# Retrofit

deepwaterooo

2021 年 12 月 20 日

## 目录

**1** 一、概述 **1**
1.1 二、注解 . . . . . 1
1.1.1 1、请求方法注解 . . . . . 1
1.1.2 2、请求头注解 . . . . . 2
1.1.3 3、请求参数注解 . . . . . 2
1.2 Retrofit2 使用详解 . . . . . 2
1.2.1 1 ： Get : 是我们最常见的请求方法，它是用来获取数据请求的。 . . . . . 3
1.2.2 2 POST : 一种用于携带传输数据的请求方式 . . . . . 5
1.2.3 3 Header : 一种用于携带消息头的请求方式 . . . . . 6
1.3 Retrofit2 从源码解析实现原理 . . . . . 7
1.3.1 1：将 method 把它转换成 ServiceMethod . . . . . 9
1.3.2 2：通过 serviceMethod, args 获取到 okHttpCall 对象 . . . . . 10
1.3.3 3：把 okHttpCall 进一步封装并返回 Call 对象 . . . . . 11

**2** **Retrofit** 库的核心实现原理是什么**?** 如果让你实现这个库的某些核心功能，你会考虑怎么去实现**?** **13**

# 1 一、概述

- 1、什么是 retrofit
  - retrofit 是现在比较流行的网络请求框架，可以理解为 okhttp 的加强版，底层封装了 Okhttp。准确来说，Retrofit 是一个 RESTful 的 http 网络请求框架的封装。因为网络请求工作本质上是由 okhttp 来完成，而 Retrofit 负责网络请求接口的封装。
  - 本质过程：App 应用程序通过 Retrofit 请求网络，实质上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 okhttp 来完成后续的请求工作。在服务端返回数据后，okhttp 将原始数据交给 Retrofit，Retrofit 根据用户需求解析。(源码在文章最后给出)
- 2、Retrofit 的优点
- 超级解耦，接口定义、接口参数、接口回调不在耦合在一起
- 可以配置不同的 httpClient 来实现网络请求，如 okhttp、httpclient
- 支持同步、异步、Rxjava
- 可以配置不同反序列化工具类来解析不同的数据，如 json、xml
- 请求速度快，使用方便灵活简洁

## 1.1 二、注解

- Retrofit 使用大量注解来简化请求，Retrofit 将 okhttp 请求抽象成 java 接口，使用注解来配置和描述网络请求参数。大概可以分为以下几类，我们先来看看各个注解的含义，再一一去实践解释。

### 1.1.1 1、请求方法注解

| 请求方法注解 | 说明 |
|---|---|
| @GET | get 请求 |
| @POST | post 请求 |
| @PUT | put 请求 |
| @DELETE | delete 请求 |
| @PATCH | patch 请求，该请求是对 put 请求的补充，用于更新局部资源 |
| @HEAD | head 请求 |
| @OPTIONS | options 请求 |
| @HTTP | 通过注解，可以替换以上所有的注解，它拥有三个属性：method、path、hasBody |

### 1.1.2 2、请求头注解

| 请求头注解 | 说明 |
|---|---|
| @Headers | 用于添加固定请求头，可同时添加多个，通过该注解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，用于添加不固定的 header，它会更新已有请求头 |

### 1.1.3 3、请求参数注解

| 请求参数注解 | 说明 |
|---|---|
| @Body | 多用于 Post 请求发送非表达数据，根据转换方式将实例对象转化为对应字符串传递参数，<br>比如使用 Post 发送 Json 数据，添加 GsonConverterFactory 则是将 body 转化为 json 字 |
| @Filed | 多用于 Post 方式传递参数，需要结合 @FromUrlEncoded 使用，即以表单的形式传递参 |
| @FiledMap | 多用于 Post 请求中的表单字段，需要结合 @FromUrlEncoded 使用 |
| @Part | 用于表单字段，Part 和 PartMap 与 @multipart 注解结合使用，适合文件上传的情况 |
| @PartMap | 用于表单字段，默认接受类型是 Map<String,RequestBody>，可用于实现多文件上传 |
| @Path | 用于 Url 中的占位符 |
| @Query | 用于 Get 请求中的参数 |
| @QueryMap | 与 Query 类似，用于不确定表单参数 |
| @Url | 指定请求路径 |

