## 简报

### 任务

|  |
| --- |
| 任务目标：制作一个插件，能够实现插件指令操作一些事件指令。（代码不需要手敲，复制粘贴即可。）  任务帮助：当前目录下，提供了两个插件模板：Drill\_SimpleCourseB1，B2，B3，分别对应课程的上，中，下。 |

### 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。

这节课，我们开始说说继承与覆盖的关系。

2.所有脚本，都是基于ES5的js格式。

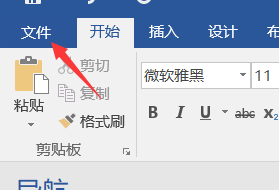
对这句话留个印象即可，后面课程会详细介绍。

### 去掉word红线

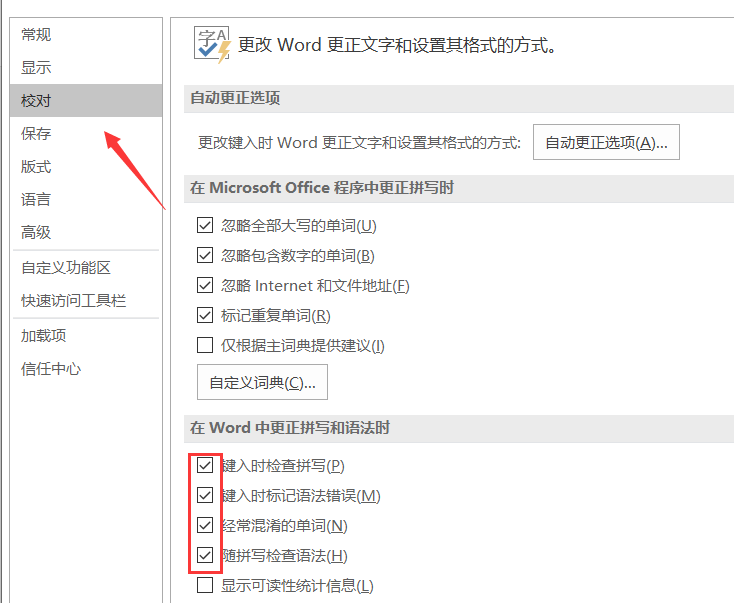
Word的红线会非常影响代码查看，这里必须去掉。



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（上）

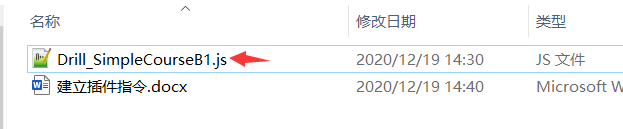
### 开始写插件

现在开始第二节课。这一节课，是上一课的下一课，也是下一课的上一课。

我们先按照下列步骤完整走一遍。

#### 打开B1脚本

比如使用notepad++打开。



#### 脚本复制到：插件指令部分

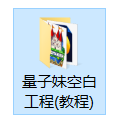
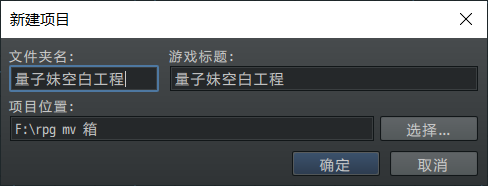
将下列代码贴到插件指令部分区域。

|  |
| --- |
| Game\_Interpreter.prototype.pluginCommand = function(command, args) {  if( command === ">插件指令A" ){  alert("执行了插件指令A。");  }  };  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  if( command === ">插件指令B" ){  alert("执行了插件指令B。");  }  }; |

