# Retrofit 网络框架

# Retrofit官网

https://square.github.io/retrofit/

# Retrofit简单使用

## 添加权限

要使用网络，先在AndroidManifest.xml添加使用网络权限

<!--网络权限-->  
<uses-permission android:name="android.permission.INTERNET" />

## 导入依赖

Retrofit是一个网络加载框架，底层由OKHttp封装的，也是出自square公司。

也因此，在build.gradle里导入依赖时，OKHttp也是要导入的。

//网络请求 Retrofit库(基于okHttp封装的网络请求库)  
implementation 'com.squareup.okhttp3:okhttp:4.1.0'  
implementation 'com.squareup.retrofit2:retrofit:2.0.2'

## 使用Retrofit编辑Service网络请求接口

编写Service网络请求接口(有参/无参各举例一个)，以请求wanandroid接口为例。

wanAndroid网站开放的API网址如下：

<https://www.wanandroid.com/blog/show/2>

https://www.wanandroid.com 作为baseurl，即所有接口都是以此为开头的。

注意Retrofit中规定baseurl以 / 结尾 ，所以使用

https://www.wanandroid.com/

首先写一个简单的GET无参数请求：

以获取首页banner为例：

1.2 首页banner

https://www.wanandroid.com/banner/json

方法：GET

参数：无

wanandroid接口具体编码如下：

public interface WanAndroidService {  
  
 */\*\*  
 \* 1.2首页banner  
 \* 原接口:https://www.wanandroid.com/banner/json  
 \* 方法：GET  
 \* 参数：无  
 \*/* @GET("banner/json")  
 Call<ResponseBody> getBannerData();  
  
}

@GET注解代表是GET请求方式，("")中的字符串即原接口去掉baseURL剩下的字符串。

原接口:https://www.wanandroid.com/banner/json

baseURL:https://www.wanandroid.com/

@GET(“banner/json”)

如果原接口有参数：

例如：

4.2 项目列表数据 某个项目类别下的文章

原接口:https://www.wanandroid.com/project/list/1/json?cid=294

方法：GET

参数：

cid 分类的id，上面项目分类接口 int类型

页码page：拼接在链接中，从1开始。 int类型

public interface WanAndroidService {  
  
 */\*\*  
 \* 1.2首页banner  
 \* 原接口:https://www.wanandroid.com/banner/json  
 \* 方法：GET  
 \* 参数：无  
 \*/* @GET("banner/json")  
 Call<ResponseBody> getBannerData();  
  
 */\*\*  
 \* 4.2 项目列表数据 某个项目类别下的文章  
 \* 原接口:https://www.wanandroid.com/project/list/1/json?cid=294  
 \* 方法：GET  
 \* 参数：  
 \* cid 分类的id，上面项目分类接口 int类型  
 \* 页码page：拼接在链接中，从1开始。 int类型  
 \*/* @GET("project/list/{page}/json")  
 Call<ResponseBody> getProjectArticle(@Path("page") int page, @Query("cid") int cid);  
  
}

@GET(“project/list/{page}/json”) 同理原接口去掉baseURL。

page作为参数使用{ }包含它，在方法中添加@Path注解，代表直接在路径中的参数，根据APi添加page作为int类型参数，完整写法为：@Path(“page”) int page

cid也是参数但是原接口含有？，对此Retrofit需使用@Query注解，此外需要在 @GET注解中去掉原接口？之后的内容：

原接口:https://www.wanandroid.com/project/list/1/json?cid=294

@GET(“project/list/{page}/json”)

使用其他wanandroid接口时，依葫芦画瓢添加到WanAndroidService接口中即可，这样就统一管理了baseurl为https://www.wanandroid.com/的所有接口。

