Table of Contents

项目简介

Ktorfit X 是一个基于 Kotlin Multiplatform 和 Ktor 构建的跨平台网络请求框架，旨在通过注解简化 RESTful API 的声明与实现。它支持客户端与服务端的代码自动生成，帮助开发者减少样板代码，提高开发效率。

依赖以及平台版本

- **Kotlin** `2.2.0`
- **Ktor**： `3.2.3`
- **KSP**： `2.2.0-2.0.2`
- **JVM** `21`

核心特点

- **多平台支持**： 支持 Android、iOS、watchOS、tvOS、Js、WasmJs、 macOS、Linux、Windows 等平台。
- **注解驱动**： 通过注解定义 API 接口与服务端路由，使用 KSP 自动生成实现类与 Ktor 路由注册代码。
- **支持 WebSocket、Mock、Timeout、Header/Cookie/Path 等常见参数类型**
- **服务端授权支持**： 内置 `@Authentication` 注解支持路由授权处理。

模块组成

Kotlin Multiplatform 客户端模块

模块名	功能
multiplatform-core	客户端核心功能
multiplatform-annotation	注解声明
multiplatform-websockets	WebSocket 支持
multiplatform-mock	Mock 支持
multiplatform-ksp	客户端 KSP 生成器

Ktor Server 服务端模块

模块名	功能
server-core	服务端核心支持
server-annotation	注解声明
server-websockets	WebSocket 支持
server-auth	授权支持
server-ksp	服务端 KSP 生成器

通用模块

模块名	功能
common-ksp-util	通用符号处理器工具

全面拥抱 3.x，从 2.x 迁移到 3.x

- 原 `cn.vividcode.multiplatform` 包名已迁移为 `cn.ktorfitx`，旧的依赖包已停止维护 ⚠️
- 新增 Ktor Server 目标平台

编译期校验

Ktorfit X 提供编译期间的错误校验支持，能够在使用错误的注解或结构时给出明确提示，帮助用户快速定位问题。

异常处理

- 当函数返回类型为 `Result<T>` 时，Ktorfit X 会自动捕获并封装网络异常；如果返回值为普通类型，则需自行处理异常逻辑。

Kotlin Multiplatform

在 Kotlin Multiplatform 中，接口生成的实现类在当前包下的 `impls` 包中， 扩展名为当前类名称的小驼峰形式

支持平台

平台	架构 / Target 编译类型
Android	JVM – androidTarget (JVM 21)
Desktop	JVM – jvm("desktop") (JVM 21)
iOS	iosX64, iosArm64, iosSimulatorArm64 (Native)
macOS	macosX64, macosArm64 (Native)
watchOS	watchosX64, watchosArm32, watchosArm64, watchosSimulatorArm64, watchosDeviceArm64 (Native)
tvOS	tvosX64, tvosArm64, tvosSimulatorArm64 (Native)
Linux	linuxX64, linuxArm64 (Native executables)
Windows	mingwX64 (Native executable via MinGW)
Js	js(IR) – outputModuleName + nodejs()
WasmJs	wasmJs (ExperimentalWasmDsl + nodejs())

全部注解

目标 ANNOTATION_CLASS

- `@HttpMethod` (@HttpMethod)： 支持自定义 HTTP 方法类型，用于非标准请求方法，需标注在自定义注解上

目标 CLASS

- `@Api` (@Api)： 定义服务接口类型，作为生成 REST 风格 Ktor 请求代码的标识
- `@ApiScope` (@ApiScope)： 定义生成接口的作用域，控制扩展方法中泛型的类型以规定此服务作用于指定 ktorfitx 配置

目标 FUNCTION

- `@GET、@POST、@PUT、@DELETE、@PATCH、@OPTIONS、@HEAD` (HTTP 请求 @GET, @POST...)： HTTP 请求
- `@BearerAuth` (@BearerAuth)： 启用 Bearer token 授权方案，自动加入 Authorization 请求头
- `@Headers` (@Headers)： 声明静态请求头，生成时自动注入多个 headers
- `@Mock` (@Mock)： 定义 Mock 行为逻辑，用于测试或演示 Mock 数据
- `@WebSocket` (@WebSocket)： 将接口函数定义为 WebSocket 通道处理入口
- `@Timeout` (@Timeout)： 配置请求的超时时间

目标 VALUE_PARAMETER

- `@Body` (@Body)： 设置请求体内容，支持配置多种序列化类型
- `@Query` (@Query)： 将参数映射为 URL 查询参数（?key=value）
- `@Field`： 以 x-www-form-urlencoded 格式提交表单字段

