# MC Mod硬编码汉化Vault Patcher编写教程

**作者：醉冰茶**

注：推荐使用Word的导航功能以获得更加的阅读体验

## 1、硬编码汉化编写门槛

必须熟练掌握JSON格式的编辑

必须足够了解原版译名，并遵守社区模组翻译规范

最好有一定程度的编程基础（而且最好是Java基础），能区分主要语法和方法体

最好有Minecraft模组编程基础，能理解模组源代码（借助辅助工具或独立阅读）

## 2、基础准备

首先必须准备好Java环境，Java版本最低应为Java8，视实际需求而定。

进行硬编码汉化编写的必要工具：

VSCode，文件编辑器，不多赘述；最好安装Extension Pack for Java插件。

反编译工具1号Recaf，我们查看mod源码的主要工具。

反编译工具2号Java Decomplifier，我们简称为JDec，某些特殊情况下才需使用，后续会特别介绍。

最新版的Vault Patcher Mod（下文简称为VP），注意这个mod是全版本通用的。

以上两个主要的反编译工具我会提供[下载链接](https://wwxj.lanzout.com/iTyPg2uef38d)，Recaf也可以去[官网下载](https://www.coley.software/Recaf/)。

然后再准备好要汉化的mod文件。

注意，为避免无用功，在硬编码汉化开始之前，建议先确认以下两点：

1.未翻译的文本是否在lang文件中。建议先获取完整翻译的lang文件再进行硬编码汉化。

2.未翻译的文本是否在配置文件（config）中。

注意，由于VP的替换原理是反编译，因此模组源码不是必要的，因为原版方法会被混淆。如果能找到模组源码，你可以将其作为参考，但在编写配置文件时不要使用源码。

## 3、VP的准备

（参考<https://github.com/3093FengMing/VaultPatcher/wiki>）

在这篇教程中，我将以宝藏猎人3核心模组The Vault与知名硬编码模组Xnet为例，介绍硬编码汉化的完整编写流程。

硬编码汉化基于VP的配置文件（config）完成，因此所有的翻译内容都是json文件。我们将mod源码（.java文件）和json文件以Consolas字体表示，这是一种编程常用的等宽字体，如下：

## test.java

// 第一行由##开头，用于区分文件名，以.java结尾代表mod反编译代码，以.json结尾代表配置文件或其他

// 第一行没有其他作用，也不在代码中生效，请勿误用

...

// 三个点（省略号）代表此处有被省略的内容

System.out.printIn(“Hello world!”)

