

Retrofit笔记

字数3159 阅读2220 评论5 喜欢21

Retrofit是什么

来自Retrofit官网的介绍：

Type-safe HTTP client for Android and Java by Square, Inc.
For more information please see [the website](#)

因为其简单与出色的性能,Retrofit 是安卓上最流行的HTTP Client库之一。

不过它的缺点是在Retrofit 1.x中没有直接取消正在进行中任务的方法。如果你想做这件事必须手动杀死,而这并不好实现。

Square几年前曾许诺这个功能将在Retrofit 2.0实现,但是几年过去了仍然没有在这个问题上有所更新。

直到不久前,Retrofit 2.0才从候选发布阶段变成Beta 1,并且公开给所有人(现在最新版本是beta 2版)。在尝试了之后,我不得不说自己对新的模式和新的功能印象深刻。有许多改进,本文将讨论它们。让我们开始吧！

导包

```
compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
```

上面那个包是目前最新的,不建议直接使用不要偷懒,还是去[github官网](#)看看吧。

Retrofit怎么用

虽然Retrofit官网已经说明了,我还是要按照我的思路说一下它的使用方法

比如你要请求这么一个api：

<https://api.github.com/repos/{owner}/{repo}/contributors>

查看github上某个repo的contributors,首先你要这样建一个接口：

```
public interface GitHub {  
    @GET("/repos/{owner}/{repo}/contributors")  
    Call<List<Contributor>> contributors(  
        @Path("owner") String owner,  
        @Path("repo") String repo);  
}
```

然后你还需要创建一个 Retrofit 对象：

```
public static final String API_URL = "https://api.github.com";  
// Create a very simple REST adapter which points the GitHub API.  
  
Retrofit retrofit = new Retrofit.Builder()  
    .baseUrl(API_URL)  
    .addConverterFactory(GsonConverterFactory.create())  
    .build();
```

再用这个 Retrofit 对象创建一个 GitHub 对象：

```
// Create an instance of our GitHub API interface.  
GitHub github = retrofit.create(GitHub.class);  
// Create a call instance for looking up Retrofit contributors.  
Call<List<Contributor>> call = github.contributors("square", "retrofit");
```

最后你就可以用这个Github对象获得数据了：

```
// Fetch and print a list of the contributors to the library.  
call.enqueue(new Callback<List<Contributor>>() {  
    @Override  
    public void onResponse(Response<List<Contributor>> response) {  
        for (Contributor contributor : response.body()) {  
            System.out.println(contributor.login + " (" + contributor.contributions + ")");  
        }  
    }  
    @Override  
    public void onFailure(Throwable t) {  
    }  
});
```

你还可以移除一个请求：

Retrofit 1.x版本没有直接取消正在进行中任务的方法的。在2.x的版本中，Service 的模式变成 Call的形式的原因是为了让正在进行的事务可以被取消。要做到这点，你只需调用call.cancel()。

```
call.cancel();
```

事务将会在之后立即被取消。

总而言之，Retrofit使用方式看上去和Volley的方式完全不一样，使用Volley时你必须先创建一个 Request 对象，包括这个请求的Method，Url，Url的参数，以及一个请求成功和失败的Listener，然后把这个请求放到 RequestQueue 中，最后NetworkDispatcher会请求服务器获得数据。而 Retrofit 只要创建一个接口就可以了，是不是太不可思议了！！

其实万变不离其宗，这两种方式本质上是一样的，只是这个框架**描述HTTP请求的方式不一样而已**。因此，**你可以发现上面的 Github 接口其实就是 Retrofit 对一个HTTP请求的描述**。

以上就是Retrofit的基本用法，下面就来讲讲它的详细用法。

URL的定义方式

Retrofit 2.0使用了新的URL定义方式。Base URL与@Url 不是简单的组合在一起而是和的处理方式一致。用下面的几个例子阐明。

```
public interface APIService {
    @POST("user")
    Call<User> login();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.demo.come/base/home")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
}

//最后的url是http://api.demo.come/base/user
```

```
public interface APIService {
    @POST("user")
    Call<User> login();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.demo.come/base/home/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
}

//最后的url是http://api.demo.come/base/home/user
```

```
public interface ApiService {
    @POST("/user")
    Call<User> login();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.demo.come/base/home/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
}

//最后的url是http://api.demo.come/user
```

不知道大家看懂了没有，没看懂的话多看几遍吧，注意baseurl和注解中的反斜杠的变化

ps：貌似第二个才符合习惯。

对于 Retrofit 2.0中新的URL定义方式，这里是我的建议：

- Base URL: 总是以 / 结尾
- @Url: 不要以 / 开头

比如:

```
public interface ApiService {
    @POST("user/list")
    Call<User> login();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.demo.come/base/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
}

//最后的url是http://api.demo.come/base/user/list
```

