Mirai-console 插件教程

My First Plugin - kotlin 教程 Mar 14 2020

前言

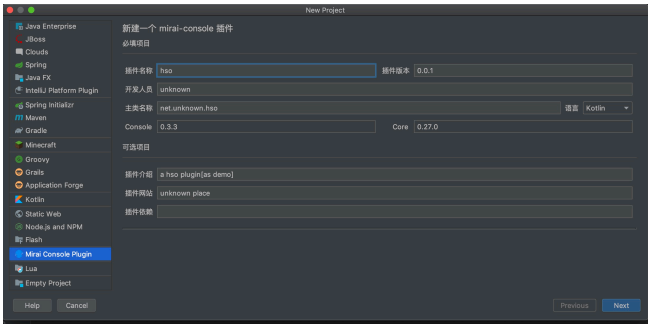
本篇教程基于你已经阅读了Mirai-console 插件开发中的如何上手, 将着重于如何实际的写出第一个有一定功能性的插件, 阅读本文不需要有很了解kotlin

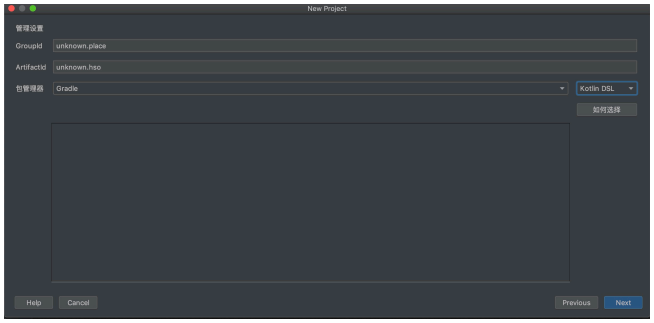
文章所开发的是一个群内涩图插件, 仅供学习参考, 公开使用请注意所在国法律与社会价值观

本文共涉及两个插件项目, 完整版工程项目源码均可在github中找到

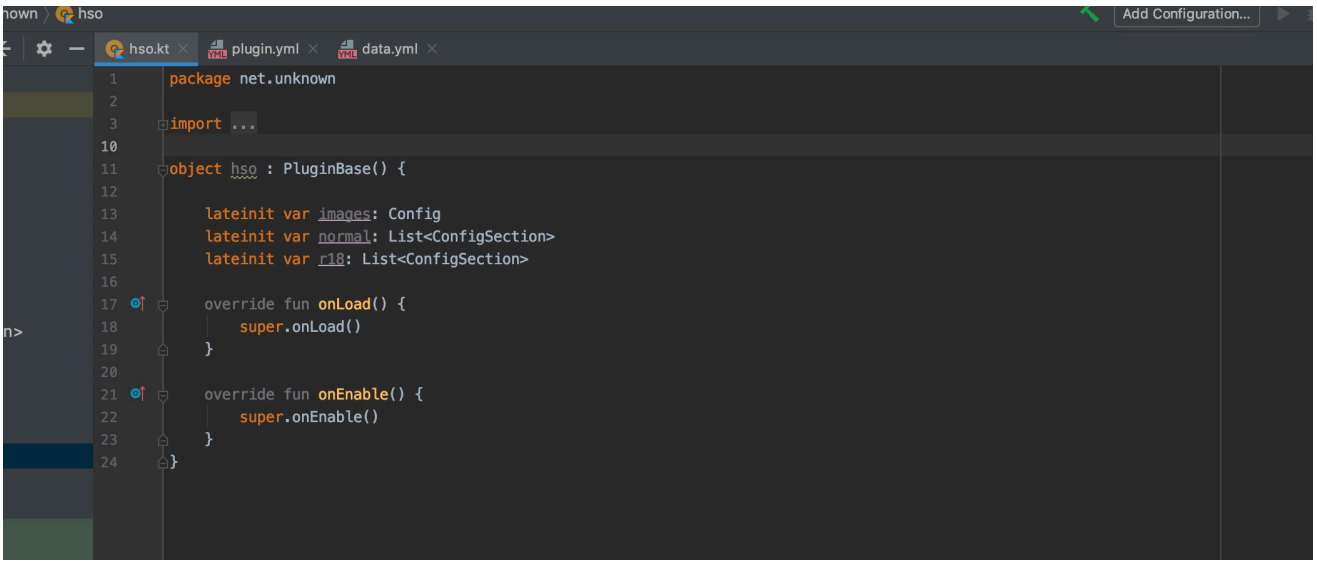
新建项目

根据上文所说, 在idea中添加插件, 启动一个插件项目， 我们给插件起名为hso, 并将开发者名设置为unknown





[Console/Core]版本可能在你阅读本文时已有更新, 请不要在意



（hso作为object名应该是Hso, 这是一个不规范的例子）

在插件项目建立完成后, 你也应当会看到类似的初始代码

onLoad() onEnable() 是插件开发中的两个核心方法, 也是插件生命周期中重要的部分, 简单的说, 当一个插件被加载, *onLoad*方法会被先调用, 全部插件的*onLoad*方法被调用后, *onEnable*的方法会被依次调用

onLoad() 正如方法名, 是插件在加载时应做的逻辑, 例如

1: 加载本地资源, 测试本地资源是否正常(或更之前)

2: 加载配置文件(或更之前)

3: 初始化一些变量

但以下逻辑不推荐或不能做:

1: 检查依赖插件是否加载

2: 注册指令

3: 注册事件监听

原因是因为你的插件在被调用*onLoad()*时, 其他插件还没有进入正式的生命周期

onEnable() 是插件启动时应做的逻辑, 如

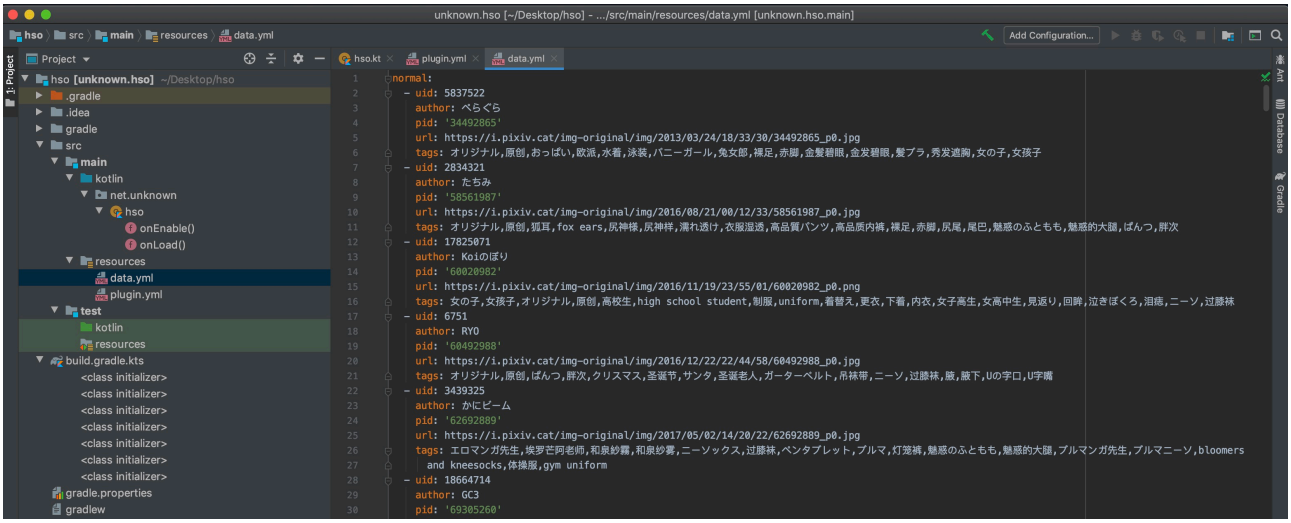
1: 注册指令

2: 注册事件监听

3: 启动插件中的循环任务, 协程, 或*worker*

两个方法均不应该出现**长时间的堵塞**, 短时间的阻塞, 如通过*http*检查最新版本是允许并推荐的, 对于这种情况, 您应当在*onLoad()*时异步请求, *onEnable*的时候等待结果, 已寻求最短的阻塞时间。

对于一个涩图插件, 图源是十分重要的, 本文采用的是 *https://api.lolicon.app/#/setu* 的涩图api, 但鉴于该站不是很稳定且过量的重复请求极为浪费, 我们以极慢的速度获取了该站的1000余张涩图数据, 将其保存在一个YAML文件, 并根据分级储存为normal/r18两档, 并将文件放到/resources/下



配置文件是插件中的重要部分

Console为插件提供了方便的配置文件API用于格式化读写文件, 插件支持了yaml, json, toml, ini(properties)等文件格式

配置文件总的来说分为两种, 可读文件与读写文件
可读的配置文件是你放在/resources/中的文件, 这类文件读取出来后只可读 不可保存

可写的配置文件是你生成在_ROOT_/plugins/插件名/ 下的配置文件, 这类文件插件可读可写, 且用户可读可写。

如果要读取一个*resources*中的配置文件, *PluginBase*中的*getResourcesConfig(fileName)* 提供了支持, 如果要读取(如不存在则创建)一个可读可写的配置文件, *PluginBase*中的 *loadConfig(fileName)* 提供了支持

*Config*是线程安全的

我们在onLoad方法中将/resources/中的涩图数据读取出来并进行储存到内存中, 并进行简单的数据统计与错误处理

```
import ...

object hso : PluginBase() {

    lateinit var images: Config
    lateinit var normal: List<ConfigSection>
    lateinit var r18: List<ConfigSection>

    override fun onLoad() {
        logger.info("loading local image data")
        try {
            images = getResourcesConfig( fileName: "data.yml")
        } catch (e: Exception) {
            e.printStackTrace()
            logger.info("无法加载本地图片")
        }
        logger.info("本地图片版本" + images.getString( key: "version"))

        r18 = images.getConfigSectionList( key: "R18")
        normal = images.getConfigSectionList( key: "normal")
        logger.info("Normal * " + normal.size)
        logger.info("R18    * " + r18.size)
    }

    override fun onEnable() {
        super.onEnable()
    }
}
```

Config 与 ConfigSection

*Config*和*ConfigSection*均为键值对应的数据结构, 他的数据结构类似于*Map<String, Any!>*, 其中键一定是*String*请牢记, *Config*与*ConfigSection*中提供了高效的获取方式, 将*Any!*转为其他数据格式, 如*getLong() getDouble() getLongList() getStringList()*等, 但这不足以满足复杂的需求, 因此出现了*ConfigSection*

ConfigSection也是一个*Map<String, Any!>*, 且可以作为一个*Any!* 被放入其他的*Config/ConfigSection*, 这样套娃, 你就可以解决99%的场景需求

以这个插件为例, *data.yml*(涩图数据文件), 就是一个使用了*Config/ConfigSection*的场景, 整个文件读取为*Config*后, 只有两个键, *normal*和*r18*, 而它们对应的则各是一个*ConfigSection List*, 因此满足了这个复杂场景的需要