4、请求和响应格式 (标记) 注解

标记类注解说明

| | |
|---|---|
| @FromUrlCoded | 表示请求发送编码表单数据，每个键值对需要使用 @Filed 注解 |
| @Multipart | 表示请求发送 form$_{encoded}$ (使用于有文件上传的场景)，<br>每个键值对需要用 @Part 来注解键名，随后的对象需要提供值 |
| @Streaming | 表示响应用字节流的形式返回，如果没有使用注解，默认会把数据全部载入到<br>内存中，该注解在下载大文件时特别有用 |

## 1.2 Retrofit2 使用详解

- 添加依赖：

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

- 因为 Retrofit2 是依赖 okHttp 请求的，而且请查看它的 META-INF->META-INF.squareup.retrofit2->pom.xml 文件
  - 这个文件是我不曾关注、不曾注意到过的，需要熟悉一下

```
<dependencies>
    <dependency>
      <groupId>com.squareup.okhttp3</groupId>
      <artifactId>okhttp</artifactId>
    </dependency>
</dependencies>
```

- 由此可见，它确实是依赖 okHttp，okHttp 有会依赖 okio 所以它会制动的把这两个包也导入进来。
- 添加权限：
  - 既然要请求网络，在我们 android 手机上是必须要有访问网络的权限的，下面把权限添加进来

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

- 好了，下面开始介绍怎么使用 Retrofit，既然它是使用注解的请求方式来完成请求 URL 的拼接，那么我们就按注解的不同来分别学习：
- 首先，我们需要创建一个 java 接口，用于存放请求方法的：

```
public interface GitHubService {
}
```

- 然后逐步在该方法中添加我们所需要的方法（按照请求方式）：

### 1.2.1 1 ： Get ： 是我们最常见的请求方法，它是用来获取数据请求的。

1. ： 直接通过 URL 获取网络内容：

```
public interface GitHubService {
    @GET("users/octocat/repos")
    Call<List<Repo>> listRepos();
}
```

- 在这里我们定义了一个 listRepos() 的方法，通过 @GET 注解标识为 get 请求，请求的 URL 为“users/octocat/repos”。
- 然后看看 Retrofit 是怎么调用的，代码如下：

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/") // 通过动态代理获取到所定义的接口
    .build();
GitHubService service = retrofit.create(GitHubService.class);
Call<List<Repo>> repos = service.listRepos();
repos.enqueue(new Callback<List<Repo>>(){
    @Override
    public void onResponse(Call<List<Repo>> call, Response<List<Repo>> response){
    }
    @Override
    public void onFailure(Call<List<Repo>> call, Throwable t){
    }
});
```

- 代码解释：首先获取 Retrofit 对象，然后通过动态代理获取到所定义的接口，通过调用接口里面的方法获取到 Call 类型返回值，最后进行网络请求操作（这里不详细说明 Retrofit 实现原理，后面会对它进行源码解析），这里必须要说的是请求 URL 的拼接：在构建 Retrofit 对象时调用 baseUrl 所传入一个 String 类型的地址，这个地址在调用 service.listRepos() 时会把 @GET(“users/octocat/repos”) 的 URL 拼接在尾部。

ok，这样就完成了，我们这次的请求，但是我们不能每次请求都要创建一个方法呀？这时我们就会想起 动态的构建 URL 了

2. @Path: ：动态获取 URL 地址：@Path

- 我们再上面的基础上进行修改，如下：

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
```

- 这里在 Get 注解中包含 {user}，它所对应的是 @Path 注解中的“user”，它所标示的正是 String user，而我们再使用 Retrofit 对象动态代理的获取到 GitHubService，当调用 listRepos 时，我们就必须传入一个 String 类型的 User，如：

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

- 如上代码，其他的代码都是不变的，而我们只需要使用 @Path 注解就完全的实现了动态的 URL 地址了，是不是很方便呢，这还不算什么，通常情况下，我们去获取一些网络信息，因为信息量太大，我们会分类去获取，也就是携带一些必要的元素进行过滤，那我们该怎么实现呢？其实也很简单，因为 Retrofit 已经为我们封装好了注解，请看下面 (官网实例)：