// 注意，我们使用双斜杠表示注释内容，实际上JSON文件中不能使用这种方式注释

// 注释文本仅作代码介绍或解析使用，不要将我的注释添加到你的代码中

我们所有的JSON文件编辑都推荐使用VSCode，这样如果有格式错误，它会提示我们问题所在。

那么我们开始吧！

注意，VP的配置文件位于根目录的config/vaultpatcher\_asm文件夹下，所有内容都在此文件夹下才能发挥作用。

### 3.1、准备config.json

准备好VP的核心配置文件config.json，示例如下：

## config.json

{

“mods”: [

“the\_vault\_vp”, // 这里换成你之后创建的模块文件的文件名

“xnet\_vp” // 如果有多个想要使用的模块文件，像这样用逗号隔开，每个文件一行

// 强烈建议每个mod只使用一个模块文件；如果有些mod的模块文件巨大无比，可以按固定原则拆分

],

“class\_patch”: false, // 反编译汉化不涉及此功能，默认为false

“debug\_mode”: {

“is\_enable”: true, // 是否启用调试模式，汉化测试时应设置为true

“use\_cache”: false, // 是否使用缓存，汉化测试时设为false以节省调试步骤，正式发布应改为true

“output\_format”: “<source> -> <target> in <class> M-[<ordinal>] | <info>”,

// output\_format代表调试模式下翻译信息的输出结果，这里的值是我自己常用的值，最为明确

“output\_mode”: 0,

“hide\_pairs”: 7,

“output\_node\_debug”: false,

“export\_class”: false,

“missing\_warn”: true

}

}

你也可以在安装VP的情况下启动一次游戏，获得一个自动生成的配置文件。

### 3.2、准备模块文件

新建一个模块文件，以The Vault模组为例，命名为“the\_vault\_vp.json”（和上文config.json中保持一致）。

然后完成模块文件的开头，示例如下：

## the\_vault\_vp.json

[

// 模块文件的第一个内容是文件的基本信息，格式如下

{

“authors”: “DrIceTea”, // 汉化作者名，不会影响汉化效果，可中文

“name”: “The Vault 硬编码汉化”, // 模块内容命名，不影响汉化效果

“desc”: “宝藏猎人核心模组汉化v1.0-教程版”, // 模块内容描述，不影响汉化效果

“mods”: “the\_vault”, // 对应的mod的信息，可随意填写，建议使用mod的命名空间（modid）

“dynamic”: false, // 是否启用动态替换，一般为false

“i18n”: false // 是否启用i18n功能，该功能会对存入的值根据所选语言替换为对应的值

},

// 这里留出空行，后续编写翻译信息就在此处继续编写

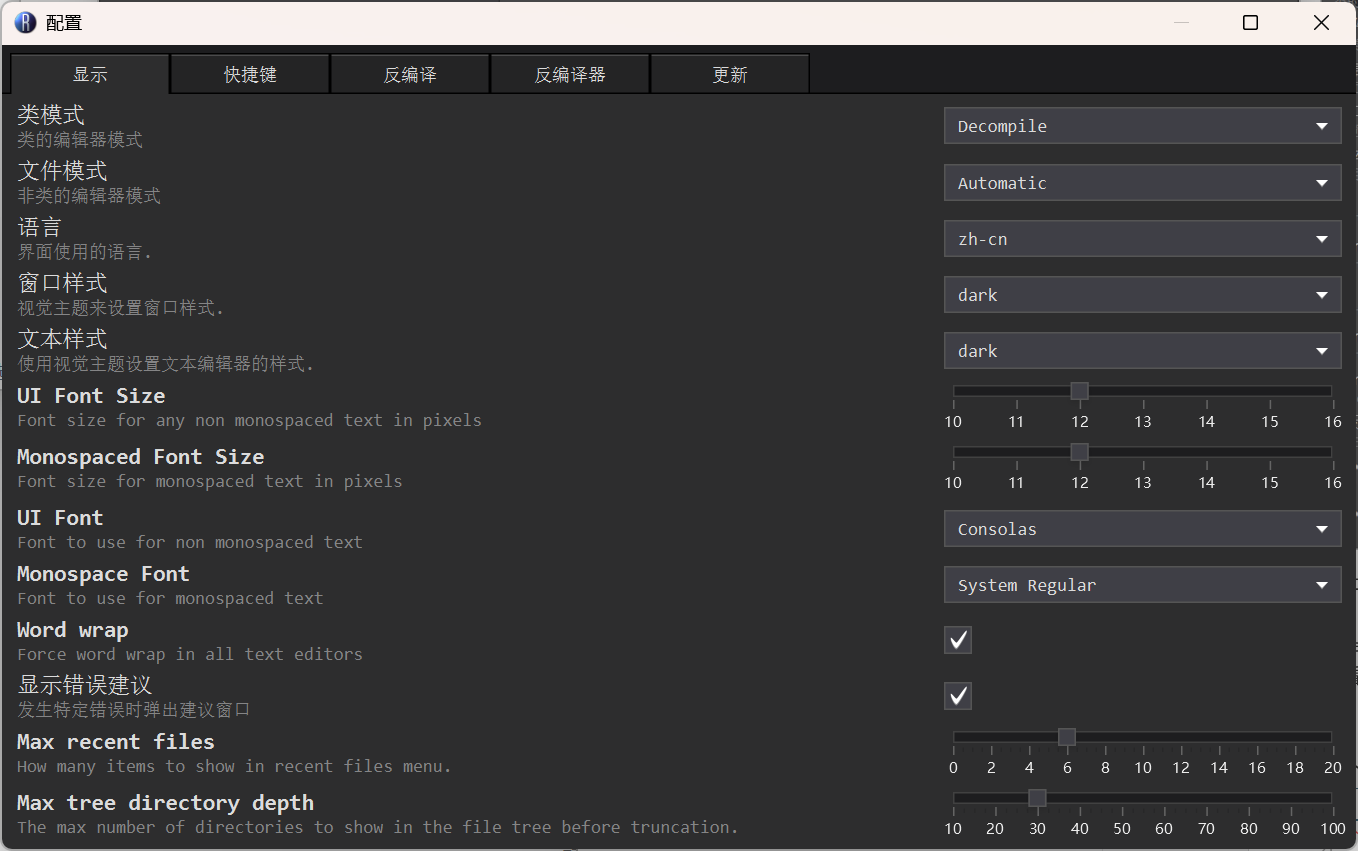
] // 结尾一定注意补好中括号

这样，我们的准备活动就完成了。

## 4、翻译信息的编写

### 4.1、配置Recaf并查看反编译代码

双击打开Recaf。如果是第一次使用Recaf，我们需要先进行一些配置。点击Recaf上边栏的“配置”：



如上图是我自己使用的Recaf配置。其他四个选项卡一般无需修改。

小提示，Word wrap选项即为开启自动换行。

配置完成后，将mod文件拖入Recaf的左侧界面，或点击“文件-加载”即可导入mod。这就是我们的主战场。

assets文件夹和data文件夹一般不含硬编码文本相关的内容，META-INF文件夹更没有任何有用的内容，我们直接查看mod反编译代码（一般就是剩余的一个文件夹）即可。

### 4.2、翻译信息的基本结构

一个完整的翻译信息由以下部分组成（仍然以The Vault为例）：

## the\_vault\_vp.json

{

“target\_class”: { // 目标对象，即待替换文本所在的位置的信息

“name”: “iskallia.vault.gear.VaultGearRarity”, // 目标类名，必填

“method”: “getDisplayName”, // 方法名，可选

“local”: “Mcapitalize”, // 变量源，可选

“ordinal”: 21 // 操作序数，可选

},

“pairs”: [ // 键值对数组，代表多个待替换文本和译文的信息

{ // 一个原文与其对应译文组成一个翻译组

“key”: “Scrappy”, // 原文本

“value”: “破烂” // 翻译文本

},

{ // 多个目标对象（代码位置）相同的键值对可以堆叠

“key”: “Common”,

“value”: “普通”

},

{

“key”: “Rare”,

“value”: “稀有”

}

]

}

接下来我们一一介绍各部分的作用。

#### 4.2.1、目标对象

target\_class为目标对象，即待替换文本所在的位置的信息。

其核心为name即目标类名。该翻译信息的所有键值对都只会在该目标类名下寻找对应对象位置替换。name是一个必填项。

当同一个类下有多个相同的待替换文本时，可以通过不同的方法将其区分，即填写不同的method值。注意这里的方法名使用的是通过Recaf或JDec反编译解析的方法名，即混淆后的方法名，不能使用源代码中的方法名。

当文本并非直接以String字符串格式表示，而是以变量或参数形式出现时，我们就需要使用local值才能完成替换。local有不同的使用前缀，用于应对不同的情况。我们在下文中详细介绍。

注意local与method不同：是否填写method对普通String的替换并无影响，但如果不使用local就无法替换以变量或参数形式出现的文本。

当相同的键值对（待翻译文本）出现在多个类、或同一个类的多个方法、或同一个类的多个不同的变量或参数中时，我们可以在同一个翻译信息中使用多个target\_class，这时需要换用target\_classes，示例如下：

## the\_vault\_vp.json

{

"target\_classes": [

{

"name": "iskallia.vault.client.gui.helper.FontHelper",

"method": "drawStringWithBorder",

"local": "Vtext"

},

{

"name": "iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget",

"method": "renderHover",

"local": "GresearchName",

"ordinal": 24

},

{

"name": "iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget",

"method": "lambda$renderHover$1",

"local": "MgetName"

},

{

"name": "iskallia.vault.research.StageManager",

},

{

"name": "iskallia.vault.item.KnowledgeBrewItem",

"local": "Vresearch"

}

],

"pairs": [

{

"key": "Backpacks",

"value": "小背包"

},

{

"key": "Big Backpacks",

"value": "大背包"

}

// 还有其他键值对，不做过多展示

]

}

可以注意到，多种不同的target\_class键的选用方式可以混用。

#### 4.2.2、键值对

下面介绍键值对pairs，这是我们主要编写翻译内容的部分。

key代表原文本，即反编译代码中出现的**完整String字符串内容**。注意必须是“完整”内容，示例如下：

## DungeonDoorTileEntity.java

...

public static class Difficulty {

...

public Component getDisplay() {

return (new TextComponent(**"Difficulty: %s"**.formatted(this.getName())))

.m\_130948\_(Style.f\_131099\_.m\_131148\_(TextColor.m\_131266\_(this.getColor())));

// 这里要替换的是显示难度的文本

}

}

...

## the\_vault\_vp.json

...

{

"target\_class": {

"name": "iskallia.vault.block.entity.DungeonDoorTileEntity$Difficulty"

},

"key": **"Difficulty: %s"**, // 这里必须填写上述代码中引号包裹的完整String字符串内容

"value": "难度：%s"

},

...

value即为待翻译文本的译文。注意保留诸如“%s”“%d”等参数文本。译文的详细规范参见[翻译规范](https://cfpa.site/TransRules/)。

注意到，当翻译信息中只有一个键值对时，我们也可以抛弃pairs的外壳，直接使用key和value，如上。

#### 4.2.3、注释

正常的JSON文件当然是不可以用双斜杠写注释的。但是作为VP的模块文件，我们可以这样写一条注释：

## the\_vault\_vp.json

...

{

“\_comment”: “这是一条模块文件中的注释”,

},

...

注释可以分隔不同部分的硬编码翻译信息，也可以帮助自己理清逻辑、记录要点、等等。多记录注释也可以方便后续维护之类的。

当然，不要将注释写在模块文件的第一条。

### 4.3 寻找待翻译文本

了解完翻译信息的基本结构，我们就需要寻找需要翻译的文本进行编写了。主要的提取方法有以下三种：

#### 4.3.1、文本搜索

在游戏内找到需要翻译的文本，然后在Recaf中点击“搜索-字符串”搜索文本，匹配模式一般选CONTAINS。

然后根据搜索结果分别查看每个类的代码，找到最有可能的一个（或多个）文本源位置。

如果文本为天然的String字符串格式，则待翻译文本（原文本）为一对英文双引号内的全部内容，如上述。

此寻找方式优点是简便，可以直接定位文本，并且基于游戏表现，较少出现漏提取的问题。缺点是对于特殊情况（如变量或参数形式的文本）无法提取。

#### 4.3.2、硬编码特征类匹配

作为硬编码文本，MC代码中有专门的类和/或方法将这些文本用于显示（称为“文本组件”），我们可以直接搜索这些类/方法来查找所有的硬编码项。

游戏版本低于1.18时，一般使用如下方法：

new TextComponent(“Example Text”);

因此，可以使用“类引用”搜索“TextComponent”类查找所有使用该方法创建的硬编码文本。

游戏版本1.19+时，一般使用如下方法：

Component.literal(“Example Text”);

因此，可以使用“成员引用”搜索，所有者为“Component”，名称为“m\_237113\_”，以查找所有使用该方法创建的硬编码文本。

如果搜索不到，可以尝试只填写“所有者”或“名称”再次搜索。此外，“m\_237113\_”是Forge对“literal”这一方法的混淆方法名，因此对fabric可能不生效，可自行尝试其他方法。

此寻找方式的优点是大量批量提取，对于简单的硬编码mod（如mcreator模组）效率奇高。缺点是较易出现漏提取情况，如部分情况下GUI显示并不使用此方法，以及部分作者使用的文本库对以上构造方法进行了包装（如RFTools系列），都可能导致搜索不到的情况。