在处理完成涩图数据后, 我们来对编写插件的配置文件(即用户可以自定义的东西)

```
object hso : PluginBase() {

    lateinit var images: Config
    lateinit var normal: List<ConfigSection>
    lateinit var r18: List<ConfigSection>

    val config = loadConfig( fileName: "setting.yml")

    val Normal_Image_Trigger by config.withDefaultWriteSave { "色图" }
    val R18_Image_Trigger by config.withDefaultWriteSave { "不够色" }
    val Image_Resize_Max_Width_Height by config.withDefaultWriteSave { 800 }

    override fun onLoad() {
        logger.info("loading local image data")
        try {
            images = getResourcesConfig( fileName: "data.yml")
        } catch (e: Exception) {
            e.printStackTrace()
            logger.info("无法加载本地图片")
        }
        logger.info("本地图片版本" + images.getString( key: "version"))

        r18 = images.getConfigSectionList( key: "R18")
        normal = images.getConfigSectionList( key: "normal")
        logger.info("Normal * " + normal.size)
        logger.info("R18    * " + r18.size)
    }

    override fun onEnable() {
        super.onEnable()
    }
}
```

首先在18行, 我们加载了一个config, 这个“setting.yml”会被创建在用户目录/plugins/hso(插件名)/下， 以供用户使用, 接着我们定义了三个简单类型的变量, 他们使用了Config提供的代理模式

Config 代理模式

Config 提供了kotlin的特性-代理(by)功能, 可以轻松的连接配置文件与代码

Var aValue by config.withDefaultWriteSave{0}的作用简单来说, 当aValue被使用时, 他会在config中寻找键为aValue(变量名)的数据, 如果没有, 则返回0, 并在配置文件写入aValue=0并保存, 当你给aValue赋值的时候, 配置文件中的数据会更改, 但不会保存

因此变量命名就变的比较重要, 这里不推荐使用驼峰命名, 推荐使用大写首字母与下划线的方式, 毕竟不是所有用户都看得懂驼峰的

与withDefaultWriteSave类似的还有

with	如果没有找到对应数据, 就会error
withDefault{}	如果没有找到对应数据, 就会返回default数值
withDefaultWrite{}	如果没有找到对应数据, 只会写入不会保存

Config代理模式仅支持基础数据格式, 如Int/Long/Double 或LongList DoubleList等
不支持ConfigSection/ ConfigSectionList
且读写list 不推荐使用代理

```kotlin
import ...

object hso : PluginBase() {

    lateinit var images: Config
    lateinit var normal: List<ConfigSection>
    lateinit var r18: List<ConfigSection>

    val config = loadConfig( fileName: "setting.yml")

    val Normal_Image_Trigger by config.withDefaultWriteSave { "色图" }
    val R18_Image_Trigger by config.withDefaultWriteSave { "不够色" }
    val Image_Resize_Max_Width_Height by config.withDefaultWriteSave { 800 }


    val groupsAllowNormal = config.getLongList( key: "Allow_Normal_Image_Groups").toMutableList()
    val groupsAllowR18 = config.getLongList( key: "Allow_R18_Image_Groups").toMutableList()

    override fun onLoad() {
        logger.info("loading local image data")
        try {
            images = getResourcesConfig( fileName: "data.yml")
        } catch (e: Exception) {
            e.printStackTrace()
            logger.info("无法加载本地图片")
        }
        logger.info("本地图片版本" + images.getString( key: "version"))

        r18 = images.getConfigSectionList( key: "R18")
        normal = images.getConfigSectionList( key: "normal")
        logger.info("Normal * " + normal.size)
        logger.info("R18    * " + r18.size)
    }

    override fun onEnable() {
        super.onEnable()
    }
```

紧接着, 我们读取出允许发送普通图片与r18图片的群组群号列表, 这里没有使用by, 因此在插件关闭时将它写回去是十分重要的

```
object hso : PluginBase() {

    lateinit var images: Config
    lateinit var normal: List<ConfigSection>
    lateinit var r18: List<ConfigSection>

    val config = loadConfig( fileName: "setting.yml")

    val Normal_Image_Trigger by config.withDefaultWriteSave { "色图" }
    val R18_Image_Trigger by config.withDefaultWriteSave { "不够色" }
    val Image_Resize_Max_Width_Height by config.withDefaultWriteSave { 800 }

    val groupsAllowNormal = config.getLongList( key: "Allow_Normal_Image_Groups").toMutableList()
    val groupsAllowR18 = config.getLongList( key: "Allow_R18_Image_Groups").toMutableList()

    override fun onDisable() {
        super.onDisable()
        logger.info("Saving data")
        config["Allow_Normal_Image_Groups"] = groupsAllowNormal
        config["Allow_R18_Image_Groups"]    = groupsAllowR18
    }

    override fun onLoad() {
        logger.info("loading local image data")
        try {
            images = getResourcesConfig( fileName: "data.yml")
        } catch (e: Exception) {
            e.printStackTrace()
            logger.info("无法加载本地图片")
        }
        logger.info("本地图片版本" + images.getString( key: "version"))

        r18 = images.getConfigSectionList( key: "R18")
        normal = images.getConfigSectionList( key: "normal")
        logger.info("Normal * " + normal.size)
```

onDisable() 与 *onLoad() onEnable()* 类似, 是在插件要被关闭前调用的

如果这样写, 你就会遇到一个问题, 因为你没有保存文件，写了白写

```
26
27      override fun onDisable() {
28          super.onDisable()
29          logger.info("Saving data")
30          config["Allow_Normal_Image_Groups"] = groupsAllowNormal
31          config["Allow_R18_Image_Groups"]    = groupsAllowR18
32          config.save()
33      }
34
```