3. @Query: ：动态指定条件获取信息：@Query

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

- 我们只需要使用 @Query 注解即可完成我们的需求，在 @Query(“sort”) 中，short 就好比是 URL 请求地址中的键，而它说对应的 String sort 中的 sort 则是它的值。
- 但是我们想，在网络请求中一般为了更精确的查找到我们所需要的数据，过滤更多不需要的无关的东西，我们往往需要携带多个请求参数，当然可以使用 @Query 注解，但是太麻烦，很长，容易遗漏和出错，那有没有更简单的方法呢，有，当然后，我们可以直接放到一个 map 键值对中：

4. @QueryMap ：动态指定条件组获取信息：@QueryMap

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

- 使用 @QueryMap 注解可以分别地从 Map 集合中获取到元素，然后进行逐个的拼接在一起。
- ok，到这里，我们使用 @Get 注解已经可以完成绝大部分的查询任务了，下面我们再来看看另一种常用的请求方式-post

### 1.2.2 2 POST：一种用于携带传输数据的请求方式

- 稍微了解点 Http 的同学们，可能都会知道：相对于 get 请求方式把数据存放在 uri 地址栏中，post 请求传输的数据时存放在请求体中，所以 post 才能做到对数据的大小无限制。而在 Retrofit 中，它又是怎么使用的呢？请看下面：

1. @Body ：携带数据类型为对象时：@Body

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

- 当我们的请求数据为某对象时 Retrofit 是这么处理使用的：
  - 首先，Retrofit 用 @POST 注解，标明这个是 post 的请求方式，里面是请求的 url；
  - 其次，Retrofit 仿照 http 直接提供了 @Body 注解，也就类似于直接把我们要传输的数据放在了 body 请求体中，这样应用可以更好的方便我们理解。
- 来看下应用：

```
Call<List<User>> repos = service.createUser(new User(1, " 管满满", "28", "http://write.blog.csdn.net/postlist"));
```

- 这样我们直接把一个新的 User 对象利用注解 @Body 存放在 body 请求体，并随着请求的执行传输过去了。
- 但是有同学在这该有疑问了，Retrofit 就只能传输的数据为对象吗？当然不是，下面请看

2. @Field ：携带数据类型为表单键值对时：@Field

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

- 当我们要携带的请求数据为表单时，通常会以键值对的方式呈现，那么 Retrofit 也为我们考虑了这种情况，它首先用到 @FormUrlEncoded 注解来标明这是一个表单请求，然后在我们的请求方法中使用 @Field 注解来标示所对应的 String 类型数据的键，从而组成一组键值对进行传递。
- 那你是不是有该有疑问了，假如我是要上传一个文件呢？

3. @Part ：单文件上传时：@Part

```
@Multipart // 表示允许多个 @Part
@PUT("user/photo")
Call<User> updateUser(@Part("photo") RequestBody photo, @Part("description") RequestBody description);
```

- 此时在上传文件时，我们需要用 @Multipart 注解注明，它表示允许多个 @Part，@Part 则对应的一个 RequestBody 对象，RequestBody 则是一个多类型的，当然也是包括文件的。下面看看使用

```
File file = new File(Environment.getExternalStorageDirectory(), "ic_launcher.png");
RequestBody photoRequestBody = RequestBody.create(MediaType.parse("image/png"), file);
RequestBody descriptionRequestBody = RequestBody.create(null, "this is photo.");
Call<User> call = service.updateUser(photoRequestBody, descriptionRequestBody);
```

- 这里我们创建了两个 RequestBody 对象，然后调用我们定义的 updateUser 方法，并把 RequestBody 传递进入，这样就实现了文件的上传。是不是很简单呢？
- 相比单文件上传，Retrofit 还进一步提供了多文件上传的方式：

4. @PartMap ： 多文件上传时：@PartMap

```
@Multipart
@PUT("user/photo")
Call<User> updateUser(@PartMap Map<String, RequestBody> photos, @Part("description") RequestBody description);
```

- 这里其实和单文件上传是差不多的，只是使用一个集合类型的 Map 封装了文件，并用 @PartMap 注解来标示起来。其他的都一样，这里就不多讲了。

### 1.2.3 3 Header : 一种用于携带消息头的请求方式

- Http 请求中，为了防止攻击或是过滤掉不安全的访问或是为添加特殊加密的访问等等以减轻服务器的压力和保证请求的安全，通常都会在消息头中携带一些特殊的消息头处理。Retrofit 也为我们提供了该请求方式：

```
@Headers("Cache-Control: max-age=640000")
@GET("widget/list")
Call<List<Widget>> widgetList();
// ------------------------------------------------------------
@Headers({
    "Accept: application/vnd.github.v3.full+json",
    "User-Agent: Retrofit-Sample-App"
})
@GET("users/{username}")
Call<User> getUser(@Path("username") String username);
```

