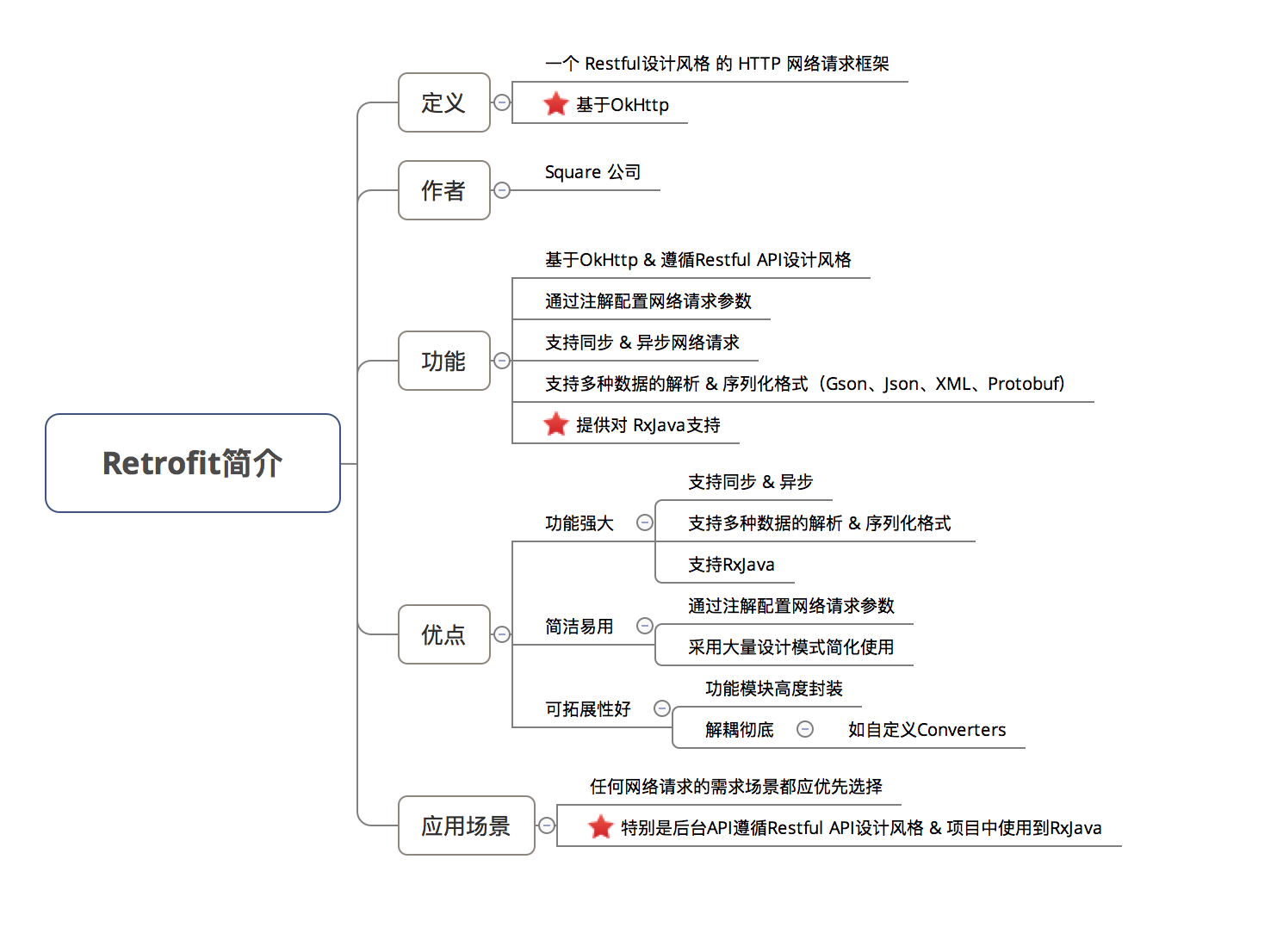
# 1、前言

在Andrroid开发中，网络请求十分常用。

而在Android网络请求库中，Retrofit是当下最热的一个网络请求库。

**（重点：retrofit的特点、 注解的使用、 使用步骤）**

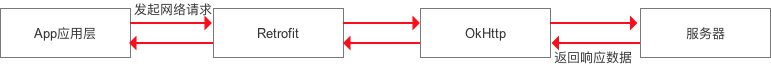
# 2、简介



**特别注意：**

准确来说，Retrofit 是一个 RESTful 的 HTTP 网络请求框架的封装。

原因：网络请求的工作本质上是 OkHttp 完成，而 Retrofit 仅负责 网络请求接口的封装。



App应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 OkHttp 完成后续的请求操作。

在服务端返回数据之后，OkHttp 将原始的结果交给 Retrofit，Retrofit根据用户的需求对结果进行解析。

# 3、与其他开源请求库对比

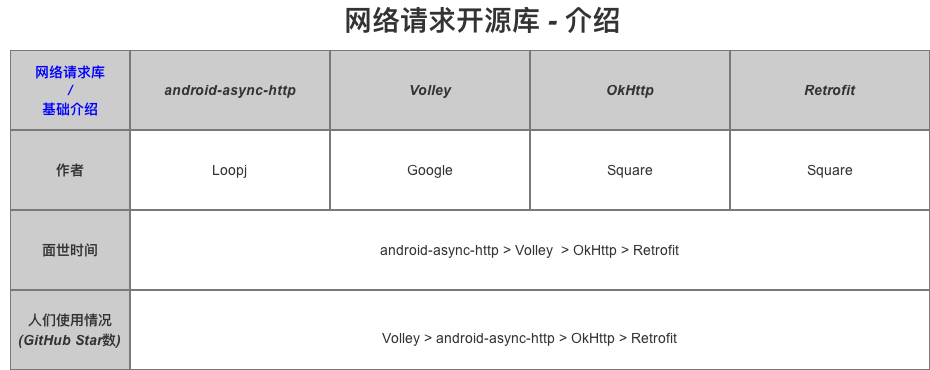
除了Retrofit，如今Android中主流的网络请求框架有：

Android-Async-Http

Volley

OkHttp

下面是简单介绍：



一图让你了解全部的网络请求库和他们之间的区别！



# 4、使用步骤介绍

使用 Retrofit 的步骤共有7个：

步骤1：添加Retrofit库的依赖

步骤2：创建 接收服务器返回数据 的类

步骤3：创建 用于描述网络请求 的接口

步骤4：创建 Retrofit 实例

步骤5：创建 网络请求接口实例 并 配置网络请求参数

步骤6：发送网络请求（异步 / 同步）

封装了 数据转换、线程切换的操作（Gson、RXJava）

步骤7：处理服务器返回的数据

接下来，我们一步步进行讲解。

步骤1：添加Retrofit库的依赖

1. 在 Gradle加入Retrofit库的依赖

build.gradle

dependencies {

compile 'com.squareup.retrofit2:retrofit:2.0.2'

// Retrofit库

}

2. 添加 网络权限

AndroidManifest.xml

<uses-permission android:name="android.permission.INTERNET"/>

步骤2：创建 接收服务器返回数据 的类

Reception.java

public class Reception {

...

// 根据返回数据的格式和数据解析方式（Json、XML等）定义

// 下面会在实例进行说明

}

步骤3：创建 用于描述网络请求 的接口

Retrofit将 Http请求 抽象成 Java接口，采用 注解 描述网络请求参数 和配置网络请求参数。

用 动态代理 动态 将该接口的注解“翻译”成一个 Http 请求，最后再执行 Http 请求。

注：接口中的每个方法的参数都需要使用注解标注，否则会报错。

GetRequest\_Interface.interface

public interface GetRequest\_Interface {

@GET("openapi.do?keyfrom=Yanzhikai&key=2032414398&type=data&doctype=json&version=1.1&q=car")

Call<Translation> getCall();