Config.save()方法将config中的数据写入文件中, 这个方法实际是阻塞的, 但为了方便开发, 它不是

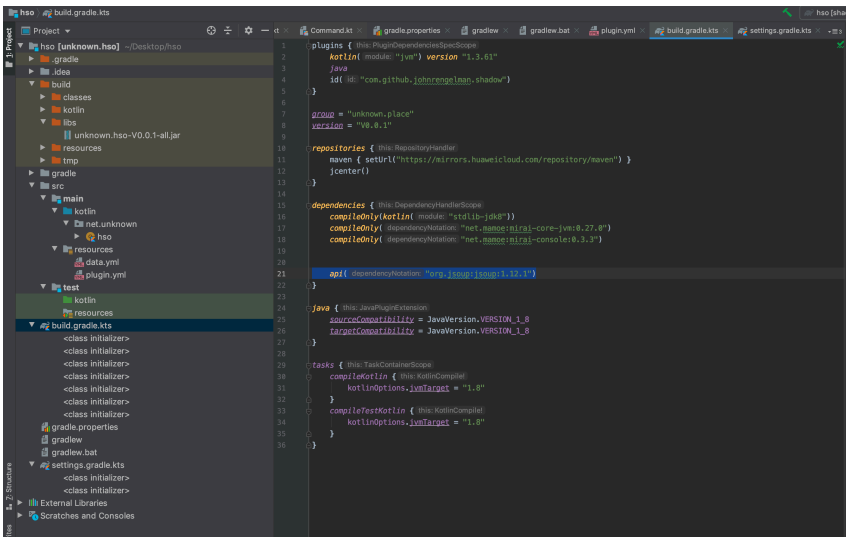
紧接着, 我们来写图片的获取, 即从pixiv下载图片, 下载图片不需要出墙
总的来说, 使用Jsoup获取图片数据, 并进行需要的压缩, 然后使用mirai的uploadAsImage方法上传到腾讯服务器, 最终返回mirai的Image

对于mirai api的教学不在本文范围之内, 请在mirai wiki学习, 那里更加齐全和详细

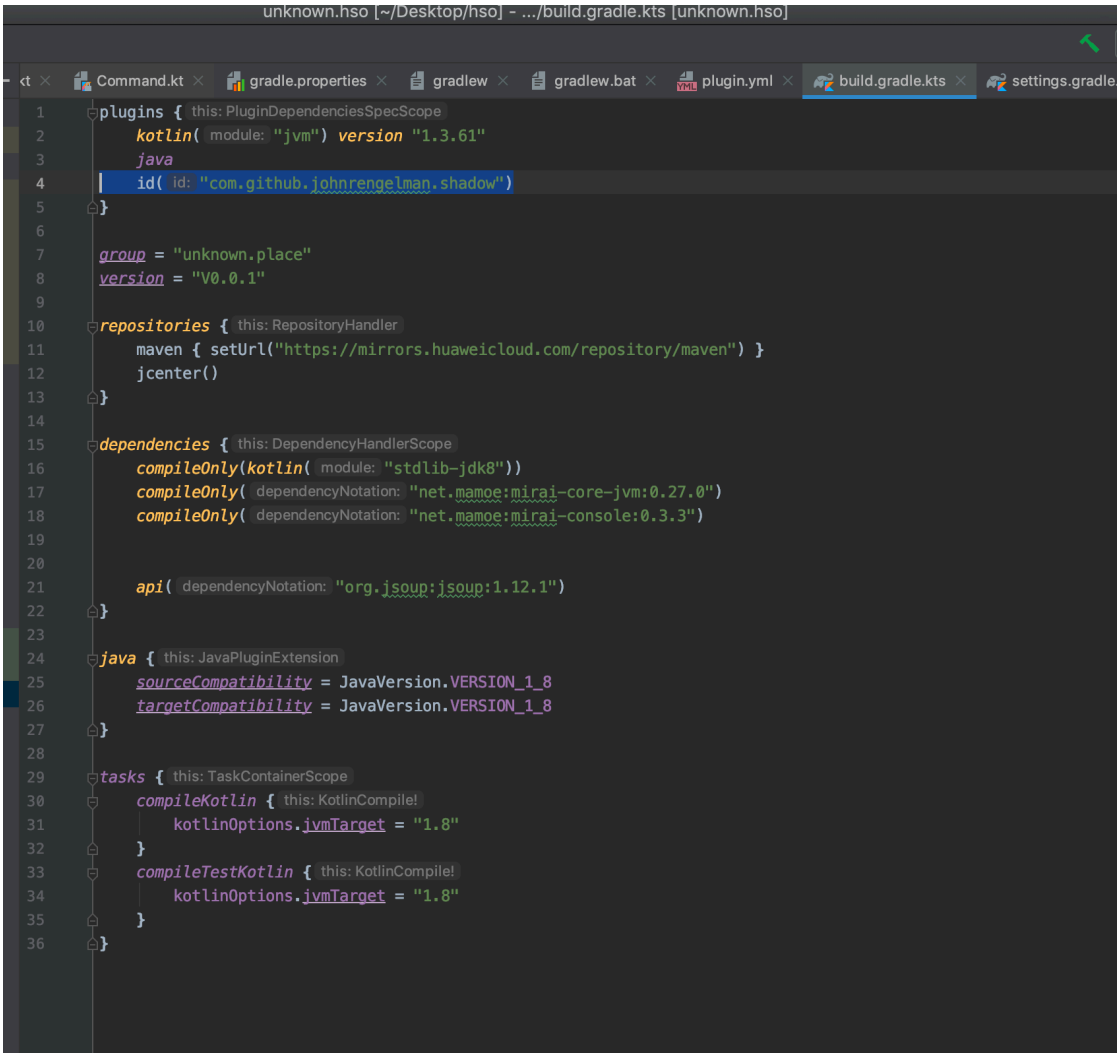
```
    private suspend fun getImage(
        contact: Contact,
        url: String,
        pid: String,
        maxWidthOrHeight: Int,
        anti: Boolean
    ): Image {
        val bodyStream = withTimeoutOrNull( timeMillis: 20 * 1000) { this: CoroutineScope
            withContext(Dispatchers.IO) { this: CoroutineScope
                Jsoup
                    .connect(url)
                    .followRedirects( followRedirects: true)
                    .timeout( millis: 180_000)
                    .ignoreContentType( ignoreContentType: true)
                    .userAgent( userAgent: "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; ja-jp) AppleWebKit/533.20.25 (KHTML, like Gecko) Version/5.0.4 Safari/533.20.27")
                    .referrer( referrer: "https://www.pixiv.net/member_illust.php?mode=medium&illust_id=$pid")
                    .ignoreHttpErrors( ignoreHttpErrors: true)
                    .maxBodySize( bytes: 100000000)
                    .execute().also { check( value: it.statusCode() == 200) { "Failed to download image" } }
            }
        }?.bodyStream() ?: error("Failed to download image")
        if (maxWidthOrHeight < 1) {
            return bodyStream.uploadAsImage(contact)
        }
        val image = withContext(Dispatchers.IO) { this: CoroutineScope
            ImageIO.read(bodyStream)
        }
        if (Image_Resize_Max_Width_Height <=1200 || image.width.coerceAtLeast(image.height) <= maxWidthOrHeight) {//send master file
            return image.upload(contact)
        }
        val rate = (maxWidthOrHeight.toFloat() / image.width.coerceAtLeast(image.height))
        val newWidth = (image.width * rate).toInt()
        val newHeight = (image.height * rate).toInt()
        return withContext(Dispatchers.IO) { this: CoroutineScope
            val dimg = BufferedImage(newWidth, newHeight, image.type)
            if (anti) {
                dimg.setRGB( x: 1,  y: 1,  rgb: 0xFFFFFF)
            }
            val g = dimg.createGraphics()
            g.setRenderingHint(RenderingHints.KEY_INTERPOLATION, RenderingHints.VALUE_INTERPOLATION_BILINEAR)
            g.drawImage(image, 0, 0, newWidth, newHeight, 0, 0, image.width, image.height, null)
            g.dispose()
            dimg ^withContext
        }.upload(contact)
    }
```