#### 4.3.3、逐个检查所有反编译代码

这是唯一完整提取所有硬编码文本的方式，即逐一打开每个反编译的代码文件，检查硬编码文本。同时打开游戏，结合游戏内表现进行处理。模块文件编写完成后，打开游戏进行测试与调试，直至所有硬编码文本都被翻译为止。

此方式优点是全面完整，缺点是费时费力。对于mcreator模组，只要使用上述方法进行一次编写后未出现缺漏，就不建议使用此方法。但对于大型mod（例如The Vault）推荐使用此方法。

注意：无论通过何种方式提取待翻译文本并编写翻译信息，在编写结束后都必须进入游戏，查看硬编码翻译是否已完全覆盖。不推荐发布未经测试的硬编码汉化。

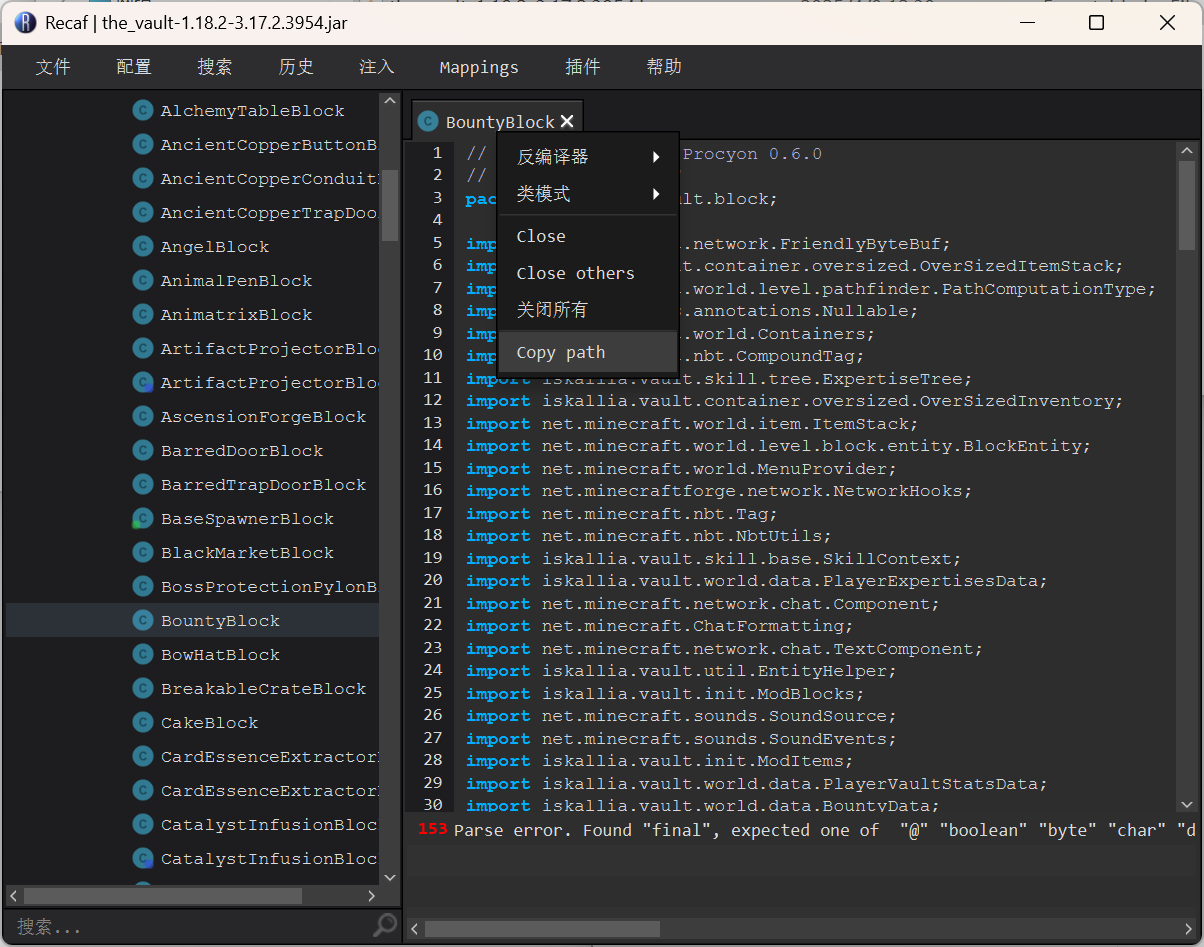
### 4.4、编写文本的翻译信息

在寻找到待翻译文本后，就需要将其编写成上文提到的翻译信息了。

#### 4.4.1、基础中的基础——获取目标类

大多数情况下，目标类（name）所填写的内容均为“package（包）名+.+文件名”。

最简单的提取方法：在Recaf中，右键点击打开的文件的顶部页签，点击“Copy Path”：



复制的路径如下：

iskallia/vault/block/BountyBlock

这段文本可以直接填入name字段中，VP可以读取；也可以将其中的“/”替换为“.”。

对于不是Recaf的情况，可以查看文件第一行的“package”后面的内容，再加上文件名，即为name的值。

注意，有些情况下，文本位于父类下定义的子类（内部类）中，因此当遇到内部类的情况时需额外注意，使用Recaf确认当前类是否定义了内部类，并查看要替换的文本实际位于哪个内部类中。示例如下：

## DungeonDoorTileEntity.java

package iskallia.vault.block.entity;

...

public class DungeonDoorTileEntity extends TreasureDoorTileEntity { // 这是这个类的定义

...

**public static class Difficulty** { // 这里是Difficulty内部类的定义

...

public Component getDisplay() { // 这就是Difficulty内部类的方法

return (new TextComponent(**"Difficulty: %s"**.formatted(this.getName())))

.m\_130948\_(Style.f\_131099\_.m\_131148\_(TextColor.m\_131266\_(this.getColor())));

// 这里要替换的是Difficulty内部类下、getDisplay方法中的，显示难度的文本

}

} // 这里是内部类结束的大括号

...

}

这里的文本就位于内部类中，我们的翻译信息需要如此编写：

## the\_vault\_vp.json

...