#### 新建测试工程

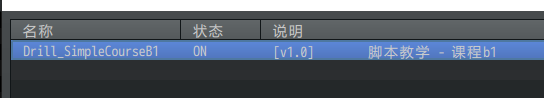
这里也不要直接在示例或你的游戏工程里面加该插件，最好新建一个工程。

（这里就不用rmmv默认的Project1了，感觉每次新建都非常臃肿，空白工程在工具箱，如果要可以去取。）



#### 加入插件

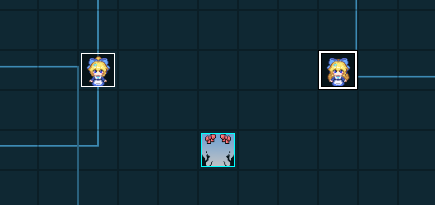
把Drill\_SimpleCourseB1插件加入到工程中，其他关闭。

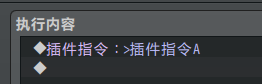


#### 放置事件

放置两个小爱丽丝，分别执行 >插件指令A 和 >插件指令B。

左边单马尾的执行A，右边长发的执行B。





#### 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。



然后分别去触发一下两个小爱丽丝。



你可以发现只有 插件指令B 能够有效。

ヽ(\*。>Д<)o゜在你完成上述流程之后，接下来我们开始剖析操作过程中的细节吧。

### 详解 - 底层函数查看（基础介绍）

经过上一节课和刚才的操作，我们已经认识了两个在最外层的函数：

|  |
| --- |
| DataManager.maxSavefiles = function() { };  Game\_Interpreter.prototype.pluginCommand = function(command, args) { } |

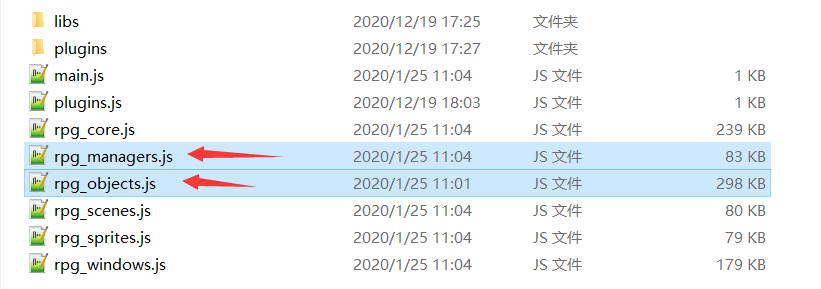
你可能会疑问，为什么一个函数有prototype，另一个没有。

因为DataManager是做为全局单例类来使用，与PluginManager.parameters(……);的单例类PluginManager一样。而Game\_Interpreter类需要在地图中实时new，每个事件都会绑定该类。

如果你不熟悉什么是prototype，可以去 萌新劝退提示 里面的课程链接去补习。

另外，这两个函数并不是凭空出现的，在rmmv内核脚本中，能够找到。

DataManager类 在rpg\_managers.js文件中，Game\_Interpreter在rpg\_objects.js就中。

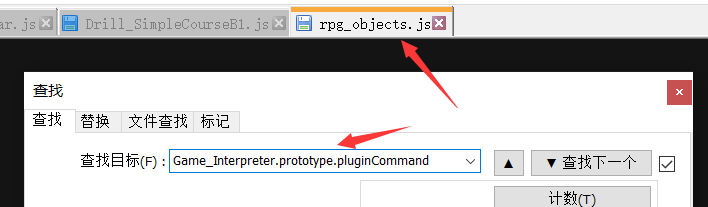


下面我们去rpg\_objects.js中找到Game\_Interpreter类吧。

进入rpg\_objects.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。

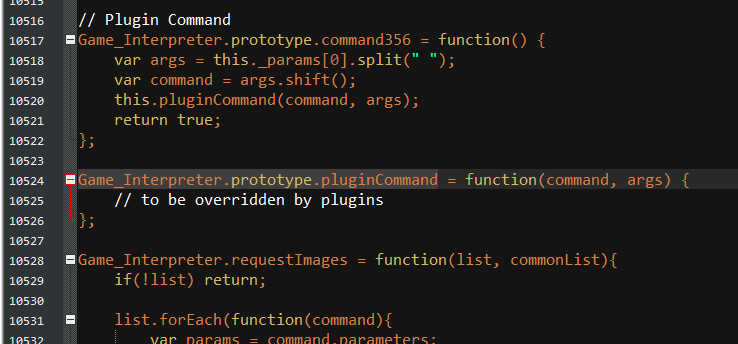


现在大多数编辑器都能够自动识别并且划分 类与函数，自动列出成员。你可以快速索引到想找的类。

但是，rmmv的脚本代码是非常原生的js，多数编辑器都不待见，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。

