# Retrofit

## 概览

- **Retrofit** 是将 API 接口转换为可调用对象的工具。
- 本章用到的所有依赖如下
  - retrofit2 : retrofit
  - retrofit2 : converter-gson
  - gson : gson

## 发送简单的 GET 请求

- 现有一个后端接口 `localhost:8080/test/currentTime`，请求后会返回当前时间（毫秒时，Long 型）。
- 我们在项目根下新建 `Contracts` 单例类用于保存全局常量，并将公共请求前缀写入：

```
// Contracts
object Contracts {
    // 不要忘记在最后补上一个斜杠
    const val BASE_URL = "https://$LOCAL_HOST:$PORT/test/"
    private const val LOCAL_HOST = "192.168.18.3"
    private const val PORT = "8080"
}
```

- 在项目根目录下新建包 `data.api` 用于存放 `Retrofit` 接口，然后在该包下新建 `TestApi` 接口：

```
// data.api.TestApi
interface TestApi {
    // 在这里定义你的第一个请求
    @GET("currentTime")
    suspend fun currentTime(): Call<Long>
}
```

> `@GET` 表明该请求是个GET请求，它接收参数 `value:String` 表示具体的请求地址

- 然后我们可以在当前接口的伴生类中使用 `Retrofit.Builder` 实例化一个默认的对象：

```
// data.api.TestApi
interface TestApi {
    // ...
    companion object {
        val Default = Retrofit.Builder()
            .baseUrl(Contracts.BASE_URL)
            .build()
            .create(TestApi::class.java)
            //或者 ".create<TestApi>()"
    }
}
```

- 现在我们在来测试这个接口：

```
class TestActivity : AppCompatActivity() {
    private val testApi = TestApi.Default
    private lateinit var button: Button
    private lateinit var textView: TextView
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        button = findViewById<Button>(R.id.button)
        textView = findViewById<TextView>(R.id.textView)
        button.setOnClickListener {
            lifecycleScope.launch(Dispatcher.IO) {
                val call = testApi.currentTime()
                call.enqueue(object : Callback<Long> {
                    override fun onResponse(
                        call: Call<Long>,
                        response: Response<Long>
                    ) {
                        textView.text = "${response.body()}ms"
                    }
                    override fun onFailure(call: Call<Long>, t: Throwable) {
                        Log.e(TAG, "onCreate", t)
                    }
                })
            }
        }
    }
}
```

## 处理 API 接口返回的 JSON 数据

- 现有一个后端接口 `localhost:8080/test/user/random` ，请求后会以JSON格式返回随机用户简略信息，比如：

```
{
    "userId": "40019",
    "name": "李亮",
    "lastOnlineAt": 1651620000000
}
```

- 我们先定义一个 `UserDTO` 数据类：

```
// data.dto.UserDTO
data class UserDTO(
    @SerializedName("userId") val id: Int,
    val name: String,
    val lastOnlineAt: Long
)
```

> 我们不需要将JSON的全部内容定义为字段，按需即可

- 在 `UserApi` 中定义该请求：

```
//data.api.UserApi
interface UserApi {
    @GET("user/random")
    suspend fun getRandomUser(): User

    companion object {
        val Default = Retrofit.Builder()
            .baseUrl(Contracts.BASE_URL)
            // 使用Gson转换器
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(UserApi::class.java)
    }
}
```

- 后端返回的 `lastOnlineAt` 是我们当前不需要的，我们可以定义一个 `User` 数据类，然后为 `UserDTO` 定义一个扩展方法 `toUser(): User` ：

```kotlin
// domain.entity.User
data class User(
    val id: Int,
    val name: String
)
```

```kotlin
// data.dto.UserDTO
data class UserDTO (...)

fun UserDTO.toUser(): User = User(id, name)
```

- 测试这个接口：

```kotlin
// ui.TestActivity
class TestActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        lifecycleScope.launch(Dispatcher.IO) {
            val user = try {
                userApi.getRandomUser().toUser()
            } catch (e: Exception) {
                Log.e(TAG, "onCreate:", e)
                null
            }
            withContext(Dispatcher.Main) {
                showUser(user)
            }
        }
    }

    @MainThread
    private fun deliverUser(user: User?) {
        textView.text = user?.name ?: "Network Error."
    }
}
```

## 使用 @Path 和 @Query 传入参数

- 现有一个API接口 `localhost:8080/test/dog/{count}` ， `count` 传入一个整数，后端会以GSON数组的格式随机返回狗狗的简略信息，例如当 `count` 为 **2** 时：

```
[
    {
        "name": "Kitty",
        "age": 2,
        "sex": "female"
    },
    {
        "name": "Lucio",
        "age": 5,
        "sex": "male"
    }
]
```

- 定义数据类 `DogDTO`

```
// data.dto.DogDTO
data class DogDTO (
    val name: String,
    val age: Int,
    val sex: String
)
```

- 定义接口 `DogApi` ，通过 `@Path` 注解实现动态请求：

```
// data.api.DogApi
interface DogApi {
    @GET("dog/{count}")
    suspend fun getDogs(@Path("count") count: Int = 1): List<DogDTO>
}
```

