vJass 完整版教程

版本： JassHelper 0.A.0.0

作者	站点
Vexorian（原作者）	*http://www.wc3c.net/*
JoyWoon（翻译和修订）	*https://www.gamesdeity.com*

前言

作者

尽管最终 WEHelper 将 World Editor 的 Jass 编译器替换为 PJass，但仍然会有一些很多的问题待解决，这就是开始这个项目的原因。

译者注：WEHelper 是非常早期的 WE 插件工具，目前已经基本过时了。

在项目的过程中，后来我感觉更进了一步，**并产生了将 Jass 扩展到面向对象编程思路的想法。**

JassHelper 是 vJass 语言的编译器，支持 vJass 的很多功能，其中包括结构，库，文本宏等等……

虽然这**不是真正的面对对象编程模式**，但我依旧希望语法能够足够的强大。

vJass 没有真正的**继承性**，这就是我不将其称之为**对象类**而是**结构**的原因。不过，vJass 仍然有着**允许多态**的**接口**功能，并且，由于还可以声明新结构的数据类型，因此，您可以写出一些具有伪继承性的代码。伪继承性在以下的教程中，会告诉你，能通过多种方式去实现。

我认为最终应该停止设计 vJass 的语言。**因此在 1.0.0 版本之后，不会再更新任何新的内容，如果您有任何需求，请紧记，在 1.0.0 版本之后进行语法更改对 vJass 是很不健康的。**

Z.0 版本在 JassHelper 中引入了 **Zinc 语言**，这只是 vJass 的简单替代，在某些方面也会更加严谨。

译者

vJass 是 Jass 的扩展，目前几乎所有的 WEer 都使用 JassHelper，因此，vJass 的通用性非常强。vJass 的基本语法和 Jass 没有任何区别，只是在此基础之上增加了更多功能性的语法，以支持面向对象式的编程方式。

我着重要求读者一定要好好了解 vJass 的结构功能。该功能基本上将魔兽争霸 3 地图的开发引上了一个新的台阶。

目录

一. 自由声明

vJass 可以自由的声明全局变量和本地函数。

全局变量

魔兽争霸 3 的 WorldEditor 在很多地方使用起来并不方便。包括声明全局变量，您需要不停的使用界面进行全局变量申明……

并且只允许您在固定的位置声明，例如在“自定义脚本”或“触发器编辑器”中声明变量。

而且在早期的时候，在不用“触发器编辑器”的情况下，甚至只能通过修改地图脚本.j 文件来添加全局变量。或者使用常函数的方式来替代。

而 vJass 完美解决了这个问题。

JassHelper 的编译器将合并地图脚本中所有的全局变量声明块，并将它们移至地图脚本的顶端，最终将所有全局变量声明统一的放在一起，以保障以暴雪的地图脚本规范保存。

```
function something takes nothing returns nothing
    set somearray[SOMETHING_INDEX]=4
endfunction

globals
```

```
    constant integer SOMETHING_INDEX = 45
    integer array somearray
endglobals
```

现在这样写可以正常工作，但 Jass 有一个限制，您不能在全局变量的默认值中使用函数或非恒定值。

例如，可以用 null、1、19923、0xFFF，true，false，“Hi”等。

您不能在变量赋值中调用任何函数或地图初始化中的内容。（尽管可以使用本地 API 函数，但是绝大数的本地 API 函数在全局变量声明中使用时往往会使地图加载线程崩溃）。 您也不能将某个全局变量分配给另一个全局变量，因为实际上没有任何方法可以控制全局变量声明的顺序。

注意：

- 请保持良好的全局命名规范。有些代码规范对全局名称的使用要求纯大写形式。在 common.j 和 blizzard.j 中，一般常量都应为大写，然后以 system_variable 作为良好的变量名。当然……您也可以坚持使用 udg_前缀。

本地函数

此功能（在 0.9.I.0 版本中添加）类似于声明全局变量，但它是针对更高级用户的，因此，如果您不想了解这段中的任何内容，可以直接跳到下一段。

Warcraft III 支持在地图脚本中声明本地 API 函数，但是只能在地图脚本的全局变量声明（globals...endglobals）块之后申明本地 API 函数。和全局变量的使用一样，JassHelper 将在地图上检测到这些声明并将它们移到地图脚本的正确位置。

到底什么是本地接口函数呢?

