## 简报

### 任务

|  |
| --- |
| 任务目标：在地图界面中放一个贴图。  任务帮助：当前目录下，提供了一个插件模板：Drill\_SimpleCourseA，你需要在模板中编写相关代码，来熟悉基本结构。 |

### 基本意识

1.所有底层和插件都是相通的，可以直接调用或覆写。也就是说你的插件函数名如果乱起名，有几率覆盖掉底层函数。

所以，函数名的命名唯一性变得极其重要，后面章节会介绍。

2.所有脚本，都是基于ES5的js格式。

没有import指令。没有const、没有let。没有箭头函数。

（你可以写并且能跑通，但是要注意有些旧机器环境是最多支持到ES5的，最好稳定性优先。）

ES5的特性：<https://www.w3school.com.cn/js/js_object_es5.asp>

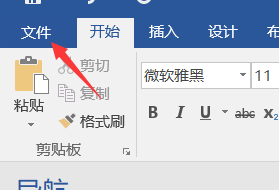
关于稳定的函数写法，可以参考当前目录的”基本函数查询表.docx”。

### 去掉word红线

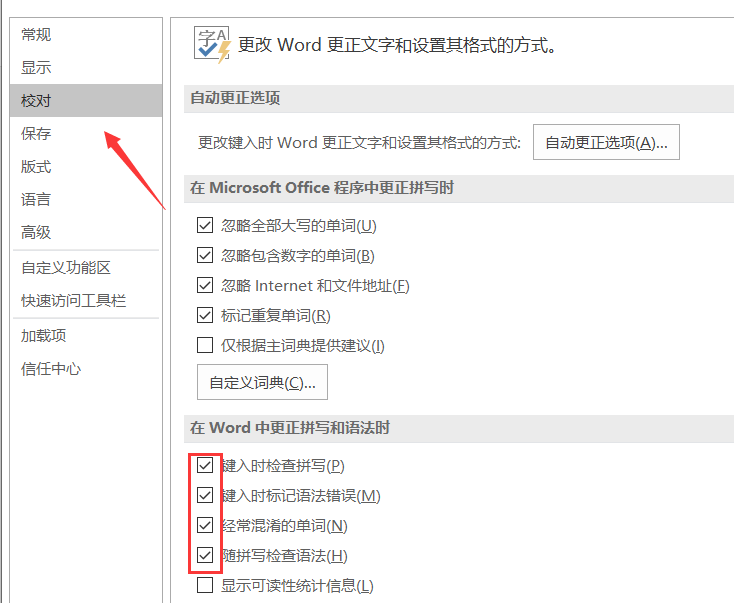
Word的红线会非常影响代码查看，这里必须去掉。



点击 文件 》 选项 ，进入选项窗口。



选择 校对 项，然后把下图的四个勾选去掉。



红线就不会出现了。

## 开始课程

### 命名

#### 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。

（c++过来的，长变量名也习惯了）

（另外，貌似跟我抱怨太长的都是写程序的女生……）

|  |
| --- |
| 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 );  } |

### 开始写插件

**了解了命名规则后，下面开始写插件，先把下列步骤完整过一遍。**

#### 1）资源文件夹代码

|  |
| --- |
| //=============================================================================  // \*\* 资源文件夹  //=============================================================================  ImageManager.load\_CourseA = function(filename) {  return this.loadBitmap('img/Course\_\_A/', filename, 0, true);  // "0, true"分别表示 色调值和抗锯齿  }; |

#### 2）插件指令代码

|  |
| --- |
| //=============================================================================  // \* 插件指令  //=============================================================================  var \_drill\_SCA\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCA\_pluginCommand.call(this, command, args);    if (command === ">脚本教学课程A指令") {  $gameTemp.\_drill\_SCA\_switch = true;  }  }; |