{

“target\_class”: {

“name”: “iskallia.vault.block.entity.DungeonDoorTileEntity**$Difficulty**”,

// 在上文介绍的基础的name值后，添加一个“$”符号，再添加内部类的类名

“method”: “getDisplay” // 方法名填写不受是否内部类影响。此处method也可省略

},

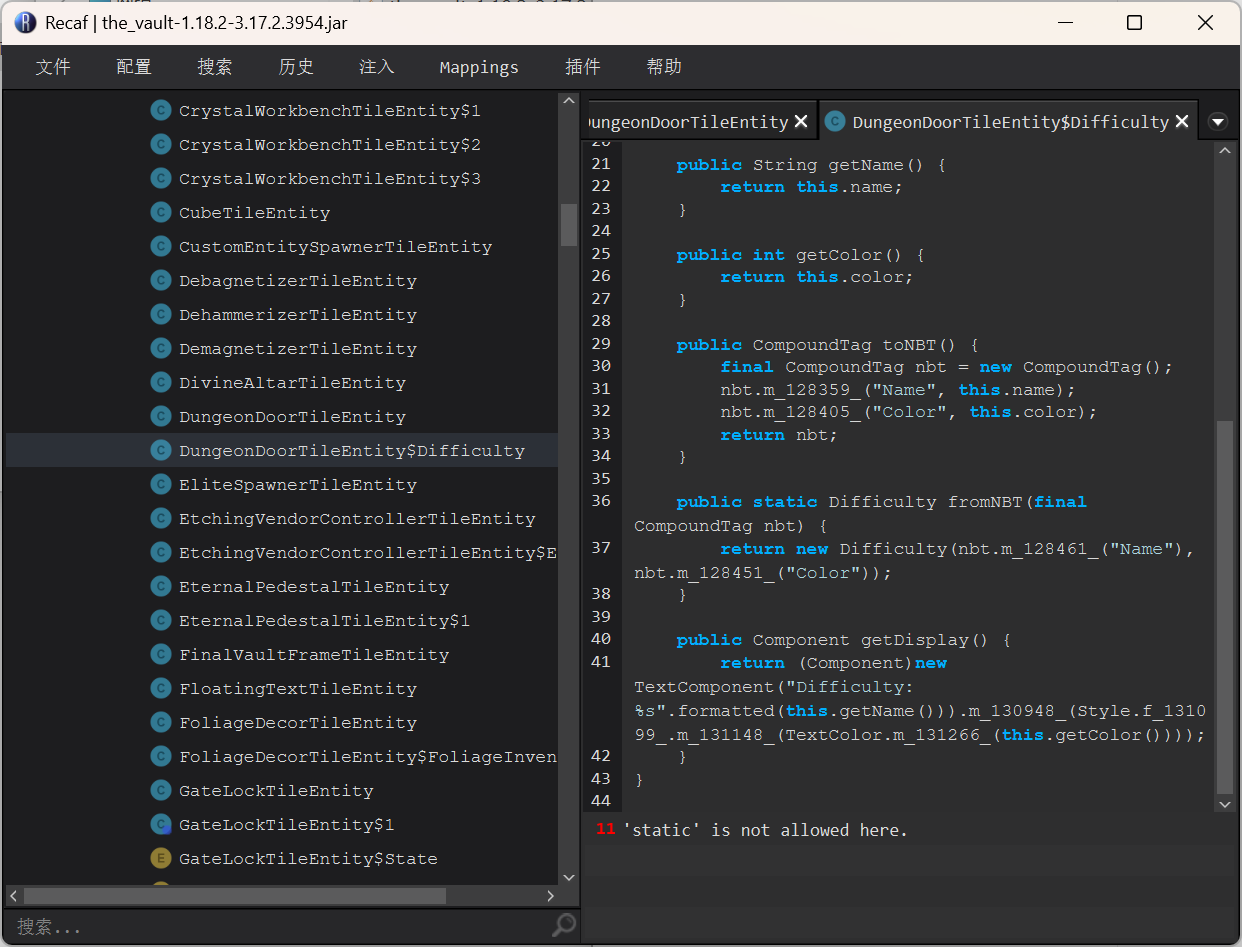
“key”: **“**Difficulty: %s”,

“value”: “难度：%s”

},

...

Recaf中也可在左侧文件列表中直接看出该类是否有内部类，仍然以上述文本为例，如下图：



对于DungeonDoorTileEntity类，有两个文件。DungeonDoorTileEntity文件会显示该类下所有内容，包括其内部类；DungeonDoorTileEntity$Difficulty文件则只会显示内部类的内容。通过这种方式也能快速判断。

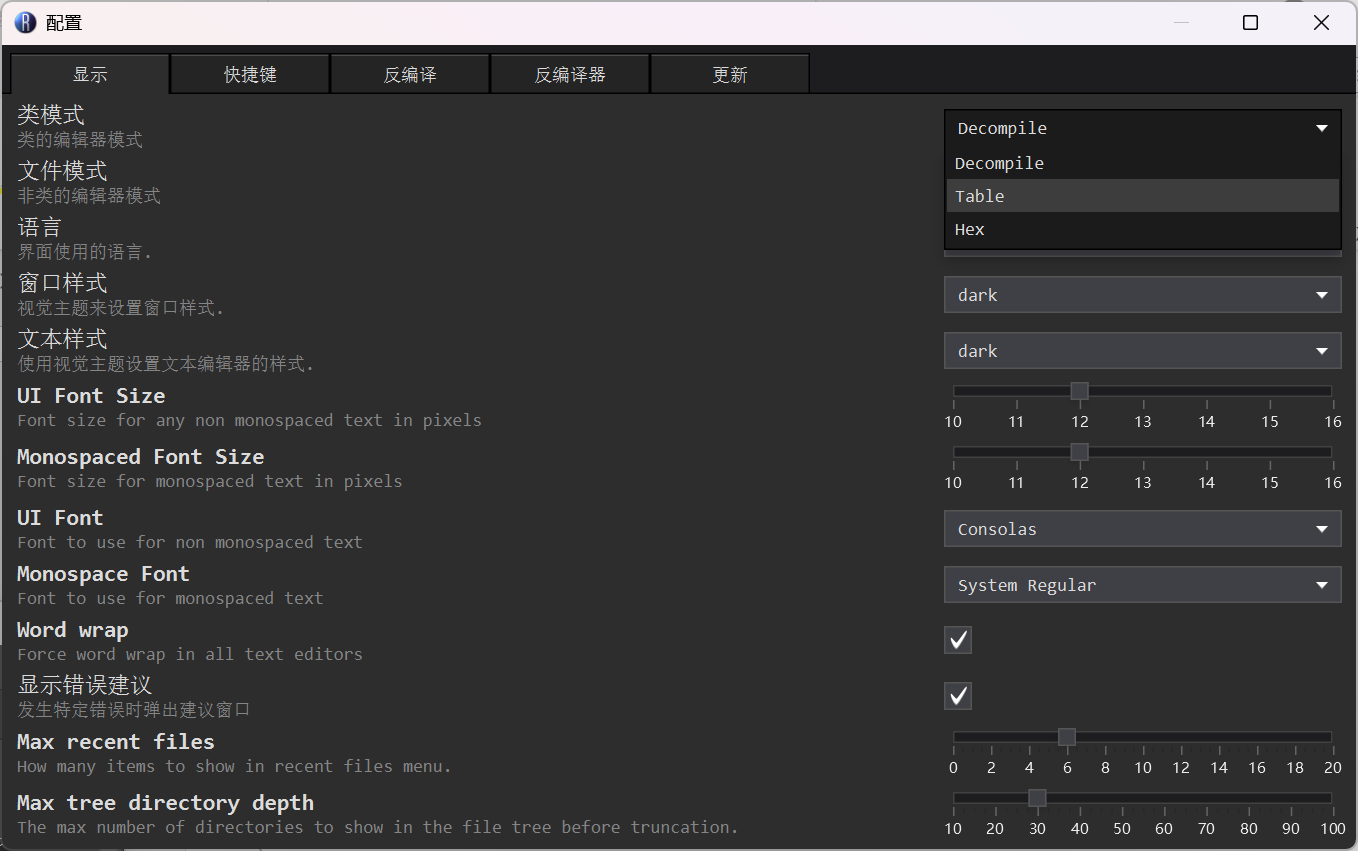
#### 4.4.2、避免字符串冲突——获取方法名

如果文本为天然的String字符串格式，且在当前类下并没有相同的字符串出现，则代表不限制method不会导致冲突，可以选择不填写method字段。

大多数情况下，method值是且必须是反编译代码中文本所在的方法名。

当文本位于类的构造方法内、类的构造器内、或lambda内时，method值不等于所在的方法名。这种情况下，你需要通过Recaf的另一种类模式查看该类下的内部类和方法。

该模式需要切换配置，如下图：



在“类模式”中选择“Table”，即可切换模式。

我们以以下待替换文本为例：

## ClassicListenersLogic.java

public void initServer(VirtualWorld world, Vault vault) {

CommonEvents.PLAYER\_REGEN.register(this, (data) -> {

...

});

CommonEvents.LISTENER\_LEAVE.register(this, (data) -> {

...

String var10002;

switch (completion) {

case COMPLETED -> var10002 = **" completed a "** + objective + **"Vault!"**;

case BAILED -> var10002 = **" survived a "** + objective + **"Vault."**;

case FAILED -> var10002 = **" was defeated in a "** + objective + **"Vault."**;

default -> throw new IncompatibleClassChangeError();

// 需替换的即为这里标为粗体的6处（5种）文本

}

...