比如为 AI 脚本创建的一些本地 API 函数未在 common.j 中声明（这些 API 函数可以在\scripts\common.ai 中找到），它们中的某些实际上有可能对 Jass 的编写者有用。

例如：

```
native UnitAlive  takes unit id returns boolean
```

该 API 函数仅在\scripts\common.ai 中被声明，如果我们在地图脚本中声明一次该函数，就可以以更直接的方式判断单位是否存活，而不是

使用 Bj 包装的间接判断函数（其实是通过判断单位生命值是否<=0 来判断存活性的）。

还有可能您用的是魔兽争霸 III 的修改版（modded）或 hacked 版，里面有很多自定义的本地 API 函数 (JAPI)，这些都需要进行一次声明。但为这些自定义的 API 函数导入一个新的 common.j 可能对您来说太麻烦了。因此，在这种情况下，您可以在地图中将新的自定义 API 函数（JAPI）自由的进行的声明。

此功能有一项保护，JassHelper 会自动删除重复的本地 API 函数声明。（即，如果已经在 common.j 中声明了一次，它将从地图脚本中删除重复的声明，以确保您的地图可正常运行）。 这取决于你提供给 JassHelper 的 common.j 是哪个版本。

如果出于某种原因，您(或newgen包)传递给JassHelper的common.j版本与您希望使用的 common.j 版本不同，则需要考虑到这一点。

范例：

```
native GetUnitGoldCost takes integer unitid returns int
eger
function test takes nothing returns nothing
```

```
    call BJDebugMsg("A footman consts : "+I2S( GetUnitG
oldCost('hfoo')+" gold coins" ) )
endfunction
```

二. 库

WorldEditor 和 Jass 的另一个问题是，无法控制地图脚本中触发器的顺序，所以要求用户在编写代码需时刻注意先后的顺序，这是一种反模块化的编程方式，可能使整个过程充满着毫无意义的操作，浪费不必要的精力。

而 vJass 基本上解决了这个问题。

库（一种针对编译器的预处理方式），可以使最重要的函数在编译时自动被放在脚本的最前端，因此可以控制每个函数的顺序。 它还具有依存关系支持，因此您可以将各种函数归纳到不同的模块库，而不必担心它们互相之间调用时，需要过多考虑其在脚本中的位置。

语法很简单：

library 库名称

或者

library 库名称 **requires** 依存库名

或者

library 库名称 **requires** 依存库名 1，依存库名 2 ...

别忘了标记库的结束，使用关键字 **endlibrary**

例子：

```
library B
    function Bfun takes nothing returns nothing
    endfunction
endlibrary

library A
    function Afun takes nothing returns nothing
    endfunction
endlibrary
```

如果 JassHelper 在编译时，找到了此命令，它将确保将 Afun 和 Bfun 函数移到地图脚本的顶端，因此地图脚本的其余部分可以自由调用 Afun() 或 Bfun()函数。

注意：不确定如果从 B 库中的函数调用 Afun()函数会发生什么。该命令只是将库移到顶部，我们并不知道 B 库是在 A 库之前还是之后。

如果 B 库中的函数需要调用 A 库中的函数，我们应该让 JassHelper 知道必须在 B 库之前添加 A 库。这就是**'requires'**关键字存在的原因：

```
    library B requires A
        function Bfun takes nothing returns nothing
```

```
        call Afun()
    endfunction
endlibrary


library A
    function Afun takes nothing returns nothing
    endfuncti
endlibrary
```

注意：出于某种原因：**requires**，**needs** 和 **uses** 都可以正常使用，并且在语法上都具有相同的功能。

它将 Afun 移动到地图的顶部，并将 Bfun 放置在其后，Bfun 现在可以自由调用 Afun()

一个库可以有多个依存关系，只需用逗号将它们分开：

```
library C needs A, B, D
    function Cfun takes nothing returns nothing
        call Afun()
        call Bfun()
        call Dfun()
```

```
    endfunction
endlibrary


library D
    function Dfun takes nothing returns nothing
    endfunction
endlibrary


library B uses A
    function Bfun takes nothing returns nothing
        call Afun()
    endfunction
endlibrary


library A
    function Afun takes nothing returns nothing
    endfunction
endlibrary
```