## 使用GsonFormat创建接口返回json数据对应的Bean类

以之前提到的第一个接口为例：

/\*\*

\* 1.2首页banner

\* 原接口:https://www.wanandroid.com/banner/json

\* 方法：GET

\* 参数：无

\*/

使用浏览器或postman请求该接口：

咋一看也看不出它的结构，使用json格式化工具网站查看：

https://www.json.cn/

复制该json字符串到该网站中即可

GsonFormat是用于根据json字符串自动生成对应Bean类的插件

选择Code->Gnerate菜单，就可以看到GsonFormat，自动生成的BnnerBean。

## 编写网络请求工具类(单例模式)与请求网络

为什么要写工具类呢？

如果不写工具类直接请求网络，每次都需要写下面的代码：

Retrofit retrofit = new Retrofit.Builder()  
 .baseUrl("https://www.wanandroid.com/")  
 .build();  
WanAndroidService wanAndroidService = *retrofit*.create(WanAndroidService.class);

操作太繁琐，可以把上面的代码写到一个工具类中去。

*/\*\*  
 \* 网络请求工具类  
 \* 基于单例模式-使用饿汉式  
 \* 网络请求库:Retrofit  
 \* 服务文件：WanAndroidService.java  
 \* app中所有网络请求的baseURL:https://www.wanandroid.com/  
 \*/*public class HttpUtil {  
  
 String TAG = "HttpUtil";  
 private static HttpUtil *httpUtil* = new HttpUtil();  
 private static WanAndroidService *wanAndroidService*;  
 private static Retrofit *retrofit*;  
  
 private HttpUtil() {  
 //使用Log验证是否单例生效,app使用过程中HttpUtil只初始化一次即可");  
 *retrofit* = new Retrofit.Builder()  
 .baseUrl("https://www.wanandroid.com/")  
 .build();  
 *wanAndroidService* = *retrofit*.create(WanAndroidService.class);  
 }  
  
 public static HttpUtil getInstance() {  
 return *httpUtil*;  
 }  
  
 //对外提供 wanAndroidService服务的 get方法 HttpUtil.getWanAndroidService()即可  
 public static WanAndroidService getWanAndroidService() {  
 return *wanAndroidService*;  
 }  
}

初始化这个工具：

在APP的主活动初始化。 HttpUtil.getInstance(); 即可

HttpUtil.*getInstance*();

使用这个工具类

以第一个WanAndroidService接口的第一个方法为例：

Call<ResponseBody> call = HttpUtil.*getWanAndroidService*().getBannerData();  
Response<ResponseBody> response = call.execute();  
System.*out*.println("=========> response: " + response.body().toString());  
String strBack = doJson(response.body());  
System.*out*.println("请求成功, 返回的数据如下:\n" + doJson(response.body()));  
JSONObject jsonObject = new JSONObject(strBack);  
Gson gson = new Gson();  
BannerBean bannerBean = gson.fromJson(jsonObject.toString(), BannerBean.class);  
System.*out*.println("请求成功, 返回的数据如下:\n" + bannerBean.toString());

# Retrofit注解详解

上面提到Retrofit 共22个注解，这节就专门介绍这22个注解，为帮助大家更好理解我将这22个注解分为三类，并用表格的形式展现出来,表格上说得并不完整，具体的见源码上的例子注释。

retrofit通过使用注解来简化请求，大体分为以下几类：

用于标注请求方式的注解

用于标记请求头的注解

用于标记请求参数的注解

用于标记请求和响应格式的注解

## HTTP请求方法的注解

|  |  |  |  |
| --- | --- | --- | --- |
| 分类 | 名称 | 说明 | 备注 |
| 请求方法 | GET | get请求 | 分别对应HTTP的请求方法 |
| 请求方法 | POST | post请求 | 都接收一个字符串表示接口 |
| 请求方法 | PUT | put请求 | 与baseUrl组成完整Url |
| 请求方法 | DELETE | delete请求 | 不过也可以不指定结合下面的@Url注解使用 |
| 请求方法 | PATCH | patch请求，该请求是对put请求的补充，用于更新局部资源 | url中也可以使用变量如{id} |
| 请求方法 | HEAD | Header请求 | 并使用@Path(“id”)注解{id}提供值 |
| 请求方法 | OPTIONS | option请求 |  |
| 请求方法 | HTTP | 通过注解，可以替换以上所有的注解，其拥有三个属性:method, path, hasBody | 可以用于替换以上7个，以及其它扩展方法 |

