

12 | 实战：用Kotlin实现一个网络请求框架KtHttp

2022-01-24 朱涛

《朱涛 · Kotlin编程第一课》

[课程介绍 >](#)



讲述：朱涛

时长 23:22 大小 21.41M



你好，我是朱涛，又到了实战环节。

在前面几节课当中，我们一起学习了 Kotlin 的委托、泛型、注解、反射这几个高级特性。那么今天这节课，我们将会运用这些特性，来写一个 **Kotlin 版本的 HTTP 网络请求框架**。由于它是纯 Kotlin 开发的，我们就把它叫做是 KtHttp 吧。

事实上，在 Java 和 Kotlin 领域，有许多出色的网络请求框架，比如 [OkHttp](#)、[Retrofit](#)、[Fuel](#)。而我们今天要实现的是 KtHttp，它的灵感来自于 Retrofit。之所以选择 Retrofit 作为借鉴的对象，是因为它的底层使用了大量的**泛型、注解和反射**的技术。如果你能跟着我一起用泛型、注解、反射来实现一个简单的网络请求框架，相信你对这几个知识点的认识也会更加透彻。

在这节课当中，我会带你从 0 开始实现这个网络请求框架。和往常一样，为了方便你理解，我们的代码会分为两个版本：

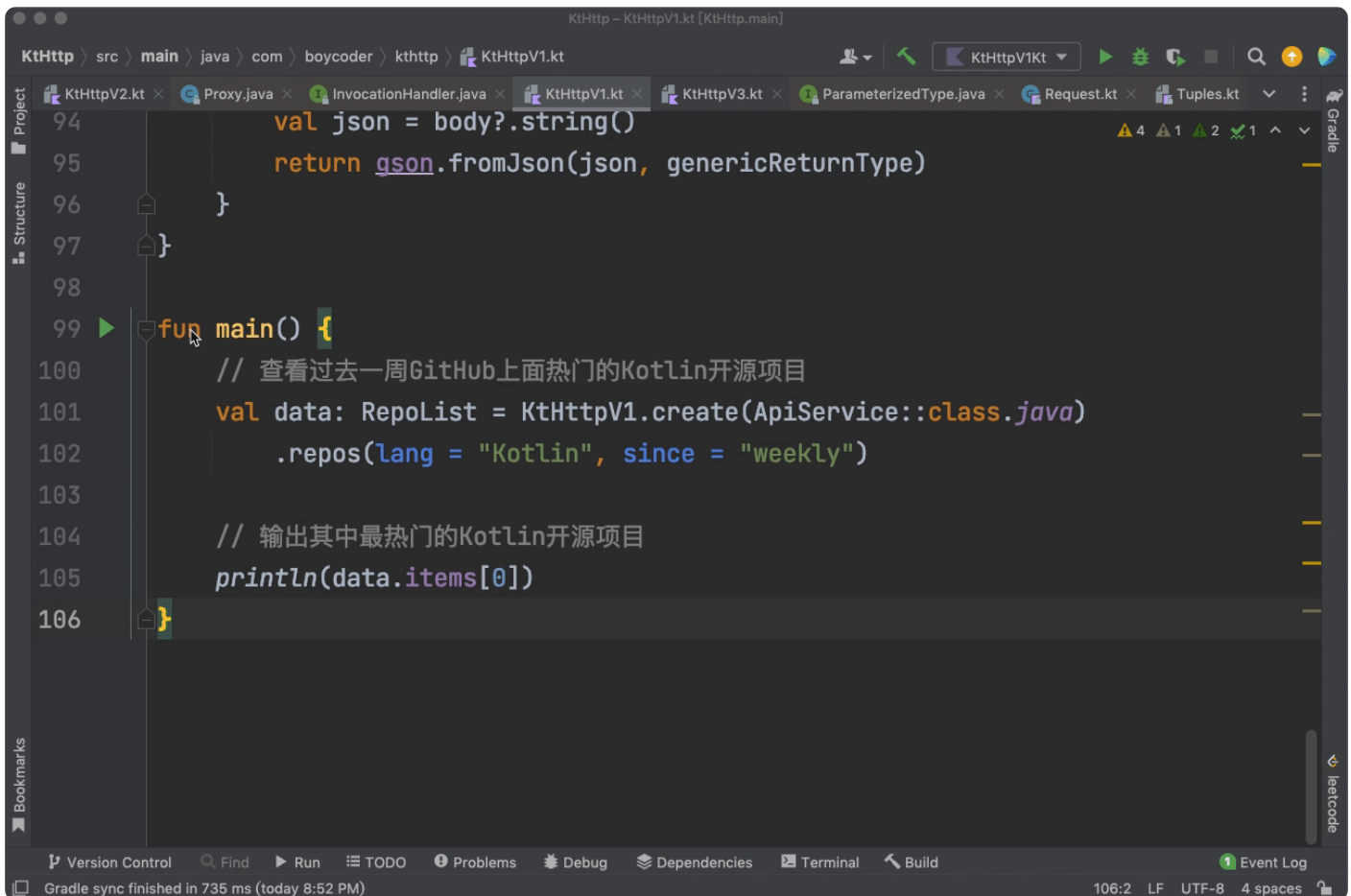
- 1.0 版本，我们会用 Java 思维，以最简单直白的方式来实现 KtHttp 的基础功能——同步式的 GET 网络请求；
- 2.0 版本，我们会用函数式思维来重构代码。