#### 3）地图界面代码

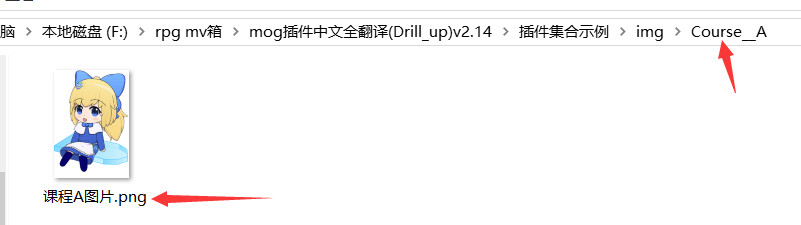
|  |
| --- |
| //=============================================================================  // \*\* 地图界面  //=============================================================================  //==============================  // \* 地图 - 帧刷新  //==============================  var \_drill\_SCA\_update = Scene\_Map.prototype.update;  Scene\_Map.prototype.update = function() {  \_drill\_SCA\_update.call(this);    this.drill\_SCA\_updateCreateSprite(); //创建一个贴图  } |

|  |
| --- |
| //==============================  // \* 帧刷新 - 创建一个贴图  //==============================  Scene\_Map.prototype.drill\_SCA\_updateCreateSprite = function() {  if( $gameTemp.\_drill\_SCA\_switch == true ){  $gameTemp.\_drill\_SCA\_switch = false;    var temp\_sprite = new Sprite();  temp\_sprite.x = 100;  temp\_sprite.y = 100;  temp\_sprite.zIndex = 10;  temp\_sprite.bitmap = ImageManager.load\_CourseA("课程A图片");    this.\_drill\_SenceTopArea.addChild( temp\_sprite ); //最顶层（层级后面再解释）    this.\_drill\_SCA\_sprite = temp\_sprite;    //this.addChild( temp\_sprite );  //你可以直接addchild，但是这种操作，会使得图片层级的先后顺序紊乱  //因为rmmv不会识别zIndex，所以层级的排序后期要自己写。  }  } |

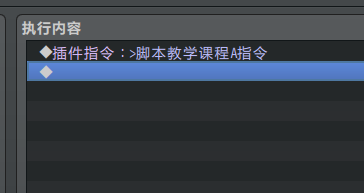
#### 4）加入插件



#### 5）文件夹放置



#### 6）创建一个事件



#### 7）功能测试

测试时，按F8，查看插件是否成功载入。如果你的插件拼写错误，控制台会出现相关报错。



进入游戏后，跟指定的事件对话，会冒出一张贴图。功能完成。



在你完成上述流程之后，接下来我们开始剖析每个过程的细节。

### 资源文件夹代码 - rmmv的manager管理模式

资源文件夹的代码如下：

|  |
| --- |
| //=============================================================================  // \*\* 资源文件夹  //=============================================================================  ImageManager.load\_CourseA = function(filename) {  return this.loadBitmap('img/Course\_\_A/', filename, 0, true);  // "0, true"分别表示 色调值和抗锯齿  }; |