// @GET注解的作用:采用Get方法发送网络请求

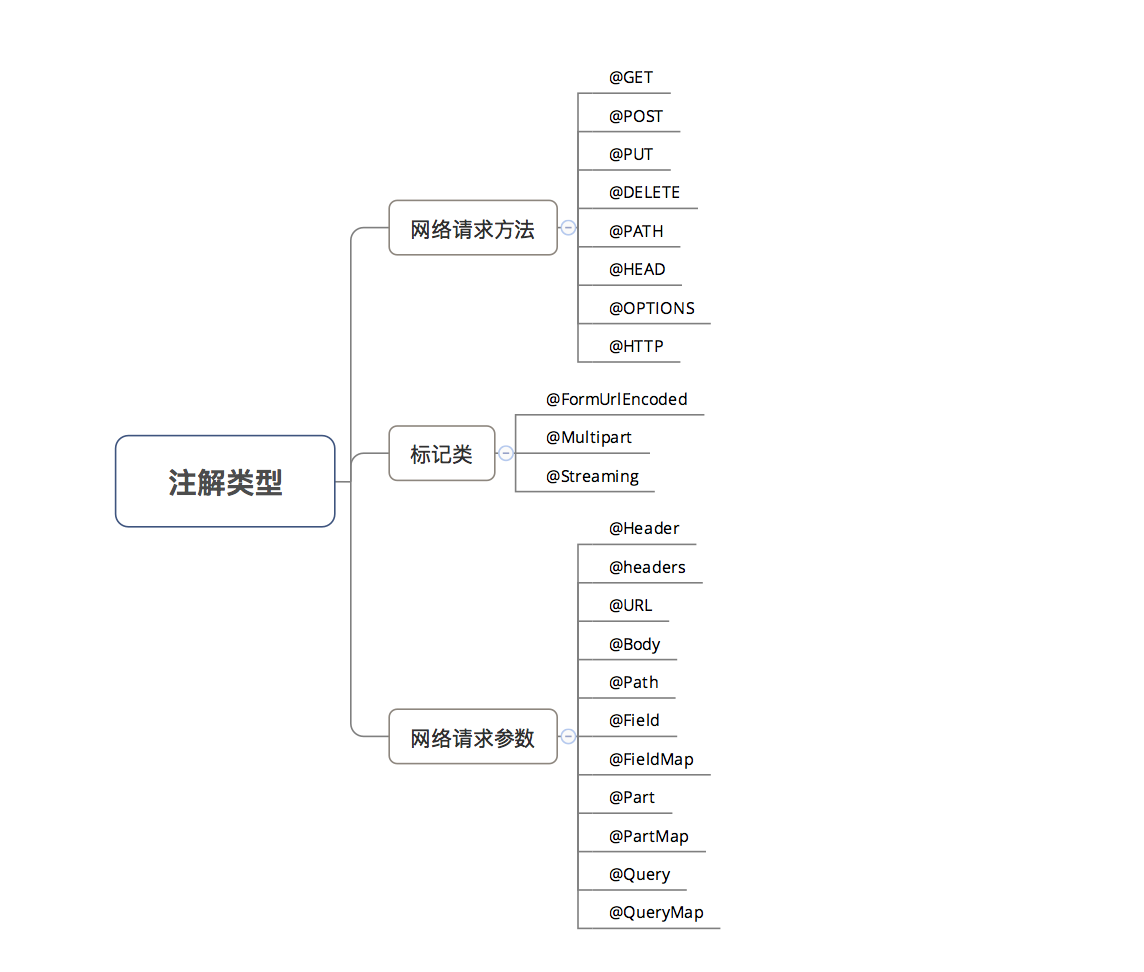
// getCall() = 接收网络请求数据的方法

// 其中返回类型为Call<\*>，\*是接收数据的类（即上面定义的Translation类）

// 如果想直接获得Responsebody中的内容，可以定义网络请求返回值为Call<ResponseBody>

}

下面详细介绍Retrofit 网络请求接口 的注解类型。



## 4.0注解说明

## 4.1第一类：网络请求方法



详细说明：

### 4.1.1 HTTP方法

HTTP请求方法有8种，分别是GET、POST、DELETE、PUT、HEAD、TRACE、CONNECT 、OPTIONS。其中PUT、DELETE、POST、GET分别对应着增删改查，对于移动开发最常用的就是POST和GET了。

GET：请求获取Request-URI所标识的资源

POST：在Request-URI所标识的资源后附加新的数据

HEAD：请求获取由Request-URI所标识的资源的响应消息报头

PUT： 请求服务器存储一个资源，并用Request-URI作为其标识

DELETE ：请求服务器删除Request-URI所标识的资源

TRACE ： 请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT： HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器

OPTIONS ：请求查询服务器的性能，或者查询与资源相关的选项和需求

### 4.1.2 @GET、@POST、@PUT、@DELETE、@HEAD

以上方法分别对应 HTTP中的网络请求方式

public interface GetRequest\_Interface {

@GET("openapi.do?keyfrom=Yanzhikai&key=2032414398&type=data&doctype=json&version=1.1&q=car")

Call<Translation> getCall();

// @GET注解的作用:采用Get方法发送网络请求

// getCall() = 接收网络请求数据的方法

// 其中返回类型为Call<\*>，\*是接收数据的类（即上面定义的Translation类）

}

此处特意说明URL的组成：Retrofit把 网络请求的URL 分成了两部分设置：

// 第1部分：在网络请求接口的注解设置

@GET("openapi.do?keyfrom=Yanzhikai&key=2032414398&type=data&doctype=json&version=1.1&q=car")

Call<Translation> getCall();

// 第2部分：在创建Retrofit实例时通过.baseUrl()设置

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://fanyi.youdao.com/") //设置网络请求的Url地址

.addConverterFactory(GsonConverterFactory.create()) //设置数据解析器

.build();

// 从上面看出：一个请求的URL可以通过 替换块 和 请求方法的参数 来进行动态的URL更新。

// 替换块是由 被{}包裹起来的字符串构成

// 即：Retrofit支持动态改变网络请求根目录

网络请求的完整 Url =在创建Retrofit实例时通过.baseUrl()设置 +网络请求接口的注解设置（下面称 “path“ ）

具体整合的规则如下：



建议采用第三种方式来配置，并尽量使用同一种路径形式。

### 4.1.2 @HTTP

作用：替换@GET、@POST、@PUT、@DELETE、@HEAD注解的作用 及 更多功能拓展