另外，在正式开始学习之前，我也建议你去 clone 我 GitHub 上面的 KtHttp 工程：

🔗 <https://github.com/chaxiu/KtHttp.git>，然后用 IntelliJ 打开，并切换到 **start** 分支跟着课程一步步敲代码。

1.0: Java 思维

在正式开始之前，我们还是先来看看程序的运行效果：



在上面的动图中，我们通过 KtHttp 请求了一个服务器的 API，然后在控制台输出了结果。这其实是我们在开发工作当中十分常见的需求。通过这个 KtHttp，我们就可以在程序当中访问任何服务器的 API，比如 🔗 [GitHub](#) 的 API。

那么，为了描述服务器返回的内容，我们定义了两个数据类：

```

1 // 这种写法是有问题的，但这节课我们先不管。
2
3 data class RepoList(
4     var count: Int?,
5     var items: List<Repo>?,
6     var msg: String?
7 )
8
9 data class Repo(
10     var added_stars: String?,
11     var avatars: List<String>?,
12     var desc: String?,
13     var forks: String?,
14     var lang: String?,
15     var repo: String?,
16     var repo_link: String?,
17     var stars: String?
18 )

```

除了数据类以外，我们还要定义一个用于网络请求的接口：

```

1 interface ApiService {
2     @GET("/repo")
3     fun repos(
4         @Field("lang") lang: String,
5         @Field("since") since: String
6     ): RepoList
7 }

```

在这个接口当中，有两个注解，我们一个个分析：

- **GET 注解**，代表了这个网络请求应该是 GET 请求，这是 [@HTTP](#) 请求的一种方式。GET 注解当中的“/repo”，代表了 API 的 path，它是和 baseUrl 拼接的；
- **Field 注解**，代表了 GET 请求的参数。Field 注解当中的值也会和 URL 拼接在一起。

也许你会好奇，**GET、Field 这两个注解是从哪里来的呢？**这其实也是需要我们自己定义的。根据上节课学过的内容，我们很容易就能写出下面的代码：

```

1 @Target(AnnotationTarget.FUNCTION)

```

```
2 @Retention(AnnotationRetention.RUNTIME)
3 annotation class GET(val value: String)
4
5 @Target(AnnotationTarget.VALUE_PARAMETER)
6 @Retention(AnnotationRetention.RUNTIME)
7 annotation class Field(val value: String)
```

从这段代码里我们可以看出，GET 注解只能用于修饰函数，Field 注解只能用于修饰参数。另外，这两个注解的 Retention 都是 AnnotationRetention.RUNTIME，这意味着这两个注解都是运行时可访问的。而这，也正好是我们后面要使用的反射的前提。

最后，我们再来看看 KtHttp 是如何使用的：


 复制代码

```
1 fun main() {
2     // ①
3     val api: ApiService = KtHttpV1.create(ApiService::class.java)
4
5     // ②
6     val data: RepoList = api.repos(lang = "Kotlin", since = "weekly")
7
8     println(data)
9 }
```

上面的代码有两个注释，我们分别来看。

- 注释①：我们调用 KtHttpV1.create() 方法，传入了 ApiService::class.java，参数的类型是 Class<T>，返回值类型是 ApiService。这就相当于创建了 ApiService 这个接口的实现类的对象。
- 注释②：我们调用 api.repos() 这个方法，传入了 Kotlin、weekly 这两个参数，代表我们想查询最近一周最热门的 Kotlin 开源项目。

看到这里，你也许会好奇，KtHttpV1.create() 是如何创建 ApiService 的实例的呢？要知道 ApiService 可是一个接口，我们要创建它的对象，必须先定义一个类实现它的接口方法，然后再用这个类来创建对象才行。

不过在这里，我们不会使用这种传统的方式，而是会用动态代理，也就是 JDK 的  Proxy。Proxy 的底层，其实也用到了反射。

不过，由于这个案例涉及到的知识点都很抽象，在正式开始编写逻辑代码之前，我们先来看看下面这个动图，对整体的程序有一个粗略的认识。

这是一个接口，api请求的关键信息存储在注解当中

```
interface ApiService {
    @GET("/repo")
    fun repos(
        @Field("lang") lang: String,
        @Field("since") since: String
    ): RepoList
}
```

现在，相信你大概就知道这个程序是如何实现的了。下面，我再带你来看看具体的代码是怎么写的。

这里我要先说明一点，为了不偏离这次实战课的主题，我们不会去深究 Proxy 的底层原理。在这里，你只需要知道，我们通过 Proxy，就可以动态地创建 ApiService 接口的实例化对象。具体的做法如下：

复制代码

```
1 fun <T> create(service: Class<T>): T {
2
3     // 调用 Proxy.newProxyInstance 就可以创建接口的实例化对象
4     return Proxy.newProxyInstance(
5         service.classLoader,
6         arrayOf<Class<*>>(service),
7         object : InvocationHandler{
8             override fun invoke(proxy: Any?, method: Method?, args: Array<out A
9                 // 省略
10         }
11     }
12 ) as T
13 }
```