调用这个函数的地方如下：

|  |
| --- |
| temp\_sprite.bitmap = ImageManager.load\_CourseA("课程A图片"); |

从上面几行代码来看，你会发现这个函数的类，是 ImageManager 。

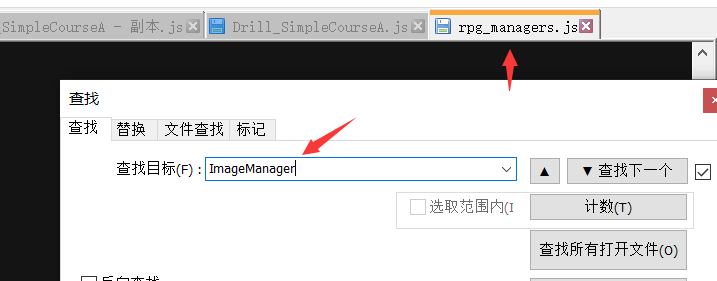
这个类在rpg\_managers.js中可以找到定义。

那么就去源码中看看结构吧。

进入rpg\_managers.js的源码翻看前，

这里介绍一个常用功能：ctrl+f**查找功能**。

没错，手动查找。



现在大多数编辑器都能够自动识别并且划分 类与函数，自动列出成员。你可以快速索引到想找的类。

但是，rmmv的脚本代码，毕竟ES5，多数编辑器不待见，所以最好的方式还是自己多用ctrl+f查找。

另外，建议用notepad++查找，vscode的查找功能一直都那么不好用，面板太小，不能并行查找。

通过查找功能，找到下面的函数：

|  |
| --- |
| ImageManager.loadSvEnemy = function(filename, hue) {  return this.loadBitmap('img/sv\_enemies/', filename, hue, true);  };  ImageManager.loadSystem = function(filename, hue) {  return this.loadBitmap('img/system/', filename, hue, false);  };  ImageManager.loadTileset = function(filename, hue) {  return this.loadBitmap('img/tilesets/', filename, hue, false);  };  ImageManager.loadTitle1 = function(filename, hue) {  return this.loadBitmap('img/titles1/', filename, hue, true);  };  ImageManager.loadTitle2 = function(filename, hue) {  return this.loadBitmap('img/titles2/', filename, hue, true);  };  ImageManager.loadBitmap = function(folder, filename, hue, smooth) {  if (filename) {  var path = folder + encodeURIComponent(filename) + '.png';  var bitmap = this.loadNormalBitmap(path, hue || 0);  bitmap.smooth = smooth;  return bitmap;  } else {  return this.loadEmptyBitmap();  }  };  ImageManager.loadNormalBitmap = function(path, hue) {  var key = this.\_generateCacheKey(path, hue);  var bitmap = this.\_imageCache.get(key);  if (!bitmap) {  bitmap = Bitmap.load(decodeURIComponent(path));  bitmap.addLoadListener(function() {  bitmap.rotateHue(hue);  });  this.\_imageCache.add(key, bitmap);  }else if(!bitmap.isReady()){  bitmap.decode();  }  return bitmap;  }; |

**下面开始读代码，读代码的过程是一个展开黑箱子的过程：**

1.你会发现ImageManager中也有许多相似的函数，他们都指向一个函数loadBitmap。

2.loadBitmap函数中，if对filename为空的情况进行了自动过滤，并且往下分为loadNormalBitmap和loadEmptyBitmap两个分支函数，这两个分支都返回了bitmap对象。

3.在loadNormalBitmap函数中，发现如果每次读取一个bitmap，都会添加缓冲this.\_imageCache.add(…)。并且，色调的变化，是在bitmap读取完成之后才进行改变的。

4.你还可以继续顺着Bitmap.load往下深入，Bitmap.load 的定义在rpg\_core.js中。进而找到 资源加载器 和资源加密器。这里暂时不继续深入了，如果你有兴趣，可以自己去看看。

【黑箱【黑箱】

**上述的进入引擎内读代码的过程，是写插件的常用过程。如果你有疑问，时不时通过关键字查询，翻出代码熟悉熟悉，自然水到渠成。**

全部游戏底层代码都在rpg\_xxx.js六个文件中，再加上一个pixi引擎底层。

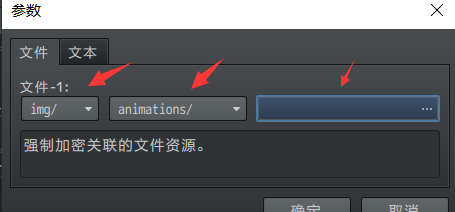
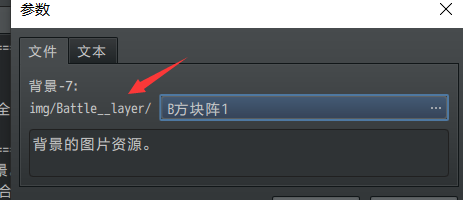
**从目前的深入的状态来看，我们可以知道下面几个知识点**：

1.反复调用loadBitmap不会重新加载资源，因为Manager有图片缓冲机制。

2.ImageManager.load\_CourseA (null) 和 ImageManager.load\_CourseA ("")都可以返回一个对象，只不过是个空对象，这不会造成程序报错。

3.你完全可以通过依葫芦画瓢，自己写一个ImageManager.load\_xxx (filename)函数，而且rmmv基本文件夹都是这样编写的，比如ImageManager.loadSystem对应img的system文件夹。

另外，需要一提的是，loadBitmap是支持多级文件夹的，’img/aaa/bbb/ccc/’这种写法可以支持。但是，rmmv的游戏编辑器，只能有两层文件夹，所以这里都统一规定为img往下只能有一级的文件夹深度。