但是由于使用了Jsoup作为依赖, 我们需要将Jsoup加入到最终打包的jar中, 请注意依赖Jsoup和依赖其他插件是两个不同的概念, 在plugin.yml的depends中写jsoup是不明智的行为

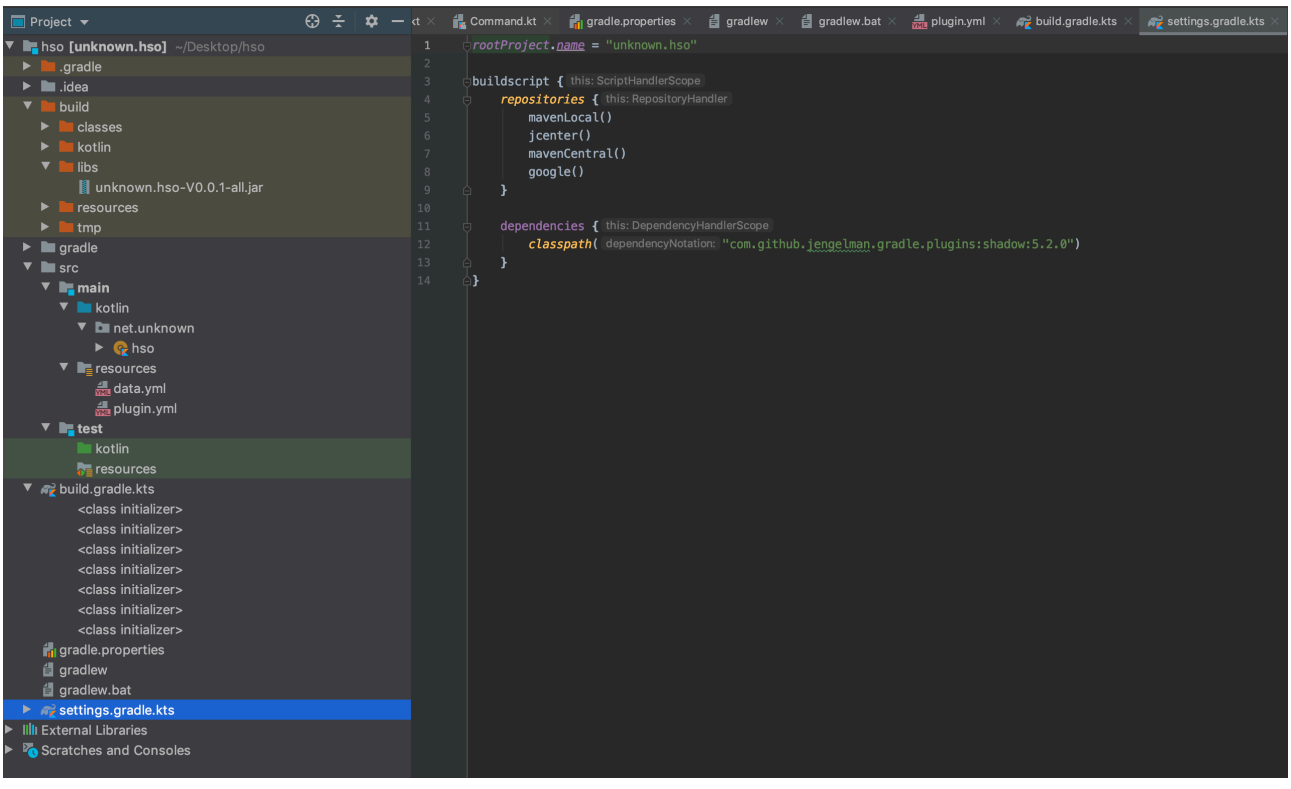
首先添加依赖



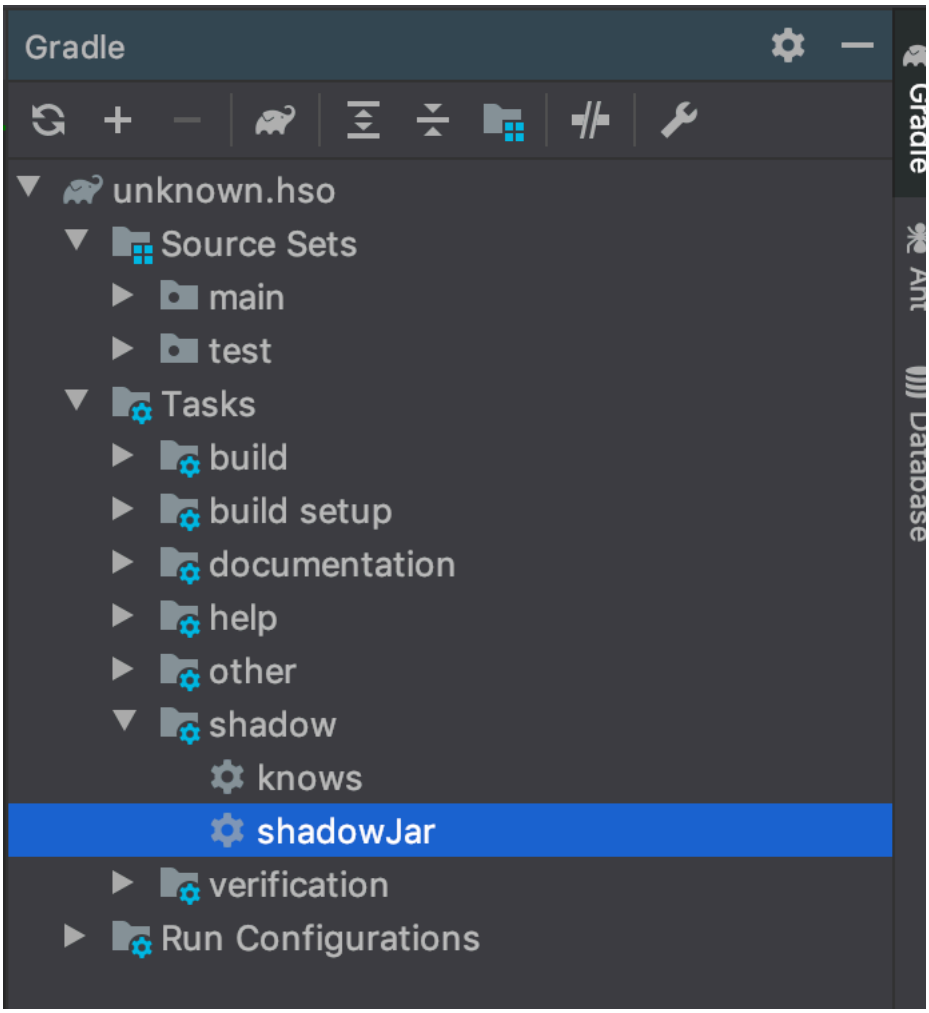
使用shadowJar



添加仓库



这样, 我们就可以使用shadowjar来进行打包, 最终的插件jar包中将包含jsoup



完成图片获取后, 我们将图片发送方法写好, 图片发送是异步执行的, 以防堵塞

```
97      private fun sendImage(contact: Contact, configSection: ConfigSection) {
98          launch { this: CoroutineScope
99              try {
100                 logger.info("正在推送图片")
101                 getImage(
102                     contact,
103                     if(Image_Resize_Max_Width_Height > 1200){
104                         configSection.getString( key: "url")
105                     }else{
106                         with(configSection.getString( key: "url").replace( oldValue: "img-original", newValue: "img-master")){ this: String
107                             val index = this.lastIndexOf( string: ".")
108                             this.substring(0,index) + "_master1200.jpg"//it has to be jpg ^with
109                         }
110                     },
111                     configSection.getString( key: "pid"),
112                     Image_Resize_Max_Width_Height,
113                     Anti_Detect
114                 ).plus(configSection.getString( key: "tags")).sendTo(contact)
115             } catch (e: Exception) {
116                 contact.sendMessage( plain: e.message ?: "unknown error")
117             }
118         }
119     }
```

最终, 我们在onEnable中增加群消息的监听, 并判断是否要发送图片

```
80      override fun onEnable() {
81          logger.info("hso plugin enabled")
82          subscribeGroupMessages { this: MessageSubscribersBuilder<GroupMessage>
83              (contains(Normal_Image_Trigger)) { this: GroupMessage
84                  if (groupsAllowNormal.contains(this.group.id)) {
85                      sendImage(subject, normal.random())
86                  }
87              }
88              (contains(R18_Image_Trigger)) { this: GroupMessage
89                  if (groupsAllowR18.contains(this.group.id)) {
90                      sendImage(subject, r18.random())
91                  }
92              } ^subscribeGroupMessages
93          }
94      }
```