具体使用：通过属性method、path、hasBody进行设置

public interface GetRequest\_Interface {

/\*\*

\* method：网络请求的方法（区分大小写）

\* path：网络请求地址路径

\* hasBody：是否有请求体

\*/

@HTTP(method = "GET", path = "blog/{id}", hasBody = false)

Call<ResponseBody> getCall(@Path("id") int id);

// {id} 表示是一个变量

// method 的值 retrofit 不会做处理，所以要自行保证准确

}

## 4.2 第二类：标记



表单请求 FormUrlEncoded 请求体是 From 表单

~~ Multipart 请求体是支持文件上传的 From 表单

标记 Streaming 响应体的数据用流的形式返回，下载文件

### 4.2.1 @FormUrlEncoded

作用：表示发送form-encoded的数据

每个键值对需要用@Filed来注解键名，随后的对象需要提供值。

### 4.2.2 @Multipart

作用：表示发送form-encoded的数据（适用于 有文件 上传的场景）

每个键值对需要用@Part来注解键名，随后的对象需要提供值。

具体使用如下：

GetRequest\_Interface

public interface GetRequest\_Interface {

/\*\*

\*表明是一个表单格式的请求（Content-Type:application/x-www-form-urlencoded）

\* <code>Field("username")</code> 表示将后面的 <code>String name</code> 中name的取值作为 username 的值

\*/

@POST("/form")

@FormUrlEncoded

Call<ResponseBody> testFormUrlEncoded1(@Field("username") String name, @Field("age") int age);

/\*\*

\* {@link Part} 后面支持三种类型，{@link RequestBody}、{@link okhttp3.MultipartBody.Part} 、任意类型

\* 除 {@link okhttp3.MultipartBody.Part} 以外，其它类型都必须带上表单字段({@link okhttp3.MultipartBody.Part} 中已经包含了表单字段的信息)，

\*/

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload1(@Part("name") RequestBody name, @Part("age") RequestBody age, @Part MultipartBody.Part file);

}

// 具体使用

GetRequest\_Interface service = retrofit.create(GetRequest\_Interface.class);

// @FormUrlEncoded

Call<ResponseBody> call1 = service.testFormUrlEncoded1("Carson", 24);

// @Multipart

RequestBody name = RequestBody.create(textType, "Carson");

RequestBody age = RequestBody.create(textType, "24");

MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", "test.txt", file);