- 如果API接口需要传入的参数包含**键值对**，如 `localhost:8080/test/dog/{count}?age={age}&sex={sex}` ，那么就需要用到 `@Query` 注解：

```
// data.api.DogApi
interface DogApi {
    @GET("dog/{count}")
    suspend fun getDogs(
        @Path("count") count: Int = 1,
        @Query("age") age: Int,
        @Query("sex") sex: String
    ): List<DogDTO>
}
```

## 为请求添加请求头信息

- 使用 `@Headers` 注解静态添加请求头信息

```
@Headers(
    "X-RapidAPI-Host: ${Contracts.X_RapidAPI_Host}",
    "X-RapidAPI-Key: ${Contracts.X_RapidAPI_Key}"
)
@GET("search/{name}")
suspend fun searchCards(@Path("name") name: String): List<Card>
```

- 使用 `@Header` 注解动态添加请求头信息

```
@GET("search/{name}")
suspend fun searchCards(
    @Header("key") key: String,
    @Path("name") name: String
): List<Card>
```

- 使用 `OKHttpClient` 设置全局请求头信息
    - 实例化 `OKHttpClient` 为一个顶级变量：

```
    val client = OkHttpClient.Builder()
                .addInterceptor {
                    val request = it.request()
                        .newBuilder()
                        // 在这里添加默认头信息
                        .addHeader("X-RapidAPI-Host",
    Contracts.X_RapidAPI_Host)
                        .addHeader("X-RapidAPI-Key",
    Contracts.X_RapidAPI_Key)
                        .build()
                    it.proceed(request)
                }.build()
```

- 在每次构造 `Retrofit` 的时候设置其client为上面这个顶级变量即可：

```
    Retrofit.Builder()
        .baseUrl(Contracts.BASE_URL)
        .client(client)
        .build()
```

## 上传一个文件到云端

- 现有一个API接口 `localhost:8080/upload` ，定义 **FileUploadService** 接口

```
interface FileUploadService {
    @Multipart
    @POST("upload")
    suspend fun postFile(
        @Part("file\"; filename=\"photo.jpg\" ") file: RequestBody
    ): Call<String>

    companion object {
        val Default = Retrofit.Builder()
            .baseUrl(Contracts.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create<FileUploadService>()
    }
}
```

- 调用这个接口

```
```

```
val requestBody = RequestBody.create(MediaType.parse("image/jpeg"), photoFile)
service.postFile(requestBody).enqueue(object : Callback<String> {
    override fun onResponse(
        call: Call<String>,
        response: Response<String>
    ) {
        // 更新成功信息到UI
    }

    override fun onFailure(call: Call<String>, t: Throwable) {
        Log.e(TAG, "postFile", t)
        // 更新错误信息到UI
    }
})
```

## 上传多个文件到云端

- 现有一个API接口 `localhost:8080/upload/files` ，定义 **FileUploadService** 接口

```
interface FileUploadService {
    // ...
    @Multipart
    @POST("upload/files")
    suspend fun postFiles(
        @PartMap Map<String, RequestBody> files,
        @Part("json") String description
    ): Call<String>
    // ...
}
```

- 构建一个 `String` 映射到 `RequestBody` 的 `MutableMap` 用于存放多个文件：

```
val files = mutableMapOf<String, RequestBody>()
```

- 将每个 `Uri` 分别构建为 `RequestBody` ，然后同文件名映射到 **files** 中：

```
uris.map(::File).forEach {
    val requestBody = it.asRequestBody("multipart/form-data".toMediaType())
    val fileName = it.name
    files.put(fileName, requestBody)
}
```

- 随后调用接口：

```
  service.postFiles(files, "string request").enqueue(object : Callback<String>
{
    override fun onResponse(
        call: Call<String>,
        response: Response<String>
    ) {
        // 更新成功信息到UI
    }

    override fun onFailure(call: Call<String>, t: Throwable) {
        Log.e(TAG, "postFile", t)
        // 更新错误信息到UI
    }
})
```

## 从云端下载文件

- 现有一个API接口 `localhost:8080/test/download/{fileName}` ，根据 **fileName** 的值返回对应文件，定义 **FileDownloadService** 接口

```
interface FileDownloadService {
    @GET("download/{filename}")
    suspend fun downloadByFileName(
        @Path("fileName") fileName: String
    ): Call<ResponseBody>

    companion object {
        val Default = Retrofit.Builder()
            .baseUrl(Contracts.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create<FileDownloadService>()
    }
}
```

- 调用这个接口：

```
service.downloadByFileName("1.png").enqueue(object : Callback<ResponseBody> {
    override fun onResponse(
        call: Call<ResponseBody>,
```

```
            response: Response<ResponseBody>
        ) {
            if (response.isSuccessful()) {
                val body = response.body()
                // 用你的方法将body写入磁盘
            }
        }

        override fun onFailure(call: Call<ResponseBody>, t: Throwable) {
            Log.e(TAG, "downloadByFileName", t)
            // 更新错误信息到UI
        }
    })
```