在脚本顶端的结果会是这样：

```
function Afun takes nothing returns nothing
```

```
endfunction
function Dfun takes nothing returns nothing
endfunction
function Bfun takes nothing returns nothing
    call Afun()
endfunction
function Cfun takes nothing returns nothing
    call Afun()
    call Bfun()
    call Dfun()
endfunction
```

或者可能是这样：

```
function Dfun takes nothing returns nothing
endfunction
function Afun takes nothing returns nothing
endfunction
function Bfun takes nothing returns nothing
    call Afun()
endfunction
function Cfun takes nothing returns nothing
```

```
        call Afun()
        call Bfun()
        call Dfun()
    endfunction
```

或者：

```
    function Afun takes nothing returns nothing
    endfunction
    function Bfun takes nothing returns nothing
        call Afun()
    endfunction
    function Dfun takes nothing returns nothing
    endfunction
    function Cfun takes nothing returns nothing
        call Afun()
        call Bfun()
        call Dfun()
    endfunction
```

依存关系将改变我们的脚本存放顺序。因为每个库都根据函数的使用情况对依存关系进行了设置，所以 3 种可能的方式均不会导致任何编译错误。

请记住以下几点：

- 库名称会区分大小写。
- 如果库 A 依存库 B，库 B 又依存库 A，则会产生一个循环，JassHelper 将报出语法错误。
- 如果库 A 依存的库依存库 B，而库 B 依存库 A，则循环仍然存在。
- 库不能嵌套库。
- 自版本 0.9.B.0 起，库会定义一个全局变量，库的依存情况决定了这个变量的添加顺序。

库的初始化

通常来说，很难确定谁先谁后，因此库还具有 initializer 关键字，可以定义库的初始化函数，该函数将在地图初始化过程中优先被执行。

您可以在库的名称后添加 initializer <函数名>，并使用 ExecuteFunc 使其优先执行，ExecuteFunc 会使用一个新的线程。因为太多的库会进行初始化，导致 init 线程进行了繁重的操作，所以我们可以更好地防

止 init 线程崩溃。 在 initializer 关键字之后，也可以使用依存语句：**requires**，**needs** 和 **uses** 。

只要使用了 initializers 关键字的库函数，都会被添加在脚本前端中运行。 因此，如果库 A 依存库 B 并且两个库都有 initializers 函数，则 B 的 initializers 函数将在 A 之前被调用。

注意，initializer 初始化函数不能有任何参数。

```
library A initializer InitA requires B
    function InitA takes nothing returns nothing
       call StoreInteger(B_gamecache , "a_rect" , Rect(
-100.0 , 100.0 , -100.0 , 100  ) )
    endfunction
endlibrary

library B initializer InitB
    globals
        gamecache B_gamecache
    endglobals
    function InitB takes nothing returns nothing
        set B_gamecache=InitGameCache("B")
```

```
    endfunction

endlibrary
```

B 的 initializer 函数 将在 A 的 initializer 函数之前在初始化线程中调用。

提示：

- library_once 关键字的工作方式与 library 完全相同，但是您可以两次声明相同的库名，它只会忽略第二个声明，并避免添加其内容而不是显示语法错误，与 textmacros 结合使用时非常有用。
- 较旧的 vJass 版本的库语法不同，以//! 开头！ 最终不建议使用，现在将弹出语法错误。

注意：从 0.9.Z.0 开始，库的声明将创建一个名为“LIBRARY_库名称”布尔常量，默认值为真。因此依存的库可以是<可选的>（在 **requires** 之后添加一个 **optional** 关键字前缀），如果未找到所需依存的库，不会发生语法错误。 并且可以使用：**static ifs** 判断库是否存在。