可以找到，插件指令定义的位置。



所有rmmv的运行脚本都在这六个rpg\_xxx.js主要文件中，没事养成习惯，多看看里面的函数。

这里暂时点到为止，接下来我们会进一步说明底层函数分析。

### 详解 - 继承与覆写

经过之前的流程操作，你会发现，插件指令B能够执行，但是插件指令A不能执行。

这是因为，直接赋值Game\_Interpreter.prototype.pluginCommand这种写法，会将上一个函数覆盖掉。

这也是基本意识中提及的：“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。”

|  |
| --- |
| Game\_Interpreter.prototype.pluginCommand = function(command, args) {  if( command === ">插件指令A" ){  alert("执行了插件指令A。");  }  };  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  if( command === ">插件指令B" ){  alert("执行了插件指令B。");  }  }; |

那么，怎样才可以让插件指令A和插件指令B都有效呢？

机智的朋友立刻想到了这种写法：

|  |
| --- |
| Game\_Interpreter.prototype.pluginCommand = function(command, args) {  if( command === ">插件指令A" ){  alert("执行了插件指令A。");  }  if( command === ">插件指令B" ){  alert("执行了插件指令B。");  }  }; |

这种方法的确解决了当前插件的覆盖问题，但是治标不治本哦。

这个时候，就需要使用继承函数的写法：

|  |
| --- |
| var \_drill\_SCB\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCB\_pluginCommand.call(this, command, args);    if( command === ">插件指令A" ){  alert("执行了插件指令A。");  }  }; |

在drill插件中，可以在大多数插件中都见到这种类似的结构：

“var \_drill\_SCB\_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。

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

（顺带一提，覆写、覆盖、重写，三个意思都是一样的。）

|  |
| --- |
| **误区**：  写call函数时，非常容易漏掉后面的参数，写成” \_drill\_SCB\_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，写函数时一定要留意参数哦！ |

去掉 插件指令部分 的代码，然后将下列函数贴到插件中，

这样，就能够实现指令AB都能够触发啦。

|  |
| --- |
| var \_drill\_SCB\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCB\_pluginCommand.call(this, command, args);    if( command === ">插件指令A" ){  alert("执行了插件指令A。");  }  };  var \_drill\_SCB\_pluginCommand2 = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCB\_pluginCommand2.call(this, command, args);    if( command === ">插件指令B" ){  alert("执行了插件指令B。");  }  }; |

