## 概述

### 序言

开头放图：(～o￣▽￣)～o o～



（希望你们不要对我写的插件产生这种感情哦。）

### 思维导图

（内容不多，等我想好了再画）

## 命名

插件的命名规则在课程中提到了多次，但是不妨再看一次哦，毕竟习惯很重要。

### 冲突问题

自己写插件，经常会遇到与其他插件冲突的问题。

但是最常见冲突的有：

1. **变量名重复**
2. **方法名重复**
3. **覆盖了rmmv的原方法以及底层函数**

要绕开这些冲突问题，需要遵循下面规则：

1）与别人变量名尽可能不重复，并且自己定义的变量名也尽可能不相互重复。

2）尽可能继承，只能重写的部分，要标注出来。

3）多封装成类，面向对象。

4）明确存储数据与临时数据。

### 命名规则

下面介绍drill插件的命名规则：

|  |
| --- |
| //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<  // 插件简称 BGi（Battle\_Gif）  // 临时全局变量 DrillUp.g\_BGi\_xxx  // 临时局部变量 this.\_drill\_BGi\_xxx  // 存储数据变量 $gameSystem.\_drill\_BGi\_xxx  // 全局存储变量 无  // 覆盖重写方法 无  // |

为了尽可能识别出我自己所写的方法与变量，大部分插件都有下面格式：

1）临时全局变量要有”g\_”前缀。

比如Drill.g\_BGi\_xxxx，由于在代码最外层，使用时所以尽可能作为只读参数。

2）所有变量要有作者简称，可以完美区别 自己的变量 与 系统变量/别人插件变量。

比如变量为.\_drill\_BGi\_xxx，函数为.drill\_BGi\_xxx()。

3）每个插件都有自己的专有简称，确定这个变量只在当前插件中作用。

比如BGi为插件简称，所有变量、函数都最好加BGi。

4) 如果其它插件调用了该插件的函数，那么两个插件的专有简称都应该写上。

比如.drill\_BGi\_CGi\_xxxx()，BGi 和CGi表示两个插件的交互。

以下为部分命名的写法，过目即可：

1）rmmv方法下 + “drill”前缀 + 插件专有简称 = 该插件的方法

|  |
| --- |
| Spriteset\_Battle.prototype.drill\_GFB\_updatePluginCommand = function() {  if( this.\_drill\_GFB\_tank.length == 0 ){ return }  } |

2）drill类下 + “\_drill”前缀 = 自己类下的自定义变量

|  |
| --- |
| Drill\_GFB\_StyleSprite.prototype.initialize = function(bossBind,enemy) {  Sprite.prototype.initialize.call(this);  this.\_drill\_bossBind = bossBind;  } |

3）drill类下 + 系统变量 = 自己类中继承/控制父类的变量内容/方法

|  |
| --- |
| Drill\_GFB\_StyleSprite.prototype.initialize = function(bossBind,enemy) {  this.opacity = 0;  } |

4）当然，有时候可能会图简单，可能会留下简单加个“\_”来区分变量的临时变量。虽然看起来比较方便，但是还是需要稍微留意一下可能重复的隐患。

|  |
| --- |
| this.\_move = 0;  this.\_time = 0; |

5）另外，要适应习惯超长的变量名与函数。不要嫌变量名太长，变量的作用越精细，就要越长。短变量名只适合局部范围临时使用。

（大多数功能中，没有命名域的区分，如果分子类，会造成更多麻烦，所以一般会起很长的函数名）

|  |
| --- |
| Game\_Map.prototype.drill\_COFA\_getCustomPointsByIdWithCondition = function( event\_id,  def\_area\_id, condition ) {  var area = this.drill\_COFA\_getCustomPointsById( event\_id, def\_area\_id );  return this.drill\_COFA\_selectPoints( area,condition );  } |

## 经验分享

### 经验短句

(´⊙ω⊙`) 一不留神发现自己已经写了一大堆插件。或许你们会从我写的脚本里面寻求一些灵感。这里我做一些经验短句的总结，给你们指一下路。

|  |
| --- |
| 了解别人的插件，比如mog插件、drill插件，只对你写rmmv插件有帮助，对你实际编程能力的帮助并不大。 |

(1. 写插件最大的缺点是，整个思路容易偏向 **面向过程** ，而很少 **面向对象**。

(2. 插件之所以是插件，是因为这些都是细小的碎片内容，而不是一个大而完整的对象系统。长期写插件不能给你带来好的编程思想。

(3. 面向过程写出来的脚本，**只能将脚本复制粘贴，耦合度太大**，如果你没有很深的面向对象基础，你会不由自主地偏向使用面向过程的方法来写脚本写插件。

|  |
| --- |
| rmmv有自己的框架，这就意味着难以移植到其他平台上。 |

(1. 实际上rmmv是一套定制化后的程序，基于pixi.js，有很多自己定义的东西在里面。最鲜明的例子是Bitmap类。

(2. 官方pixi定义了Sprite、texture、render等基础结构，

(3. rmmv将这些结构封装起来，形成一个中介对象Bitmap，你只要操作bitmap就可以几乎做到所有的渲染、贴图效果。这就意味着，如果你的sprite要移植到其它平台上，比如使unity可以用，那么你必须将rmmv定义的bitmap也移植过去。（另外unity，早已不支持js，现在游戏引擎之间隔阂都那么深嘛……）