表1 请求方法

以上表格中除了HTTP以外都对应了HTTP标准中的请求方法，而HTTP注解则可以代替以上方法中任意一个注解，有三个属性: method, path, hasBody, 下面是用HTTP注解实现的接口

*/\*\*  
 \* method 表示请求的方法，区分大小写  
 \* path表示路径  
 \* hasBody表示是否有请求体  
 \*/*@HTTP(method = "GET", path = "project/list/{page}/json", hasBody = false)  
Call<ResponseBody> getBannerData(@Path("page") int page, @Query("cid") int cid);

注: method 的值 retrofit 不会做处理，所以要自行保证其准确性，之前使用小写也可以是因为示例源码中的服务器不区分大小写。

## 请求头注解

|  |  |
| --- | --- |
| 注解 | 说明 |
| @Headers | 用于添加固定请求头，可以同时添加多个。通过该注解添加的请求头不会相互覆盖，而是共同存在 |
| @Header | 作为方法的参数传入，用于添加不固定值的Header，该注解会更新已有的请求头 |

## 请求和响应格式注解

|  |  |  |
| --- | --- | --- |
| 分类 | 名称 | 备注 |
| 表单请求 | @FormUrlEncoded | 表示请求发送编码Form表单数据，每个键值对需要使用@Field注解，你在网站上看到登录页面就是用这种请求方式Content-Type:application/x-www-form-urlencoded |
| 表单请求 | @Multipart | 表示请求体是一个支持文件上传的Form表单，你看到带文件上传的网页就是用的这种请求方式Content-Type:multipart/form-data |
| 标记 | Streaming | 表示响应体的数据用流的形式返回如果没有使用该注解，默认会把数据全部载入内存，之后你通过流获取数据也不过是读取内存中的数据所以如果你的返回数据比较大，你就需要使用这个注解 |

表2 标记类注解

## 请求参数注解

|  |  |  |
| --- | --- | --- |
| 分类 | 名称 | 备注 |
| \* | @Body | 多用于非表单请求体， 比如想要以post方式传递json格式数据 |
| \* | @Field | 用于post表单字段， Field和FieldMa需要FormUrlEncoded结合使用 |
| \* | @FieldMap | FiledMap和@Field作用一致，用于不确定表单参数 |
| \* | @Part | 用于表单字段， Part和PartMap与Multipart注解配合，适合有文件上传的情况，FieldMap的接受类型是Map<String, String>, 非String类型会调用其toString方法。 |
| \* | @PartMap | 用于表单字段，PartMap的默认接受类型是Map<String, RequestBody>， 非RequestBody类型会通过Convert转换，可用于实现多文件上传 |
| \* | @Path | 用于URL占位 Query和QueryMap与Field和FieldMap功能一样，不同的是Query和QueryMap的数据体现在Url上， 而Field和FieldMap的数据是请求体，但生成的数据形式是一样的。 |
| \* | @Query | 用于Get中指定参数 |
| \* | @QueryMap | 和Query使用类似 |
| \* | @Url | 指定请求路径 |

注1:{占位符}和PATH尽量只用在URL的path部分，url中的参数使用Query和QueryMap 代替，保证接口定义的简洁

注2: Query、Field和Part这三者都支持数组和实现了Iterable接口的类型，如List，Set等，方便向后台传递数组。

Call<ResponseBody> foo(@Query("ids[]") List<Integer> ids);  
//结果：ids[]=0&ids[]=1&ids[]=2

**用注解描述Http请求**

支持URL参数占位以及查询参数占位

请求体的对象转换(例如json以及协议buffers)

分片请求和文件上传

**API定义**

注解在接口方法上,参数声明了请求如何被处理

**请求方法**

每个方法必须有一个HTTP注解用于提供请求的方法和相对URL.有五种内置的注解请求方法:

GET，POST，PUT，DELETE，and HEAD。

相对URL路径在注解内部.