- 以上两种是静态的为 Http 请求添加消息头，只需要使用 @Headers 注解，以键值对的方式存放即可，如果需要添加多个消息头，则使用 {} 包含起来，如上所示。但要注意，即使有相同名字得消息头也不会被覆盖，并共同的存放在消息头中。
- 当然有静态添加那相对的也就有动态的添加消息头了，方法如下：

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

- 使用 @Header 注解可以为一个请求动态的添加消息头，假如 @Header 对应的消息头为空的话，则会被忽略，否则会以它的.toString() 方式输出。
- ok，到这里已基本讲解完 Retrofit 的使用，还有两个重要但简单的方法也必须在这里提一下：
  - 1 call.cancel(); 它可以终止正在进行的请求，程序只要一旦调用到它，不管请求是否在终止都会被停止掉。

- 2 call.clone(); 当你想要多次请求一个接口的时候，直接用 clone 的方法来生产一个新的，否则将会报错，因为当你得到一个 call 实例，我们调用它的 execute 方法，但是这个方法只能调用一次。多次调用则发生异常。

- 好了，关于 Retrofit 的使用我们就讲这么多，接下来我们从源码的角度简单的解析下它的实现原理。

## 1.3 Retrofit2 从源码解析实现原理

- 首先先看一下 Retrofit2 标准示例

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build();
GitHubService service = retrofit.create(GitHubService.class);
service.enqueue();
```

- 由上面我们基本可以看出，Retrofit 是通过构造者模式创建出来的，那么我们就来看看 Builder 这个构造器的源码：

```
public static final class Builder {
    Builder(Platform platform) {
        this.platform = platform;
        converterFactories.add(new BuiltInConverters());
    }
    public Builder() {
        this(Platform.get());
    }

    public Builder client(OkHttpClient client) {
        return callFactory(checkNotNull(client, "client == null"));
    }
    public Builder callFactory(okhttp3.Call.Factory factory) {
        this.callFactory = checkNotNull(factory, "factory == null");
        return this;
    }

    public Builder baseUrl(String baseUrl) {
        checkNotNull(baseUrl, "baseUrl == null");
        HttpUrl httpUrl = HttpUrl.parse(baseUrl);
        if (httpUrl == null)
            throw new IllegalArgumentException("Illegal URL: " + baseUrl);
        return baseUrl(httpUrl);
    }
    public Builder baseUrl(HttpUrl baseUrl) {
        checkNotNull(baseUrl, "baseUrl == null");
        List<String> pathSegments = baseUrl.pathSegments();
        if (!"".equals(pathSegments.get(pathSegments.size() - 1)))
            throw new IllegalArgumentException("baseUrl must end in /: " + baseUrl);
        this.baseUrl = baseUrl;
        return this;
    }

    public Builder addConverterFactory(Converter.Factory factory) {
        converterFactories.add(checkNotNull(factory, "factory == null"));
        return this;
    }
    public Builder addCallAdapterFactory(CallAdapter.Factory factory) {
        adapterFactories.add(checkNotNull(factory, "factory == null"));
        return this;
    }

    public Builder callbackExecutor(Executor executor) {
        this.callbackExecutor = checkNotNull(executor, "executor == null");
        return this;
    }
    public Builder validateEagerly(boolean validateEagerly) {
```

```
        this.validateEagerly = validateEagerly;
        return this;
    }
    public Retrofit build() {
        if (baseUrl == null)
            throw new IllegalStateException("Base URL required.");

        okhttp3.Call.Factory callFactory = this.callFactory;
        if (callFactory == null)
            callFactory = new OkHttpClient();

        Executor callbackExecutor = this.callbackExecutor;
        if (callbackExecutor == null)
            callbackExecutor = platform.defaultCallbackExecutor();