Call<ResponseBody> call3 = service.testFileUpload1(name, age, filePart);

|  |
| --- |
| 例子：  #上传文件使用：Content-Type:multipart/form-data  //传单个文件  @Multipart  @POST("v1/create")  Call<ResponseBody> create(@Part("pictureName") RequestBody pictureName,  @Part MultipartBody.Part picture)  RequestBody pictureNameBody = RequestBody.create(MediaType.parse(AppConstants.CONTENT\_TYPE\_FILE), "pictureName");  File picture= new File(path);  RequestBody requestFile = RequestBody.create(MediaType.parse(AppConstants.CONTENT\_TYPE\_FILE), picture);  // MultipartBody.Part is used to send also the actual file name  MultipartBody.Part picturePart = MultipartBody.Part.createFormData("picture", picture.getName(), requestFile);  //调接口  create(pictureNameBody, picturePart);  //传多个文件  @Multipart  @POST("v1/create")  Call<ResponseBody> create(@Part("pictureName") RequestBody pictureName, @PartMap Map<String, RequestBody> map)  RequestBody pictureNameBody = RequestBody.create(MediaType.parse(AppConstants.CONTENT\_TYPE\_FILE), "pictureName");  File picture= new File(path);  RequestBody requestFile = RequestBody.create(MediaType.parse(AppConstants.CONTENT\_TYPE\_FILE), picture);  Map<String, RequestBody> params = new HashMap<>();  params.put("picture\"; filename=\"" + picture.getName() + "", requestFile);  //调接口  create(pictureNameBody, params); |

### 4.2.3 @Streaming

响应体的数据用流的形式返回

未使用该注解，默认会把数据全部载入内存，之后通过流获取数据也是读取内存中数据，所以返回数据较大时，需要使用该注解。

|  |
| --- |
| public interface ApiInterface {  /\*\*  \* 下载视频  \* @param fileUrl  \* @return  \*/  @Streaming //大文件时要加不然会OOM  @GET  Call<ResponseBody> downloadFile(@Url String fileUrl);  } |

//获取数据的代码

ResponseBody body = response.body();

long fileSize = body.contentLength();

InputStream inputStream = body.byteStream();

## 4.3 第三类：网络请求参数



详细说明

### 4.3.1 @Header & @Headers

作用：添加请求头 &添加不固定的请求头

具体使用如下：

// @Header

@GET("user")

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

// @Headers

@Headers("Authorization: authorization")

@GET("user")

Call<User> getUser()

// 以上的效果是一致的。

// 区别在于使用场景和使用方式

// 1. 使用场景：@Header用于添加不固定的请求头，@Headers用于添加固定的请求头

// 2. 使用方式：@Header作用于方法的参数；@Headers作用于方法

### 4.3.2 @Body

使用 @Body 注解，指定一个对象作为 request body 。

作用：以 Post方式 传递 自定义数据类型 给服务器。

特别注意：如果提交的是一个Map，那么作用相当于 @Field。

不过Map要经过 FormBody.Builder 类处理成为符合 Okhttp 格式的表单，如：

FormBody.Builder builder = new FormBody.Builder();

builder.add("key","value");

### 4.3.3 @Field & @FieldMap

作用：发送 Post请求 时提交请求的表单字段

具体使用：与 @FormUrlEncoded 注解配合使用

public interface GetRequest\_Interface {

/\*\*

\*表明是一个表单格式的请求（Content-Type:application/x-www-form-urlencoded）

\* <code>Field("username")</code> 表示将后面的 <code>String name</code> 中name的取值作为 username 的值

\*/

@POST("/form")

@FormUrlEncoded

Call<ResponseBody> testFormUrlEncoded1(@Field("username") String name, @Field("age") int age);

/\*\*

\* Map的key作为表单的键

\*/

@POST("/form")

@FormUrlEncoded

Call<ResponseBody> testFormUrlEncoded2(@FieldMap Map<String, Object> map);

}

// 具体使用

// @Field

Call<ResponseBody> call1 = service.testFormUrlEncoded1("Carson", 24);

// @FieldMap

// 实现的效果与上面相同，但要传入Map

Map<String, Object> map = new HashMap<>();

map.put("username", "Carson");

map.put("age", 24);

Call<ResponseBody> call2 = service.testFormUrlEncoded2(map);

### 4.3.4 @Part & @PartMap

作用：发送 Post请求 时提交请求的表单字段

与@Field的区别：功能相同，但携带的参数类型更加丰富，包括数据流，所以适用于 有文件上传 的场景

具体使用：与 @Multipart 注解配合使用

public interface GetRequest\_Interface {

/\*\*

\* {@link Part} 后面支持三种类型，{@link RequestBody}、{@link okhttp3.MultipartBody.Part} 、任意类型

\* 除 {@link okhttp3.MultipartBody.Part} 以外，其它类型都必须带上表单字段({@link okhttp3.MultipartBody.Part} 中已经包含了表单字段的信息)，

\*/

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload1(@Part("name") RequestBody name, @Part("age") RequestBody age, @Part MultipartBody.Part file);

/\*\*

\* PartMap 注解支持一个Map作为参数，支持 {@link RequestBody } 类型，

\* 如果有其它的类型，会被{@link retrofit2.Converter}转换，如后面会介绍的 使用{@link com.google.gson.Gson} 的 {@link retrofit2.converter.gson.GsonRequestBodyConverter}

\* 所以{@link MultipartBody.Part} 就不适用了,所以文件只能用<b> @Part MultipartBody.Part </b>

\*/

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload2(@PartMap Map<String, RequestBody> args, @Part MultipartBody.Part file);

@POST("/form")

@Multipart

Call<ResponseBody> testFileUpload3(@PartMap Map<String, RequestBody> args);

}

// 具体使用

MediaType textType = MediaType.parse("text/plain");

RequestBody name = RequestBody.create(textType, "Carson");