});

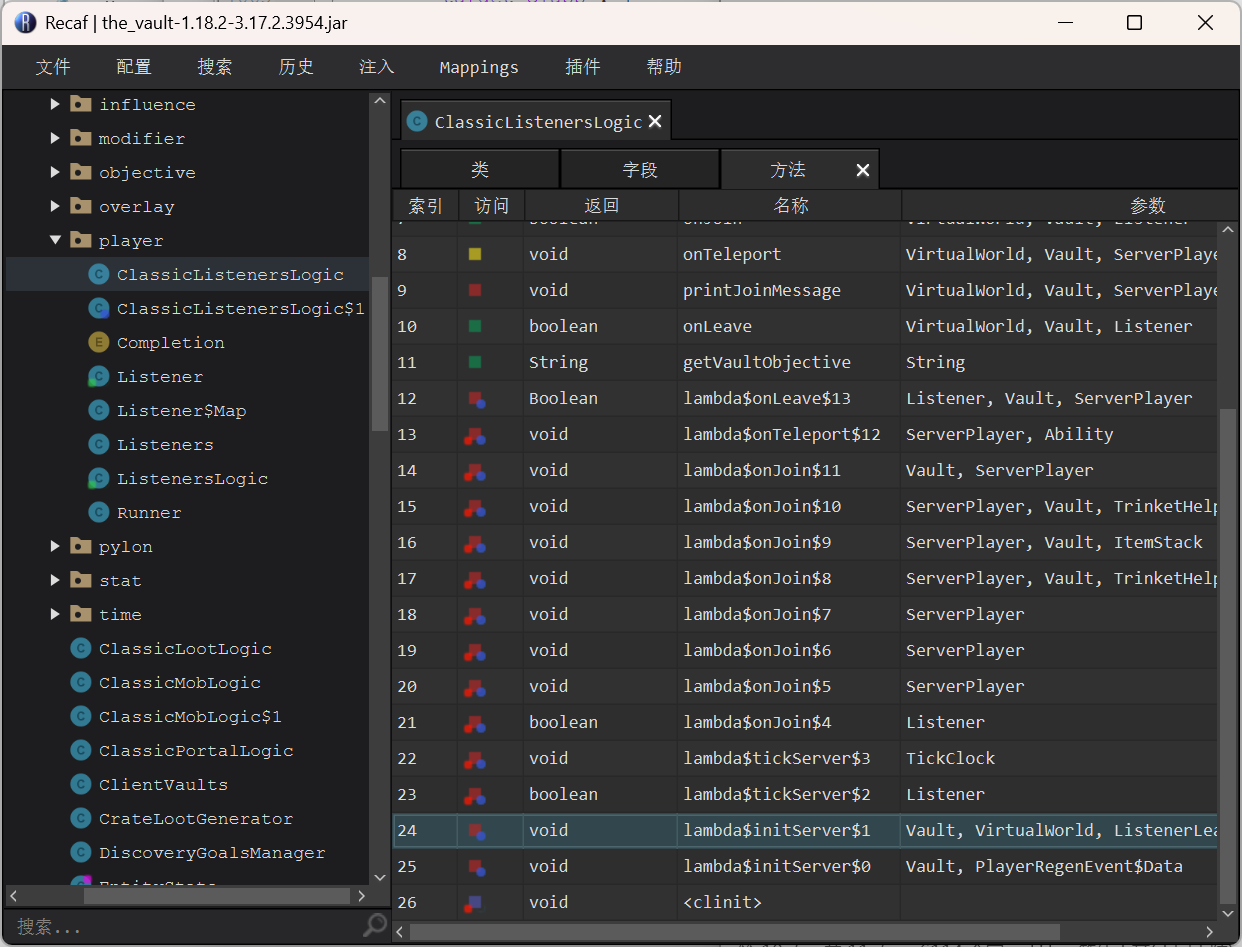
}

诸如上述代码中：

(data) -> {...}

的格式即为“lambda”。详细介绍请自行百度。

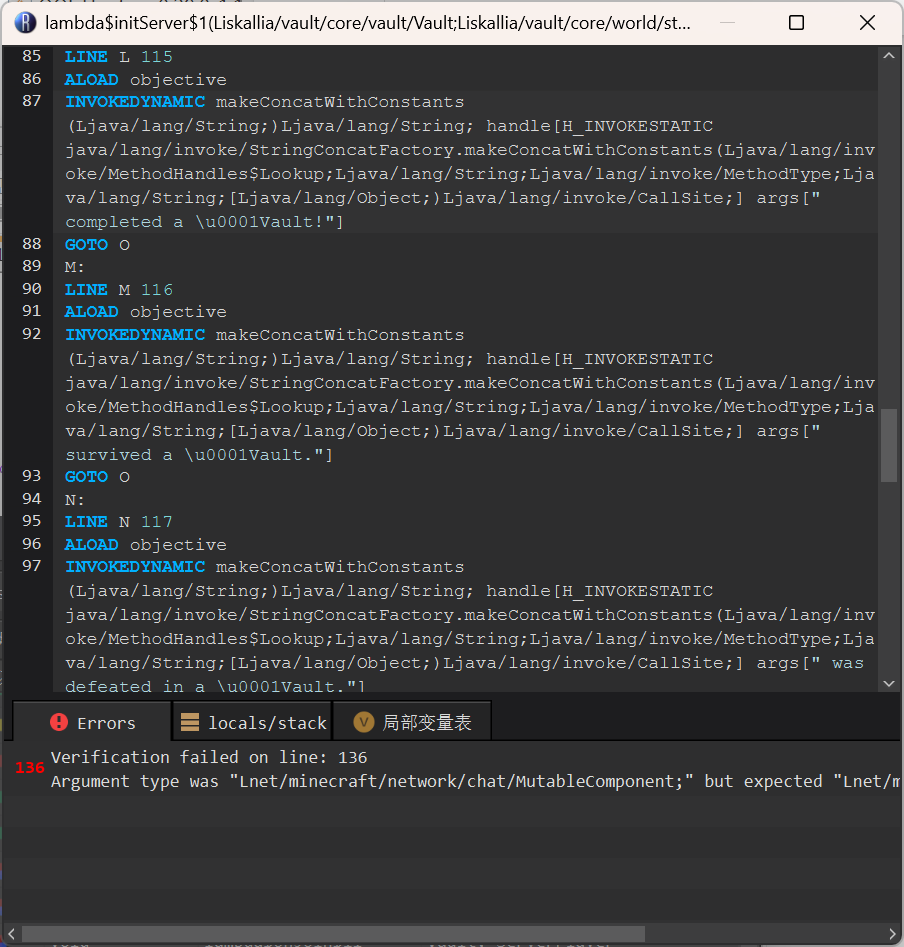
lambda并没有命名，我们该如何编写method呢？对于反编译来说，未命名的内容一般按在当前类中的出现顺序（序数）以数字命名。我们通过上述方法查看Recaf中的对应类：



可以看到，所有的lambda都命名为了“lambda$<所在方法名>$[序数]”的格式，且一个类中第1个出现的lambda的序数为0。

通过分析上述反编译代码，我们可以发现，我们要替换的文本位于initServer方法下的、当前类出现的第2个lambda中，那么序数应该为1。也就是说，我们正确的method值应该是这里的“lambda$initServer$1”。

为了确认我们的猜想是否正确，我们可以右键点击“lambda$initServer$1”这一行，选择“查看汇编代码”：



嗯，我们确实在这里找到了我们想替换的文本，证明我们的猜想没有问题。

需要注意，这里的“\u0001”代表的是文本中掺杂的其他变量，并无实际意义，我们编写翻译信息也不需要参考汇编代码。查看汇编代码只是为了确定我们要替换的文本是否在该方法中。

那么就可以编写翻译信息如下：

## the\_vault\_vp.json

...

{

"target\_class": {

"name": "iskallia.vault.core.vault.player.ClassicListenersLogic",

"method": "**lambda$initServer$1**"

},

"pairs": [

{

"key": "Vault!",

"value": "宝库世界！"

},

{

"key": "Vault.",

"value": "宝库世界。"

},

{

"key": " was defeated in a ",

"value": "被击败并离开了"

},

{

"key": " completed a ",

"value": "完成了"

},

{

"key": " survived a ",

"value": "成功存活并离开了"

}

]

},

...