@GET("users/list")

你也可以将查询参数放到URL中

@GET("users/list?sort=desc")

**URL操作**

请求的URL可以通过占位块和接口方法的参数进行动态更新, 格式如下:

@GET("group/{id}/users")  
Call<List<User>> groupList(@Path("id") int groupId);

查询参数也可以放在接口方法中, 通过@Query注解实现

@GET("group/{id}/users")  
Call<List<User>> groupList(@Path("id") int groupId, @Query("sort") String sort);

对于复杂的查询参数可以用Map操作,代码格式如下:

@GET("group/{id}/users")  
Call<List<User>> groupList(@Path("id") int groupId, @QueryMap Map<String, String> options);

**请求Body**

一个对象可以作为一个HTTP的请求体,通过@Body注解实现

@POST("users/new")  
Call<User> createUser(@Body User user);

Retrofit实例中声明了转化器, 对象才可以被转化.如果没有添加转换器,只能用RequestBody

**格式化编码和分片操作**

请求接口也可以被定义为可以发送格式化编码和分片数据.

通过@FormUrlEncoded注解声明格式化编码的数据.每个键值对通过@Field注解声明.

@FormUrlEncoded  
@POST("user/edit")  
Call<User> updateUser(@Field("first\_name") String first, @Field("last\_name") String last);

分片请求通过注解@Multipart完成,示例代码如下:

@Multipart  
@PUT("user/photo")  
Call<User> updateUser(@Part("photo") RequestBody photo, @Part("description") RequestBody description);

分片的片数据可以用Reftrofit定义的转换器,也可以通过实现RequestBody处理自己实现序列化.

**头部操作**

可以通过注解@Heather来设置静态的头信息

@Headers("Cache-Control: max-age=640000")  
@GET("widget/list")  
Call<List<Widget>> widgetList();

相同头名称的字段不会相互覆盖,都会包含在头请求里, 头信息可以通过@Header注解动态更新, 头信息的值未null,则该头信息将被忽略,否则tostring被调用,然后用该函数转换后值.

@GET("user")  
Call<User> getUser(@Header("Authorization") String authorization)

同查询参数类似,复杂的头信息组合,可以用Map操作,示例代码如下:

@GET("user")  
Call<User> getUser(@HeaderMap Map<String, String> headers)

每个请求都需要的头信息,可以在OkHttp 的拦截器中处理.

**同步与异步**

Call实例可以以同步或者异步方式执行.每个实例只能被执行一次,但是可以通过clone()方法创建一个新的实例来使用.在Android系统中,回调方法在主线程中执行.在JVM中,回调方法将在与执行HTTP请求线程相同的线程中执行.

**Retrofit配置**

Retrofit可以被配置.

# 转换器/Gson与Converter

默认情况下,Retrofit只可以反序列化Http bodies成OkHttp的ResponseBody类型, 仅仅接收RequestBody类型作为@Body参数。这也是为什么我在前面的例子接口的返回值都是 Call，但如果响应体只是支持转换为ResponseBody的话何必要引入泛型呢，返回值直接用一个Call就行了嘛，既然支持泛型，那说明泛型参数可以是其它类型的，而Converter就是Retrofit为我们提供用于将ResponseBody转换为我们想要的类型，有了Converter之后我们就可以写把我们的第一个例子的接口写成这个样子了：

public interface BlogService {

@GET("blog/{id}")

Call<Result<Blog>> getBlog(@Path("id") int id);

}

1

2

3

4

5

当然只改变泛型的类型是不行的，我们在创建Retrofit时需要明确告知用于将ResponseBody转换我们泛型中的类型时需要使用的Converter

引入Gson支持:

compile 'com.squareup.retrofit2:converter-gson:2.0.2'

1

2

通过GsonConverterFactory为Retrofit添加Gson支持：

Gson gson = new GsonBuilder()

//配置你的Gson

.setDateFormat("yyyy-MM-dd hh:mm:ss")

.create();

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

//可以接收自定义的Gson，当然也可以不传

.addConverterFactory(GsonConverterFactory.create(gson))

.build();

1

2

3

4

5

6

7

8

9

10

11

这样Retrofit就会使用Gson将ResponseBody转换我们想要的类型。

这是时候我们终于可以演示如使创建一个Blog了！

@POST("blog")

Call<Result<Blog>> createBlog(@Body Blog blog);