RequestBody age = RequestBody.create(textType, "24");

RequestBody file = RequestBody.create(MediaType.parse("application/octet-stream"), "这里是模拟文件的内容");

// @Part

MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", "test.txt", file);

Call<ResponseBody> call3 = service.testFileUpload1(name, age, filePart);

ResponseBodyPrinter.printResponseBody(call3);

// @PartMap

// 实现和上面同样的效果

Map<String, RequestBody> fileUpload2Args = new HashMap<>();

fileUpload2Args.put("name", name);

fileUpload2Args.put("age", age);

//这里并不会被当成文件，因为没有文件名(包含在Content-Disposition请求头中)，但上面的 filePart 有

//fileUpload2Args.put("file", file);

Call<ResponseBody> call4 = service.testFileUpload2(fileUpload2Args, filePart); //单独处理文件

ResponseBodyPrinter.printResponseBody(call4);

}

### 4.3.5 @Query & @QueryMap

作用：用于 @GET 方法的查询参数（Query = Url 中 ‘?’ 后面的 key-value）

如：url = http://www.println.net/?cate=android，其中，Query = cate

具体使用：配置时只需要在接口方法中增加一个参数即可：

{

@GET("/")

Call<String> cate(@Query("cate") String cate);

}

// 其使用方式同 @Field与@FieldMap，这里不作过多描述

### 4.3.6 @Path

作用：URL地址的缺省值

具体使用：

public interface GetRequest\_Interface {

@GET("users/{user}/repos")

Call<ResponseBody> getBlog（@Path("user") String user ）;

// 访问的API是：https://api.github.com/users/{user}/repos

// 在发起请求时， {user} 会被替换为方法的第一个参数 user（被@Path注解作用）

}

### 4.3.7 @Url

作用：直接传入一个请求的 URL变量 用于URL设置

具体使用：

public interface GetRequest\_Interface {

@GET

Call<ResponseBody> testUrlAndQuery(@Url String url, @Query("showAll") boolean showAll);

// 当有URL注解时，@GET传入的URL就可以省略

// 当GET、POST...HTTP等方法中没有设置Url时，则必须使用 {@link Url}提供

}

### 4.3.8 小结

Path是网址中的参数,例如:trades/{userId}，相当于半静态的地址。而url则是动态的Url地址。

Query是get方法问号后面的参数,例如:trades/{userId}?token={token}；

QueryMap 相当于多个@Query。

Field用于Post请求,提交单个数据,然后要加@FormUrlEncoded。

Body相当于多个@Field,以对象的方式提交。

@Streaming:用于下载大文件。

@Header,@Headers添加请求头。



## 4.4 步骤4：创建 Retrofit 实例

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://fanyi.youdao.com/") // 设置网络请求的Url地址

.addConverterFactory(GsonConverterFactory.create()) // 设置数据解析器

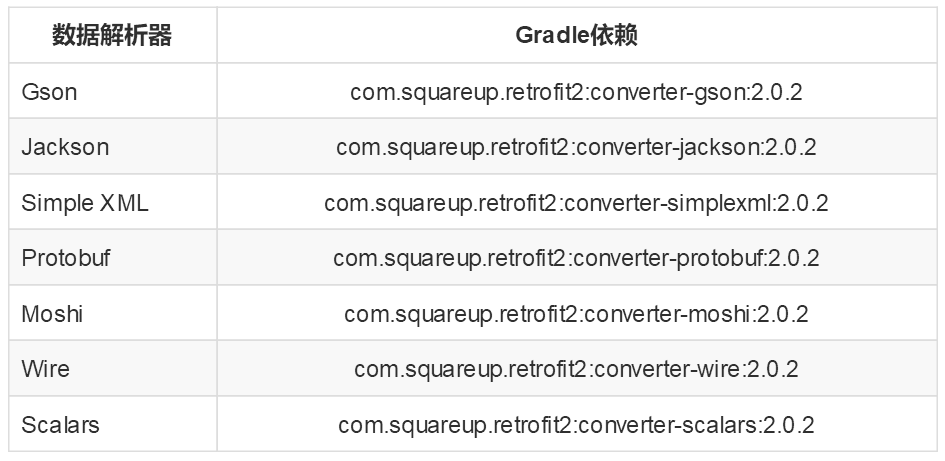
.addCallAdapterFactory(RxJavaCallAdapterFactory.create()) // 支持RxJava平台

.build();

a. 关于数据解析器（Converter）

Retrofit支持多种数据解析方式

使用时需要在Gradle添加依赖

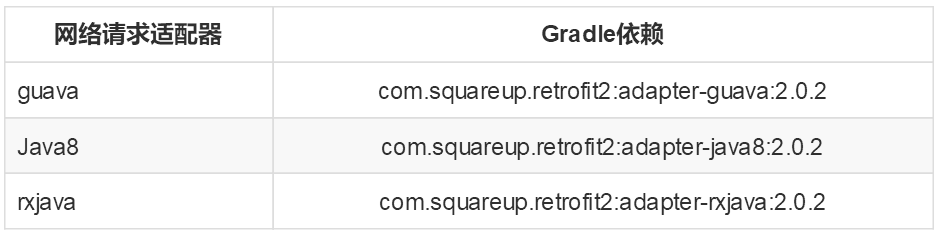


b. 关于网络请求适配器（CallAdapter）

Retrofit支持多种网络请求适配器方式：guava、Java8和rxjava。

使用时如使用的是 Android 默认的 CallAdapter，则不需要添加网络请求适配器的依赖，否则需要按照需求进行添加。

使用时需要在Gradle添加依赖：



步骤5：创建 网络请求接口实例

// 创建 网络请求接口 的实例

GetRequest\_Interface request = retrofit.create(GetRequest\_Interface.class);