### 插件指令代码 - 继承与覆写

插件指令的代码如下：

|  |
| --- |
| //=============================================================================  // \* 插件指令  //=============================================================================  var \_drill\_SCA\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCA\_pluginCommand.call(this, command, args);    if (command === ">脚本教学课程A指令") {  $gameTemp.\_drill\_SCA\_switch = true;  }  }; |

你可以在很多插件中见到这种类似的结构：

“var \_drill\_SCA\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;”

通过给一个全局函数变量赋值，再把这个函数覆写，覆写时通过call调用原函数即可实现函数继承。

需要注意的是，由于\_drill\_SCA\_pluginCommand是在最外层的全局变量，你需要确保函数名必须唯一。

**另外，写call函数时，非常容易漏掉后面的参数，写成” \_drill\_SCA\_pluginCommand.call(this)”，这样不但破坏了原函数的结构，而且还不会报语法错误。该bug只能在运行测试时发现问题，代价很大。**

你可以在一个插件里反复写这种结构，继承同一个函数。

因为所有底层与插件之间都是平铺的，多次继承不会影响。

而且部分大功能，需要相互区分，揉在一起会造成很多读代码的困难。

|  |
| --- |
| //=============================================================================  // \* 插件 – 大功能A  //=============================================================================  var \_drill\_SCA\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCA\_pluginCommand.call(this, command, args);  //...  };  //=============================================================================  // \* 插件 – 大功能B  //=============================================================================  var \_drill\_SCA\_pluginCommand2 = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCA\_pluginCommand2.call(this, command, args);  //...  }; |

### 地图界面代码 - 什么是贴图

**Sprite**：通常称呼为贴图，是pixi库中提供的sprite类。其它可能会有别名，比如精灵、图片等。

**Bitmap**：通常都被称呼为资源，是rpg\_core中的私有封装类，用于隔离pixi的sprite中的texture材质渲染结构。

简单来说底层是这样的关系：

原Pixi库： Sprite -> texture -> render渲染器 => 图像

Rmmv：Sprite -> Bitmap -> Manager场景界面 => 图像

*Rmmv经过了一层封装，使得你完全不需要了解pixi的官方渲染结构，直接用bitmap提供好的结构就可以了。不需要考虑****渲染器结构****。*

于是，就出现了下面的代码：

|  |
| --- |
| //==============================  // \* 帧刷新 - 创建一个贴图  //==============================  Scene\_Map.prototype.drill\_SCA\_updateCreateSprite = function() {  if( $gameTemp.\_drill\_SCA\_switch == true ){  $gameTemp.\_drill\_SCA\_switch = false;    var temp\_sprite = new Sprite();  temp\_sprite.x = 100;  temp\_sprite.y = 100;  temp\_sprite.zIndex = 10;  temp\_sprite.bitmap = ImageManager.load\_CourseA("课程A图片");    this.\_drill\_SenceTopArea.addChild( temp\_sprite ); //最顶层（层级后面再解释）    this.\_drill\_SCA\_sprite = temp\_sprite;    //this.addChild( temp\_sprite );  //你可以直接addchild，但是这种操作，会使得图片层级的先后顺序紊乱  //因为rmmv不会识别zIndex，所以层级的排序后期要自己写。  }  } |