还有两种情况。如果文本位于类的构造方法或构造器中，即类似于：

## Example.java

...

private String name; // 这是Example类定义的全局变量（或称为字段）

private String desc;

public Example() { // 与类名相同的方法即为构造方法。内部类也可能有构造方法

this.name = “Example Text”;

...

}

static { // 类似static{...}的方法体即为构造器。这个方法会在类初始化时静态初始化全局变量值

this.desc = “Example Description.”

...

}

private String modVersion = “1.0.0”; // 这种直接初始化全局变量值的写法也视为位于构造器内

// 以上代码中desc的初始化方法与modVersion的初始化方法是等效的

...

则method值为特殊的指定值。

构造方法的method值为“<init>”。即使有多个构造方法，也只能使用这个值定位构造方法，或者说这个值会定位所有构造方法。

构造器的method值为“<clinit>”。

#### 4.4.3、隐藏文本不放过——编写local

当文本不以天然String字符串格式出现，即以变量或参数形式出现时，此时使用local参数即可解决。

变量源（local）的使用方法为“标志+目标名”。“标志”是local使用的前缀，根据标志的不同，local拥有不同的用法或作用：

·方法调用：文本是调用其他方法获得的返回值时，可使用标志M，目标名是所调用的方法的名称。

·局部变量：文本是所在方法内的局部变量（包括实参）时，可使用标志V，目标名是变量的名称。

·方法返回：文本是当前所在方法的返回值时，使用标志R，目标名是返回值的名称。部分情况下等效于M或V。

·全局变量：如果文本是所在类的全局变量，使用标志G，目标名是该全局变量的名称。注意此用法会修改指定方法内该全局变量的值，可能影响其他代码。

接下来，我们为上述用法提供示例。（为完整显示缩进，缩小了字体）（宝藏猎人还是太全面了）

## ResearchWidget.java

...

private final String **researchName**; // 这是一个String类型的全局变量

...

private void **renderHover**(PoseStack matrixStack, int mouseX, int mouseY, float partialTicks) {

if (this.m\_5953\_((double)mouseX, (double)mouseY)) {

List tTip = new ArrayList();

tTip.add(FormattedCharSequence.m\_13714\_(this.**researchName**, Style.f\_131099\_));

// 这里调用了全局变量researchName

List conflicts;

if (this.locked) {

conflicts = ModConfigs.SKILL\_GATES.getGates().getDependencyResearches(this.researchName);

if (!conflicts.isEmpty()) {

String **tTipRequirement** = ModConfigs.SKILL\_GATES.getGates().hasEitherSkillGate(**this.**getResearchName**()**) ? " any of" : "";

// 构造局部变量tTipRequirement

tTip.add(FormattedCharSequence.m\_13714\_("Requires" + **tTipRequirement** + ":", Style.f\_131099\_.m\_131140\_(ChatFormatting.RED)));

// 这里调用了局部变量tTipRequirement

conflicts.forEach((research) -> { // 当前类中第2个lambda

tTip.add(FormattedCharSequence.m\_13714\_("- " + research.**getName**(), Style.f\_131099\_.m\_131140\_(ChatFormatting.RED)));

// 这里调用了方法getName的返回值

});

}

}

conflicts = ModConfigs.SKILL\_GATES.getGates().getLockedByResearches(this.researchName);

if (!conflicts.isEmpty()) {

tTip.add(FormattedCharSequence.m\_13714\_("Cannot be unlocked alongside:", Style.f\_131099\_.m\_131140\_(ChatFormatting.RED)));

conflicts.forEach((research) -> { // 当前类中第3个lambda

tTip.add(FormattedCharSequence.m\_13714\_("- " + research.**getName**(), Style.f\_131099\_.m\_131140\_(ChatFormatting.RED)));

// 这里也调用了方法getName的返回值

});

}

TooltipUtil.renderTooltip(matrixStack, tTip, mouseX, mouseY, Integer.MAX\_VALUE, Integer.MAX\_VALUE);

RenderSystem.m\_69478\_();

}

}

...

以上代码将分别介绍。

对于：

conflicts.forEach((research) -> { // 当前类中第2个lambda

tTip.add(FormattedCharSequence.m\_13714\_("- " + research.**getName**(), Style.f\_131099\_.m\_131140\_(ChatFormatting.RED)));

// 这里调用了方法getName的返回值

});

我们的待替换文本是getName的返回值。这个类并不是我们当前所在的类，因此使用M来编写local：

{

"name": "iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget",

"method": "lambda$renderHover$1",

"local": "MgetName" // 使用M定位getName方法的返回值

},

对于下文中第3个lambda同理。

对于：

String **tTipRequirement** = ModConfigs.SKILL\_GATES.getGates().hasEitherSkillGate(**this.**getResearchName**()**) ? " any of" : "";

// 构造局部变量tTipRequirement

tTip.add(FormattedCharSequence.m\_13714\_("Requires" + **tTipRequirement** + ":", Style.f\_131099\_.m\_131140\_(ChatFormatting.RED)));

// 这里调用了局部变量tTipRequirement

我们的待替换文本是局部变量tTipRequirement的值。在此该变量已经用String明文构造，我们可以使用：

{

“target\_class”: {

“name”: “iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget”,

“method”: “renderHover”

},

“key”: “ any of”,

“value”: “任意”

},

但是，我们正在演示local的用法（doge）。所以我们使用以下方法达成相同效果：

{

“target\_class”: {

“name”: “iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget”,

“method”: “renderHover”,

“local”: “VtTipRequirement” // 使用V定位局部变量tTipRequirement

},

“key”: “ any of”,

“value”: “任意”

},

对于：

private final String **researchName**; // 这是一个String类型的全局变量

...

tTip.add(FormattedCharSequence.m\_13714\_(this.**researchName**, Style.f\_131099\_)); // 这里调用了全局变量researchName

我们的待替换文本是类的全局变量researchName的值，因此使用G来编写local：

{

"name": "iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget",

"method": "renderHover",

"local": "GresearchName" // 使用G定位全局变量researchName

},

还有一个R没有介绍，使用示例代码：

## Example.java

...

public static String getName() {

String result = ...;

...

return result; // 需要替换该方法的返回值

}

则编写local：

{

"name": "com.dricetea.examplemod.Example",

"method": "getName",

"local": "Rresult" // 这里也可以是“RgetName”

},

如果在该方法体中，局部变量result被大量修改，则直接使用“Vresult”无法定位到其最终的值。因此直接使用“Rresult”，可以避免消耗脑力。

local的用法介绍完毕。特别注意，VP只能替换String类型的对象，如果对象不是String类型将替换失败。也就是说，无论如何，编写的目标对象（local指代的返回值/变量/参数）必须指向一个String类型的值。如果不是String类型，则需要找到对象从String类型封装前的位置进行替换。

#### 4.4.4、终极防冲突大法——编写ordinal

当我们真正执行前面编写的local时，我们发现游戏崩溃了。发生了什么？我们再回来看看这段代码：

## ResearchWidget.java

...

private final String **researchName**; // 这是一个String类型的全局变量

...

private void **renderHover**(PoseStack matrixStack, int mouseX, int mouseY, float partialTicks) {

if (this.m\_5953\_((double)mouseX, (double)mouseY)) {

List tTip = new ArrayList();

tTip.add(FormattedCharSequence.m\_13714\_(this.**researchName**, Style.f\_131099\_));

// 这里调用了全局变量researchName

List conflicts;

if (this.locked) {

conflicts = ModConfigs.SKILL\_GATES.getGates().getDependencyResearches(this.**researchName**);

// 这里也调用了全局变量researchName

...

}

conflicts = ModConfigs.SKILL\_GATES.getGates().getLockedByResearches(this.**researchName**);

// 这里也调用了全局变量researchName

...

}

}

...