//对 发送请求 进行封装

Call<Reception> call = request.getCall();

步骤6：发送网络请求（异步 / 同步）

封装了 数据转换、线程切换的操作

//发送网络请求(异步)

call.enqueue(new Callback<Translation>() {

//请求成功时回调

@Override

public void onResponse(Call<Translation> call, Response<Translation> response) {

//请求处理,输出结果

response.body().show();

}

//请求失败时候的回调

@Override

public void onFailure(Call<Translation> call, Throwable throwable) {

System.out.println("连接失败");

}

});

*// 发送网络请求（同步）*

Response<Reception> response = call.execute();

步骤7：处理返回数据

通过response类的 body（）对返回的数据进行处理

//发送网络请求(异步)

call.enqueue(new Callback<Translation>() {

//请求成功时回调

@Override

public void onResponse(Call<Translation> call, Response<Translation> response) {

// 对返回数据进行处理

response.body().show();

}

//请求失败时候的回调

@Override

public void onFailure(Call<Translation> call, Throwable throwable) {

System.out.println("连接失败");

}

});

// 发送网络请求（同步）

Response<Reception> response = call.execute();

// 对返回数据进行处理

response.body().show();

# Retrofit + RxJava结合使用

首先在build.gradle文件中加入依赖：

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

compile 'io.reactivex:rxandroid:1.2.1'

将RetrofitService类中getPostInfo方法的返回值修改为Observable（被观察者）：

public interface RetrofitService {

/\*\*

\* 获取快递信息

\* Rx方式

\* @param type 快递类型

\* @param postid 快递单号

\* @return Observable<PostInfo>

\*/

@GET("query")

Observable<PostInfo> getPostInfoRx(@Query("type") String type, @Query("postid") String postid);

}

在创建Retrofit时添加RxJava支持：

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://www.kuaidi100.com/")

.addConverterFactory(GsonConverterFactory.create())

.addCallAdapterFactory(RxJavaCallAdapterFactory.create()) // 支持RxJava

.build();

获取被观察者：

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

Observable<PostInfo> observable = service.getPostInfoRx("yuantong", "11111111111");

订阅：

observable.subscribeOn(Schedulers.io()) // 在子线程中进行Http访问

.observeOn(AndroidSchedulers.mainThread()) // UI线程处理返回接口

.subscribe(new Observer<PostInfo>() { // 订阅

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable e) {

}

@Override

public void onNext(PostInfo postInfo) {

Log.i("http返回：", postInfo.toString() + "");

}

});

在RxJava中，由于链式调用的影响，是被观察者订阅观察者。

到这里Retrofit + RxJava结合使用就讲完了，由于Retrofit、RxJava属于同门师兄弟，结合使用还是很容易的。

# 关于retrofit的封装使用

拦截器的使用：

addInterceptor()；

addNetworkInterceptor();

|  |
| --- |
| public class MyRetrofit {  private static Retrofit mRetrofit;  public static Retrofit getRetrofit() {  //如果mRetrofit为空 或者服务器地址改变 重新创建  if (mRetrofit == null) {  OkHttpClient httpClient;  OkHttpClient.Builder builder=new OkHttpClient.Builder();  /\*\*  \* 1,拦截器：实现原理就是拦截之前的请求，添加完参数，再传递新的请求。  \* 然后再Retrofit初始化的时候配置。  \*/  builder.addInterceptor(new CommonInterceptor());  /\*\*  \* 2,当处于开发阶段的时候，设置监听日志的拦截器。拦截有4个级别，分别是  \* BODY、HEADERS、BASIC、NONE  \* 其中BODY输出的日志是最全的。  \*/  //阶段分为开发和发布阶段，当前为开发阶段设置拦截器  if (BuildConfig.DEBUG) {  HttpLoggingInterceptor logging = new HttpLoggingInterceptor();  //设置拦截器级别  logging.setLevel(HttpLoggingInterceptor.Level.BODY);  builder.addInterceptor(logging);  }  httpClient=builder.build();  //构建Retrofit  mRetrofit = new Retrofit.Builder()  //配置服务器路径  .baseUrl(UrlConfig.net\_HTTP\_Host)  //返回的数据通过Gson解析  .addConverterFactory(GsonConverterFactory.create())  //配置回调库，采用RxJava  .addCallAdapterFactory(RxJavaCallAdapterFactory.create())  //设置OKHttp模板  .client(httpClient)  .build();  }  return mRetrofit;  }  /\*\*  \* 3,处理约定错误  \* 除了常见的404，500等异常，网络请求中我们往往还会约定些异常，比如token失效，账号异常等等。  \* 以token失效为例，每次请求我们都需要验证是否失效，如果在每个接口都处理一遍错误就有点太繁琐了。  \* 我们可以统一处理下错误。  \*/  } |
| /\*\*  \* 自定义拦截器：添加相同的请求参数  \*  \* 为了更好的管理迭代版本，一般每次发起请求的时候都传输当前程序的版本号到服务器。  \* 有些项目我们每次还会传用户id，token令牌等相同的参数。  \* 如果在每个请求的接口都添加这些参数太繁琐。Retrofit可以通过拦截器添加相同的请求参数，  \* 无需再每个接口添加了。  \*/  class CommonInterceptor implements Interceptor {  @Override  public Response intercept(Interceptor.Chain chain) throws IOException {  Request oldRequest = chain.request();  // 添加新的参数  HttpUrl.Builder authorizedUrlBuilder = oldRequest.url()  .newBuilder()  .scheme(oldRequest.url().scheme())  .host(oldRequest.url().host())  .addQueryParameter("device\_type", "1")  .addQueryParameter("version", BuildConfig.VERSION\_NAME)  .addQueryParameter("token", "Token");  // 新的请求  Request newRequest = oldRequest.newBuilder()  .method(oldRequest.method(), oldRequest.body())  .url(authorizedUrlBuilder.build())  .build();  return chain.proceed(newRequest);  }  } |