Sprite具体函数和成员可以进入pixi或者rpg\_core底层查看，这里我总结了下列常用的属性：

|  |  |  |
| --- | --- | --- |
| 属性名 | 含义 | 功能 |
| .x  .y | 坐标xy | 贴图在父类中的相对坐标位置。  如果有多级父类，那么相对位置会被无限叠加嵌套。所有sprite基本都是根据自身相对位置来定的，很少有绝对位置的说法。 |
| .bitmap | 资源对象 | 通过ImageManager获得的资源对象。  可以反复赋值不出问题且不影响性能，因为赋值的是对象指针。  Gif的实现原理基于此，存储一个bitmap容器，然后每帧按规律变bitmap。 |
| .anchor.x  .anchor.y | 中心锚点xy | 锚点决定图片的起始位置。  (0.0,0.0)表示贴图左上角，(0.5,0.5)表示贴图中心。  锚点影响 缩放/旋转 效果比较多。事件贴图的锚点都为(0.5,1.0)正下方。 |
| .blendMode | 混合模式 | 只有0正常、1叠加、2乘积、3滤色 四种混合模式可用。  0和2是常用颜色模式，内部硬性赋值比较好，容易让其它人理解。  比如纯色滤镜就是通过混合模式2实现的，蓝色会过滤出蓝色的光，过滤方式与红绿蓝底色有关。若直接摆出混合模式0123会让其它人较难理解。 |
| .opacity | 透明度 | Rmmv的封装属性，范围为 0 - 255。  原Pixi库中为alpha，范围为0 - 1.0。  如果opacity接受了一个undefined或者NaN值，则rmmv会报“clamp”错误，这里留意一下，因为这个错误比较容易出现。 |
| .visible | 显示 | 如果为false，则图片不渲染。  如果为true，则图片渲染。  从图形层面来看，不渲染比渲染要节省很多性能，但是千万不要滥用.visible=false，因为用多了你会经常因为图片不显示而半天找不到问题。 |
| .rotation | 旋转 | 贴图围绕锚点旋转。  单位为弧度，2 \* Math.PI为一周（2\* 3.14）。  正数顺时针，负数逆时针。（以0朝右为基准，则1.57朝下，-1.57朝上）  如果你让一个贴图单纯自旋转，比如魔法圈，是没问题的。  旋转令人疼的地方，是与事件的东南西北朝向对应过渡的问题，极坐标与直角象限比较难对应。 |
| .scale.x  .scale.y | 缩放 | 贴图围绕锚点缩放。  1.0为原比例缩放，2.0为两倍大小，0.5为一半大小。  锚点非常容易影响缩放的视觉效果，注意控制。 |
| .skew.x  .skew.y | 斜切 | 0.0表示正常，x 1.0时，形成向右上倾斜平行四边形，-1.0时左上倾斜。  功能不常用。 |

其中.zIndex图片层级顺序是自写的额外属性，底层中并不存在这个属性。

你可以去Drill\_LayerGround多层地图背景插件中找到zIndex的定义，参考结构。

相关插件中的 地图层级/战斗层级/菜单层级 是共享的。如果你要划分层级，建议跟随drill划分好的层级。

|  |  |  |
| --- | --- | --- |
| 函数名 | 含义 | 功能 |
| .addChild( sprite )  .removeChild( sprite ) | sprite嵌套 | 常用函数，子类addChild加入父类后，将会跟随父类一起变换。  建议养成一个习惯，每次创建贴图的前，先创建一个父类层，比如\_xxx\_layer某某层。插件的所有贴图统一放入该层级，这样方便和其它插件作层级顺序控制。 |
| .setFrame( x, y, width, height ) | 切割框架 | 设置框架后，将会把bitmap资源，切割成一小块。  可以放入update中反复调用，不影响性能。  一般的图片集合，比如图标/符号都会使用setFrame。  rmmv原动画也是setFrame实现的，不过不建议将多个大图片组合在一起然后还要再setFrame，这样不但复杂而且设置麻烦。 |
| .setBlendColor( [r,g,b,a] ) | 混合颜色 | 能够对bitmap资源进行整体填色。参数为一个数组。  不能放入update中反复调用，会严重影响性能。  填充滤镜就是基于该原理。不过，该功能不常用。 |

**以上为Sprite的基本内容，80%的贴图插件都是围绕这些功能展开，你可以在插件中稍微修改属性、调用指定函数看看效果。**

Sprite还有一些其他的特殊属性与函数，比如.filter滤镜、.mask遮罩等，这些需要你先用熟悉了上述所有内容后，再深入。你如果感兴趣，可以去pixi或者rpg\_core底层用ctrl+f多看看。

另外，Window窗口也是基于上述的贴图结构进行封装的类，具体内容后面章节将会提及。

### 加入插件 - 插件顺序关系

大家都知道，插件之间是有顺序关系的，核心插件尽量往前放，扩展插件尽量往后放。

在插件编辑器中加入插件：



为什么插件的顺序会造成那么大的影响呢？因为继承/覆写顺序不一样，会造成不同的效果。

前面 插件指令代码 章节中 提到了继承和覆写的关系，即使是继承，也会存在顺序问题，因为先继承的对象会被后继承的对象包裹。**其中，功能A是可以操作功能B中的属性的，因为A在后面，包裹了B。**

