Mirai-console 插件教程

My First Plugin - java 教程 Apr 6 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,这是一个Java写插件的例子

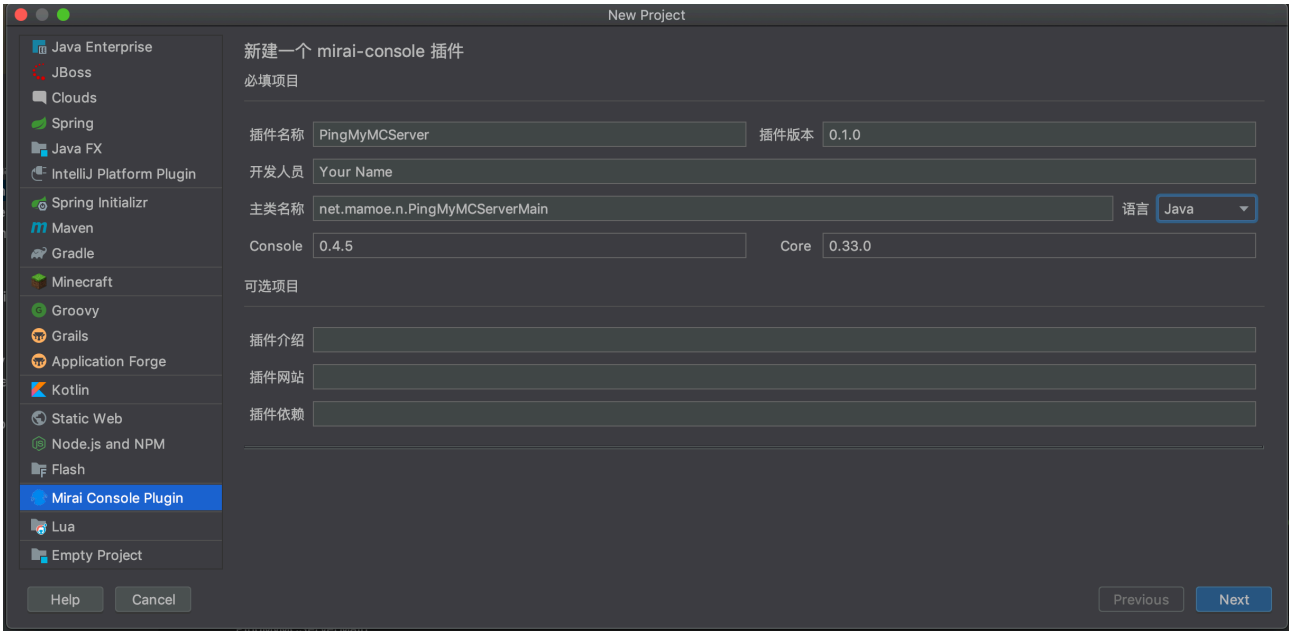
文章所开发的是一个ping MC服务器插件

本文共涉及一个插件项目, 源码可以在github中找到

本文使用Console版本>0.4.5, 大多数JavaAPI只有在0.4.5以上才可以使用