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。

后期写比较复杂的插件时，各功能直接需要相互区分独立，可以用多次继承的方式将函数的功能分门别类。

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

### 详解 - debug方式

Js常用的debug方式有下面两种：

|  |
| --- |
| alert("执行了插件指令A。");  console.log("执行了插件指令A。"); |

常用的只有这两种，没法打断点、监听实时参数值。（非常原始的debug方式）

因为后期经常会要监听update中的函数，由于执行速度太快，经常是一闪而过，所以只能自己手写alert。

另外console.log()函数会将信息显示到开发者界面。

**1）进入开发者界面**

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。

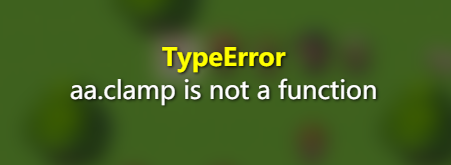
而如果使用火狐浏览器（只剩火狐了）运行，按f12可进入开发者界面。



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，才发现插件根本没开。

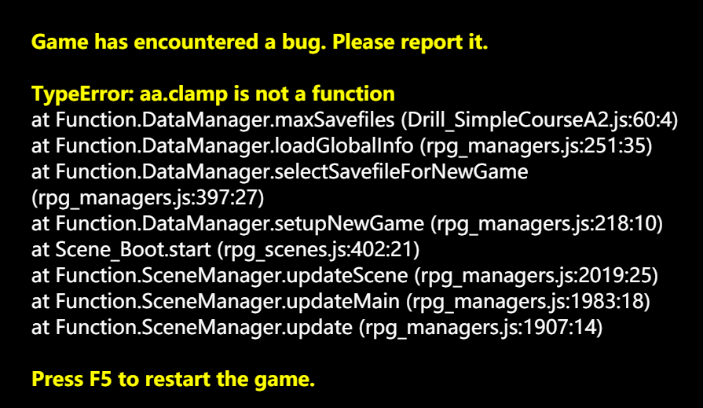
**2）yep插件的报错优化**

使用默认的工程进行测试时，系统如果遇到错误，会显示下面的情况：



而如果你使用了yep的核心引擎，遇到错误时，会将简单的报错优化成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。

不过这个优化可以使得玩家在出现错误时，能直接从截图中就能找到问题，而不需要再多让玩家按f8才能定位到问题了。



**3）查看fps**

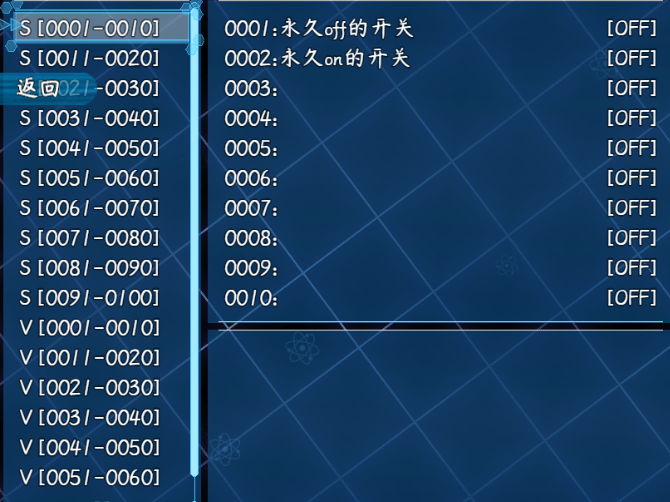
另外提及一下，rmmv按F2可以调出fps以查看帧率性能。

关于性能的详细介绍，可以去看看”0.性能测试报告 > 关于插件性能.docx”。



**4）查看变量与开关**

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）



## 开始课程（中）

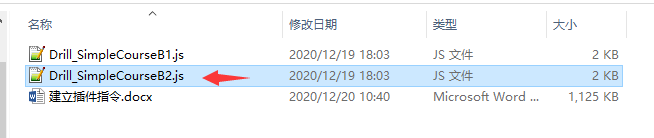
### 开始写插件

经过了上半部分的学习，我们了解了函数、底层函数、继承与覆盖的关系，以及debug的各类操作。

接下来，我们按照新的步骤完整走一遍。

#### 打开B2脚本

打开脚本。



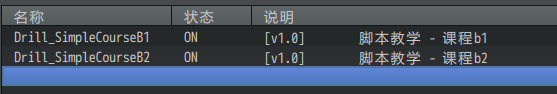
#### 脚本复制到：插件指令部分

将下列脚本复制或者手敲到B2插件的指定位置。

|  |
| --- |
| var \_drill\_SCB\_pluginCommand3 = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCB\_pluginCommand3.call(this, command, args);  if( command === ">插件指令C" ){ // >插件指令C : 这是一串字符串  if( args.length == 2 ){  var temp1 = String(args[1]);  alert("获取到字符串：" + temp1);  }  }  }; |

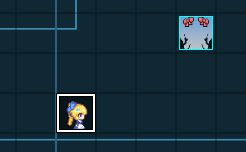
#### 加入插件

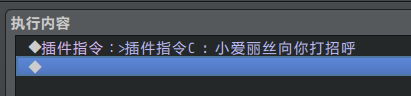
把Drill\_SimpleCourseB2插件加入到工程中，保留B1为开启状态。