在上面的代码当中，我们在 `create()` 方法当中，直接返回了 `Proxy.newProxyInstance()` 这个方法的返回值，最后再将其转换成了 `T` 类型。

那么，`newProxyInstance()` 这个方法又是如何定义的呢？

 复制代码

```
1 public static Object newProxyInstance(ClassLoader loader,
2                                     Class<?>[] interfaces,
3                                     InvocationHandler h){
4     ...
5 }
6
7 public interface InvocationHandler {
8     public Object invoke(Object proxy, Method method, Object[] args)
9         throws Throwable;
10 }
```

从上面的代码当中，我们可以看到，最后一个参数，`InvocationHandler` 其实是符合 `SAM` 转换要求的，所以我们的 `create()` 方法可以进一步简化成这样：

 复制代码

```
1 fun <T> create(service: Class<T>): T {
2
3     return Proxy.newProxyInstance(
4         service.classLoader,
5         arrayOf<Class<*>>(service)
6     ) { proxy, method, args ->
7         // 待完成
8     } as T
9 }
```

那么到这里，我们程序的基本框架也就搭建好了。

细心的你一定发现了，我们**程序的主要逻辑还没实现**，所以接下来，我们就一起看看上面那个“待完成”的 `InvocationHandler`，这个 `Lambda` 表达式应该怎么写。这个换句话说，也就是 `Proxy.newProxyInstance()`，会帮我们创建 `ApiService` 的实例对象，而 `ApiService` 当中的接口方法的具体逻辑，我们需要在 `Lambda` 表达式当中实现。

好了，让我们回过头来看看 `ApiService` 当中的代码细节：

```

1 interface ApiService {
2 // 假设我们的baseUrl是: https://baseUrl.com
3 // 这里拼接结果会是这样: https://baseUrl.com/repo
4 //           ↓
5     @GET("/repo")
6     fun repos(
7 //           Field注解当中的lang, 最终会拼接到url当中去
8 //           ↓
9         @Field("lang") lang: String, // https://baseUrl.com/repo?lang=Kotlin
10        @Field("since") since: String // https://baseUrl.com/repo?lang=Kotlin&s
11    ): RepoList
12 }

```

从代码注释中可以看出，其实我们真正需要实现的逻辑，就是想办法把注解当中的值 /repo、lang、since 取出来，然后拼接到 URL 当中去。那么，我们如何才能得到注解当中的值呢？

答案自然就是我们在上节课学过的：**反射**。

```

1 object KtHttpV1 {
2
3     // 底层使用 OkHttp
4     private var okHttpClient: OkHttpClient = OkHttpClient()
5     // 使用 Gson 解析 JSON
6     private var gson: Gson = Gson()
7
8     // 这里以baseUrl.com为例，实际上我们的KtHttpV1可以请求任意API
9     var baseUrl = "https://baseUrl.com"
10
11     fun <T> create(service: Class<T>): T {
12         return Proxy.newProxyInstance(
13             service.classLoader,
14             arrayOf<Class<*>>(service)
15             //           ①           ②
16             //           ↓           ↓
17         ) { proxy, method, args ->
18             // ③
19             val annotations = method.annotations
20             for (annotation in annotations) {
21                 // ④
22                 if (annotation is GET) {
23                     // ⑤
24                     val url = baseUrl + annotation.value
25                     // ⑥

```

```


26         return@newProxyInstance invoke(url, method, args!!)
27     }
28 }
29 return@newProxyInstance null
30
31 } as T
32 }
33
34 private fun invoke(url: String, method: Method, args: Array<Any>): Any? {
35     // 待完成
36 }
37 }

```

在上面的代码中，一共有 6 个注释，我们一个个看。

- 注释①：method 的类型是反射后的 Method，在我们这个例子当中，它最终会代表被调用的方法，也就是 ApiService 接口里面的 repos() 这个方法。
- 注释②：args 的类型是对象的数组，在我们的例子当中，它最终会代表方法的参数的值，也就是“api.repos("Kotlin", "weekly")”当中的“Kotlin”和“weekly”。
- 注释③：method.annotations，代表了我们会取出 repos() 这个方法上面的所有注解，由于 repos() 这个方法上面可能会有多个注解，因此它是数组类型。
- 注释④：我们使用 for 循环，遍历所有的注解，找到 GET 注解。
- 注释⑤：我们找到 GET 注解以后，要取出 @GET("/repo”) 当中的“/repo”，也就是“annotation.value”。这时候我们只需要用它与 baseUrl 进行拼接，就可以得到完整的 URL；
- 注释⑥：return@newProxyInstance，用的是 Lambda 表达式当中的返回语法，在得到完整的 URL 以后，我们将剩下的逻辑都交给了 invoke() 这个方法。

接下来，我们再来看看 invoke() 当中的“待完成代码”应该怎么写。

 复制代码

```

1 private fun invoke(url: String, method: Method, args: Array<Any>): Any? {
2     // ① 根据url拼接参数，也就是：url + ?lang=Kotlin&since=weekly
3     // ② 使用okHttpClient进行网络请求
4     // ③ 使用gson进行JSON解析
5     // ④ 返回结果
6 }