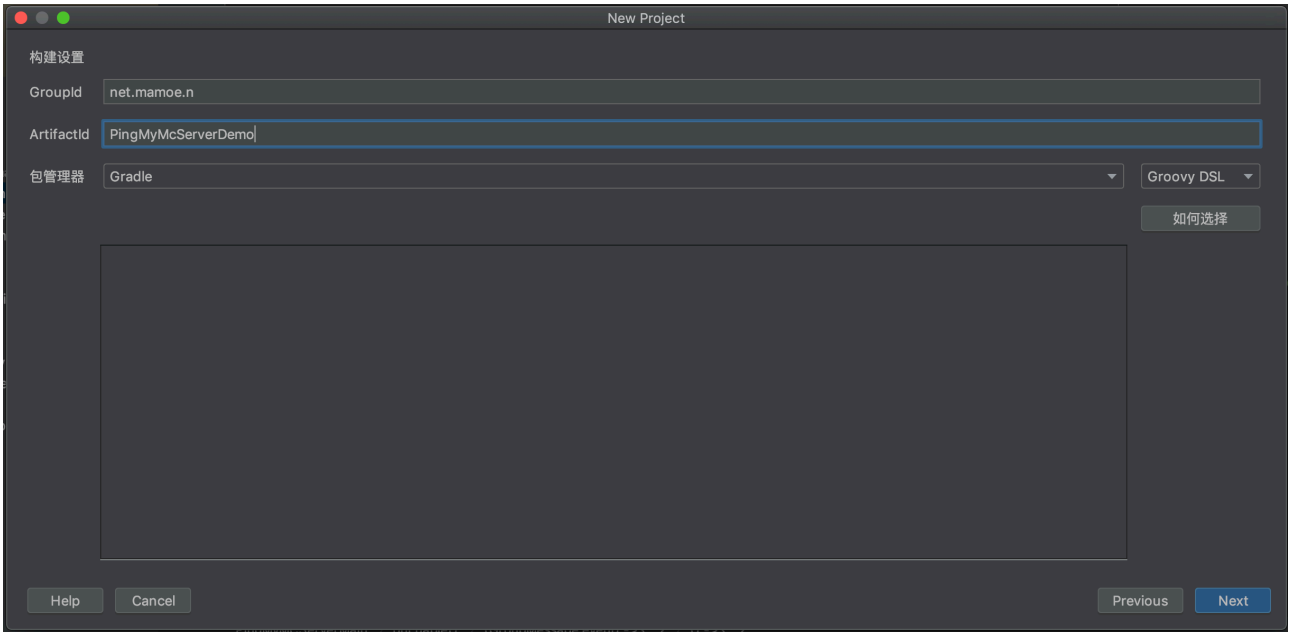
新建项目

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PingMyMCServer, 并将开发者名设置为你自己



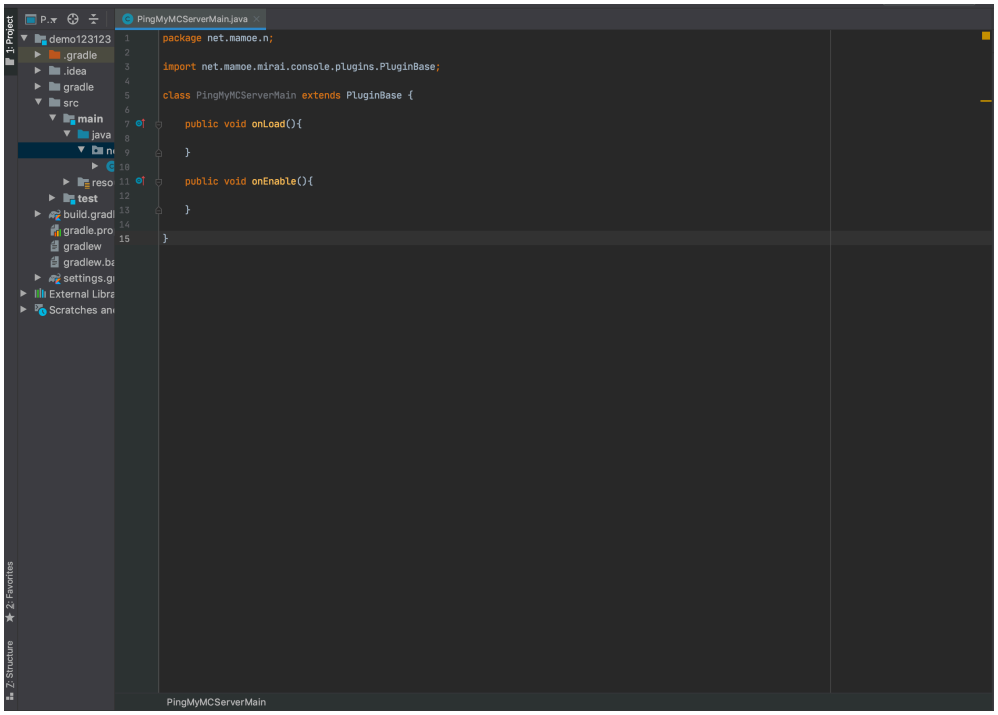
你的主类名称不应该叫Example.. 应包含你名字和插件名字来保证独特

[Console/Core]版本可能在你阅读本文时已有更新, 请不要在意



选择Groovy DSL, 既然你选择了Java你不想写kotlin DSL.