静态 ifs 语句

static ifs 与普通 ifs 相似，不同之处在于

a）条件必须使用布尔值常量，**and** 操作符以及 **not** 操作符。

b）在编译期间会对此进行分析。条件不匹配的代码都会被忽略编译。

```
library OptionalCode requires optional UnitKiller
        globals
            constant boolean DO_KILL_LIB = true
        endglobals

        function fun takes nothing returns nothing
            local unit u = GetTriggerUnit();
            //以下代码会杀死该单位，但是也许可以使用外部库的
''UnitKiller' 函数去执行。
            //仅当 DO_KILL_LIB 为 true 并且 UnitKiller 的库在
脚本中时会执行其中代码。
            static if DO_KILL_LIB and LIBRARY_UnitKille
r then
            //请使用 static if 来判断 UnitKiller 库是否存在。
            //常规的 if 不会在编译时删除这些代码，因此将导致
语法错误。
                call UnitKiller(u);
```

```
        else
            call KillUnit(u);
        endif
    endfunction

endlibrary

library UnitKiller

    function UnitKiller(unit u)
        call BJDebugMsg("Unit kill!");
        call KillUnit(u);
    endfunction

endfunction
```

静态成员

通过添加一些库来对域进行控制是一个好主意，而私有成员（private members）则是保护用户避免冲突的好方法。

```
library privatetest
    globals
        private integer N=0
    endglobals
    private function x takes nothing returns nothing
        set N=N+1
    endfunction

    function privatetest takes nothing returns nothing
        call x()
        call x()
    endfunction
endlibrary

library otherprivatetest
    globals
        private integer N=5
    endglobals
    private function x takes nothing returns nothing
        set N=N+1
    endfunction
```

```
    function otherprivatetest takes nothing returns not
hing
        call x()
        call x()
    endfunction
endlibrary
```

请注意，这两个库都有相同名称的全局变量和函数，但是这不会引起任何语法上的错误，因为 **private** 预处符将确保私有成员在仅可使用的域内使用，并且不会与域外名称相同的内容冲突。

但在这种情况下，私有成员仅能在声明它们的库的域范围内使用。

Scope 域

有时，您不希望代码到达脚本的前端（它实际上并不算是函数库），但仍想对一组全局变量和函数使用私有关键字 private。 这就是我们定义域 **scope** 关键字的原因。

scope 关键字的语法：

scope 名称

[...脚本内容...]

endscope

如此，域内的函数和声明可以自由的在域内使用，但外部代码将无法使用。 （注意，库应被视为具有内部范围的域）

此域内有许多应用程序：

```
scope GetUnitDebugStr


    private function H2I takes handle h returns integer
        return h
        return 0
    endfunction


    function GetUnitDebugStr takes unit u returns strin
g
        return GetUnitName(u)+"_"+I2S(H2I(u))
    endfunction
endscope
```

在这种情况下，该函数使用 H2I，但是 H2I 是一个非常常见的函数名称，因此可能与外部声明同样名称的脚本发生冲突，您可以自己为 H2I 函数

名称添加一个独立的前缀，或者将库设置为依存其他拥有 H2I 的库，但有时可能过于复杂，而且为了降低模块之间的耦合性，建议使用私有关键字 **private**，这样您可以在该域中自由使用 H2I，而不必担心，如果在其他地方声明了另一个 H2I，并且它不是私有函数也没关系。域对私有成员的运用保持最高优先级（当私有成员名称和外部公共名称一样时）。

这对于全局变量更为重要，例如为了封装，则可能希望禁止外部直接访问域内的全局变量，而只允许访问某些开放的函数，以保持一种包络感。

私有（**private**）工作的方式实际上是通过重命名 scopename（随机数）__自动前缀到私有成员的标识符名称上。 随机数是一种使其真正保密的方法，因此人们甚至不能通过添加前缀来使用它们。 使用 double __ 是因为我们认为这是识别预处理程序生成变量/函数的方法，因此您应避免在人工声明的标识符名称中使用 double __。 读取输出文件时（例如，当 PJass 返回语法错误时），能够识别预处理器生成的标识符非常有用。

使用 ExecuteFunc 或真值更改事件，您必须使用 SCOPE_PRIVATE（请参见下面的内容）

提示：作用域支持初始化函数，就像库一样，实现上也有所不同，这是因为它们使用普通调用而不是 ExecuteFunc 调用，如果您需要初始化繁重的函数，最好使用库初始化程序，或使用 ExecuteFunc 调用子函数。