        List<CallAdapter.Factory> adapterFactories = new ArrayList<>(this.adapterFactories);
        adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));
        List<Converter.Factory> converterFactories = new ArrayList<>(this.converterFactories);

        return new Retrofit(callFactory, baseUrl, converterFactories, adapterFactories, callbackExecutor, validateEagerly);
    }
}
```

- 源码讲解：
  - 1：当我们使用 new Retrofit.Builder() 来创建时，在 Builder 构造器中，首先就获得当前的设备平台信息，并且把内置的转换器工厂（BuiltInConverters）加添到工厂集合中，它的主要作用就是当使用多种 Converters 的时候能够正确的引导并找到可以消耗该类型的转化器。
  - 2：从我们的基本示例中看到有调用到.baseUrl(BASE$_{URL}$) 这个方法，实际上没当使用 Retrofit 时，该方法都是必须传入的，并且还不能为空，从源码中可以看出，当 baseUrl 方法传进的参数来看，如果为空的话将会抛出 NullPointerException 空指针异常。
  - 3：addConverterFactory 该方法是传入一个转换器工厂，它主要是对数据转化用的，请网络请求获取的数据，将会在这里被转化成我们所需要的数据类型，比如通过 Gson 将 json 数据转化成对象类型。
  - 4：从源码中，我们看到还有一个 client 方法，这个是可选的，如果没有传入则就默认为 OkHttpClient，在这里可以对 OkHttpClient 做一些操作，比如添加拦截器打印 log 等
  - 5：callbackExecutor 该方法从名字上看可以得知应该是回调执行者，也就是 Call 对象从网络服务获取数据之后转换到 UI 主线程中。
  - 6：addCallAdapterFactory 该方法主要是针对 Call 转换了，比如对 Rxjava 的支持，从返回的 call 对象转化为 Observable 对象。
  - 7：最后调用 build() 方法，通过 new Retrofit(callFactory, baseUrl, converterFactories, adapterFactories, callbackExecutor, validateEagerly); 构造方法把所需要的对象传递到 Retrofit 对象中。
- ok，当我们通过 Builder 构造器构造出 Retrofit 对象时，然后通过 Retrofit.create() 方法是怎么把我们所定义的接口转化成接口实例的呢？来看下 create() 源码：

```
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);
```

```
                ServiceMethod serviceMethod = loadServiceMethod(method);
                OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
                return serviceMethod.callAdapter.adapt(okHttpCall);
            }
        });
}
```

- 当看到 Proxy 时，是不是多少有点明悟了呢？没错就是动态代理，动态代理其实已经封装的很简单了，主要使用 newProxyInstance() 方法来返回一个类的代理实例，其中它内部需要传递一个类的加载器，类本身以及一个 InvocationHandler 处理器，主要的动作都是在 InvocationHandler 中进行的，它里面只有一个方法 invoke() 方法，每当我们调用代理类里面的方法时 invoke() 都会被执行，并且我们可以从该方法的参数中获取到所需要的一切信息，比如从 method 中获取到方法名，从 args 中获取到方法名中的参数信息等。
- 而 Retrofit 在这里使用到动态代理也不会例外：
  - 首先，通过 method 把它转换成 ServiceMethod ；
  - 然后，通过 serviceMethod, args 获取到 okHttpCall 对象；
  - 最后，再把 okHttpCall 进一步封装并返回 Call 对象。
- 下面来逐步详解。

### 1.3.1 1：将 method 把它转换成 ServiceMethod

```
ServiceMethod serviceMethod = loadServiceMethod(method);
ServiceMethod loadServiceMethod(Method method) {
    ServiceMethod result;
    synchronized (serviceMethodCache) {
        result = serviceMethodCache.get(method);
        if (result == null) {
            result = new ServiceMethod.Builder(this, method).build();
            serviceMethodCache.put(method, result);
        }
    }
    return result;
}
```

- loadServiceMethod 源码方法中非常的好理解，主要就是通过 ServiceMethod.Builder() 方法来构建 ServiceMethod，并把它给缓存取来，以便下次可以直接回去 ServiceMethod。那下面我们再来看看它是怎么构建 ServiceMethod 方法的：