而且在Retrofit 2.0中我们还可以在@Url里面定义完整的URL：

```
public interface ApiService {
    @POST("http://api.demo.come/base/user/list")
    Call<User> login();
}
```

这种情况下Base URL会被忽略。

解析

在Retrofit 2.0中，Converter 不再包含在package 中了。你需要自己插入一个Converter 不然的话Retrofit 只能接收字符串结果。同样的，Retrofit 2.0也不再依赖于Gson 。

如果你想接收json 结果并解析成DAO，你必须把Gson Converter 作为一个独立的依赖添加进来。

```
compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
```

然后使用addConverterFactory把它添加进来。注意RestAdapter的别名仍然为Retrofit。

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://api.demo.come/base/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();
service = retrofit.create(APIService.class);
```

这里是Square提供的官方Converter modules列表。选择一个最满足你需求的。

Gson: com.squareup.retrofit:converter-gson
Jackson: com.squareup.retrofit:converter-jackson
Moshi: com.squareup.retrofit:converter-moshi
Protobuf: com.squareup.retrofit:converter-protobuf
Wire: com.squareup.retrofit:converter-wire
Simple XML: com.squareup.retrofit:converter-simplexml

你也可以通过实现Converter.Factory接口来创建一个自定义的converter 。

我比较赞同这种新的模式。它让Retrofit对自己要做的事情看起来更清晰。

自定义Gson对象

为了以防你需要调整json里面的一些格式，比如，Date Format。你可以创建一个Gson 对象并把它传递给GsonConverterFactory.create()。

```
Gson gson = new GsonBuilder()
    .setDateFormat("yyyy-MM-dd'T'HH:mm:ssZ")
    .create();

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://api.demo.come/base/")
    .addConverterFactory(GsonConverterFactory.create(gson))
    .build();
```

```
service = retrofit.create(APIService.class);
```

完成。

现在需要OkHttp的支持

在Retrofit 2.0中，OkHttp 是必须的，并且自动设置为了依赖。下面的代码是从Retrofit 2.0的pom文件中抓取的。你不需要再做任何事情了。

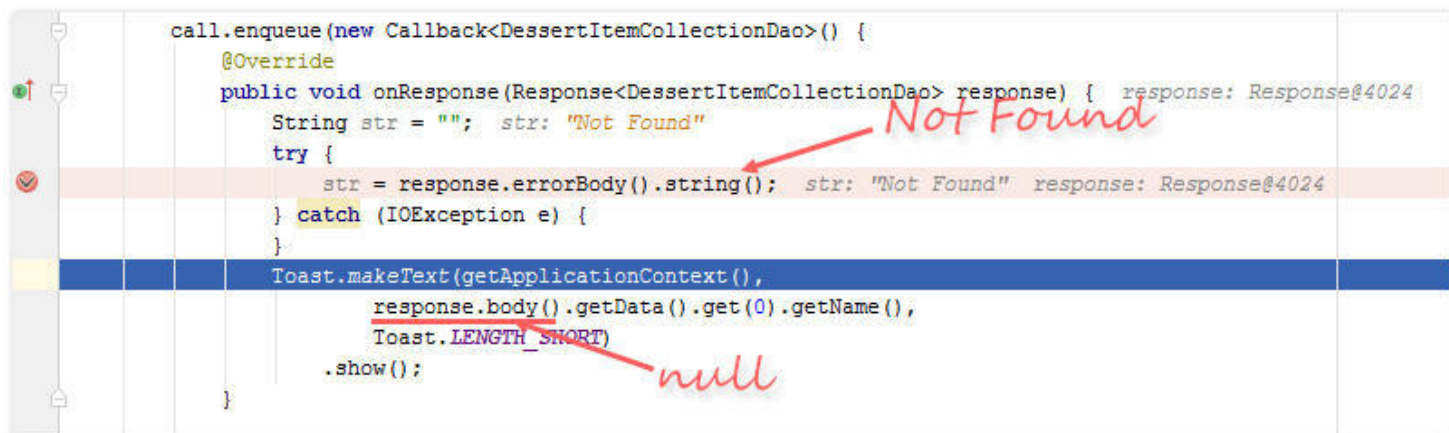
```
<dependencies>
<dependency>
<groupId>com.squareup.okhttp</groupId>
<artifactId>okhttp</artifactId>
</dependency>
...
</dependencies>
```

为了让OkHttp 的Call模式成为可能，在Retrofit 2.0中OkHttp 自动被用作HTTP 接口。

即使response存在问题onResponse依然被调用

在Retrofit 1.9中，如果获取的 response 不能解析成定义好的对象，则会调用failure。但是在 Retrofit 2.0中，不管 response 是否能被解析。onResponse总是会被调用。但是在结果不能解析的情况下，response.body()会返回null。别忘了处理这种情况。

如果 response 存在什么问题，比如 404什么的，onResponse 也会被调用。你可以从 response.errorBody().string()中获取错误信息的主体。