我们突然发现，上述代码中调用全局变量researchName调用了3次，而后2次并不用于实际显示，而是用于其他的代码逻辑。而我们的代码直接指定了GresearchName，导致后2次所调用的researchName变量值也被替换，从而导致代码报错。我们有办法在指定GresearchName的时候，只生效于第1个用于实际显示的调用，而不生效于后2次调用吗？

操作序数（ordinal）并不是一个常规使用的参数，它是专门用于处理这种冲突情况的。当指定了ordinal时，该翻译信息将只针对所指定的ordinal进行替换，而不影响其他定位相同、ordinal不同的调用。

获取ordinal十分复杂。首先我们需要单独创建一个模块文件，其中只编写我们需要获取ordinal的这个翻译信息，且不编写ordinal。如下：

## the\_vault\_vp\_ordinalTest.json

[

{

“authors”: “DrIceTea”,

“name”: “The Vault 硬编码汉化测试”,

“desc”: “宝藏猎人核心模组汉化v1.0-教程版-提取ordinal测试”,

“mods”: “the\_vault”,

“dynamic”: false,

“i18n”: false

},

{

“target\_class”: {

"name": "iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget",

"method": "renderHover",

"local": "GresearchName"

},

“key”: “Backpacks”,

“value”: “小背包”

}

]

然后要修改config.json：

## config.json

{

“mods”: [

“the\_vault\_vp\_ordinalTest” // 我们只使用这一个模块文件

],

“class\_patch”: false,

“debug\_mode”: {

“is\_enable”: true, // 必须启用调试模式

“use\_cache”: false, // 必须不使用缓存

“output\_format”: “<source> -> <target> in <class> M-[<ordinal>] | <info>”,

// output\_format为最重要的获取ordinal的值，建议和我的内容保持一致

... // 其余内容同前

}

}

然后我们启动一次游戏，让游戏崩溃。崩溃后，我们前往根目录下的logs文件夹，打开latest.log。

接下来，我们使用VSCode的搜索功能，搜索“Trying replacing!”。每一个搜索结果代表我们的模块文件令VP进行了一次替换。文件中从上到下的每个搜索结果与类文件中从上到下的每次引用一一对应。

“Trying replacing!”所在行的下一行即为文本替换信息。

注意，并不是所有替换操作都是我们要找的操作。当替换操作没有触发local的指定效果时，ordinal会显示为“unknown”。这类替换操作应当被忽略。

按照我们的反编译代码，我们应该会找到3个带有ordinal的操作，需要的应该是第一次替换的结果，即：

## latest.log

...

[254月2025 00:53:49.818] [modloading-worker-0/INFO] [VaultPatcher/]: [VaultPatcher] **Trying replacing!**

Backpacks -> 小背包 in iskallia/vault/client/gui/screen/player/legacy/widget/ResearchWidget **M-[24]** | TranslationInfo{targetClassInfo=TargetClassInfo{name='iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget', method='renderHover', local='GresearchName', ordinal=-1, matchMode=FULL, localMode=GLOBAL\_VARIABLE}, pairs={['Backpacks'='小背包'}}

...

按照我们使用的output\_format，“M-[24]”的中括号内的数字即为操作序数（ordinal），将操作序数填入翻译信息即可：

## the\_vault\_vp.json

...

{

“target\_class”: {

“name”: “iskallia.vault.client.gui.screen.player.legacy.widget.ResearchWidget”,

“method”: “renderHover”,

“local": “GresearchName”,

“ordinal”: 24 // 注意ordinal直接为Integer格式，因此不需要引号

},

“pairs”: [

{

“key”: “Backpacks”,

“value”: “小背包”

},

... // 其他键值对

]

},

...

将config.json改回去之后，你可以再启动游戏，就能发现替换成功了。

## 5、检查与调试

VSCode会自动检查你的JSON文件格式是否正确，请及时修改。

完成编写后，可将配置文件放入正确的config位置，然后启动游戏测试，查看待替换文本是否成功替换。

替换失败可能是以上任意环节出现错误，请返回逐一检查。

经过完整的检查与调试后，将config.json的debug模式关掉：

## config.json

...

“debug\_mode”: {

“is\_enable”: false, // 不启用调试模式

“use\_cache”: true, // 启用VP缓存，可以加快第2次及之后的游戏启动速度

...

}

...

然后就可以发布你的硬编码汉化啦！

特别注意：硬编码汉化是基于特定mod文件编写的，因此跨游戏版本使用、甚至跨mod版本使用都有一定的失效隐患，因此建议在模块文件的文件名特别标注mod的游戏版本和mod版本，最好以mod文件名为基础编写。

## 6、进阶教程

接下来请欣赏《MC模组硬编码大赏之类人的群星闪耀时》（doge）。

### 6.1、超大mod文件

当编写宝藏猎人的硬编码汉化时，我发现：由于The Vault模组体积过大，过于复杂，Recaf有时会罢工，反编译并不能完整生成。这对编辑硬编码汉化是巨大的阻碍，而Recaf确实有这个问题：生成并查看大体积的反编译代码时效率不佳，且容易出现bug。

因此我们可以使用替代方案：使用Java Decompiler（我们简称为JDec）获取mod反编译代码。

首先我们要获取JDec。你可以通过我提供的链接获取这个工具。可以查看前面的“基础准备”章节。

获取到JDec之后，你需要将它放到一个文件夹中，然后在该文件夹下创建一个txt文件：

## start.txt

java -jar java-decompiler.jar ".\ori\the\_vault-1.18.2-3.17.2.3954.jar" .\out

这段代码中，你需要将“java-decompiler.jar”替换为你获取到的JDec的文件名，“the\_vault-1.18.2-3.17.2.3954.jar”替换为你想要进行反编译的mod文件名。

然后，将以上文件改名为“start.cmd”。注意，改名后，你依然可以将其用VSCode打开以进行编辑。

再然后，在该文件夹下新建一个ori文件夹。

每次你要反编译一个新的mod文件时，你就先更改上述mod文件名，然后将你要进行反编译的mod文件放置在这个文件夹下的ori文件夹中，接下来双击运行start.cmd，等待窗口的工作结束。

窗口结束后，你会发现文件夹中的out文件夹里面有一个和你的mod文件名相同的“.jar”格式的文件。使用压缩软件将其解压，即可获得一个经过反编译后的文件夹。

将这个文件夹用VSCode打开，你就可以像用Recaf一样查看反编译文件了！

注意，这个方法并不能完全替代Recaf，当你开始编辑就知道我说的意思了。

### 6.2、硬编码汉化的更新

当mod更新时，硬编码汉化可能也需要更新。但是我怎么知道哪些部分是需要更新的呢？

现在并没有较好的方案。我自己的方案是：

1. 通过JDec工具获取模组的反编译文件（见上文6.1）；
2. 通过文本比较工具（我自己在用的是Beyond Compare 4）比对更新前与更新后的mod反编译文件，获取mod的更新内容；
3. 逐一查阅更新内容，查看有无文本更新，以及对已有的硬编码汉化所对应的代码内容的修改，通过这些更改对模块文件进行修改；
4. 完成修改后启动游戏查看是否完整覆盖（同“检查与调试”章节）。

如果你在汉化的是MCreator模组，更新可能更加方便快捷，因为硬编码内容出现的位置十分固定且明确。

当然，持续维护本身就是一件难事，谁又愿意做呢……

### 6.3、怎么替换不上？

我明明已经定位到文本的位置了，但是替换一直失败？可能是以下原因：

1. 对于较远古的版本（1.12及以前）VP可能会有部分bug，请多尝试各种不同的替换姿势。
2. 你的键值对的key必须是**完整的字符串**，即不能有任何缺漏的、两个引号之间的所有字符，一个空格也不能放过！如果还是local定位，那你就要确定这段文本**到底是**什么样的，可能它前面或后面有空格，也可能原文本中间的空格是其他空白字符——谁知道呢？
3. 当编写了多个指定同一个类的翻译信息时，只有**最后的一个翻译信息**能够生效！这里的“最后一个翻译信息”指的是所有翻译信息的目标对象【target\_class(es)】中最后一个指向该类的目标对象。这是VP现阶段的bug，所以你可以尝试使用动态替换（仅限1.17+），详见下文“动态替换”。