```
public Builder(Retrofit retrofit, Method method) {
    this.retrofit = retrofit;
    this.method = method;
    this.methodAnnotations = method.getAnnotations();
    this.parameterTypes = method.getGenericParameterTypes();
    this.parameterAnnotationsArray = method.getParameterAnnotations();
}
public ServiceMethod build() {
    callAdapter = createCallAdapter();
    responseType = callAdapter.responseType();
    if (responseType == Response.class || responseType == okhttp3.Response.class)
        throw methodError("'" + Utils.getRawType(responseType).getName()
                          + "' is not a valid response body type. Did you mean ResponseBody?");
    responseConverter = createResponseConverter();
    return new ServiceMethod<>(this);
}
```

- 首先在 Builder() 中初始化一些参数，然后在 build() 中返回一个 new ServiceMethod<>(this) 对象。

- 下面来详细的解释下 build() 方法，完全理解了该方法则便于理解下面的所有执行流程。

1. ：构建 CallAdapter 对象，该对象将会在第三步中起着至关重要的作用。
   - 现在我们先看看它是怎么构建 CallAdapter 对象的：createCallAdapter() 方法源码如下：

```
private CallAdapter<?> createCallAdapter() {
    Type returnType = method.getGenericReturnType();
    if (Utils.hasUnresolvableType(returnType))
        throw methodError("Method return type must not include a type variable or wildcard: %s", returnType);
    if (returnType == void.class)
        throw methodError("Service methods cannot return void.");
    Annotation[] annotations = method.getAnnotations();
    try {
        return retrofit.callAdapter(returnType, annotations);
    } catch (RuntimeException e) { // Wide exception range because factories are user code.
        throw methodError(e, "Unable to create call adapter for %s", returnType);
    }
}
```

   - 在 createCallAdapter 方法中主要做的是事情就是获取到 method 的类型和注解，然后调用 retrofit.callAdapter(returnType, annotations); 方法：

```
public CallAdapter<?> callAdapter(Type returnType, Annotation[] annotations) {
    return nextCallAdapter(null, returnType, annotations);
}
public CallAdapter<?> nextCallAdapter(CallAdapter.Factory skipPast, Type returnType, Annotation[] annotations) {
    checkNotNull(returnType, "returnType == null");
    checkNotNull(annotations, "annotations == null");
    int start = adapterFactories.indexOf(skipPast) + 1;
    for (int i = start, count = adapterFactories.size(); i < count; i++) {
        CallAdapter<?> adapter = adapterFactories.get(i).get(returnType, annotations, this);
        if (adapter != null)
            return adapter;
    }
}
```

   - 辗转到 Retrofit 中 nextCallAdapter() 中，在 for 循环中分别从 adapterFactories 中来获取 CallAdapter 对象，但是 adapterFactories 中有哪些 CallAdapter 对象呢，这就需要返回到构建 Retrofit 对象中的 Builder 构造器中查看了

```
public static final class Builder {
    public Builder addCallAdapterFactory(CallAdapter.Factory factory) {
        adapterFactories.add(checkNotNull(factory, "factory == null"));
        return this;
    }
    public Retrofit build() {
        List<CallAdapter.Factory> adapterFactories = new ArrayList<>(this.adapterFactories);
        adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));
    }
}
CallAdapter.Factory defaultCallAdapterFactory(Executor callbackExecutor) {
    if (callbackExecutor != null)
        return new ExecutorCallAdapterFactory(callbackExecutor);
    return DefaultCallAdapterFactory.INSTANCE;
}
```

   - 从上面的代码中可以看到，不管有没有通过 addCallAdapterFactory 添加 CallAdapter，adapterFactories 集合至少都会有一个 ExecutorCallAdapterFactory 对象。当我们从 adapterFactories 集合中回去 CallAdapter 对象时，那我们都会获得 ExecutorCallAdapterFactory 这个对象。而这个对象将在第三步中和后面执行同步或异步请求时起着至关重要的作用。

2. ：构建 responseConverter 转换器对象，它的作用是寻找适合的数据类型转化

- 该对象的构建和构建 CallAdapter 对象的流程基本是一致的，这里就不在赘述。同学们可自行查看源码。

### 1.3.2 2：通过 serviceMethod, args 获取到 okHttpCall 对象

- 第二步相对比较简单，就是对象传递：

```
OkHttpCall(ServiceMethod<T> serviceMethod, Object[] args) {
    this.serviceMethod = serviceMethod;
    this.args = args;
}
```

### 1.3.3 3：把 okHttpCall 进一步封装并返回 Call 对象

- 这一步也是一句话 return serviceMethod.callAdapter.adapt(okHttpCall); 但是想理解清楚必须先把第一步理解透彻，通过第一步我们找得到 serviceMethod.callAdapter 就是 ExecutorCallAdapterFactory 对象，那么调用.adapt(okHttpCall) 把 okHttpCall 怎么进行封装呢？看看源码：