```
call.enqueue(new Callback<DessertItemCollectionDao>() {
    @Override
    public void onResponse(Response<DessertItemCollectionDao> response) { response: Response@4024
        String str = ""; str: "Not Found"
        try {
            str = response.errorBody().string(); str: "Not Found" response: Response@4024
        } catch (IOException e) {
        }
        Toast.makeText(getApplicationContext(),
            response.body().getData().get(0).getName(),
            Toast.LENGTH_SHORT)
            .show();
    }
})
```

缺少INTERNET权限会导致SecurityException异常

在Retrofit 2.0中，当你调用call.enqueue或者call.execute，将立即抛出SecurityException，如果你不使用try-catch会导致崩溃。

```
java.lang.SecurityException: Permission denied (missing INTERNET permission?)
    at java.net.InetAddress.lookupHostByName (InetAddress.java:451)
    at java.net.InetAddress.getAllByNameImpl (InetAddress.java:252)
    at java.net.InetAddress.getAllByName (InetAddress.java:215)
    at com.squareup.okhttp.internal.Network$1.resolveInetAddresses (Network.java:29)
    at com.squareup.okhttp.internal.http.RouteSelector.resetNextInetSocketAddress (RouteSelector.java:187)
    at com.squareup.okhttp.internal.http.RouteSelector.nextProxy (RouteSelector.java:156)
    at com.squareup.okhttp.internal.http.RouteSelector.next (RouteSelector.java:98)
    at com.squareup.okhttp.internal.http.HttpEngine.createNextConnection (HttpEngine.java:344)
    at com.squareup.okhttp.internal.http.HttpEngine.connect (HttpEngine.java:327)
    at com.squareup.okhttp.internal.http.HttpEngine.sendRequest (HttpEngine.java:245)
    at com.squareup.okhttp.Call.getResponse (Call.java:267)
    at com.squareup.okhttp.Call$ApplicationInterceptorChain.proceed (Call.java:224)
    at com.squareup.okhttp.Call.getResponseWithInterceptorChain (Call.java:195)
    at com.squareup.okhttp.Call.access$100 (Call.java:34)
    at com.squareup.okhttp.Call$AsyncCall.execute (Call.java:162)
    at com.squareup.okhttp.internal.NamedRunnable.run (NamedRunnable.java:33)
    at java.util.concurrent.ThreadPoolExecutor.runWorker (ThreadPoolExecutor.java:1112)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run (ThreadPoolExecutor.java:587)
    at java.lang.Thread.run (Thread.java:818)
```

1.png

这类似于在手动调用HttpURLConnection时候的行为。不过这不是什么大问题，因为当INTERNET权限添加到了AndroidManifest.xml中就没有什么需要考虑的了。

Use an Interceptor from OkHttp

我们必须使用OkHttp里面的Interceptor。首先你需要实用Interceptor创建一个OkHttpClient对象，如下：

```
OkHttpClient client = new OkHttpClient();
client.interceptors().add(new Interceptor() {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Response response = chain.proceed(chain.request());
        // Do anything with response here
        return response;
    }
});
```

然后传递创建的client到Retrofit的Builder链中。

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://api.demo.come/base/")
```

```
.addConverterFactory(GsonConverterFactory.create())
.client(client)
.build();
```

以上为全部内容。

学习关于OkHttp Interceptor的知识，请到[OkHttp Interceptors](#)。

终于到放大招的时候啦，RxJava支持

RxJava 是笔者当前最喜欢的一种编程方式要学习的可以去[这里](#)。

RxJava Integration with CallAdapter

除了使用Call模式来定义接口，我们也可以定义自己的type，比如MyCall。。我们把Retrofit 2.0的这个机制称为CallAdapter。

Retrofit团队有已经准备好了的CallAdapter module。其中最著名的module可能是为RxJava准备的CallAdapter，它将作为Observable返回。要使用它，你的项目依赖中必须包含两个modules。

```
compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2'
compile 'io.reactivex:rxandroid:1.1.0'
```

