对于需要重新实现的部分，应该会有重新实现的代码。

  同理，各个结构之间的关系，就是在其实例化调用的接口上展示出来的。

- 例子分析至关重要。

  有时候一些bug出现的时候，是在一系列操作中出现的，此时找出bug显得尤为困难，那么尝试回归之前已经实现过的比较简单的单纯的代码中进行查错。

  排查有时候确实显得很笨拙，但是往往非常有效。