```


在上面的代码中，我们的 `invoke()` 方法一共分成了四个步骤，其中的③、④两个步骤其实很容易实现：

 复制代码

```
1 private fun invoke(url: String, method: Method, args: Array<Any>): Any? {
2     // ① 根据url拼接参数，也就是: url + ?lang=Kotlin&since=weekly
3
4     // 使用okHttpClient进行网络请求
5     val request = Request.Builder()
6         .url(url)
7         .build()
8     val response = okHttpClient.newCall(request).execute()
9
10    // ② 获取repos()的返回值类型 genericReturnType
11
12    // 使用gson进行JSON解析
13    val body = response.body
14    val json = body?.string()
15    //                                     根据repos()的返回值类型解析JSON
16    //                                     ↓
17    val result = gson.fromJson<Any?>(json, genericReturnType)
18
19    // 返回结果
20    return result
21 }
```

继续看，经过我们的分解，现在的问题变成了下面这样：

- 注释①，利用反射，解析出“`api.repos("Kotlin", "weekly")`”这个方法当中的“`Kotlin`”和“`weekly`”，将其与 URL 进行拼接得到：`url + ?lang=Kotlin&since=weekly`
- 注释②，利用反射，解析出 `repos()` 的返回值类型，用于 JSON 解析。

我们来看看最终的代码：

 复制代码

```
1 private fun invoke(path: String, method: Method, args: Array<Any>): Any? {
2     // 条件判断
3     if (method.parameterAnnotations.size != args.size) return null
4
5     // 解析完整的url
6     var url = path
7     // ①
```

```

8      val parameterAnnotations = method.parameterAnnotations
9      for (i in parameterAnnotations.indices) {
10         for (parameterAnnotation in parameterAnnotations[i]) {
11             // ②
12             if (parameterAnnotation is Field) {
13                 val key = parameterAnnotation.value
14                 val value = args[i].toString()
15                 if (!url.contains("?")) {
16                     // ③
17                     url += "?$key=$value"
18                 } else {
19                     // ④
20                     url += "&$key=$value"
21                 }
22             }
23         }
24     }
25 }
26 // 最终的url会是这样:
27 // https://baseurl.com/repo?lang=Kotlin&since=weekly
28
29 // 执行网络请求
30 val request = Request.Builder()
31     .url(url)
32     .build()
33 val response = okHttpClient.newCall(request).execute()
34
35 // ⑤
36 val genericReturnType = method.genericReturnType
37 val body = response.body
38 val json = body?.string()
39 // JSON解析
40 val result = gson.fromJson<Any?>(json, genericReturnType)
41
42 // 返回值
43 return result
44 }

```

上面的代码一共涉及五个注释，它们都是跟注解与反射这两个知识点相关的。

- 注释①，`method.parameterAnnotations`，它的作用是取出方法参数当中的所有注解，在我们这个案例当中，`repos()` 这个方法当中涉及到两个注解，它们分别是`@Field("lang")`、`@Field("since")`。
- 注释②，由于方法当中可能存在其他注解，因此要筛选出我们想要的 `Field` 注解。
- 注释③，这里是取出注解当中的值“`lang`”，以及参数当中对应的值“`Kotlin`”进行拼接，URL 第一次拼接参数的时候，要用“`?`”分隔。

- 注释④，这里是取出注解当中的值“since”，以及参数当中对应的值“weekly”进行拼接，后面的参数拼接格式，是用“&”分隔。
- 注释⑤，`method.genericReturnType` 取出 `repos()` 的返回值类型，也就是 `RepoList`，最终，我们用它来解析 JSON。

说实话，动态代理的这种模式，由于它大量应用了反射，加之我们的代码当中还牵涉到了泛型和注解，导致这个案例的代码不是那么容易理解。不过，我们其实可以利用**调试**的手段，去查看代码当中每一步执行的结果，这样就能对注解、反射、动态代理有一个更具体的认识。

前面带你看过的这个动图，其实就是在向你展示代码在调试过程中的关键节点，我们可以再来回顾一下整个代码的执行流程：

这是一个接口，api请求的关键信息存储在注解当中

```
interface ApiService {
    @GET("/repo")
    fun repos(
        @Field("lang") lang: String,
        @Field("since") since: String
    ): RepoList
}
```

相信现在，你已经能够体会我们使用 **动态代理 + 注解 + 反射** 实现这个网络请求框架的原因了。通过这样的方式，我们就不必在代码当中去实现每一个接口，而是只要是符合这样的代码模式，任意的接口和方法，我们都可以直接传进去。在这个例子当中，我们用的是 `ApiService` 这个接口，如果下次我们定义了另一个接口，比如说：

复制代码

```
1 interface GitHubService {
2     @GET("/search")
3     fun search(
4         @Field("id") id: String
```

```
5     ): User
6 }
```

这时候，我们的 `KtHttp` 根本不需要做任何改动，直接这样调用即可：

 复制代码

```
1 fun main() {
2     KtHttpV1.baseUrl = "https://api.github.com"
3     //          换一个接口名即可          换一个接口名即可
4     //          ↓          ↓
5     val api: GitHubService = KtHttpV1.create(GitHubService::class.java)
6     val data: User = api.search(id = "JetBrains")
7 }
```