- `@Part`： 表示 multipart/form-data 的表单部分，用于文件上传或复杂表单
- `@Header`： 动态注入 HTTP 请求头参数
- `@Path`： 映射路径参数，对 URL 模板中的 `{}` 片段进行替换
- `@Cookie`： 从 Cookie 中提取参数值注入接口
- `@Attribute`： 用于传递请求属性（attribute），可携带元数据或上下文信息

注意事项⚠️

- `@Body`、 `@Field`、 `@Part` 这三个注解不能在同一个请求中同时使用

@Api

定义服务接口类型，作为生成 REST 风格 Ktor 请求代码的标识

代码示例

```
/**
 * 在这里使用 @Api 注解
 */
@Api(url = "user")
interface UserApi {

    @POST(url = "login")
    suspend fun login(
        @Field username: String,
        @Field password: String
    ): LoginDTO
}

@Serializable
data class LoginDTO(
    val token: String
)
```

生成实现

以下代码将会自动生成，您只需要通过 `Ktorfitx<DefaultApiScope>` 的扩展属性 `userApi` 获取实例即可

```
private class UserApiImpl private constructor(
    private val config: KtorfitxConfig,
) : UserApi {

    override suspend fun login(username: String, password: String):
LoginDTO {
        val response = this.config.httpClient.post {
```

```
            this.url("user/login")
            this.contentType(ContentType.Application.FormUrlEncoded)
            this.setBody(listOf("username" to username, "password" to
password).formUrlEncode())
        }
        return response.body()
    }

    @OptIn(InternalAPI::class)
    companion object {

        private var defaultApiScopeInstance: UserApi? = null

        private val defaultApiScopeSynchronizedObject:
SynchronizedObject = SynchronizedObject()

        fun getInstanceByDefaultApiScope(ktorfitx:
Ktorfitx<DefaultApiScope>): UserApi = defaultApiScopeInstance ?:
synchronized(defaultApiScopeSynchronizedObject) {
            defaultApiScopeInstance ?:
UserApiImpl(ktorfitx.config).also { defaultApiScopeInstance = it }
        }
    }
}

val Ktorfitx<DefaultApiScope>.userApi: UserApi
    @JvmName("userApiByDefaultApiScope")
    get() = UserApiImpl.getInstanceByDefaultApiScope(this)
```

注意事项 ⚠️

除 `@Api` 和 `@ApiScope` 文档，其余文档将不展示接口和类的代码，以方便阅读查看

@ApiScope

定义生成接口的作用域，控制扩展方法中泛型的类型以规定此服务作用于指定 ktorfitx 配置

代码示例

```
/**
 * 在这里使用 @ApiScope 注解，并传递 自定义的 UserScope 作用域
 */
@ApiScope(scopes = [UserScope::class])
@Api(url = "user")
interface UserApi

sealed interface UserScope
```

生成实现

注意，这里生成的 `userApi` 是 `Ktorfitx<UserScope>` 的扩展属性，而不是默认的 `Ktorfitx<DefaultApiScope>` 的了

```
private class UserApiImpl private constructor(
    private val config: KtorfitxConfig,
) : UserApi {

    @OptIn(InternalAPI::class)
    companion object {

        private var userScopeInstance: UserApi? = null

        private val userScopeSynchronizedObject: SynchronizedObject =
SynchronizedObject()

        fun getInstanceByUserScope(ktorfitx: Ktorfitx<UserScope>):
UserApi = userScopeInstance ?:
synchronized(userScopeSynchronizedObject) {
            userScopeInstance ?: UserApiImpl(ktorfitx.config).also {
```

```
userScopeInstance = it }
        }
    }
}

val Ktorfitx<UserScope>.userApi: UserApi
    @JvmName("userApiByUserScope")
    get() = UserApiImpl.getInstanceByUserScope(this)
```

注意事项 ⚠️

除 `@Api` 和 `@ApiScope` 文档，其余文档将不展示接口和类的代码，以方便阅读查看

@HttpMethod

支持自定义 HTTP 方法类型，用于非标准请求方法，需标注在自定义注解上

示例代码

在这里使用 `@HttpMethod` 注解，method 参数默认会使用自定义注解的注解名称

```
@HttpMethod(method = "CUSTOM")
@Retention(AnnotationRetention.SOURCE)
@Target(AnnotationTarget.FUNCTION)
annotation class CUSTOM(
    val url: String
)
```

注意事项

- 必须标记 `@Retention` 注解，并将值设置为 `AnnotationRetention.SOURCE`
- 必须标记 `@Target` 注解，并将值设置为 `AnnotationTarget.FUNCTION`
- 必须添加一个名称为 `url` 类型为 `String` 的参数
- `@HttpMethod` 的 `method` 默认值为自定义注解名称

HTTP 请求 @GET, @POST...