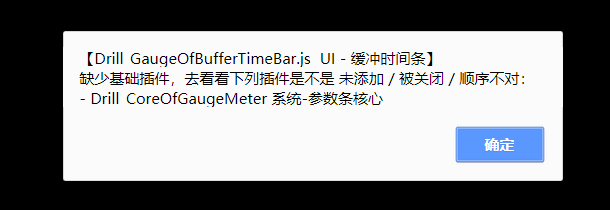
|  |
| --- |
| //=============================================================================  // \* 插件 – 大功能B  //=============================================================================  var \_drill\_SCA\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCA\_pluginCommand.call(this, command, args);  //...  };  //=============================================================================  // \* 插件 – 大功能A  //=============================================================================  var \_drill\_SCA\_pluginCommand2 = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCA\_pluginCommand2.call(this, command, args);  //...  }; |

如果A需要 操作B的属性或函数，那么就可以看做 A -> B 的需求关系。如果没有B，那么A的功能就不能实现。这就出现了 **基于/作用于/扩展于** 的关系。

**另外，call的位置也会影响继承的相互关系，如果你的代码写在call函数的前面，也会产生不同的效果**。

**基于关系**：基于是硬性关系。

要求必须装有核心，才能运作插件。你可以使用下面的方式，给插件全部代码加一个大的if外壳。下图为UI-缓冲时间条 插件的外壳。



|  |
| --- |
| //=============================================================================  // \* >>>>基于插件检测>>>>  //=============================================================================  if( Imported.Drill\_CoreOfGaugeMeter ){    //...  // 插件的全部代码  //...    //=============================================================================  // \* <<<<基于插件检测<<<<  //=============================================================================  }else{  Imported.Drill\_GaugeOfBufferTimeBar = false;  alert(  "【Drill\_GaugeOfBufferTimeBar.js UI - 缓冲时间条】\n缺少基础插件，去看看下列插件是不是 未添加 / 被关闭 / 顺序不对："+  "\n- Drill\_CoreOfGaugeMeter 系统-参数条核心"  );  } |

**作用于/扩展于 关系**：这里的关系比较弱，通常写在插件代码中，加个判断插件，然后多一步操作。

如果没有作用于的插件，则功能操作不执行即可。在插件中搜索Imported即可找到。

|  |
| --- |
| //==============================  // \* 普通跳跃 - 声音  //==============================  SoundManager.drill\_EJu\_playSE = function(fileName,character){  var se = {};  se.name = fileName;  se.pitch = 100;  se.volume = 100;  if( Imported.Drill\_EventSound && AudioManager.drill\_ESo\_playCharacterSe ){ //适应声音距离化  AudioManager.drill\_ESo\_playCharacterSe(se,character);  }else{  AudioManager.playSe(se);  }  }; |

另外，如果你的插件对外部插件进行了引用控制，你需要在注释中进行告知，以防玩家因为插件顺序出现问题，而半天找不到原因。

|  |
| --- |
| \*  \* -----------------------------------------------------------------------------  \* ----插件扩展  \* 该插件可以单独使用，也可以作用于其他插件。  \* 可作用于：  \* - Drill\_Jump 互动 - 跳跃能力  \* 目标插件基于此插件，可以使得控制台控制玩家的跳跃。  \* |

有群友提出过让插件最后载入的写法，比如(function () { })()。

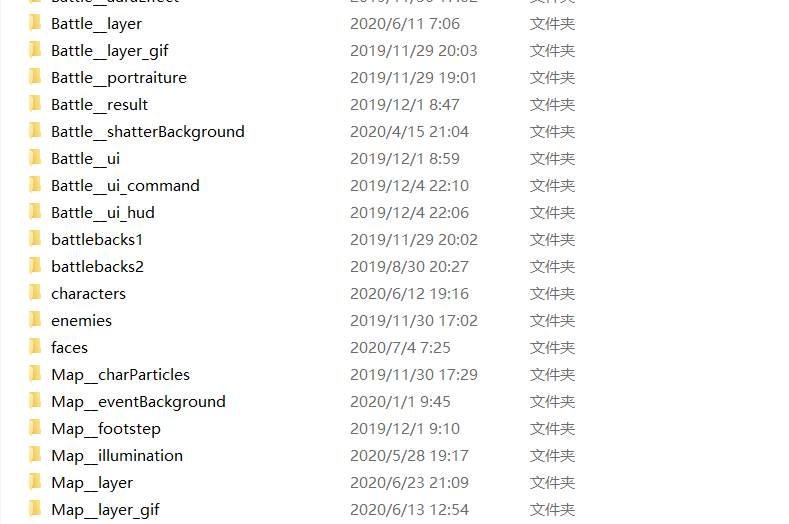
**这里，不建议使用(function () { })()，虽然这个方法会使你的插件 顺序 变在最后。但是你写，别人也会写。如果其他插件都加这个函数，那么具有相同写法的函数又会被排一次顺序，反而插件顺序变得更加混乱了**。