可以发现，使用动态代理实现网络请求的优势，它的**灵活性**是非常好的。只要我们定义的 `Service` 接口拥有对应的注解 `GET`、`Field`，我们就可以通过注解与反射，将这些信息拼凑在一起。下面这个动图就展示了它们整体的流程：

```
var baseUrl = "https://baseurl.com"

interface ApiService {
    @GET("/repo")
    fun repos(
        @Field("lang") a: String,
        @Field("since") b: String
    ): RepoList
}

fun main() {
    val api: ApiService = KtHttpV1.create(ApiService::class.java)
    val data: RepoList = api.repos("Kotlin", "weekly")
}
```

实际上，我们的 `KtHttp`，就是将 `URL` 的信息存储在了注解当中（比如 `lang` 和 `since`），而实际的参数值，是在函数调用的时候传进来的（比如 `Kotlin` 和 `weekly`）。我们通过泛型、注解、反射的结合，将这些信息集到一起，完成整个 `URL` 的拼接，最后才通过 `OkHttp` 完成的网络请求、`Gson` 完成的解析。

好，到这里，我们 1.0 版本的开发就算是完成了。这里的单元测试代码很容易写，我就不贴出来了，**单元测试是个好习惯，我们不能忘。**

接下来，我们正式进入 2.0 版本的开发。

2.0：函数式思维

其实，如果你理解了 1.0 版本的代码，2.0 版本的程序也就不难实现了。因为这个程序的主要功能都已经完成了，现在要做的只是：**换一种思路重构代码。**

我们先来看看 KtHttpV1 这个单例的成员变量：

 复制代码

```
1 object KtHttpV1 {
2     private var okHttpClient: OkHttpClient = OkHttpClient()
3     private var gson: Gson = Gson()
4
5     fun <T> create(service: Class<T>): T {}
6     fun invoke(url: String, method: Method, args: Array<Any>): Any? {}
7 }
```

okHttpClient、gson 这两个成员是不支持懒加载的，因此我们首先应该让它们**支持懒加载**。

 复制代码

```
1 object KtHttpV2 {
2     private val okHttpClient by lazy { OkHttpClient() }
3     private val gson by lazy { Gson() }
4
5     fun <T> create(service: Class<T>): T {}
6     fun invoke(url: String, method: Method, args: Array<Any>): Any? {}
7 }
```

这里，我们直接使用了 **by lazy** 委托的方式，它简洁的语法可以让我们快速实现懒加载。

接下来，我们再来看看 **create()** 这个方法的定义：

 复制代码

```
1 //          注意这里
2 //
```

↓

```

3 fun <T> create(service: Class<T>): T {
4     return Proxy.newProxyInstance(
5         service.classLoader,
6         arrayOf<Class<*>>(service)
7     ) { proxy, method, args ->
8     }
9 }

```

在上面的代码中，`create()` 会接收一个 `Class<T>` 类型的参数。其实，针对这样的情况，我们完全可以省略掉这个参数。具体做法，是使用我们前面学过的 `inline`，来实现**类型实化**（**Reified Type**）。我们常说，Java 的泛型是伪泛型，而这里我们要实现的就是真泛型。

 复制代码

```

1 // 注意这两个关键字
2 // ↓           ↓
3 inline fun <reified T> create(): T {
4     return Proxy.newProxyInstance(
5         T::class.java.classLoader, // ① 变化在这里
6         arrayOf(T::class.java) // ② 变化在这里
7     ) { proxy, method, args ->
8         // 待重构
9     }
10 }

```

正常情况下，泛型参数 `类型会被擦除`，这就是 Java 的泛型被称为“伪泛型”的原因。而通过使用 **inline** 和 **reified** 这两个关键字，我们就能实现类型实化，也就是“真泛型”，进一步，我们就可以在代码注释①、②的地方，使用“`T::class.java`”来得到 `Class` 对象。

下面，我们来看看 `KtHttp` 的主要逻辑该如何重构。

为了方便理解，我们会使用 Kotlin 标准库当中已有的高阶函数，尽量不去涉及函数式编程里的高级概念。**在这里我强烈建议你打开 IDE 一边敲代码一边阅读**，这样一来，当你遇到不熟悉的标准函数时，就可以随时去看它的实现源码了。相信在学习过第 7 讲的 `高阶函数` 以后，这些库函数都不会难倒你。

首先，我们来看看 `create()` 里面“待重构”的代码该如何写。在这个方法当中，我们需要读取 `method` 当中的 `GET` 注解，解析出它的值，然后与 `baseUrl` 拼接。这里我们完全可以借助 **Kotlin 的标准库函数**来实现：

```

1 inline fun <reified T> create(): T {
2     return Proxy.newProxyInstance(
3         T::class.java.classLoader,
4         arrayOf(T::class.java)
5     ) { proxy, method, args ->
6
7         return@newProxyInstance method.annotations
8             .filterIsInstance<GET>()
9             .takeIf { it.size == 1 }
10            ?.let { invoke("$baseUrl${it[0].value}", method, args) }
11    } as T
12 }