HTTP 请求

请求类型

注解	请求类型	说明
`@GET`	GET	用于获取资源或数据，不应包含请求体。常用于查询或读取操作
`@POST`	POST	用于创建资源或提交数据。支持请求体（如 JSON、表单等）
`@PUT`	PUT	用于整体更新资源，需提供完整的资源内容
`@DELETE`	DELETE	用于删除指定资源，常用于数据删除操作
`@PATCH`	PATCH	用于部分更新资源，适合修改资源中的部分字段
`@HEAD`	HEAD	类似 GET 请求，但不返回响应体。用于获取资源的元信息（如是否存在）
`@OPTIONS`	OPTIONS	获取服务器支持的通信选项。常用于 CORS 预检请求

代码示例

在这里使用 `@GET` 注解并设置 `url`，生成的 url 会自动拼接上 `@Api` 上提供的 `url`

```
@BearerAuth
@GET(url = "userDetail/query")
suspend fun queryUserDetail(): UserDetailDTO
```

你也可以使用 `http://` 或 `https://` 前缀的 url

```
@BearerAuth
@POST(url = "https://ktorfitx.cn:8080/api/userDetail/update")
suspend fun updateUserDetail(
    @Field key: String,
    @Field value: String?
): Boolean
```

生成实现

```
override suspend fun queryUserDetail(): UserDetailDTO {
    val token = this.config.token?.invoke()
    val response = this.config.httpClient.get {
        this.url("query")
        if (token != null) {
            this.bearerAuth(token)
        }
    }
    return response.body()
}
```

```
override suspend fun updateUserDetail(key: String, value: String?):
Boolean {
    val token = this.config.token?.invoke()
    val response = this.config.httpClient.post {
        this.url("userDetail/update")
        if (token != null) {
            this.bearerAuth(token)
        }
        this.contentType(ContentType.Application.FormUrlEncoded)
        this.setBody(listOf("key" to key, "value" to
value).formUrlEncode())
    }
    return response.body()
}
```

@BearerAuth

启用 Bearer token 授权方案，自动加入 Authorization 请求头

配置 token

```
val ktorfitx = ktorfitx {
    /**
     * 在这里配置获取 token
     */
    token { "<token>" }

    // 省略其他代码
}
```

示例代码

```
@BearerAuth
@GET(url = "query")
suspend fun query(): UserDetailDTO
```

生成实现

在生成的代码中会首先生成获取 token 的代码，然后判空后调用 ktor 提供的 `this.bearerAuth(token)` 方法

这里的 `this.config` 是通过生成类的构造参数获取，具体代码可以查看 `@Api` (@Api) 的文档

```
override suspend fun query(): UserDetailDTO {
    val token = this.config.token?.invoke()
    val response = this.config.httpClient.get {
        this.url("query")
        if (token != null) {
            this.bearerAuth(token)
        }
    }
```

```
    return response.body()
}
```

@Headers

声明静态请求头，生成时自动注入多个 headers。

示例代码

```
@Headers(
    "X-App-Name: KtorfitxDocument",
    "X-Locale: zh-CN"
)
@GET(url = "info")
suspend fun fetchInfo(): Info
```

生成实现

```
override suspend fun fetchInfo(): Info {
    val response = this.config.httpClient.`get` {
        this.url("info")
        this.headers {
            this.append("X-App-Name", "KtorfitxDocument")
            this.append("X-Locale", "zh-CN")
        }
    }
    return response.body()
}
```

@Mock

定义 Mock 行为逻辑，用于测试或演示 Mock 数据

示例代码

这里使用 `@Mock` 注解，并设置 `provider` 参数

```
@POST(url = "friend/{friendId}")
@Mock(provider = FriendMockProvider::class)
suspend fun fetchFriend(
    @Path friendId: Int,
): FriendDTO
```

定义 `FriendMockProvider` Mock 提供者，必须是 `object` 类型

`MockProvider<R>` 的泛型必须和请求函数的返回类型一致，如果返回类型是 `Result<T>` 则需要和类型 `T` 保持一致

```
object FriendMockProvider : MockProvider<FriendDTO> {

    override fun provide(): FriendDTO {
        return TODO()
    }
}
```

生成实现

```
override suspend fun fetchFriend(friendId: Int): FriendDTO =
this.config.mockClient.request(
    method = HttpMethod.Post,
    mockProvider = FriendMockProvider,
) {
    this.url("friend/${friendId}")
    this.paths {
        this.append("friendId", friendId)
```

```
    }
}
```

注意事项 ⚠️

目前暂不支持在 provide() 函数中获取请求参数，计划在未来版本中支持，但目前还未设计如何实现此功能

@WebSocket

将接口函数定义为 WebSocket 通道处理入口

代码示例

在这里使用 `@WebSocket` 注解，并设置 `WebSocketSessionHandler` 类型，他是 `suspend DefaultClientWebSocketSession.() -> Unit` 类型的别名

```
@BearerAuth
@WebSocket(url = "friendMessageSocket")
suspend fun friendMessageSocket(handler: WebSocketSessionHandler)
```