Sync Gradle并在Retrofit Builder链表中如下调用addCallAdapterFactory：

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://api.demo.come/base/")
    .addConverterFactory(GsonConverterFactory.create())
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
    .build();
```

你的Service接口现在可以作为Observable返回了！

你可以完全像RxJava那样使用它，如果你想让subscribe部分的代码在主线程被调用，需要把observeOn(AndroidSchedulers.mainThread())添加到链表中。

```
Observable<DessertItemCollectionDao> observable = service.loadDessertListRx();

observable.observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Subscriber<DessertItemCollectionDao>() {
        @Override
        public void onComplete() {
            Toast.makeText(getApplicationContext(),
                "Completed",
```



```

        Toast.LENGTH_SHORT)
        .show();
    }

    @Override
    public void onError(Throwable e) {
        Toast.makeText(getApplicationContext(),
            e.getMessage(),
            Toast.LENGTH_SHORT)
            .show();
    }

    @Override
    public void onNext(DessertItemCollectionDao dessertItemCollectionDao) {
        Toast.makeText(getApplicationContext(),
            dessertItemCollectionDao.getData().get(0).getName(),
            Toast.LENGTH_SHORT)
            .show();
    }
});

```

完成！我相信RxJava的粉丝对这个变化相当满意。

Retrofit的原理

Volley描述一个HTTP请求是需要创建一个 `Request` 对象，而执行这个请求呢，就是把这个请求对象放到一个队列中，让网络线程去处理。

Retrofit是怎么做的呢？答案就是Java的**动态代理**

动态代理

当开始看Retrofit的代码，我对下面这句代码感到很困惑：

```
// Create an instance of our GitHub API interface.GitHub github = retrofit.create(GitHub.class);
```

我给Retrofit对象传了一个 `GitHub` 接口的Class对象，怎么又返回一个 `GitHub` 对象呢？进入 `create` 方法一看，没几行代码，但是我觉得这几行代码就是Retrofit的精妙的地方：

```

/** Create an implementation of the API defined by the {@code service} interface. */
@SuppressWarnings("unchecked") // Single-interface proxy creation guarded by parameter safety.
public <T> T create(final Class<T> service) {
    Utils.validateServiceInterface(service);
    if (validateEagerly) {
        eagerlyValidateMethods(service);
    }
}

```

```
    }

    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },
        new InvocationHandler() {
            private final Platform platform = Platform.get();

            @Override
            public Object invoke(Object proxy, Method method, Object... args) throws Throwable {
                // If the method is a method from Object then defer to normal invocation.
                if (method.getDeclaringClass() == Object.class) {
                    return method.invoke(this, args);
                }
                if (platform.isDefaultMethod(method)) {
                    return platform.invokeDefaultMethod(method, service, proxy, args);
                }
                return loadMethodHandler(method).invoke(args);
            }
        });
}
```

看，create方法重要就是返回了一个动态代理对象。那么问题来了...

动态代理是个什么东西？

看Retrofit代码之前我知道Java动态代理是一个很重要的东西，比如在Spring框架里大量的用到，但是它有什么用呢？

Java动态代理就是Java开发给了开发人员一种可能：当你要调用某个类的方法前，插入你想要执行的代码

比如你要执行某个操作前，你必须要判断这个用户是否登录，或者你在付款前，你需要判断这个人的账户中存在这么多钱。这么简单的一句话，我相信可以把一个不懂技术的人也讲明白Java动态代理是什么东西了。

为什么要使用动态代理

你看上面代码，获取数据的代码就是这句：

```
// Create a call instance for looking up Retrofit contributors.
Call<List<Contributor>> call = github.contributors("square", "retrofit");
```

上面github对象其实是一个动态代理对象，并不是一个真正的Github接口的implements对象，当github对象调用contributors方法时，执行的是动态代理方法（你debug一下就知道了）

此时，动态代理发挥了它的作用，你看上去是调用了contributors方法，其实此时Retrofit把Github接口翻译成一个HTTP请求，也就是Retrofit中的MethodHandler对象，这个对象中包含了：

- OkHttpClient：发送网络请求的工具
- RequestFactory：类似于Volley中的Request，包含了HTTP请求的Url、Header信息，MediaType、Method以及RequestAction数组
- CallAdapter：HTTP请求返回数据的类型
- Converter：数据转换器

简单来说，Retrofit就是在你调用 `Call<List<Contributor>> call = github.contributors("square", "retrofit");` 后为你生成了一个Http请求，然后，你调用 `call.enqueue` 方法时就发送了这个请求，然后你就可以处理Response的数据了，从原理上讲，就是这样的。如果要再往细节处说，就可以再说很多了