```

这段代码的可读性很好，我们可以像读英语文本一样来阅读：

- 首先，我们通过 `method.annotations`，来获取 `method` 的所有注解；
- 接着，我们用 `filterIsInstance<GET>()`，来筛选出我们想要找的 `GET` 注解。这里的 `filterIsInstance` 其实是 `filter` 的升级版，也就是过滤的意思；
- 之后，我们判断 `GET` 注解的数量，它的数量必须是 `1`，其他的都不行，这里的 `takeIf` 其实相当于我们的 `if`；
- 最后，我们通过拼接出 `URL`，然后将程序执行流程交给 `invoke()` 方法。这里的 `"?.let{}"` 相当于判空。

好了，`create()` 方法的重构已经完成，接下来我们来看看 `invoke()` 方法该如何重构。

```

1 fun invoke(url: String, method: Method, args: Array<Any>): Any? =
2     method.parameterAnnotations
3         .takeIf { method.parameterAnnotations.size == args.size }
4         ?.mapIndexed { index, it -> Pair(it, args[index]) }
5         ?.fold(url, ::parseUrl)
6         ?.let { Request.Builder().url(it).build() }
7         ?.let { okHttpClient.newCall(it).execute().body?.string() }
8         ?.let { gson.fromJson(it, method.genericReturnType) }

```

这段代码读起来也不难，我们一行一行来分析。

- 第一步，我们通过 `method.parameterAnnotations`，获取方法当中所有的参数注解，在这里也就是 `@Field("lang")`、`@Field("since")`。
- 第二步，我们通过 `takeIf` 来判断，参数注解数组的数量与参数的数量相等，也就是说 `@Field("lang")`、`@Field("since")` 的数量是 2，那么 `["Kotlin", "weekly"]` 的 `size` 也应该是 2，它必须是一一对应的关系。
- 第三步，我们将 `@Field("lang")` 与 "Kotlin" 进行配对，将 `@Field("since")` 与 "weekly" 进行配对。这里的 `mapIndexed`，其实就是 `map` 的升级版，它本质还是一种映射的语法，“注解数组类型”映射成了“Pair 数组”，只是多了一个 `index` 而已。
- 第四步，我们使用 `fold` 与 `parseUrl()` 这个方法，拼接出完整的 URL，也就是：
<https://baseurl.com/repo?lang=Kotlin&since=weekly>。这里我们使用了函数引用的语法 `::parseUrl`。而 `fold` 这个操作符，其实就是高阶函数版的 `for` 循环。
- 第五步，我们构建出 `OkHttp` 的 `Request` 对象，并且将 URL 传入了进去，准备做网络请求。
- 第六步，我们通过 `okHttpClient` 发起了网络请求，并且拿到了 `String` 类型的 JSON 数据。
- 最后，我们通过 `Gson` 解析出 JSON 的内容，并且返回 `RepoList` 对象。

到目前为止，我们的 `invoke()` 方法的主要流程就分析完了，接下来我们再来看看用于实现 URL 拼接的 `parseUrl()` 是如何实现的。

 复制代码

```
1 private fun parseUrl(acc: String, pair: Pair<Array<Annotation>, Any>) =
2     pair.first.filterIsInstance<Field>()
3         .first()
4         .let { field ->
5             if (acc.contains("?")) {
6                 "$acc&${field.value}=${pair.second}"
7             } else {
8                 "$acc?${field.value}=${pair.second}"
9             }
10        }
```

可以看到，这里我们只是把从前的 `for` 循环代码，换成了 **Kotlin 的集合操作符**而已。大致流程如下：

- 首先，我们从注解的数组里筛选出 `Field` 类型的注解；

- 接着，通过 `first()` 取出第一个 `Field` 注解，这里它也应该是唯一的；
- 最后，我们判断当前的 `acc` 是否已经拼接过参数，如果没有拼接过，就用“?”分隔，如果已经拼接过参数，我们就用“&”分隔。

至此，我们 2.0 版本的代码就完成了，完整的代码如下：

 复制代码

```
1 object KtHttpV2 {
2
3     private val okHttpClient by lazy { OkHttpClient() }
4     private val gson by lazy { Gson() }
5     var baseUrl = "https://baseurl.com" // 可改成任意url
6
7     inline fun <reified T> create(): T {
8         return Proxy.newProxyInstance(
9             T::class.java.classLoader,
10             arrayOf(T::class.java)
11         ) { proxy, method, args ->
12
13             return@newProxyInstance method.annotations
14                 .filterIsInstance<GET>()
15                 .takeIf { it.size == 1 }
16                 ?.let { invoke("$baseUrl${it[0].value}", method, args) }
17         } as T
18     }
19
20     fun invoke(url: String, method: Method, args: Array<Any>): Any? =
21         method.parameterAnnotations
22             .takeIf { method.parameterAnnotations.size == args.size }
23             ?.mapIndexed { index, it -> Pair(it, args[index]) }
24             ?.fold(url, ::parseUrl)
25             ?.let { Request.Builder().url(it).build() }
26             ?.let { okHttpClient.newCall(it).execute().body?.string() }
27             ?.let { gson.fromJson(it, method.genericReturnType) }
28
29
30     private fun parseUrl(acc: String, pair: Pair<Array<Annotation>, Any>) =
31         pair.first.filterIsInstance<Field>()
32             .first()
33             .let { field ->
34                 if (acc.contains("?")) {
35                     "$acc&${field.value}=${pair.second}"
36                 } else {
37                     "$acc?${field.value}=${pair.second}"
38                 }
39             }
40 }
```