注意：类似于库的方式，域曾经的语法是//！。目前已不支持旧语法，这将导致错误。

公共成员

公共成员与私人成员密切相关，因为他们的行为基本相同，不同之处在于公共成员的名字不会随机化，并且可以在域范围之外使用。 对于在称为 SCP 域中声明为公共的变量/函数，您可以仅在作用域内使用声明的函数/变量名称，但是要在作用域外使用它，则可以使用 SCP_前缀将其调用。

一个例子应该更容易理解：

```
    library cookiesystem

        public function ko takes nothing returns nothin
g

            call BJDebugMsg("a")
```

```
        endfunction


        function thisisnotpublicnorprivate takes nothin
g returns nothing
            call ko()
            call cookiesystem_ko() //cookiesystem_ 字首
可用可不用
        endfunction
    endlibrary


    function outside takes nothing returns nothing
         call cookiesystem_ko() //cookiesystem_ 字首是必
须的
    endfunction
```

公用函数成员可由 ExecuteFunc 或实数变量事件使用，但是当用作字符串时，它们始终需要其库/域名称的前缀：

```
    library cookiesystem
        public function ko takes nothing returns nothin
g
            call BJDebugMsg("a")
```

```
        endfunction


        function thisisnotpublicnorprivate takes nothin
g returns nothing
            call ExecuteFunc("cookiesystem_ko")
                //无论是否在库范围内都需要前缀
            call ExecuteFunc("ko")
                //这很可能会使游戏崩溃。
            call cookiesystem_ko()
                //不需要前缀，但可以这么用。
            call ko()
                //因为不需要前缀，所以该行正常运行。
        endfunction
    endlibrary
```

或者，您可以使用 SCOPE PREFIX（请参见下文）

注意：如果在名为 InitTrig 的函数上使用公共成员关键字 public，则会以特殊方式处理它，而不是成为 ScopeName_InitTrig，它将变为 InitTrig_ScopeName，因此您可以在触发器中使用对应的作用域名称

使其存在于一个域/库内，并使其成为公共成员。而不需要手动再回到域的内部创建 InitTrig_Correctname（不推荐）。

域的嵌套

域可以嵌套，不要将这句话和“库可以嵌套”'混淆，实际上，您甚至不能在**域**的范围内定义**库**。 但是，您可以在库的范围内或者其他域的范围内定义**域**。

一个域在另外一个域内被定义，被视为子域。 子域被认为是父域的公共成员。

子域不能被声明为私有或全局成员。

子域相对于其父域，与普通域相对于整个地图脚本的情况相同。

由于子域始终是公共成员，因此您可以在父域之外访问子域的公共成员，但是它需要父项的前缀和子项的前缀。

一个例子 ：

```
    library nestedtest
        scope A
          globals
            private integer N=4
          endglobals
```

```
        public function display takes nothing returns
 nothing
            call BJDebugMsg(I2S(N))
        endfunction
    endscope

    scope B
        globals
            public integer N=5
        endglobals

        public function display takes nothing retur
ns nothing
            call BJDebugMsg(I2S(N))
        endfunction
    endscope

    function nestedDoTest takes nothing returns not
hing
        call B_display()
```

```
            call A_display()
        endfunction


    endlibrary


    public function outside takes nothing returns nothi
ng
        set nestedtest_B_N= -4
        call nestedDoTest()
        call nestedtest_A_display()
    endfunction
```

下一个示例将导致语法错误：

```
    library nestedtest
        globals
            private integer N=3
        endglobals


        scope A
          globals
            private integer N=4 //错误：N 已经被申明
```

```
        endglobals
    endscope
endlibrary
```

它实际上是由解析器中的一个限制引起的，存在一个冲突，该冲突是由于将 N 用作父级，然后再将其声明给子级而引起的。 但是，以下版本不会导致语法错误：

```
library nestedtest
    scope A
      globals
        private integer N=4
      endglobals
    Endscope

    globals
        private integer N=3
    endglobals

endlibrary
```