1

2

3

被@Body注解的的Blog将会被Gson转换成RequestBody发送到服务器。

BlogService service = retrofit.create(BlogService.class);

Blog blog = new Blog();

blog.content = "新建的Blog";

blog.title = "测试";

blog.author = "怪盗kidou";

Call<Result<Blog>> call = service.createBlog(blog);

1

2

3

4

5

6

结果：

Result{code=200, msg='OK', data=Blog{id=20, date='2016-04-21 05:29:58', author='怪盗kidou', title='测试', content='新建的Blog'}, count=0, page=0}

1

下面以GsonConverterFactory类作为GitHubService的实现用于Gson的反序列化,示例代码如下:

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("https://api.github.com")

.addConverterFactory(GsonConverterFactory.create())

.build();

GitHubService service = retrofit.create(GitHubService.class);

1

2

3

4

5

6

定制转换器

如果你需要定制一个转化器用于解析诸如YAML,txt,自定义格式,或者你希望用一个不同的库来实现存在的格式,你可以通过集成Converter.Factory类来满足需求.

4 RxJava与CallAdapter

说到Retrofit就不得说到另一个火到不行的库RxJava，网上已经不少文章讲如何与Retrofit结合，但这里还是会有一个RxJava的例子，不过这里主要目的是介绍使用CallAdapter所带来的效果。

第3节介绍的Converter是对于Call中T的转换，而CallAdapter则可以对Call转换，这样的话Call中的Call也是可以被替换的，而返回值的类型就决定你后续的处理程序逻辑，同样Retrofit提供了多个CallAdapter，这里以RxJava的为例，用Observable代替Call：

引入RxJava支持:

compile 'com.squareup.retrofit2:adapter-rxjava:2.0.2'

// 针对rxjava2.x（adapter-rxjava2的版本要 >= 2.2.0）

compile 'com.squareup.retrofit2:adapter-rxjava2:2.3.0'

1

2

3

通过RxJavaCallAdapterFactory为Retrofit添加RxJava支持：

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(RxJavaCallAdapterFactory.create())

// 针对rxjava2.x

.addCallAdapterFactory(RxJava2CallAdapterFactory.create())

.build();

1

2

3

4

5

6

7

接口设计：

public interface BlogService {

@POST("/blog")

Observable<Result<List<Blog>>> getBlogs();

}

1

2

3

4

使用：

BlogService service = retrofit.create(BlogService.class);

service.getBlogs(1)

.subscribeOn(Schedulers.io())

.subscribe(new Subscriber<Result<List<Blog>>>() {

@Override

public void onCompleted() {

System.out.println("onCompleted");

}

@Override

public void onError(Throwable e) {

System.err.println("onError");

}

@Override

public void onNext(Result<List<Blog>> blogsResult) {

System.out.println(blogsResult);

}

});

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

结果：

Result{code=200, msg='OK', data=[Blog{id=1, date='2016-04-15 03:17:50', author='怪盗kidou', title='Retrofit2 测试1', content='这里是 Retrofit2 Demo 测试服务器1'},.....], count=20, page=1}

1

2

像上面的这种情况最后我们无法获取到返回的Header和响应码的，如果我们需要这两者，提供两种方案：

1、用

Observable<Response<T>>

1

代替

Observable<T>

1

,这里的Response指retrofit2.Response.

2、用

Observable<Result<T>>

1

代替

Observable<T>

1

，这里的Result是指retrofit2.adapter.rxjava.Result,这个Result中包含了Response的实例.