```
T adapt(Call call);
```

- 一看,吓死宝宝了,就这么一句,这是嘛呀,但是经过第一步的分析,我们已知道 serviceMethod.callAdap 就是 ExecutorCallAdapterFactory，那么我们可以看看在 ExecutorCallAdapterFactory 类中有没有发现 CallAdapter 的另类应用呢，一看，果不其然在重写父类的 get() 方法中我们找到了答案：

```
@Override
public CallAdapter<Call<?>> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {
    if (getRawType(returnType) != Call.class)
        return null;
    final Type responseType = Utils.getCallResponseType(returnType);
    return new CallAdapter<Call<?>>() {
        @Override public Type responseType() {
            return responseType;
        }
        @Override public <R> Call<R> adapt(Call<R> call) {
            return new ExecutorCallbackCall<>(callbackExecutor, call);
        }
    };
}
```

- 当看到 return new CallAdapter 中的 adapt(Call call) 我们就完全知其所以然了，至于 ExecutorCallbackCall 怎么应用的我们在发起网络请求的时候讲解。

- ok，当我们得到接口的代理实例之后，通过代理接口调用里面的方法，就会触发 InvocationHandler 对象中的 invoke 方法，从而完成上面的三个步骤并且返回一个 Call 对象，通过 Call 对象就可以去完成我们的请求了，Retrofit 为我们提供两种请求方式，一种是同步，一种是异步。我们这里就以异步方式来讲解：

```
service.enqueue(new Callback<List<User>>() {
    @Override
    public void onResponse(Call<List<User>> call, Response<List<User>> response) {
        Log.d("response body",response.body());
    }
    @Override
    public void onFailure(Call<BoardInfo> call, Throwable t) {
        Log.i("response Throwable",t.getMessage().toString());
    }
});
```

- 从上面我们可以看到 enqueue 方法中有一个回调函数，回调函数里面重写了两个方法分别代表请求成功和失败的方法，但是我们想知道它是怎么实现的原理呢？那么请往下面看：
- 在上面获取接口的代理实例时，通过代理接口调用里面的方法获取一个 Call 对象，我们上面也分析了其实这个 Call 对象就是 ExecutorCallbackCall，那么我们来看看它里面是怎么实现的？

```
static final class ExecutorCallbackCall<T> implements Call<T> {
    final Executor callbackExecutor;
    final Call<T> delegate;
    ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
        this.callbackExecutor = callbackExecutor;
        this.delegate = delegate;
    }
    @Override public void enqueue(final Callback<T> callback) {
        if (callback == null) throw new NullPointerException("callback == null");

        delegate.enqueue(new Callback<T>() {
                @Override public void onResponse(Call<T> call, final Response<T> response) {
                    callbackExecutor.execute(new Runnable() {
                            @Override public void run() {
                                if (delegate.isCanceled())
                                    callback.onFailure(ExecutorCallbackCall.this, new IOException("Canceled"));
                                else
                                    callback.onResponse(ExecutorCallbackCall.this, response);
                            }
                        });
                }
                @Override public void onFailure(Call<T> call, final Throwable t) {
                    callbackExecutor.execute(new Runnable() {
                            @Override public void run() {
                                callback.onFailure(ExecutorCallbackCall.this, t);
                            }
                        });
                }
            });
    }
}
```

- 在 ExecutorCallbackCall 类中，封装了两个对象一个是 callbackExecutor，它主要是把现在运行的线程切换到主线程中去，一个是 delegate 对象，这个对象就是真真正正的执行网络操作的对象，那么它的真身到底是什么呢？还记得我们在获取代理接口第三步执行的 serviceMethod.callAdapter.adapt(okHttpCall) 的分析吧，经过辗转几步终于把 okHttpCall 传递到了 new ExecutorCallbackCall<>(callbackExecutor, call); 中，然后看看 ExecutorCallbackCall 的构造方法：

```
ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
      this.callbackExecutor = callbackExecutor;
      this.delegate = delegate;
    }
```

- 由此可以明白 delegate 就是 okHttpCall 对象，那么我们在看看 okHttpCall 是怎么执行异步网络请求的：