这样做看起来的确是一样的，但是由于在父库之前声明了子域的 N，所以解析器不再感到困惑。

要记住的另一件事是，与普通全局变量不同，私有/公共全局变量不能在声明之前使用，否则 JassHelper 会认为它们只是普通变量。

域不能重复声明，不能有两个名称相同的域。 但是也有例外，如果两个子域是不同父域范围的子域（例如不同库中的），虽然它们具有相同的名称，但是因为编译器在编译时，实际上根据其父域或库编译成了不同的名称。

```
    library nestedtest
      scope A
        function kkk takes nothing returns nothing
            set N=N+5
            //当 JassHelper 解析器到达这一行时，它还没有看到私有整数 N 的声明，因此它假定 N 是使用的全局变量并且不进行任何替换
        endfunction
      endscope

   endlibrary
```

```
   scope X
       scope A
       //再次声明域 A 不会造成任何问题，因为实际上是 X_A，所以先前声明的 A 域实际上是 nestedtest_A
           function DoSomething takes nothing returns nothing
           endfunction
       endscope
    endscope
```

域没有任何嵌套限制，但是请注意，根据域嵌套的深度，其私有/公共成员的变量和函数名称会越来越长。 较长的变量名可能会影响游戏运行时的性能，但并不是很多。可以通过地图优化器来防止此效率问题，地图优化器一般会自动缩短函数、变量的名称。

SCOPE_PREFIX 和 SCOPE_PRIVATE 语句

在域/库中，SCOPE_PREFIX 和 SCOPE_PRIVATE 都是可以使用的字符串常量。

SCOPE_PREFIX 将返回当前域的名称（以 Jass 字符串形式），并与一个下码连接。（为公共成员添加了前缀）

SCOPE_PRIVATE 将为私有成员返回当前前缀的名称（作为 Jass 字符串）。

```
scope test
    private function kol takes nothing returns nothing
        call BJDebugMsg("...")
    endfunction
    function lala takes nothing returns nothing
         call ExecuteFunc(SCOPE_PRIVATE+"kol")
    endfunction
endscope
```

在示例中，我们允许 lala() 通过 ExecuteFunc 调用私有函数 kol。

keyword 关键字语句

keyword 关键字语句允许您在不声明实际函数/变量/等的情况下为作用域范围内声明的替换指令。出于多种原因，它很有用，最重要的原因是，您不能在其范围内私有/公共成员声明之前就使用它们，在大多数情况下，此限制仅是一个麻烦，要求您更改对应的声明位置，在某些情况下，这也是 JASS 的局限性。

例如，两个相互递归的函数可以使用.evaluate 相互调用，但是如果您还希望函数是私有的，那么不使用关键字就无法做到这一点：

```
scope myScope


    private keyword B
     //为避免与域的外部冲突，我们可以将 B 声明为私有。
    private function A takes integer i returns nothing
        if(i!=0) then
            return B.evaluate(i-1)*2
     //因为 B 被声明为作用域的私有成员，我们现在可以使用
evaluate 安全调用它
        endif
        return 0
    endfunction


    private function B takes integer i returns nothing
        if(i!=0) then
            return A(i-1)*3
        endif
        return 0
    endfunction
```

```
endscope
```

三. 结构

结构将 Jass 引入面向对象的编程范畴。

如果不先举一个例子，我将无法解释它：

```
struct pair
    integer x
    integer y
endstruct
function testpairs takes nothing returns nothing
    local pair A=pair.create()
    set A.x=5
    set A.x=8

    call BJDebugMsg(I2S(A.x)+" : "+I2S(A.y))

    call pair.destroy(A)
endfunction
```

如您所见，您可以在一个结构中存储多个值，然后就可以像使用另一个 Jass 类型一样使用该结构。请注意，这里的成员的语法类似于大多数编程语言的语法。

声明结构

使用结构之前，您需要先声明它。 语法只是 struct <结构名>和 endstruct 关键字。

要声明成员，您只需使用<类型> <名称> [=初始值]

在上面的示例中，我们声明了一个名为 pair 的结构类型，该结构类型具有 2 个成员：x 和 y，它们没有设置初始值。

通常，将初始值分配给成员是一个好主意，这样您就不必在创建结构对象之后再手动初始化它们，通常的默认值为空值，但是取决于您要解决的问题，取决于您是否需要其他的初始值。

创建和销毁结构

结构是伪动态的，您经常需要创建和销毁结构，并且应该创建一个结构之后将其分配给变量。

创建结构的语法是（实际上是获取其唯一 ID）：结构类型名称.create()

