# HenCoder Plus 第 4 课 讲义

# Retrofit

## Retrofit 使用方法简介

1. 创建一个 interface 作为 Web Service 的请求集合，在里面用注解（Annotation）写入需要配置的请求方法

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

2. 在正式代码里用 `Retrofit` 创建出 interface 的实例

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

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

3. 调用创建出的 Service 实例的对应方法，创建出相应的可以用来发起网络请求的 `Call` 对象

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

4. 使用 `Call.execute()` 或者 `Call.enqueue()` 来发起请求

```
repos.enqueue(callback);
```

## Retrofit 源码结构总结

- 通过 `Retrofit.create(Class)` 方法创建出 Service interface 的实例，从而使得 Service 中配置的方法变得可用，这是 Retrofit 代码结构的核心；
- `Retrofit.create()` 方法内部，使用的是 `Proxy.newProxyInstance()` 方法来创建 Service 实例。这个方法会为参数中的多个 interface （具体到 Retrofit 来说，是固定传入一个 interface）创建一个对象，这个对象实现了所有 interface 的每个方法，并且每个方法的实现都是雷同的：调用对象实例内部的一个 `InvocationHandler` 成员变量的 `invoke()` 方法，并把自己的方法信息传递进去。这样就在实质上实现了代理逻辑：interface 中的方法全部由一个另外设定的 `InvocationHandler` 对象来进行代理操作。并且，这些方法的具体实现是在运行时生成 interface 实例时才确定的，而不是在编译时（虽然在编译时就已经可以通过代码逻辑推断出来）。这就是网上所说的「动态代理机制」的具体含义。

具体的概念结构可以回顾课上画的图。

- 因此，`invoke()` 方法中的逻辑，就是 Retrofit 创建 Service 实例的关键。这个方法内有三行关键代码，共同组成了具体逻辑：

  1. `ServiceMethod` 的创建：

```
ServiceMethod<Object, Object> serviceMethod =
        (ServiceMethod<Object, Object>) loadServiceMethod(method);
```

  这行代码负责读取 interface 中原方法的信息（包括返回值类型、方法注解、参数类型、参数注解），并将这些信息做初步分析。

  2. `OkHttpCall` 的创建：

```
OkHttpCall<Object> okHttpCall = new OkHttpCall<>(serviceMethod, args);
```

  `OkHttpCall` 是 `retrofit2.Call` 的子类。这行代码负责将 ServiceMethod 封装进一个 `retrofit2.Call` 对象；而这个对象可以在需要的时候（例如它的 `enqueue()` 方法被调用的时候，利用 ServiceMethod 中包含的信息来创建一个 `okhttp3.Call` 对象，并调用这个 `okhttp3.Call` 对象来进行网络请求的发起，然后对结果进行预处理（如类型转换）。

  3. `adapt()` 方法：

```
return serviceMethod.adapt(okHttpCall);
```

  这个方法会使用 ServiceMethod 中的 callAdapter 对象来把 okHttpCall 对象进行转换，生成一个新的 `retrofit2.Call` 对象，在这个新的 `Call` 对象中，后台线程发起的请求，会在相应返回后，从主线程中调用回调方法，实现线程的自动切换。

  另外，这个方法不止可以生成新的 `retrofit2.Call` 对象，也可以生成别的类型对象，例如 RxJava 的 `Obervable`，来让 Retrofit 可以和 RxJava 结合使用。

- 更细的代码逻辑（例如 ServiceMethod 如果做方法解析、CallAdapter 如果做 adapt，就不在讲义里再总结一遍了，可以看课上的分析）

## 扔物线读源码的思路与方式

- 寻找切入点，而不是逐行通读
  - 理想情况下，逐行通读可以最高效率读通一个项目的代码，因为每行代码都只需要读一遍；但实时情况下，逐行通读会导致脑中积累太多没有成体系的代码，因此一点也不实用。而从切入点开始读，可以在最快时间内把看到的代码体系化，形成一个「完整的小世界」；在把「小世界」看明白之后，再去一步步扩大和深入，就能够逐渐掌握更多的细节。
  - 寻找切入点的方式：离你最近的位置就是切入点，通常是业务代码中的最后一行。
  - 以 Retrofit 为例，最后的 `Call.enqueue()` 会被我作为切入点；在尝试从 `Call.enqueue()` 切入失败后，会退到 `Retrofit.create()` 方法，找到项目结构的核心，然后开始逐步发散和深入。
- 在阅读过程中，始终保有把看过的代码逻辑完整化的意识

    - 代码阅读过程中，不懂的代码会越来越多，脑子就会越来越乱。如果不断尝试把看到的代码结合起来组合成完整逻辑，就可以把多行或多段代码在脑子里（或者笔记里）组合成一整块，从而让代码结构更清晰，让阅读过程不断增加进度感，也减小继续阅读的难度。
    - 以 Retrofit 为例，当读到 `Proxy.newProxyInstance()` 方法实际上是创建了一个代理对象的时候，可以停下来做一个总结：「这是 Retrofit 的大框架」，在脑子里或者笔记上都可以。总结消化过后，继续阅读，就能顺畅一点。
- 尽量让每一刻都有一个确定的目标
    - 读代码经常会出现「横向逻辑还没看清晰，纵向深度也没挖透」的情况。到底是要横向扩展阅读结构，还是纵向挖深度，在每次遇到这种分岔路口的时候就先做好决定。不能在每个分岔路口都想也不想地看到不懂的就追下去，容易迷路。
    - 在遇到「横向也广，纵向也深」的时候，根据情况选择其中一个就好，并没有必然哪种选择更优的铁律。而如果遇到越钻越头大的情况，可以退回之前的某一步，换条路继续走。换路的时候记得做好标记，我在哪里探路失败了。

虽然这些思路与方式的东西，听人说不如自己做，但还是希望我的这些总结能够多少帮到大家一些。

## 作业

其实很想留作业，因为往往你听过之后再练练才会更容易记住，但这期内容不太适合留作业，那就请大家亲自去读一下 Retrofit 的源码吧。读的时候如果你觉得合适，可以试一试扔物线的心法。