# 缓存的使用

## 7.1 缓存的实现

需要通过设置cache和Interceptor即可，cache的作用是指定缓存文件地址、大小，Interceptor稍微复杂一些，分为应用拦截、网络拦截两种，主要控制缓存的时间、方式、过滤等。

拦截器是针对所有的请求一视同仁的。如果需要对某个请求特殊处理，可以通过使用Headers添加请求头cacheControl来控制。所以，要通过在拦截器中将请求头的CacheControl放到响应头里面去。

最后实现的效果是：有网的时候根据你每个接口设置的需要缓存的时间（1分钟、5分钟等）进行缓存，过了时间重新请求；没网的时候读缓存。

（1）配置okhttp中的Cache

OkHttpClient okHttpClient = new OkHttpClient();

File cacheFile = new File(context.getCacheDir(), "[缓存目录]");

Cache cache = new Cache(cacheFile, 1024 \* 1024 \* 100); //100Mb

okHttpClient.setCache(cache);

（2）配置请求头中的cache-control

在Retrofit中，我们可以通过@Headers来配置，如：

@Headers("Cache-Control: public, max-age=3600)

@GET("merchants/{shopId}/icon")

Observable<ShopIconEntity> getShopIcon(@Path("shopId") long shopId);

没有设置的可以即为有网的时候不进行缓存。

或者你所有接口在有网的时候都不需要缓存或者都需要缓存且时间一样，那么也不用配置每个接口的@Headers的Cache-Control了。

（3）云端配合设置响应头或者自己写拦截器修改响应头response中cache-control

**到这一步缓存就已经待在你的缓存目录了。**

**如果云端有处里cache的话，就已经可以了。**

**但是很可能云端没有处理，所以返回的响应头中cache-control是no-cache，这时候你还是无法做缓存，大家可以用okhttp的写日志拦截器查看响应头的内容。**

如果云端现在不方便处理的话，你也可以自己搞定缓存的，那就是写拦截器修改响应头中的cache-control。**我把请求头中的cache-control读出来然后设置到了响应头中。**

设置拦截器：

REWRITE\_CACHE\_CONTROL\_INTERCEPTOR拦截器需要同时设置networkInterceptors和interceptors（OKHTTP3.0配置是否有效待我测试）

okHttpClient.interceptors().add(LoggingInterceptor);

okHttpClient.networkInterceptors().add(REWRITE\_CACHE\_CONTROL\_INTERCEPTOR);

okHttpClient.interceptors().add(REWRITE\_CACHE\_CONTROL\_INTERCEPTOR);

拦截器如下：云端响应头拦截器，用来配置缓存策略

/\*\*

\* 云端响应头拦截器，用来配置缓存策略

\* Dangerous interceptor that rewrites the server's cache-control header.

\*/private final Interceptor REWRITE\_CACHE\_CONTROL\_INTERCEPTOR = chain -> {

Request request = chain.request();

if(!NetUtils.hasNetwork(context)){

request = request.newBuilder()

.cacheControl(CacheControl.FORCE\_CACHE)

.build();

Logger.t(TAG).w("no network");

}

Response originalResponse = chain.proceed(request);

if(NetUtils.hasNetwork(context)){

//有网的时候读接口上的@Headers里的配置，你可以在这里进行统一的设置String cacheControl = request.cacheControl().toString();

return originalResponse.newBuilder()

.header("Cache-Control", cacheControl)

.removeHeader("Pragma")

.build();

}else{

return originalResponse.newBuilder()

.header("Cache-Control", "public, only-if-cached, max-stale=2419200")

.removeHeader("Pragma")

.build();

}

};

最后日志拦截器也贴上来吧

private final Interceptor LoggingInterceptor = chain -> {

Request request = chain.request();

long t1 = System.nanoTime();

Logger.t(TAG).i(String.format("Sending request %s on %s%n%s", request.url(), chain.connection(), request.headers()));

Response response = chain.proceed(request);

long t2 = System.nanoTime();

Logger.t(TAG).i(String.format("Received response for %s in %.1fms%n%s", response.request().url(), (t2 - t1) / 1e6d, response.headers()));

return response;

};

以下测试Cache-Control的配置在请求头和响应头中都有且一样。

max-stale在请求头设置有效，在响应头设置无效。

max-stale和max-age同时设置的时候，缓存失效的时间按最长的算。

关于max-age和max-stale我这里做了一个测试：

测试结果：

我在请求头中设置了：Cache-Control: public, max-age=60,max-stale=120,响应头的Cache-Control和请求头一样。

•在第一次请求数据到一分钟之内，响应头有：Cache-Control: public, max-age=60,max-stale=120

•在1分钟到3分钟在之间，响应头有：Cache-Control: public, max-age=60,max-stale=120

Warning: 110 HttpURLConnection "Response is stale"

可以发现多了一个Warning。

•三分钟的时候：重新请求了数据，如此循环，如果到了重新请求的节点此时没有网，则请求失败。

## 7.2 另一篇实现

说说我的需求:

1.有网的时候所有接口不使用缓存

2.指定的接口产生缓存文件,其他接口不会产生缓存文件

3.无网的时候指定的接口使用缓存数据.其他接口不使用缓存数据

1.网络拦截器(关键)

提示:只能缓存Get请求

Interceptor cacheInterceptor = new Interceptor() {

@Override

public Response intercept(Chain chain) throws IOException {

//拿到请求体

Request request = chain.request();