#### 放置事件

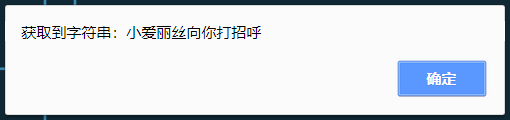
再添加一个小爱丽丝，执行插件指令C。





#### 功能测试

测试游戏，去触发一下小爱丽丝，顺带再去触发之前的小爱丽丝。



你会发现小爱丽丝都能够触发各自相应的插件指令。

ヽ(\*。>Д<)o゜在你完成上述流程之后，接下来我们开始剖析操作过程中的细节吧。

### 详解 - 命名规则

在复制代码时，你会发现函数继承时使用了：\_drill\_SCB\_pluginCommand3

这是因为之前写了两个函数：\_drill\_SCB\_pluginCommand 和 \_drill\_SCB\_pluginCommand2。

因为底层相通，为了防止覆盖，这里加了3。

|  |
| --- |
| **提示**：  由于课程中，B1插件和B2插件的简称都为 SCB ，所以会出现覆盖的情况。  这时候，你可以将简称改为：SCB1 和 SCB2。  （通常写插件，很少会遇到简写相同的情况，这里因为是教学中的同一个插件，所以需要修改区分一下。）  替换时，因为SCB是大写的英文标识，在代码中几乎不会出现两个英文大写的变量名与函数名，所以ctrl+f打开一键全部替换就可以啦。 |

后面的课程，都将变为简写SCC1，SCC2，将不会出现覆盖情况。

#### 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）尽可能继承函数，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。

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

### 详解 - 底层函数查看II（插件指令底层）

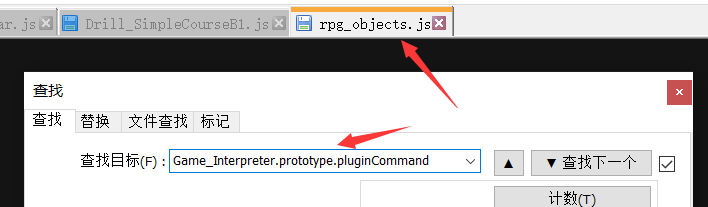
这部分与上部分相比较，多了args的使用。

插件指令字符串是如何被拆成 command 和 args 呢？这两个参数应该如何使用？

|  |
| --- |
| if( command === ">插件指令C" ){ // >插件指令C : 这是一串字符串  if( args.length == 2 ){  var temp1 = String(args[1]);  alert("获取到字符串：" + temp1);  }  } |

下面我们不妨去底层函数再去看看。（需要养成随时ctrl+f查找的习惯哦！）

去rpg\_objects.js中找到Game\_Interpreter类。



找到函数后，我们会发现这个函数的上面，就有调用执行它的函数。

|  |
| --- |
| // 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  }; |

目前，虽然我们不清楚this.\_params[0]是从哪里来的，但是我们看见了split函数。

以此可以推理出，args是一个根据 空格 分隔成的字符串数组。

这也就解释了为什么插件指令是通过空格分隔的。

经过分析，我们可以尝试理解

”>插件指令C : 这是一串字符串”

被分成了：

|  |
| --- |
| command：”>插件指令C”  args[0]： ”: ” （冒号）  args[1]： ”这是一串字符串”  args长度为2。 |

如果按照drill的格式来，比如一个长插件指令：

“>位置与位移 : 本事件 : 移动到 : 位置[3,3]”

就可以被分成：

|  |
| --- |
| command：”>位置与位移”  args[0]： ”: ” （冒号）  args[1]： ”本事件”  args[2]： ”: ” （冒号）  args[3]： ”移动到”  args[4]： ”: ” （冒号）  args[5]： ”位置[3,3]”  args长度为6。 |

获取时要用1,3,5,7,9这种方式获取哦，这也就是为什么你经常会在drill插件中看见下面的情况：

|  |
| --- |
| /\*-----------------移动到相对位置------------------\*/  if(args.length == 6){ //>位置与位移 : 玩家 : 移动到 : 位置[3,3]  var unit = String(args[1]);  var type = String(args[3]);  var pos = String(args[5]);  //...  } |