你也可以使用 `ws://` 或者 `wss://` 前缀的 url，当然你也可以直接使用 `suspend DefaultClientWebSocketSession.() -> Unit` 类型

```
@WebSocket("ws://ktorfitx.cn:8080/api/keepAlive")
suspend fun keepAlive(handler: suspend DefaultClientWebSocketSession.
() -> Unit)
```

生成实现

```
override suspend fun friendMessageSocket(handler:
WebSocketSessionHandler) {
    val token = this.config.token?.invoke()
    this.config.httpClient.webSocket(
        urlString = "friendMessageSocket",
        request = {
            if (token != null) {
                this.bearerAuth(token)
            }
        },
        block = handler
    )
}
```

```
override suspend fun keepAlive(handler: suspend
DefaultClientWebSocketSession.() -> Unit) {
    this.config.httpClient.webSocket(
        urlString = "ws://ktorfitx.cn:8080/api/keepAlive",
        block = handler
    )
}
```

@Timeout

配置请求超时时间，用于控制方法的最长执行时长

示例代码

在这里使用 `@Timeout` 注解，配置：请求超时时间、连接超时时间 和 套接字超时时间

```
@Timeout(
    requestTimeoutMillis = 10_000L,
    connectTimeoutMillis = 10_000L,
    socketTimeoutMillis = 10_000L
)
@BearerAuth
@GET(url = "user/info")
suspend fun getUserInfo(): UserInfo
```

生成实现

在 `@Timeout` 中配置的参数会在实现中生成 `this.timeout { }`

```
override suspend fun getUserInfo(): UserInfo {
    val token = this.config.token?.invoke()
    val response = this.config.httpClient.`get` {
        this.url("user/info")
        this.timeout {
            this.requestTimeoutMillis = 10_000L
            this.connectTimeoutMillis = 10_000L
            this.socketTimeoutMillis = 10_000L
        }
        if (token != null) {
            this.bearerAuth(token)
        }
    }
```

```
    return response.body()
}
```

@Body

设置请求体内容，支持配置多种序列化类型

示例代码

`@Body` 注解可用于设置请求体的序列化方式。通过 `format` 参数，可以指定使用的 `SerializationFormat` ，从而生成对应的 `ContentType` 类型的请求体

可选的 `SerializationFormat` 类型及对应的 `ContentType` 如下：

- `SerializationFormat.JSON` 对应 `ContentType.Application.Json` 默认值
- `SerializationFormat.XML` 对应 `ContentType.Application.Xml`
- `SerializationFormat.CBOR` 对应 `ContentType.Application.Cbor`
- `SerializationFormat.PROTO_BUF` 对应 `ContentType.Application.ProtoBuf`

在这里使用 `@Body` 注解，这里为 `format` 设置了 `SerializationFormat.XML` 值

```
@BearerAuth
@POST(url = "user/detail/update")
suspend fun updateUserDetail(
    @Body(format = SerializationFormat.XML) data: UserDetail
): Boolean
```

生成的实现

在实现中这里为 `contentType()` 函数设置了 `ContentType.Application.Xml` 值，对应注解中设置的 `SerializationFormat.XML`

```
override suspend fun updateUserDetail(`data`: UserDetail): Boolean {
    val token = this.config.token?.invoke()
    val response = this.config.httpClient.post {
        this.url("user/detail/update")
        if (token != null) {
            this.bearerAuth(token)
```

```
        }
        this.contentType(ContentType.Application.Xml)
        this.setBody(`data`)
    }
    return response.body()
}
```

@Query

将参数映射为 URL 查询参数（?key=value）

示例代码

在这里使用 `@Query` 注解，此注解目的是为了将参数映射成 URL 查询参数，生成 Ktor 提供的 `this.parameter()` 函数

它有个 `name` 参数，用来设置查询参数的键，默认为变量名，混淆不会影响结果，因为这是编译时处理的

```
@BearerAuth
@GET(url = "friend/infos")
suspend fun queryFriendInfos(
    @Query pageSize: Int,
    @Query(name = "custom") pageNumber: Int
): List<FriendInfo>
```

生成实现

```
override suspend fun queryFriendInfos(pageSize: Int, pageNumber:
Int): List<FriendInfo> {
    val token = this.config.token?.invoke()
    val response = this.config.httpClient.`get` {
        this.url("friend/infos")
        if (token != null) {
            this.bearerAuth(token)
        }
        this.parameter("pageSize", pageSize)
        this.parameter("custom", pageNumber)
    }
    return response.body()
}
```