新建完成后, 点击Import, 加载所需的依赖



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。

我们在*onLoad*中初始化配置文件, 配置文件可以储存插件的一些数据, 以及用户的偏好

```
class PingMyMCServerMain extends PluginBase {

    private String defaultServerName;
    private ConfigSection serverMap;
    private Config setting;
    private String API;
    private String responseTemplate;

    public void onLoad(){
        super.onLoad();

        this.setting = this.loadConfig( fileName: "setting.yml");

        this.setting.setIfAbsent( s: "API", t: "https://api-mping.loliboy.com/ping/{address}/{port}");
        this.setting.setIfAbsent( s: "ServerList", ConfigSectionFactory.create());
        this.setting.setIfAbsent( s: "DefaultServerName", t: "");
        this.setting.setIfAbsent( s: "ResponseTemplate", t: "Ping {serverName}: \nGame: {game}, {version}\nName: {fullName}\nPlayer: {currentPlayers}/{m


        this.API = this.setting.getString( s: "API");
        this.defaultServerName = this.setting.getString( s: "DefaultServerName");
        this.serverMap = this.setting.getConfigSection( s: "ServerList");
        this.responseTemplate = this.setting.getString( s: "ResponseTemplate");
    }
```

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

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, Object>*, 且可以作为一个*Object* 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, 就是一个使用了*Config/ConfigSection*的场景,
整个文件读取为*Config*后, *serverMap*为一个*ConfigSection*, 其中*index*为服务器名字, *value*为另一个*ConfigSection*, 包含服务器信息。

在处理完*Config*后, 我们对群消息进行监听, 并作出逻辑。

```
51  public void onEnable(){
52      this.getEventListener().subscribeAlways(GroupMessage.class, (GroupMessage event) -> {
53
54          String messageInString = event.getMessage().toString();
55
56          if(!messageInString.contains("ping ")) {
57              return;
58          }
59
60          String serverName = messageInString.replace( target: "ping ",  replacement: "").toLowerCase().trim();
61
62          if(!this.serverMap.containsKey(serverName)) {
63              serverName = this.defaultServerName;
64          }
65
```

*PluginBase instance*的*getEventListener*方法可以让你获取到插件对应的*EventListener instance,* 用这个*instance*所订阅*(*监听*)*的事件, 是可以在插件关闭时自动取消的。

在这个例子中, 我们订阅了*GroupMessage*事件, 之后的*lambda(*闭包*)*就是你的逻辑, 请注意这个闭包不会被并行调用, 如果闭包内容正在执行, 新的事件会被加入队列, 在当前处理结束后立刻开始处理新的, 长时间的阻塞会导致有延迟出现。

在上面的代码中， 我们通过基础的字符传操作，获得群消息中如果要*ping*服务器的服务器名

我们在*serverMap*中尝试找到有没有这个服务器数据，如果没有则认为用户在*ping* 默认的服务器

```
if(!this.serverMap.containsKey(serverName)){
    event.getSubject().sendMessage("Bot管理员没有设置任何可ping的服务器, 请使用/mcserver 来增加");
    return;
}

ConfigSection serverInfo = this.serverMap.getConfigSection(serverName);

final String serverName_ = serverName;
```

之后检查是否有默认服务器设置，如果没有则返回提示，如果有则读出来。

复制一份服务器名字为*final* 方便后续的*lambda*使用

```
getScheduler().async(() -> {
    event.getSubject().sendMessage("正在获取中..");
    try {
        String response = Utils.tryNTimes( n: 2, () ->
                Jsoup.connect(API
                        .replace( target: "{address}", serverInfo.getString( s: "address"))
                        .replace( target: "{port}", serverInfo.getString( s: "port"))
                ).ignoreContentType(true).timeout(8000).execute().body()
        );
        JsonObject resObj = JsonParser.parseString(response).getAsJsonObject();
        JsonObject addressObj = resObj.get("rinfo").getAsJsonObject();
        event.getSubject().sendMessage(this.responseTemplate
                .replace( target: "{connected}",resObj.get("connected").getAsString())
                .replace( target: "{currentPlayers}",resObj.get("currentPlayers").getAsString())
                .replace( target: "{maxPlayers}",resObj.get("maxPlayers").getAsString())
                .replace( target: "{serverName}",serverName_)
                .replace( target: "{fullName}",resObj.get("cleanName").getAsString())
                .replace( target: "{game}",resObj.get("game").getAsString())
                .replace( target: "{version}",resObj.get("version").getAsString())
                .replace( target: "{address}",addressObj.get("address").getAsString())
                .replace( target: "{port}",addressObj.get("port").getAsString())
        );
    } catch (Exception e) {
        event.getSubject().sendMessage("获取失败.." + e.getMessage());
        e.printStackTrace();
    }
});
```