对应的，我们可以再看看 1.0 版本的完整代码：

 复制代码

```
1 object KtHttpV1 {
2
3     private var okHttpClient: OkHttpClient = OkHttpClient()
4     private var gson: Gson = Gson()
5     var baseUrl = "https://baseurl.com" // 可改成任意url
6
7     fun <T> create(service: Class<T>): T {
8         return Proxy.newProxyInstance(
9             service.classLoader,
10             arrayOf<Class<*>>(service)
11         ) { proxy, method, args ->
12             val annotations = method.annotations
13             for (annotation in annotations) {
14                 if (annotation is GET) {
15                     val url = baseUrl + annotation.value
16                     return@newProxyInstance invoke(url, method, args!!)
17                 }
18             }
19             return@newProxyInstance null
20
21         } as T
22     }
23
24     private fun invoke(path: String, method: Method, args: Array<Any>): Any? {
25         if (method.parameterAnnotations.size != args.size) return null
26
27         var url = path
28         val parameterAnnotations = method.parameterAnnotations
29         for (i in parameterAnnotations.indices) {
30             for (parameterAnnotation in parameterAnnotations[i]) {
31                 if (parameterAnnotation is Field) {
32                     val key = parameterAnnotation.value
33                     val value = args[i].toString()
34                     if (!url.contains("?")) {
35                         url += "?$key=$value"
36                     } else {
37                         url += "&$key=$value"
38                     }
39                 }
40             }
41         }
42
43         val request = Request.Builder()
44             .url(url)
45             .build()
46
47         val response = okHttpClient.newCall(request).execute()
```

```

49         val genericReturnType = method.genericReturnType
50         val body = response.body
51         val json = body?.string()
52         val result = gson.fromJson<Any?>(json, genericReturnType)
53
54         return result
55     }
56 }
57 }

```

可见，1.0 版本、2.0 版本，它们之间可以说是天壤之别。

小结

好了，这节实战就到这里。接下来我们来简单总结一下：

- 在 1.0 版本的代码中，我们灵活利用了**动态代理**、**泛型**、**注解**、**反射**这几个技术，实现了 KtHttp 的基础功能。
- **动态代理**，由于它的底层原理比较复杂，课程当中我是通过 `ApiImpl` 这个类，来模拟了它动态生成的 `Proxy` 类。用这种直观的方式来帮助你理解它存在的意义。
- **泛型**方面，我们将其用在了动态代理的 `create()` 方法上，后面我们还使用了“类型实化”的技术，也就是 `inline + reified` 关键字。
- **注解**方面，我们首先自定义了两个注解，分别是 `GET`、`Field`。其中，`@GET` 用于标记接口的方法，它的值是 URL 的 `path`；`@Field` 用于标记参数，它的值是参数的 `key`。
- **反射**方面，这个技术点，几乎是贯穿于整个代码实现流程的。我们通过反射的自省能力，去分析 `repos()` 方法，从 `GET` 注解当中取出了“/repo”这个 `path`，从注解 `Field` 当中取出了 `lang`、`since`，还取出了 `repos()` 方法的返回值 `RepoList`，用于 JSON 数据的解析。
- 在 2.0 版本的代码中，我们几乎删除了之前所有的代码，**以函数式的思维重写了** KtHttp 的内部逻辑。在这个版本当中，我们大量地使用了 Kotlin 标准库里的高阶函数，进一步提升了代码的可读性。

在前面的 [🍷加餐](#)课程当中，我们也讨论过 Kotlin 的编程范式问题。**命令式还是函数式，这完全取决于我们开发者自身。**

相比起前面实战课中的 [🍷单词频率统计程序](#)，这一次我们的函数式范式的代码，实现起来就没有那么得流畅了。原因其实也很简单，Kotlin 提供了强大的集合操作符，这就让 Kotlin 十分擅

长“集合操作”的场景，因此词频统计程序，我们不到 10 行代码就解决了。而对于注解、反射相关的场景，函数式的编程范式就没那么擅长了。


在这节课里，我之所以费尽心思地用函数式风格，重构出 KtHttp 2.0 版本，主要还是想让你看到函数式编程在它不那么擅长的领域表现会如何。毕竟，我们在工作中什么问题都可能会遇到。

思考题

好了，学完这节课以后，请问你有哪些感悟和收获？请在评论区里分享出来，我们一起交流吧！

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 11 | 注解与反射：进阶必备技能

[下一篇](#) 加餐二 | 什么是“表达式思维”？

精选留言 (15)

 写留言



面无表情的生鱼片

2022-02-12

请教老师，如果 `method.genericType` 是 kotlin 的 Basic Type 的话（例如：`String`、`Int`），要怎么做兼容比较好呢？