## 开始课程（下）

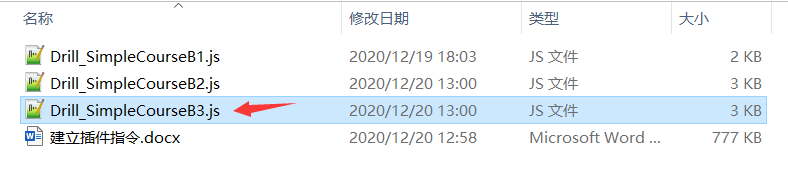
### 开始写插件

经过了上半部分的学习，我们了解了函数、底层函数、继承与覆盖的关系，以及debug的各类操作。

接下来，我们按照新的步骤完整走一遍。

#### 打开B3脚本

打开脚本。



#### 脚本复制到：插件指令部分

将下列脚本复制或者手敲到B3插件的指定位置。

代码行数比较多，但是基于的原理非常简单，不要怕。（如果你到现在还缺少js基础，赶紧去补习哦！）

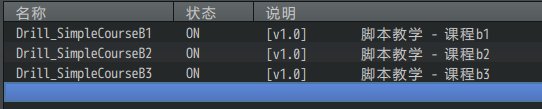
另外，有部分群友对于作者这种注释方式感到过于复杂，你可以不用按照作者的格式来，仅提供参考。

不过，后期插件数量上来了，还是要考虑一下插件指令的多样性。

|  |
| --- |
| var \_drill\_SCB3\_pluginCommand = Game\_Interpreter.prototype.pluginCommand;  Game\_Interpreter.prototype.pluginCommand = function(command, args) {  \_drill\_SCB3\_pluginCommand.call(this, command, args);    if( command === ">变量控制" ){ // >变量控制 : 变量[21] : 增加 : 10  if( args.length == 6 ){  var temp1 = String(args[1]); //变量id  var type = String(args[3]); //操作类型  var temp2 = String(args[5]); //操作的值    temp1 = temp1.replace("变量[","");  temp1 = temp1.replace("]","");  temp1 = Number(temp1);    temp2 = Number(temp2);    if( type == "增加" ){  var v\_value = $gameVariables.value( temp1 );  $gameVariables.setValue( temp1, v\_value+temp2 );  }  if( type == "减少" ){  var v\_value = $gameVariables.value( temp1 );  $gameVariables.setValue( temp1, v\_value-temp2 );  }  alert("经过插件指令执行后，变量值"+ temp1 +"结果为：" + $gameVariables.value( temp1 ));  }  }  }; |

#### 加入插件

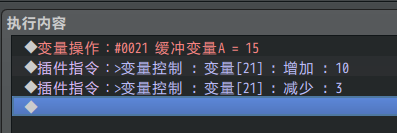
把Drill\_SimpleCourseB3插件加入到工程中，保留B1和B2为开启状态。



#### 放置事件

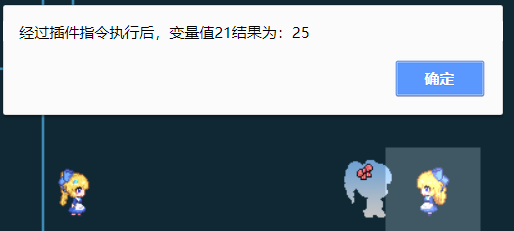
建立一个小爱丽丝，执行变量赋值，然后执行下图的两个插件指令。

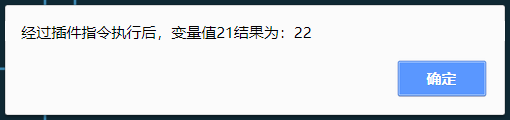




#### 功能测试

测试游戏，去触发放置的小爱丽丝。





可以看到先是15+10的结果，然后是 15+10-3 的结果。

ヽ(\*。>Д<)o゜在你完成上述流程之后，接下来我们开始剖析操作过程中的细节吧。

### 详解 - 变量的脚本II

我们上节课学习了变量的取值和赋值脚本：

|  |
| --- |
| var aa = $gameSwitches.value(21); //获取21号开关值（true/false）  var bb = $gameVariables.value(22); //获取22号变量值（整数）    $gameSwitches.setValue(21,false); //设置21号开关值为false  $gameVariables.setValue(22,100); //设置22号变量值为100 |