至此, 涩图插件基本功能就完成了, 但是我们希望他变得更适用, 因此我们要引入指令这项功能

```
168     private fun registerCommands() {
169         registerCommand { this: CommandBuilder
170             name = "hso"
171             alias = listOf("setu")
172             description = "hso plugin management"
173             usage = "[/hso enable]    允许本群发送普通图片\n" +
174                     "[/hso enable r18]允许本群发送r18图片\n" +
175                     "[/hso size x]    将发送的图片大小变为x\n" +
176                     "[/hso anti]      开启反过滤和谐\n" +
177                     "对于色图插件触发色图的词汇请在配置文件更改"
178             onCommand { this: CommandSender
179                 val operatingGroup = if (this is ContactCommandSender && this.contact is Group) {
180                     this.contact.id
181                 } else {
182                     0
183                 }
184                 if (it.isEmpty()) {
185                     return@onCommand false
186                 }
187                 when (it[0]) {
188                     "enable" -> {
189                         if (operatingGroup == 0L) {
190                             return@onCommand false
191                         }
192                         if (it.size == 2 && it[1] == "r18") {
193                             groupsAllowR18.add(operatingGroup)
194                             this.sendMessage("以允许" + operatingGroup + "发送R18的图")
195                         } else {
196                             groupsAllowNormal.add(operatingGroup)
197                             this.sendMessage("以允许" + operatingGroup + "发送R普通的图")
198                         }
199                     }
200                     "size" -> {
201                         if (it.size < 2) {
202                             return@onCommand false
203                         }
204                         val to = try {
205                             it[1].toInt()
206                         } catch (e: Throwable) {
207                             this.sendMessage(it[1] + "无法被转为数字")
208                             return@onCommand false
209                         }
210                         Image_Resize_Max_Width_Height = to
211                         this.sendMessage("更新完成")
211                         this.sendMessage("更新完成")
212                     }
213                     "anti" -> {
214                         Anti_Detect = true
215                         this.sendMessage("更新完成")
216                     }
217                     else -> {
218                         return@onCommand false
219                     }
220                 }
221                 return@onCommand true
222             }
223         }
224     }
225 }
```

并在onEnable的时候调用这个方法

```
80  override fun onEnable() {
81      registerCommands()
82      logger.info("hso plugin enabled")
83      subscribeGroupMessages { this: MessageSubscribersBuilder<GroupMessage>
84          (contains(Normal_Image_Trigger)) { this: GroupMessage
85              if (groupsAllowNormal.contains(this.group.id)) {
86                  sendImage(subject, normal.random())
87              }
88          }
89          (contains(R18_Image_Trigger)) { this: GroupMessage
90              if (groupsAllowR18.contains(this.group.id)) {
91                  sendImage(subject, r18.random())
92              }
93          } ^subscribeGroupMessages
94      }
95  }
```

指令(Command)系统
指令是mirai-console中核心的权限系统之一, mirai-console中的manager(bot主人)系统是独立于mirai-core所存在的, 而一条指令则是 bot主人在任何bot在的地方说的一句以/开头的话或是在console后台输入的指令

指令的存在使得bot的管理变得更加简单

指令结构
/commandName args[0] agrs[1] args[2] args[3] ….

指令发送者
ContactCommandSender Manager在qq中使用指令
ConsoleCommandSender 后台使用指令
使用sendMessage()会立即发送一句话(如果是qq则会回复, 如果是后台则会打印),
appendMessage()写入一些文字, 会在全部处理结束后一起发送给使用者

指令注册
插件可以注册属于自己的指令, kotlin中应使用PluginBase方法registerCommands{}进行注册
registerCommand **{**
name = “hso”(commandName指令名)
alias = *listOf*(“setu") (command别名)
description = “hso插件总指令”(指令介绍)
usage = "[/hso normal]　允许本群发送普通图片\n" (指令使用方法)
onCommand = { 最高级监听器, return true代表执行顺利, false代表不顺利
在这个{}里, this为CommandSender it为args[]
}
}

指令的生命周期
当一个指令被使用, 他会首先交给注册该指令的插件处理, 如果该插件返回true, 则代表指令正常, 会再交给所有插件的onCommand()方法监听, 如果返回false, 则不会给其他插件监听的机会, 且会给使用者usage进行帮助

```
81    override fun onCommand(command: Command, sender: CommandSender, args: List<String>) {
82        if(command.name == "login"){
83
84        }
85    }
86
```