异步处理*ping*请求，在这里使用了*Jsoup*作为*Http*库，*Gson*解析

另外你需要理解*getScheduler()* Utils.*tryNTimes()* 这两个*console*便捷方法

Scheduler(for JAVA only)

通过PluginBase instance的getScheduler() 方法你可以获得一个PluginScheduler实例, 绑定PluginBase, 因此使用PluginScheduler进行的任务, 都会在插件关闭时关闭

PluginScheduler为java API实现接口, kotlin协程实现底层, 有极高的效率, 它提供了4个方法

```
public void demo(){
    //异步做一个任务，并获得一个Future
    Future<Integer> asyncResult = getScheduler().async(() -> {
        //do some task
        return 100;
    });

    //异步做一个任务，不需要结果
    getScheduler().async(() -> {
        //do some task
        //no result
    });

    //重复任务
    PluginScheduler.RepeatTaskReceipt receipt = getScheduler().repeat(() -> {
        //repeat some task once 1 sec.
    }, intervalMs: 1000);

    //延迟任务
    getScheduler().delay(() -> {
        //do some task after 10s
        receipt.setCancelled(true);//cancel the repeat task 取消上面的重复任务
    }, delayMs: 10000);
}
```

Scheduler(for JAVA only)

不要在任何TASK中使用Thread.sleep()[长时间] 这会导致一个IO协程池中的线程罢工，如果有类似的需求，应该继续用new Thread()

```
String response = Utils.tryNTimes( n: 2, () ->
        Jsoup.connect(API
                .replace( target: "{address}", serverInfo.getString( s: "address"))
                .replace( target: "{port}", serverInfo.getString( s: "port"))
        ).ignoreContentType(true).timeout(8000).execute().body()
);
```

*tryNTimes*是*Utils*提供的一个小工具, 可以将这个*lambda*执行, 如果失败则重试一共*N*次，成功一次就会立刻返回，全部失败会丢出失败错误，适合不稳定的*IO*操作

在之后我们要了解一下*Console*内置的权限和指令系统, 他帮助用户统一的使用插件。

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

后台中使用/manager 指令添加Bot主人

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sendMessageBlocking()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),
appendMessageBlocking()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, java中应该使用
JCommandManager.*getInstance*().register(this, new BlockingCommand())
来注册

```
JCommandManager.getInstance().register( pluginBase: this, new BlockingCommand(
        name: "mcserver", new ArrayList<>(), description: "管理可以ping的MC服务器", usage: "/mcserver add/remove"
) {
    @Override
    public boolean onCommandBlocking(@NotNull CommandSender commandSender, @NotNull List<String> list) {
        if(list.size() < 1){
            return false;
        }
        switch (list.get(0)){
            case "add":
                if(list.size() < 4){
                    commandSender.sendMessageBlocking( s: "/mcserver add 服务器名字 IP 端口");
                    return true;
```