Js语言中，直接使用 20+20，即可完成两个数的相加。

变量也同理，$gameVariables.value(21) + $gameVariables.value(22)

另外提及一下，20+20在编程语言中称作 表达式（去百度了解下），并不是什么数据都能随便相加的，虽然 字符串+数字=字符串 的设定在js语言中是有效的，但是其它语言可会被视作为语法错误哦！

那么如何实现控制游戏中的变量修改呢？

由于变量的值只能通过 setValue进行赋值，那么就出现了下面的用法：

加数 + 加数 = 得数

|  |
| --- |
| var v\_value = $gameVariables.value( 加数 );  $gameVariables.setValue( temp1, v\_value+加数 ); |

被减数 – 减数 = 得数

|  |
| --- |
| var v\_value = $gameVariables.value( 减数 );  $gameVariables.setValue( temp1, v\_value-减数 ); |

通过这种方式，就可以实现对变量值的加减操作。

虽然 加数 + 负数 的得数值变少了，但这仍然表示两者是相加关系，而不是相减哦！

结合课程中间部分获取插件指令参数值的方法，我们就可以写出：

|  |
| --- |
| if( command === ">变量控制" ){ // >变量控制 : 变量[21] : 增加 : 10  if( args.length == 6 ){  var temp1 = String(args[1]); //变量id  var type = String(args[3]); //操作类型  var temp2 = String(args[5]); //操作的值  temp1 = temp1.replace("变量[","");  temp1 = temp1.replace("]","");  temp1 = Number(temp1);  temp2 = Number(temp2);  if( type == "增加" ){  var v\_value = $gameVariables.value( temp1 );  $gameVariables.setValue( temp1, v\_value+temp2 );  }  }  } |

除此之外，还可以多添加 乘以、除以、取余、赋值 等操作。

|  |
| --- |
| if( type == "乘以" ){  var v\_value = $gameVariables.value( temp1 );  $gameVariables.setValue( temp1, v\_value\*temp2 );  }  if( type == "除以" ){  var v\_value = $gameVariables.value( temp1 );  $gameVariables.setValue( temp1, v\_value/temp2 );  }  //... |

## 课程小结

下面来总结一下课程的全部内容，如果你对下面的知识点仍然感到不好把握，可以回去多看看，自己上手试验试验，熟悉基本知识。

1）**基本意识**：所有底层函数和插件都是相通的，不能乱起名，需要根据一套命名方式来防止覆盖。

2）**底层函数查看**：打开notepad++，通过ctrl+f查找rpg\_object中的插件指令功能，手动查找。游戏主要函数在六个rpg\_xxx文件中，以后遇到想了解函数，直接ctrl+f去找。

3）**继承与覆写**：直接赋值函数的方法，会将之前的函数给覆盖掉。需要采用继承的方式，先让一个全局变量捕获到函数，然后在新函数中进行call即可实现函数继承。

4）**debug方式**：通过alert和console.log的方式进行参数值输出。按f8打开开发者页面，必要时最好加入yep的核心引擎，按f2可以查看fps，按f9可以查看变量和开关情况。

5）**命名规则**：临时全局变量要有”g\_”前缀；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；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；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。

6）**底层函数查看II**：打开notepad++，再次使用ctrl+f查找rpg\_object中的插件指令功能。发现args是根据插件指令字符串split之后生成的字符串数组。

7）**变量的脚本II**：根据插件指令和变量的知识，可以编写出 通过插件指令来控制变量值 的功能。并且能够支持加减乘除、取余、赋值等操作。

|  |
| --- |
| 课后作业：写一个插件，能够通过插件指令，来控制存档的数量。 |

如果你能完成这个作业，就说明你入门啦！

|  |
| --- |
| **挖的坑**：  基本意识2，后面课程会介绍。 *->课程C* |