### 文件夹放置 - 多资源管理技巧

由于rmmv编辑器的限制，所有文件夹只能有一级的深度，不过这样也很方便，所有资源全部被平铺开来，文件夹的命名就会变得非常重要了。

尽量保持一个插件控制一个文件夹，这样就不会造成资源分布错综复杂的问题。如果你去除了指定的插件，要去除资源，就把该插件的文件夹删掉就可以了。

插件示例中的资源文件夹如下图。



建议大类分为Battle、Map、Menu，分别对应 战斗界面、地图界面、菜单界面 。注意图中的下划线有两个。

如果插件的资源在多个界面之间交织，则用Special前缀。

这样既可以区分rmmv原装文件夹，还可以对资源的作用域进行划分。

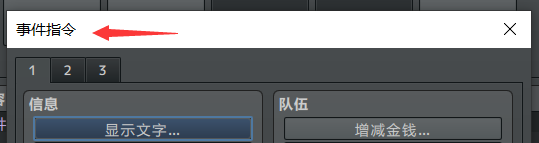
（另外提及一下，如果你制作的是一套新的战斗系统，完全脱离了rmmv的 战斗/地图/菜单 结构，比如STG战斗界面，这种界面和rmmv原分类界面完全不一样，你可以另起一个大类。）

不过，新写一个战斗系统或者独立小游戏系统，需要很强的代码功力，这可不仅仅是建立一个Scene界面那么简单，而是一个新的游戏生态系统。萌新不要尝试。

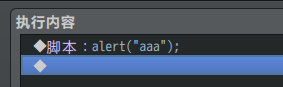
### 创建一个事件 - 事件指令介绍

**事件执行内容中的所有指令，都称作事件指令。**

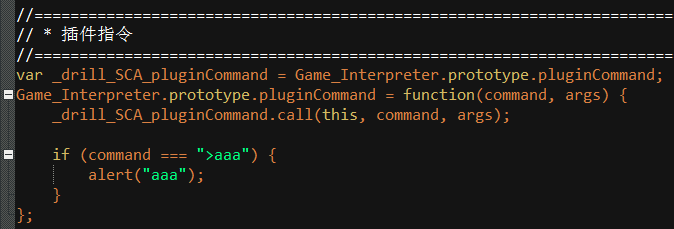
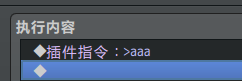
插件指令、脚本、事件注释、分支条件等都属于事件指令。注意区分。



Rmmv事件指令中，脚本与插件指令功能相似，下图是脚本直接写法：



插件指令相当于多进行了一次字符串指令识别：



**写插件时，最好少用脚本，多用插件指令，这样可以有效避免设计者直接改脚本出现问题**。

另外提及一下，rmmv的事件指令是通过事件的拦截器控制的，称为Game\_Interpreter，课程插件中覆写的插件指令函数在rpg\_objects.js中第10518行左右。（用ctrl+f查找）

|  |
| --- |
| // Plugin Command  Game\_Interpreter.prototype.command356 = function() {  var args = this.\_params[0].split(" ");  var command = args.shift();  this.pluginCommand(command, args);  return true;  };  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  // to be overridden by plugins  }; |

command356函数比较特殊，你在游戏数据中添加每一个事件指令都有自己的序号，而356就是这个插件指令的序号，rmmv通过指令栈来调用 command+数字 的函数。在8940行。