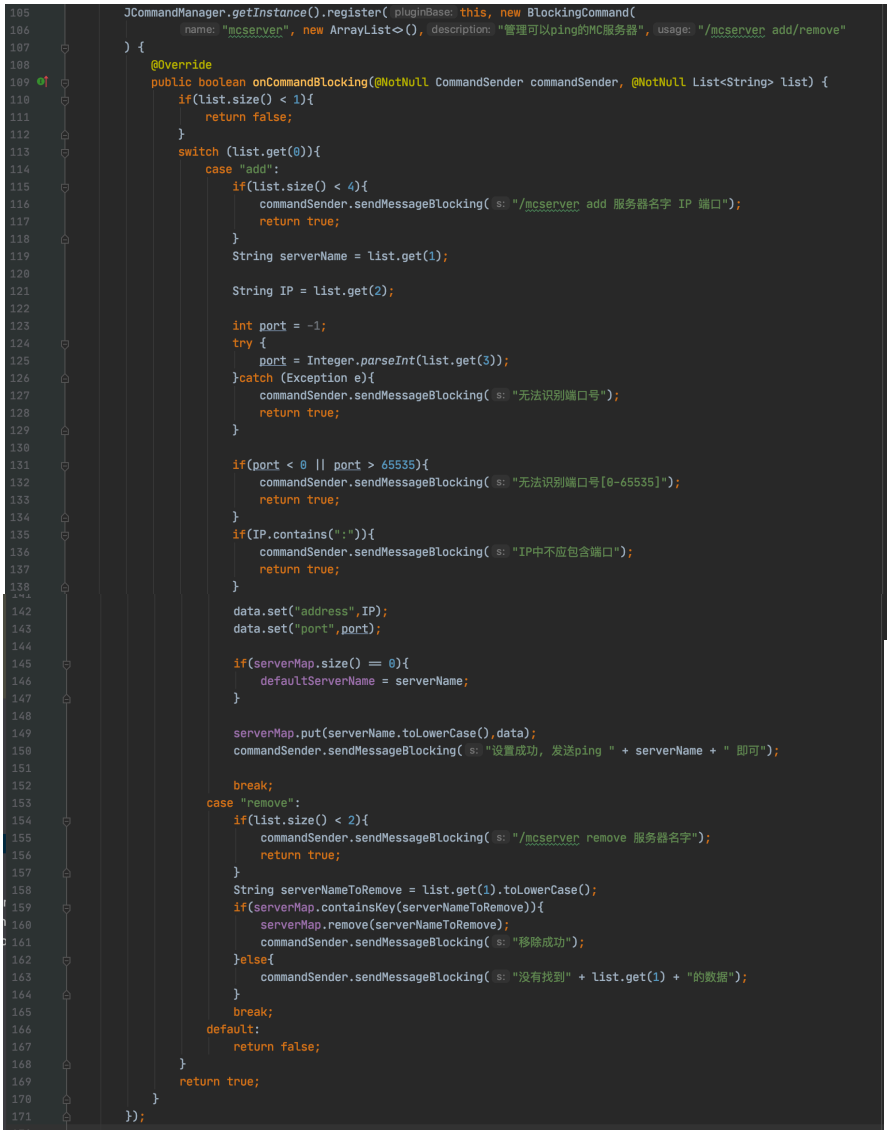
指令的生命周期

当一个指令被使用, 他会首先交给注册该指令的插件处理（即上文的onCommandBlocking）, 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

举个例子, 任意插件都可以监听“login”指令, 如果登录成功

```
@Override
public void onCommand(@NotNull Command command, @NotNull CommandSender sender, @NotNull List<String> args) {
    if(command.getName().equals("login")){
        sender.sendMessageBlocking( s: "hi");
    }
}
```

这个插件中的指令注册代码



```
JCommandManager.getInstance().register( pluginBase: this, new BlockingCommand(
        name: "mcserver", new ArrayList<>(), description: "管理可以ping的MC服务器", usage: "/mcserver add/remove"
) {
    @Override
    public boolean onCommandBlocking(@NotNull CommandSender commandSender, @NotNull List<String> list) {
        if(list.size() < 1){
            return false;
        }
        switch (list.get(0)){
            case "add":
                if(list.size() < 4){
                    commandSender.sendMessageBlocking( s: "/mcserver add 服务器名字 IP 端口");
                    return true;
                }
                String serverName = list.get(1);

                String IP = list.get(2);

                int port = -1;
                try {
                    port = Integer.parseInt(list.get(3));
                }catch (Exception e){
                    commandSender.sendMessageBlocking( s: "无法识别端口号");
                    return true;
                }

                if(port < 0 || port > 65535){
                    commandSender.sendMessageBlocking( s: "无法识别端口号[0-65535]");
                    return true;
                }
                if(IP.contains(":")){
                    commandSender.sendMessageBlocking( s: "IP中不应包含端口");
                    return true;
                }
                data.set("address",IP);
                data.set("port",port);

                if(serverMap.size() == 0){
                    defaultServerName = serverName;
                }

                serverMap.put(serverName.toLowerCase(),data);
                commandSender.sendMessageBlocking( s: "设置成功, 发送ping " + serverName + " 即可");

                break;
            case "remove":
                if(list.size() < 2){
                    commandSender.sendMessageBlocking( s: "/mcserver remove 服务器名字");
                    return true;
                }
                String serverNameToRemove = list.get(1).toLowerCase();
                if(serverMap.containsKey(serverNameToRemove)){
                    serverMap.remove(serverNameToRemove);
                    commandSender.sendMessageBlocking( s: "移除成功");
                }else{
                    commandSender.sendMessageBlocking( s: "没有找到" + list.get(1) + "的数据");
                }
                break;
            default:
                return false;
        }
        return true;
    }
});
```

而这还不是完, 最后的一步则是书写使用说明, 来帮助你的用户知道你在写什么

完

Citations

---

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

文章作者： mamoe. NaturalHG