或者，你可以发在群里请求开发者和其他编写人员的帮助！

### 6.4、动态替换

我的文本真的替换不上！VP反编译就这？难道就到此为止了？

别急，我们有高级方案——使用“**动态替换**”！

首先声明：此方法仅限1.17以上。

动态替换是指VP在游戏运行中**实时**将显示的特定文本替换为其他文本的替换方式。

注意：除非asm替换始终失败，否则不到万不得已，尽量不要使用动态替换，它对性能的影响比asm大得多。

#### 6.4.1、动态替换的具体方法

首先，新建一个模块文件，开启动态替换：

## the\_vault\_vp\_dynamic.json

[

{

“authors”: “DrIceTea”,

“name”: “The Vault 硬编码汉化-动态替换”,

“desc”: “宝藏猎人核心模组汉化v1.0-教程版-动态替换内容”,

“mods”: “the\_vault”,

“dynamic”: **true**, // 启用动态替换功能

“i18n”: false

},

// 这里留出空行，后续编写翻译信息就在此处继续编写

]

然后，在config.json中添加这个使用动态替换的模块文件。然后就像其他模块文件一样，完成该文件即可。

动态替换的翻译信息编写方法和asm替换相同，只有一部分小区别，我很快就能讲完：

#### 6.4.2、空白的target\_class

在动态替换模式下，target\_class(es)可以被省略，但要有一个空的target\_class。例如：

## the\_vault\_vp\_dynamic.json

...

{

“target\_class”: {},

“pairs”: [

{

“key”: “Crumpled Paper”,

“value”: “皱巴巴的纸”

},

... // 其他键值对

]

},

...

（Tips：这个示例甚至是宝藏猎人正在用的。它的用途是翻译宝库中某些彩蛋生成的、改了名字的原版物品。这些物品的名字写在了mod附带的结构NBT文件中……它不是mod代码，VP对它毫无办法。）

这样编写之后，VP就会将**所有的**符合key值的显示文本**实时**替换为对应的value值。

#### 6.4.3、半匹配

正常的键值对使用全匹配，即当且仅当整个字符串与key的值相匹配时，才会进行替换。

对于动态替换，如果key的第一个字符是“@”，那么该键值对会自动优化为**半匹配**键值对：

## the\_vault\_vp\_dynamic.json

...

{

"target\_class": {},

"pairs": [

{

"key": "COMMON",

"value": "@普通"

},

...

]

},

...

（Tips：这个示例甚至也是宝藏猎人正在用的……能体会到作为宝藏猎人汉化维护者的我的艰辛吗……）

半匹配键值对会替换整个字符串中与key的值**相匹配**的那部分内容。此例中，物品提示中显示的文本为：

Rarity: COMMON

这个文本是坚不可摧的一个整体文本，所以当我们使用动态替换进行全匹配时，理所当然的替换失败。

而当我们启用半匹配后，这段文本在游戏中就会替换成功：

Rarity: 普通

#### 6.4.4、循环调用？截获！

接下来想象下面这种情况：

在mod代码中，作者将用于显示的文本在另一处读取，并用于其他的代码逻辑，也就是将这段文本变形，然后用于读取对应的物品/实体/标签等等。

很出生的写法对不对？这也是宝藏猎人的实际案例。

读取显示的文本的方法为：

## GroupListElement.json

...

public GroupListElement(ISpatial spatial, BestiaryScreen parentScreen) {

...

Iterator var4 = GroupUtils.**getEntityGroupNames**().iterator();

// 这里读取的getEntityGroupNames方法的结果在另一个地方直接用于显示

// getEntityGroupNames的返回值甚至还是一个List<String>，不是直接的String

while(true) {

String groupName;

PartialEntityGroup group;

do {

if (!var4.hasNext()) {

return;

}

**groupName** = (String)var4.next();

// 这里读取上面获取的var4的文本

Optional byName = GroupUtils.getFilterByName(**groupName**);

// 这里将读取的groupName用于逻辑判断，逻辑判断中只有英文的原文本才能正常返回结果

if (!byName.isPresent()) {

break;

}

Object var8 = byName.get();

if (!(var8 instanceof PartialEntityGroup)) {

break;

}

group = (PartialEntityGroup)var8;

} while(ModConfigs.BESTIARY.getHiddenGroups().contains(group.getId()));

...

}

...

GroupUtils工具类主要方法如下：

## GroupUtils.json

...

public static List getEntityGroupNames() { // 这是读取文本的方法

return ENTITY\_GROUPS.keySet().stream().map(GroupUtils::**getEntityName**)

.map(Component::getString).sorted().toList();

// getEntityName方法为生成文本的方法

}

public static Component getEntityName(EntityPredicate filter) { // 生成文本的方法

// 注意出生代码：这里直接封装成文本组件（Component）了

if (filter instanceof PartialEntityGroup group) {

return group.getId().m\_135815\_().equalsIgnoreCase("fighter")

? new TextComponent("**Dweller**")

: TextUtil.**formatLocationPathAsProperNoun**(group.getId());

// 这里就是将文本封装的主要方法

} else {...} // 这里就是不重要的代码了

}

...

然后是formatLocationPathAsProperNoun方法：

## TextUtil.java

...

public static TextComponent formatLocationPathAsProperNoun(ResourceLocation location) {

String[] split = location.m\_135815\_().split("\_");

AtomicReference name = new AtomicReference("");

Arrays.stream(split).forEach((s) -> {

String var10001 = (String)name.get();

name.set(var10001 + org.apache.commons.lang3.StringUtils.capitalize(s) + " ");

});

return new TextComponent(((String)name.get()).**trim**());

// 总之封装String的最后一步是这个trim方法

// 这里不能用local的R，看过前面教程的你请想想为什么

}

...

（好笑吗？这全都是我的辛酸泪……）

在这么出生的写法之下，我们还有办法替换显示的文本吗？答案是可以！

宝藏猎人中采用的处理方法是：分别找到显示和读取的方法，然后在显示方法中将英文替换为中文，在读取方法中将中文替换为英文，这样即可完美解决。

## the\_vault\_vp.json

...

{

"target\_class": { // 用于将显示的文本替换

"name": "iskallia.vault.util.TextUtil",

"method": "formatLocationPathAsProperNoun",

"local": "Mtrim"

},

"pairs": [

{

"key": "Guardian",

"value": "守卫"

},

{

"key": "Horde",

"value": "集群怪物"

},

...

},

{

"target\_classes": [ // 用于在代码逻辑前将文本替换回原文本

{

"name": "iskallia.vault.client.gui.screen.bestiary.element.GroupListElement",

"method": "<init>",

"local": "Mnext"

},

{

"name": "iskallia.vault.client.gui.screen.bestiary.BestiaryScreen",

"method": "selectGroup",

"local": "VgroupName"

} // 这个目标对象是另一处用于处理的用法，不再赘述

],

"pairs": [

{

"key": "守卫",

"value": "Guardian"

},

{

"key": "集群怪物",

"value": "Horde"

},

...

]

},

...

这段代码至今运行得十分良好……

## 7、硬编码汉化的原则

1.最核心的原则是对源码造成尽可能小的影响，如果文本兼有数据存储或计算的作用，例如会写入nbt或玩家数据的内容，尽量从最终显示或渲染的方法去替换，而不是从来源处。

2.鉴于硬编码汉化的强大功能，进行硬编码汉化的译者应尽力保证汉化的100%覆盖，应译尽译，这才是使用VP的最终目的。

3.尽量使用asm替换，即非动态替换，这样更加减少运行时的性能消耗。实在无法使用asm替换时，再使用动态替换。

4.一定要经过完整测试之后再发布硬编码汉化！这样是对自己和对玩家负责。

最后，希望各位译者成功掌握VP这个顶级硬编码汉化工具，愿Minecraft社区翻译越来越好！

by 醉冰茶DrIceTea