//读接口上的@Headers里的注解配置

String cacheControl = request.cacheControl().toString();

//判断没有网络并且添加了@Headers注解,才使用网络缓存.

if (!Utils.isOpenInternet()&&!TextUtils.isEmpty(cacheControl)){

//重置请求体;

request = request.newBuilder()

//强制使用缓存

.cacheControl(CacheControl.FORCE\_CACHE)

.build();

}

//如果没有添加注解,则不缓存

if (TextUtils.isEmpty(cacheControl) || "no-store" .contains(cacheControl)) {

//响应头设置成无缓存

cacheControl = "no-store";

} else if (Utils.isOpenInternet()) {

//如果有网络,则将缓存的过期时间,设置为0,获取最新数据

cacheControl = "public, max-age=" + 0;

}else {

//...如果无网络,则根据@headers注解的设置进行缓存.

}

Response response = chain.proceed(request);

HLog.i("httpInterceptor", cacheControl);

return response.newBuilder()

.header("Cache-Control", cacheControl)

.removeHeader("Pragma")

.build();

};

具体接口中的使用,添加headers:

/\*\*

\* 只能缓存get请求.

\* 这里我设置了1天的缓存时间

\* 接口随便写的.哈哈,除了 @Headers(...),其它代码没啥参考价值.

\*/

@Headers("Cache-Control: public, max-age=" + 24 \* 3600)

@GET("url")

Observable<?> queryInfo(@Query("userName") String userName);

关于Cache-Control头的参数说明:

public 所有内容都将被缓存(客户端和代理服务器都可缓存)

private 内容只缓存到私有缓存中(仅客户端可以缓存，代理服务器不可缓存)

no-cache no-cache是会被缓存的，只不过每次在向客户端（浏览器）提供响应数据时，缓存都要向服务器评估缓存响应的有效性。

no-store 所有内容都不会被缓存到缓存或 Internet 临时文件中

max-age=xxx (xxx is numeric) 缓存的内容将在 xxx 秒后失效, 这个选项只在HTTP 1.1可用, 并如果和Last-Modified一起使用时, 优先级较高

max-stale和max-age一样，只能设置在请求头里面。

同时设置max-stale和max-age，缓存失效的时间按最长的算。(这个其实不用纠结)

还有2个参数:

CacheControl.FORCE\_CACHE 强制使用缓存,如果没有缓存数据,则抛出504(only-if-cached)

CacheControl.FORCE\_NETWORK 强制使用网络,不使用任何缓存.

这两个设置,不会判断是否有网.需要自己写判断.

设置错误会导致,数据不刷新,或者有网情况下,请求不到数据

这两个很关键..可以根据自己的需求,进行切换.

2.设置OkHttpClient

OkHttpClient client = new OkHttpClient.Builder()

//添加log拦截器,打印log信息,代码后面贴出

.addInterceptor(loggingInterceptor)

//添加上面代码的拦截器,设置缓存

.addNetworkInterceptor(cacheInterceptor)

//这个也要添加,否则无网的时候,缓存设置不会生效

.addInterceptor(cacheInterceptor)

//设置缓存目录,以及最大缓存的大小,这里是设置10M

.cache(new Cache(MyApplication.getContext().getCacheDir(), 10240 \* 1024))

.build();

3.完整的代码:

public class RetrofitUtil {

/\*\*

\* 服务器地址

\*/

private static final String API\_HOST = Constant.URLS.BASEURL;

private RetrofitUtil() {

}

public static Retrofit getRetrofit() {

return Instanace.retrofit;

}

private static Retrofit getInstanace() {

HttpLoggingInterceptor loggingInterceptor = new HttpLoggingInterceptor(new HttpLoggingInterceptor.Logger() {

@Override

public void log(String message) {

HLog.i("RxJava", message);

}

});

Interceptor cacheInterceptor = new Interceptor() {

@Override

public Response intercept(Chain chain) throws IOException {

Request request = chain.request();

//有网的时候,读接口上的@Headers里的注解配置

String cacheControl = request.cacheControl().toString();

//没有网络并且添加了注解,才使用缓存.

if (!Utils.isOpenInternet()&&!TextUtils.isEmpty(cacheControl)){

//重置请求体;

request = request.newBuilder()

.cacheControl(CacheControl.FORCE\_CACHE)

.build();

}

//如果没有添加注解,则不缓存

if (TextUtils.isEmpty(cacheControl) || "no-store" .contains(cacheControl)) {

//响应头设置成无缓存

cacheControl = "no-store";

} else if (Utils.isOpenInternet()) {

//如果有网络,则将缓存的过期事件,设置为0,获取最新数据

cacheControl = "public, max-age=" + 0;

}else {

//...如果无网络,则根据@headers注解的设置进行缓存.

}

Response response = chain.proceed(request);

HLog.i("httpInterceptor", cacheControl);

return response.newBuilder()

.header("Cache-Control", cacheControl)

.removeHeader("Pragma")

.build();

}

};

OkHttpClient client = new OkHttpClient.Builder()

.addInterceptor(loggingInterceptor)

.addNetworkInterceptor(cacheInterceptor)

.addInterceptor(cacheInterceptor)

.cache(new Cache(MyApplication.getContext().getCacheDir(), 10240 \* 1024))

.build();

return new Retrofit.Builder()

.client(client)

.baseUrl(API\_HOST)

.addConverterFactory(FastjsonConverterFactory.create())

.addCallAdapterFactory(RxJavaCallAdapterFactory.create())

.build();

}

private static class Instanace {

private static final Retrofit retrofit = getInstanace();

}

}