（Bitmap定义在 rpg\_core.js中）

|  |
| --- |
| 不算注释，写出100行代码的功能需要一小时，写出200行代码的功能需要四小时，写出400行代码的功能需要一天，写出800行代码的功能需要四天，写出1600行代码的功能需要三个星期。我没有做到写出5000行代码，因为那样的代码都是一个巨大的引擎系统。你需要留意达到5000行等级的人，因为他们是 真 大神，那样的代码才真的值得学习。 |

(1. 代码需要经过反复提炼、打磨，才能形成最终的结果。并不是简单的代码累加。把数个功能合并到一个插件里面。（忍不住吐槽……mog很多代码都是功能集合体，看起来量大，而实际上可以划分成很多小的部分。而且定制化严重。yep才是真神仙，mog就是个业余……）

(2. **你会发现超过5000行代码的脚本一般都是一个诺大的游戏系统。比如STG系统，卡牌系统、战旗系统。这些系统，都不是一般程序员能做出来的。**不过从用户角度来说，不管多少行，配置太少，就不是好脚本。

（其实我也觉得那些大神的脑袋有洞，用了那么多时间写出的好程序，为什么不提供文档、不提供可规划参数……他们为什么一点写注释、写软件文档的习惯都木有……）

|  |
| --- |
| 如果你想更深入地学习语言，不要从js、python等脚本语言开始。要从C++、java等开始打基础。 |

(1. 有许多重要概念js是无法教你的，比如：指针、映射、堆栈、链表、泛型 等。

(2. js中的概念：传地址、传引用、深拷贝、浅拷贝。这些都是c++的指针进化后的产物。

举个简单的例子，你如果只学js，你会无法理解为什么js中：

console.log( {}=={} ); 和 console.log( []==[] ); 输出的是false。

（另外。我的工作用的C++。我恨C++。）

|  |
| --- |
| 现在我写插件是这样的流程：灵感 -> 快速实现 -> 写一个例子 -> 完善插件指令与注释 -> 完善大部分例子 -> 检查与其他插件兼容性 -> 设计示例 -> 大量事件性能测试 -> 性能测试表记录 -> 插件说明文档 -> 其它细节补充 -> 最后封装版本 |

(1. 也可能是个人强迫症所在，经常会写文档和性能测试，毕竟自己是在精雕细琢，而不是滥竽充数。不过，在一般正常情况下，程序员对于文档和性能并没有特别的追求，只要实现功能就好。（毕竟大部分程序员不会为自己的代码全权负责，卷一波就跳槽。）

(2. 很多定制化的游戏/插件，只执行了前三四个步骤，就没继续往下了。因为它们一般只要实现功能达到目的就可以了。所以经常会出现下图的有趣现象：



### rmmv脚本的坑

如果你是**自己写脚本、写插件**，这里提供一些插件的千万注意的地方：

1. 不要把存储用的变量数据写外面，要扔进$gameSystem中。

我的插件中，将DrillUp.xxx叫做：临时全局变量。

**临时，指关掉游戏后变量会被重新赋值。**

**全局，指如果不关游戏进另一个存档，值是跨存档的。**

比如，DrillUp.xxx = 12，能直接改变当前游戏的赋值情况。

但是，你存档，读档，DrillUp.xxx如果不变，仍然 = 12。

1. $gameSystem里面千万不要写入object。

$gameSystem会识别字符串、json和数组，然后转码成存档进行存储。但是如果写入了object，插件会报错无法识别，**这时候，报错的位置完全不是你犯错的地方**，你需要付出代价地毯式搜索每个gameSystem用到的地方。W(ﾟДﾟ)w

如果你要全局，并且写入object，（比如预加载一些图片）写入$gameTemp里面。

1. 尽量继承方法，如果是覆写方法，要留下记录。

**我写完插件后，我就不记得我写了哪些方法了**。(´⊙ω⊙`) 系统的方法和我写的方法完完全全柔和在一起，完全不知道哪些方法是否真的起作用了，还是仅仅写了·一个废方法扔在里面。

为此，我将变量、方法都加了drill\_ 前缀，就是用来防止不知道方法的具体作用又不敢删的情况。

1. 不要在update函数里面new Sprite()，任何对象都不应该在update里new。

这个是基础常规操作了，sprite是个大类，频繁new会导致大类反复销毁重建。一次性new100个，都比写在update强。

不过，有时候我也经常会当心卡顿的问题，因为你写的代码经常感觉不到哪里出现了浪费计算量的情况，好像有轻微卡顿，又好像没有。

最典型的例子就是，存档掉帧的问题，一直找不到原因，时好时坏。

总之，尽量避免在update里面new任何对象。

1. 尽可能复用rmmv类。如果你定义了一个比较大的类，你最好尽可能地去封装成完整对象，并写文档和注释，说明调用情况，有哪些功能及子功能。

Rmmv有很多精华一般类。但是，没注释(ノ｀⊿´)ノ。

如果不知道那些类，你很可能会选择重复造轮子，这样会浪费编写时间，还会消耗后期维护的成本。

如果你开始编写大类，尽可能地封装它。这些类就像一张巨大的根茎网，在各个插件和结构中扎根。如果不及时封装，后期会像藤蔓一样，不仅斩不断，还不能丢。