5 自定义Converter

本节的内容是教大家实现在一简易的Converter，这里以返回格式为Call为例。

在此之前先了解一下Converter接口及其作用：

public interface Converter<F, T> {

// 实现从 F(rom) 到 T(o)的转换

T convert(F value) throws IOException;

// 用于向Retrofit提供相应Converter的工厂

abstract class Factory {

// 这里创建从ResponseBody其它类型的Converter，如果不能处理返回null

// 主要用于对响应体的处理

public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[] annotations,

Retrofit retrofit) {

return null;

}

// 在这里创建 从自定类型到ResponseBody 的Converter,不能处理就返回null，

// 主要用于对Part、PartMap、Body注解的处理

public Converter<?, RequestBody> requestBodyConverter(Type type,

Annotation[] parameterAnnotations, Annotation[] methodAnnotations, Retrofit retrofit) {

return null;

}

// 这里用于对Field、FieldMap、Header、Path、Query、QueryMap注解的处理

// Retrfofit对于上面的几个注解默认使用的是调用toString方法

public Converter<?, String> stringConverter(Type type, Annotation[] annotations,

Retrofit retrofit) {

return null;

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

我们要想从Call 转换为 Call 那么对应的F和T则分别对应ResponseBody和String，我们定义一个StringConverter并实现Converter接口。

public static class StringConverter implements Converter<ResponseBody, String> {

public static final StringConverter INSTANCE = new StringConverter();

@Override

public String convert(ResponseBody value) throws IOException {

return value.string();

}

}

1

2

3

4

5

6

7

8

9

10

我们需要一个Fractory来向Retrofit注册StringConverter

public static class StringConverterFactory extends Converter.Factory {

public static final StringConverterFactory INSTANCE = new StringConverterFactory();

public static StringConverterFactory create() {

return INSTANCE;

}

// 我们只关实现从ResponseBody 到 String 的转换，所以其它方法可不覆盖

@Override

public Converter<ResponseBody, ?> responseBodyConverter(Type type, Annotation[] annotations, Retrofit retrofit) {

if (type == String.class) {

return StringConverter.INSTANCE;

}

//其它类型我们不处理，返回null就行

return null;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

使用Retrofit.Builder.addConverterFactory向Retrofit注册我们StringConverterFactory：

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

// 我们自定义的一定要放在Gson这类的Converter前面

.addConverterFactory(StringConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())

.build();

1

2

3

4

5

6

7

注: addConverterFactory是有先后顺序的，如果有多个ConverterFactory都支持同一种类型，那么就是只有第一个才会被使用，而GsonConverterFactory是不判断是否支持的，所以这里交换了顺序还会有一个异常抛出，原因是类型不匹配。

只要返回值类型的泛型参数就会由我们的StringConverter处理,不管是Call还是Observable

有没有很简单?如果你有其它的需求处理的就自己实现吧。

6 自定义CallAdapter

本节将介绍如何自定一个CallAdapter，并验证是否所有的String都会使用我们第5节中自定义的Converter。

先看一下CallAdapter接口定义及各方法的作用：

public interface CallAdapter<T> {

// 直正数据的类型 如Call<T> 中的 T

// 这个 T 会作为Converter.Factory.responseBodyConverter 的第一个参数

// 可以参照上面的自定义Converter

Type responseType();

<R> T adapt(Call<R> call);

// 用于向Retrofit提供CallAdapter的工厂类

abstract class Factory {

// 在这个方法中判断是否是我们支持的类型，returnType 即Call<Requestbody>和`Observable<Requestbody>`

// RxJavaCallAdapterFactory 就是判断returnType是不是Observable<?> 类型

// 不支持时返回null

public abstract CallAdapter<?> get(Type returnType, Annotation[] annotations,

Retrofit retrofit);

// 用于获取泛型的参数 如 Call<Requestbody> 中 Requestbody

protected static Type getParameterUpperBound(int index, ParameterizedType type) {

return Utils.getParameterUpperBound(index, type);

}

// 用于获取泛型的原始类型 如 Call<Requestbody> 中的 Call

// 上面的get方法需要使用该方法。

protected static Class<?> getRawType(Type type) {

return Utils.getRawType(type);

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

了解了CallAdapter的结构和其作用之后，我们就可以开始自定义我们的CallAdapter了，本节以CustomCall为例。

在此我们需要定义一个CustomCall，不过这里的CustomCall作为演示只是对Call的一个包装，并没有实际的用途。

public static class CustomCall<R> {

public final Call<R> call;

public CustomCall(Call<R> call) {

this.call = call;

}

public R get() throws IOException {

return call.execute().body();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