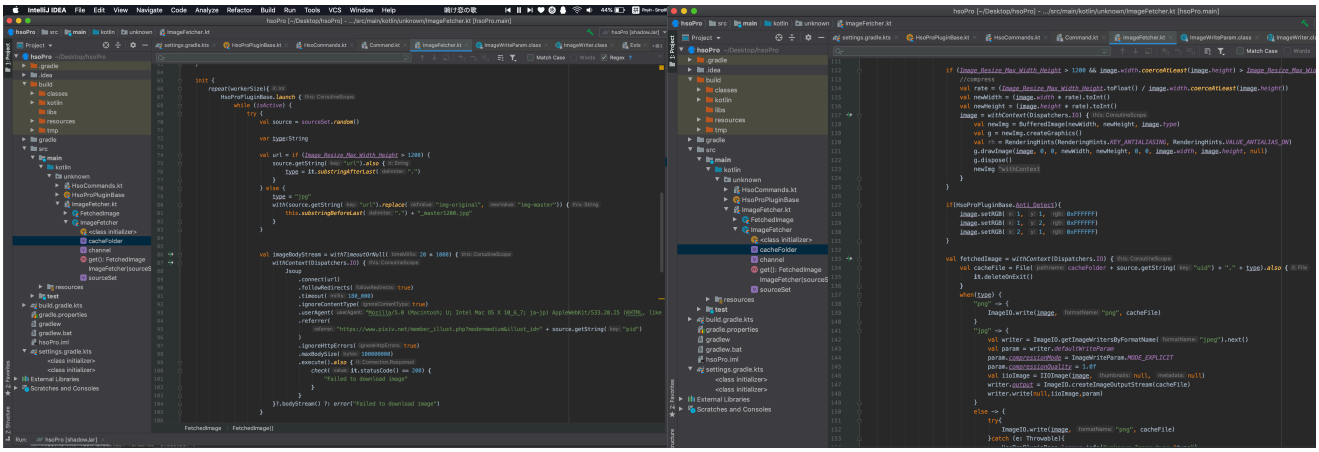
任意插件都可以监听“login”指令, 如果登录成功

至此, 涩图插件就可以运行了, 完整源码可以在github中找到

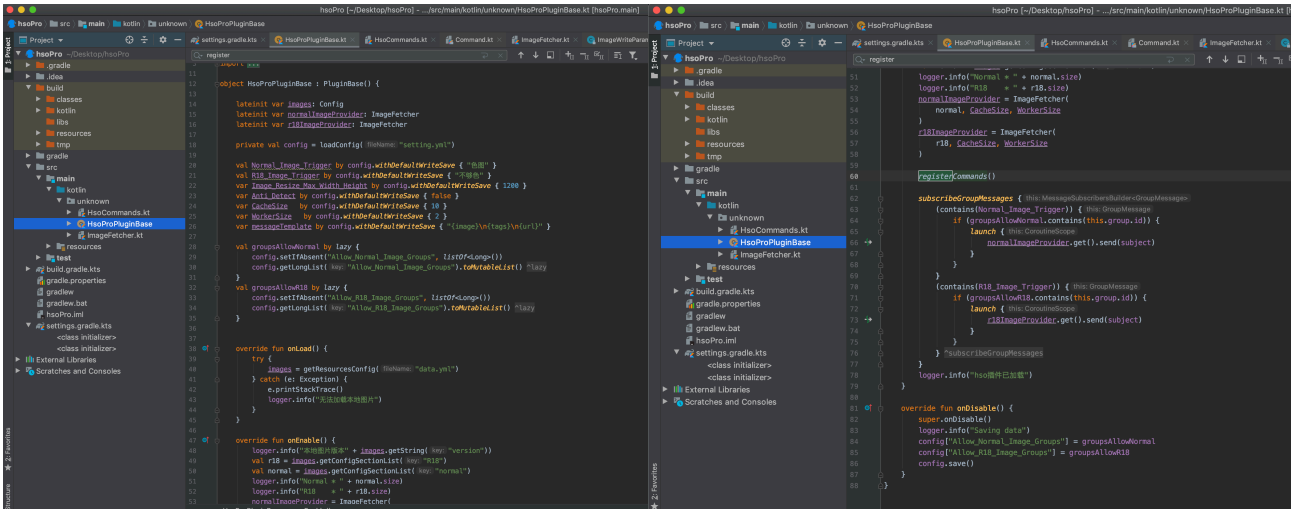
但现在写出的成品只是一个可以加载的插件，但绝对谈不上是一个好用的插件，作为一名插件开发者，你不应该满足于此。

1: 图片发送速度极慢
2: 图片有色块, 失真现象
3: 代码混乱, 不够清晰

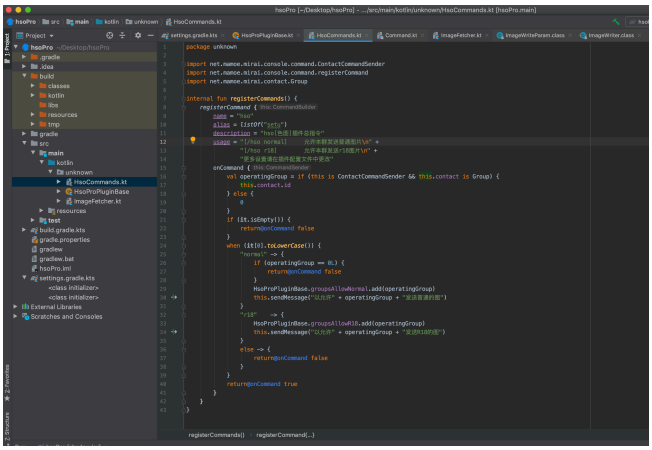
针对第一个问题, 我们将图片转为ExternalImage并与数据一起使用kotlin的Channel进行缓存，这样我们在需要发送图片的时候不需要漫长的等待图片下载
针对第二个问题, 我们将图片使用文件储存, 并将自动压缩关闭
针对第三个问题, 我们将插件分为三个文件, 各司其职



ImageFetcher负责图片的缓存, 他们将图片缓存到/plugins/插件名字/cache 下



我们的PluginBase则只负责插件的生命周期, 数据管理



而HsoCommands负责Command的注册与逻辑

至此, 涩图插件的第二版写完, 解决了第一版的问题, 带来了很好的用户体验。
（完整源码可以在github找到）

而这还不是完, 最后的一步则是书写使用说明, 就想这样(这里只展示了插件配置文件的帮助, 但你应当顺便提供指令帮助等...)

```yaml
Allow_Normal_Image_Groups:          ####允许普通色图的群号 由插件自动生成 也可以自己改
- 655057127
Allow_R18_Image_Groups: []          ####允许R18色图的群号 由插件自动生成 也可以自己改
Normal_Image_Trigger: 色图           ####使Bot发普通色图的暗号
R18_Image_Trigger: 不够色            ####使Bot发R18色图的暗号
CacheSize: 10                       ####缓存的色图数量, 缓存并非在内存中, 增加不会导致内存暴涨, 但推荐够用就好
WorkerSize: 2                       ####用于缓存色图的工作线程数量, 增大数值会提高缓存速度, 增加宽带负载
Image_Resize_Max_Width_Height: 1200 ####图片压缩, 1200为最低 超过1200会获取原图并压缩到该大小 增加内存负载
Anti_Detect: false                  ####防止图片被检测 会增加内存负载
messageTemplate: |-                 ####发送出去的图片整体消息, 使用{image}{tags}{url}三个模版物进行替换
  {image}
  {tags}
  {url}
```

完

Citations

mirai-console:https://github.com/mamoe/mirai-console
mirai: https://github.com/mamoe/mirai

文章作者： mamoe. NaturalHG