作者回复: 在工作中，有时候确实会遇到String类型的需求。要解决这个问题，我们只需要将JSON解析相关逻辑抽离出去，然后将逻辑交给业务层去实现即可。



白乾涛

2022-02-19

勉强能看明白，但这代码谁能手写的出来呀？

就算写出来了，谁保证没 **bug**？谁能保证别人能看明白？谁能保证后续能维护？

作者回复: 是的，这也取决于团队的默契程度。



木易杨

2022-01-24

Kotlin这语法越写越变态。**Java**啰嗦吧，起码能看懂，没那么多语法题

作者回复: 别怕，适应了就好。另外，等到了源码篇，我也会集中分析一波Kotlin的高阶函数的原理和意义，请留意哈。



阿康

2022-01-24

Lambda 表达式当中的返回语法 能讲下吗？或者给个相关的博客连接

作者回复: 其实很好理解：

`return@newProxyInstance`代表返回Lambda；而直接的`return`，代表了返回`create()`这个函数。

参考链接: <https://kotlinlang.org/docs/lambdas.html#underscore-for-unused-variables>

共 3 条评论 >



\$Kotlin

2022-01-24

动图看起来不太方便，不能暂停，而且这个动图好长。

作者回复: 后面的长动图我会改成视频形式哈。至于.....这节课的动图，就辛苦你多看几遍啦。



河山

2022-03-09

请问老师 像如下代码

```
fun <T> Int.toType():T{
    return (this as T)
}
class Animal{}
fun main() {
    println(100.toType<Animal>())
}
```

这个不应该有类型转换异常吗 为什么我运行没有报异常 而且会输出100 但是debug模式 去运行100.toType<Animal>() 这个表达式 却的确会提示类型转换异常 老师 为什么运行没问题啊

作者回复: 如果你将上面的代码进行反编译就会发现问题了。其中主要的原因还是在于Kotlin的泛型是伪泛型。这里我们调用asT的时候, 其实它只是做了一个object强转, 所以并不会出问题。如果你将代码改成这样, 就肯定会在运行时出现崩溃了:

```
...
fun <T> Int.toType():T{
    return (this as T)
}
class Animal{
    // 变化在这里
    fun getName() = "Animal"
}
fun main() {
    println(100.toType<Animal>().getName())
}
...
```



syz

2022-03-09

动态代理的那张动图, 播放中不能暂停, 要懂这样过一遍没毛病。建议将每一次停顿变成带序号的标注, 贴代码上来感觉会好点。

作者回复: 好的, 感谢你的建议, 后续课程中比较长的动图我都做成视频了, 随时可以暂停的。



DeBlue



POPiUS

2022-02-27

操作符太多了，日常写业务不常用的话很快就忘了。不知道老师是如何知道这么多没听过的操作符（`filterIsInstance`、`fold`）。

作者回复: 其实还是靠练习，熟能生巧。我会在工作里尽量用上来，而在工作之余，我也会写点其他练手的代码，也会刷点算法题之类的。

共 2 条评论 >



山河入梦

2022-02-16

// 这种写法是有问题的，但这节课我们先不管。

我想问下老师，这种写法的问题在哪，因为我一直这样写来着，从昨天看了文章，就一直纠结着

作者回复: 改进的方向是：我们应该尽可能消灭数据类的可空性（加餐四有提到）。具体来说，应该使用非空类型，具体怎么做，我会在后面的课程里提到哈。



jim

2022-02-15

kotlin确实很优雅，有时候写着写着看不懂了！

作者回复: 确实，所以要多练习，多适应。



梦佳

2022-01-31

运行不起来

作者回复: 程序运行需要下载Gradle的依赖，需要一些科学上网的手段。（如果解决了上网问题还是不行的话，可以把错误的日志发出来给我看看。）



只为你停留

2022-01-28

```
mapIndexed { index, it -> Pair(it, args[index]) }
```

这个函数中 `it -> Pair(it, args[index])` 怎么理解呢，尤其不理解 `it ->`

作者回复: 这里其实是对`method.parameterAnnotations`当中每一个注解进行map, 所以: { index: Int, it: Annotation-> ... }

共 3 条评论 >



l-zesong

2022-01-25

`return@newProxyInstance` 是什么意思啊? 没看懂

作者回复: 它代表了返回Lambda的函数体, 也就是退出`InvocationHandler`的`invoke`方法。

具体语法可以看这里: <https://kotlinlang.org/docs/lambda.html#returning-a-value-from-a-lambda-expression>



sunlight

2022-01-25

有个地方疑惑, 动态代理一般会有两种使用方式吗?

方式一 `create()`方法中会多传个被代理对象, 通过`method.invoke(被代理对象)`, 实现拦截。外层返回代理对象

方式二 `create()`方法中只会有接口, 没有手动实现被代理对象。因为我们不关心接口的具体实现, 只关心接口中的注解参数, 拦截获取到参数即可

文中是使用第二种, 并没有手动实现被代理对象, 只是最终返回了代理对象。请问这样理解对么

作者回复: 你的理解是对的。不过, 如果你想深入了解动态代理部分的内容, 你去搜索一下“Java Proxy 动态代理”即可, 这是一个单独的Java知识点。



7Promise

2022-01-24

深奥的东西在经过学习原理后都是会有恍然开朗的感觉。

作者回复: 这也说明你在进步, 真替你感到高兴, 加油~