```
@Override
public void enqueue(final Callback<T> callback) {
    if (callback == null) throw new NullPointerException("callback == null");
    okhttp3.Call call;
    Throwable failure;
    synchronized (this) {
        if (executed) throw new IllegalStateException("Already executed.");
        executed = true;
        call = rawCall;
        failure = creationFailure;
        if (call == null && failure == null)
```

```
            try {
                call = rawCall = createRawCall();
            } catch (Throwable t) {
                failure = creationFailure = t;
            }
        }
        if (failure != null) {
            callback.onFailure(this, failure);
            return;
        }
        if (canceled)
            call.cancel();
        call.enqueue(new okhttp3.Callback() {
            @Override public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse)
                throws IOException {
                Response<T> response;
                try {
                    response = parseResponse(rawResponse);
                } catch (Throwable e) {
                    callFailure(e);
                    return;
                }
                callSuccess(response);
            }
            @Override public void onFailure(okhttp3.Call call, IOException e) {
                try {
                    callback.onFailure(OkHttpCall.this, e);
                } catch (Throwable t) {
                    t.printStackTrace();
                }
            }
            private void callFailure(Throwable e) {
                try {
                    callback.onFailure(OkHttpCall.this, e);
                } catch (Throwable t) {
                    t.printStackTrace();
                }
            }
            private void callSuccess(Response<T> response) {
                try {
                    callback.onResponse(OkHttpCall.this, response);
                } catch (Throwable t) {
                    t.printStackTrace();
                }
            }
        });
}
```

- 从上面代码中，我们很容易就看出，其实它就是这里面封装了一个 okhttp3.Call，直接利用 okhttp 进行网络的异步操作，至于 okhttp 是怎么进行网络请求的我们就不再这里讲解了，感兴趣的朋友可以自己去查看源码。

# 2 Retrofit 库的核心实现原理是什么？如果让你实现这个库的某些核心功能，你会考虑怎么去实现？

- Retrofit 主要是在 create 方法中采用动态代理模式 (通过访问代理对象的方式来间接访问目标对象) 实现接口方法，这个过程构建了一个 ServiceMethod 对象，根据方法注解获取请求方式，参数类型和参数注解拼接请求的链接，当一切都准备好之后会把数据添加到 Retrofit 的 RequestBuilder 中。然后当我们主动发起网络请求的时候会调用 okhttp 发起网络请求，okhttp 的配置包括请求方式，URL 等在 Retrofit 的 RequestBuilder 的 build() 方法中实现，并发起真正的网络请求。
- 你从这个库中学到什么有价值的或者说可借鉴的设计思想?
- 内部使用了优秀的架构设计和大量的设计模式，在我分析过 Retrofit 最新版的源码和大量优秀

的 Retrofit 源码分析文章后，我发现，要想真正理解 Retrofit 内部的核心源码流程和设计思想，首先，需要对它使用到的九大设计模式有一定的了解，下面我简单说一说:

- 1、创建 Retrofit 实例: 使用 建造者模式 通过内部 Builder 类建立了一个 Retroift 实例。网络请求工厂使用了 工厂模式 方法。
- 2、创建网络请求接口的实例:
  - 首先，使用 外观模式 统一调用创建网络请求接口实例和网络请求参数配置的方法。然后，使用 动态代理 动态地去创建网络请求接口实例。
  - 接着，使用了 建造者模式 & 单例模式 创建了 serviceMethod 对象。
  - 再者，使用了 策略模式 对 serviceMethod 对象进行网络请求参数配置，即通过解析网络请求接口方法的参数、返回值和注解类型，从 Retrofit 对象中获取对应的网络的 url 地址、网络请求执行器、网络请求适配器和数据转换器。
  - 最后，使用了 装饰者模式 ExecuteCallBack 为 serviceMethod 对象加入线程切换的操作，便于接受数据后通过 Handler 从子线程切换到主线程从而对返回数据结果进行处理。
- 3、发送网络请求: 在异步请求时，通过静态 delegate 代理对网络请求接口的方法中的每个参数使用对应的 ParameterHanlder 进行解析。
- 4、解析数据
- 5、切换线程: 使用了 适配器模式 通过检测不同的 Platform 使用不同的回调执行器，然后使用回调执行器切换线程，这里同样是使用了装饰模式。
- 6、处理结果