有了CustomCall，我们还需要一个CustomCallAdapter来实现 Call 到 CustomCall的转换，这里需要注意的是最后的泛型，是我们要返回的类型。

public static class CustomCallAdapter implements CallAdapter<CustomCall<?>> {

private final Type responseType;

// 下面的 responseType 方法需要数据的类型

CustomCallAdapter(Type responseType) {

this.responseType = responseType;

}

@Override

public Type responseType() {

return responseType;

}

@Override

public <R> CustomCall<R> adapt(Call<R> call) {

// 由 CustomCall 决定如何使用

return new CustomCall<>(call);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

提供一个CustomCallAdapterFactory用于向Retrofit提供CustomCallAdapter：

public static class CustomCallAdapterFactory extends CallAdapter.Factory {

public static final CustomCallAdapterFactory INSTANCE = new CustomCallAdapterFactory();

@Override

public CallAdapter<?> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {

// 获取原始类型

Class<?> rawType = getRawType(returnType);

// 返回值必须是CustomCall并且带有泛型

if (rawType == CustomCall.class && returnType instanceof ParameterizedType) {

Type callReturnType = getParameterUpperBound(0, (ParameterizedType) returnType);

return new CustomCallAdapter(callReturnType);

}

return null;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

使用addCallAdapterFactory向Retrofit注册CustomCallAdapterFactory

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://localhost:4567/")

.addConverterFactory(Example09.StringConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(CustomCallAdapterFactory.INSTANCE)

.build();

1

2

3

4

5

6

7

注： addCallAdapterFactory与addConverterFactory同理，也有先后顺序。

7 其它说明

7.1 Retrofit.Builder

前面用到了 Retrofit.Builder 中的baseUrl、addCallAdapterFactory、addConverterFactory、build方法，还有callbackExecutor、callFactory、client、validateEagerly这四个方法没有用到，这里简单的介绍一下。

方法 用途

callbackExecutor(Executor) 指定Call.enqueue时使用的Executor，所以该设置只对返回值为Call的方法有效

callFactory(Factory) 设置一个自定义的okhttp3.Call.Factory，那什么是Factory呢?OkHttpClient就实现了okhttp3.Call.Factory接口，下面的client(OkHttpClient)最终也是调用了该方法，也就是说两者不能共用

client(OkHttpClient) 设置自定义的OkHttpClient,以前的Retrofit版本中不同的Retrofit对象共用同OkHttpClient,在2.0各对象各自持有不同的OkHttpClient实例，所以当你需要共用OkHttpClient或需要自定义时则可以使用该方法，如：处理Cookie、使用stetho 调式等

validateEagerly(boolean) 是否在调用create(Class)时检测接口定义是否正确，而不是在调用方法才检测，适合在开发、测试时使用

7.2 Retrofit的Url组合规则

如果你在注解中提供的url是完整的url，则url将作为请求的url。

如果你在注解中提供的url是不完整的url，且不以 / 开头，则请求的url为baseUrl+注解中提供的值

如果你在注解中提供的url是不完整的url，且以 / 开头，则请求的url为baseUrl的主机部分+注解中提供的值

7.3 Retrofit中提供的Convert

Converter Gradle依赖

Gson com.squareup.retrofit2:converter-gson:2.0.2

Jackson com.squareup.retrofit2:converter-jackson:2.0.2

Moshi com.squareup.retrofit2:converter-moshi:2.0.2

Protobuf com.squareup.retrofit2:converter-protobuf:2.0.2

Wire com.squareup.retrofit2:converter-wire:2.0.2

Simple XML com.squareup.retrofit2:converter-simplexml:2.0.2

Scalars com.squareup.retrofit2:converter-scalars:2.0.2

7.4 Retrofit提供的CallAdapter：

CallAdapter Gradle依赖

guava com.squareup.retrofit2:adapter-guava:2.0.2

Java8 com.squareup.retrofit2:adapter-java8:2.0.2

rxjava com.squareup.retrofit2:adapter-rxjava:2.0.2

————————————————

版权声明：本文为CSDN博主「Calvin880828」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/u011897062/article/details/109643349

式样

微软雅黑 Light 五号

段落 固定值 23磅

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5