|  |
| --- |
| var methodName = 'command' + command.code; |

这就是事件指令的运行机制，你可以翻翻源码了解一下。

### 功能测试 - F8与报错

游戏通过node.js进行运行，rmmv默认配置了f8，按f8可以进入开发者界面。

而如果使用火狐浏览器（只剩火狐了）运行，按f12可进入开发者界面。



**插件如果出现常见的拼写错误，游戏会直接关闭插件并运行。所以在测试插件时，最好先按f8先确认插件中是否有语法错误 或 JSON.parse()数据 错误**。防止游戏测试运行了半天，一直找不准插件什么时候生效。

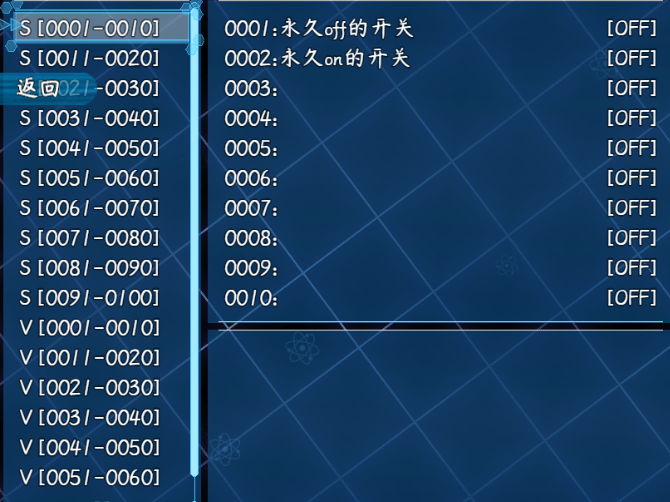
如果你使用了yep的核心引擎，遇到错误时，会出现下面的黄色提醒界面，这个部分的信息都是插件辅助显示的，实际上都在f8输出中都可以看见：



另外提及一下，rmmv按F2可以调出fps以查看帧率性能。



按F9可以进入debug界面查看控制变量与开关状态。（浏览器中按F9没有效果）



### 当前插件的局限性

课程插件只是贴了一个贴图，虽然功能实现了，但是存在很多后续问题，比如：

1.切换菜单 或 离开地图，贴图就没了。

2.不停地执行插件指令，你会发现大量贴图不断地积压，严重影响性能。

这部分与rmmv的底层原理有关，后面章节将会详细介绍。

## 课程小结

下面来总结一下课程的全部内容，内容密度有点大，如果你对下面的知识点仍然感到不好把握，可以回去看看，或者自己试验试验。

1）**基本意识**：所有底层函数和插件都是相通的，不能乱起名，并且要用ES5的兼容写法。

2）**命名**：每个函数、变量、插件都应该有专有唯一且不冲突名字，你可以借鉴drill插件的命名规则。

3）**上手**：通过写好的模板，你可以尝试完成 在地图界面中放一个贴图 的功能。

4）**参考**：”基本函数查询表.docx”中有一些稳定的ES5的写法，你可以根据其中的规则，确保插件稳定运行。

5）**查看底层**：进入rpg\_xxx文件后，通过ctrl+f的手动查找想看的函数。

6）**黑盒展开**：通过阅读底层代码，将有疑问的函数黑盒慢慢展开，了解这个函数的特性，熟能生巧。

7）**继承与覆写**：注意写call函数时，不要漏掉后面的参数，是大坑。

8）**贴图**：贴图是sprite，单独的sprite不能构成图像，只相当于一个层级。它可以通过成员bitmap资源对象来构建图像。同时sprite可以通过addChild无限嵌套新的sprite，并且具有x、y、opacity、visible……等大量可修改属性。

9）**插件顺序**：插件之间的顺序关系，与继承与覆写的先后顺序有关系。根据顺序关系，插件分为 基于/作用于/扩展于 的关系。

10）**资源文件夹**：文件夹只有一级深度，可根据插件的作用范围进行命名，并且一个插件只对应一个文件夹。

11）**事件指令**：事件示例是事件执行内容的一个统称，注意不要把 插件指令和事件指令 记混了。

12）**报错检查**：按f8可以进入开发者界面，查看插件的语法错误，或者执行时错误。

13）**课程插件局限性**：该课程插件产生的贴图容易被销毁，还需更多知识来完善。