在上述结构的情况下，您将必须使用 pair.create() 获得新的结构。

JassHelper 只是一个预处理器，而不是 hack，因此，vJass 还是受 Jass 自身的局限性，在这种情况下，结构体使用数组的值限制为 8191 个，而且我们不能使用索引 0 (结构体为 null)， 因此限制为 8190 个实例。此限制适用于 vJass 每种类型的实例，因此您具有 8190 个成对结构类型的对象，并且仍然能够具有许多其他类型结构的实例。这意味着，如果您继续创建更多其他类型的结构而不会受到破坏。

在创建实例达到结构实例极限的情况下，structtype.create()将返回 0。

通常情况下，实例极限不是一个令人担心的问题，除非您想创建链表或类似的东西，否则 8190 是一个庞大的数目。

例如，如果仅将结构用于法术实例数据，则甚至不能达到 9 个以上的实例。 并且许多其他实际应用程序基本不会超过 2000 个实例。

除非，不再使用的实例结构不会被摧毁（Destroy）。 在那种情况下，我们应该想办法解决这种问题。（对于结构，与句柄不同，不销毁它们不会增加内存使用率，但是有达到极限的风险）。

通过计算，每秒创建一个结构实例，并且忘记删除它，则地图需要 2 小时 16 分钟才能达到该结构类型的极限。

无论如何，如果您想要让其他人使用它，并且你不确定在运行过程中是否会达到极限，可以在调用 create()之后使用 0 进行判断，并以某种方式阻止该过程，以防发生错误。

使用结构，如果达到限制并且您没有办法去捕获，可能在之后会引起一些冲突，根据其使用的方式，冲突的强度可能为 null 或很大。

如果在编译脚本时打开调试模式，则一旦达到限制，create() 将显示警告消息。

要销毁一个结构，您只需使用 destroy 方法，该方法可以用作实例方法或类方法，在上面的示例中，调用 pair.destroy(a) 用于销毁实例，但是您也可以调用 a.destroy()达到同样的效果。

在您尝试销毁 0（null）结构的情况下，如果打开了调试模式，则 destroy 将不起作用或显示警告消息。

结构运用

只需以声明普通类型的变量/函数/参数的方式声明结构值即可。

声明结构并创建成员后，就可能需要访问它们，通常是(struct value).(member name)。

访问成员后，其用法与变量的用法非常相似。 您可以使用 set 语句，也可以在表达式中将其用作值。

```
struct pair
    integer x=1
integer y=2
endstruct
function pair_sum takes pair A, pair B returns pair
 local pair C=pair.create()
    set C.x=A.x+B.x
    set C.y=A.y+B.y
 return C
endfunction
function testpairs takes nothing returns nothing
 local pair A=pair.create()
 local pair B=pair_sum(A, A)
```

```
 local pair C=pair_sum(A,B)


    call BJDebugMsg(I2S(C.x)+" : "+I2S(C.y))
    //用完结构实例后，别忘了摧毁它们，避免溢出
    call B.destroy()
    call C.destroy()
    call pair.destroy(A)
endfunction
```

将显示“ 3：6”

实例成员

因此，您可以声明任何类型的结构成员，甚至是结构类型。 但是，您不能声明数组成员，在更高版本中此限制应该已被解除。

```
struct pairpair
    pair x=0
    //您不能使用 pair.create()，而应该只对初始默认值使用常量。
    pair y=0
endstruct
function testpairs takes nothing returns nothing
    local pairpair A=pairpair.create()
    local pair x

    set A.x=pair.create()
    set A.y=pair.create()

    set x=A.y
    //注意，我们将 A.y 保存在备用变量中，以便我们可以销毁它。
    set A.y= pair_sum(A.x,A.y)
    //这取代了 A.y，这就是我们保存它的原因

    call BJDebugMsg(I2S(  A.y.x )+" : "+I2S( A.y.y ))
```

```
    //注意 .嵌套
    call A.x.destroy()
    call A.y.destroy()
    call A.destroy()
    call x.destroy()
endfunction
```

结构类型的全局变量

您可以具有结构类型的全局变量。 由于 Jass 的限制，您无法直接对其进行初始化。

```
globals
     pair globalpair=0 //合法
     pair globalpair2= pair.create() //不合法
endglobals
```

您将不得不在 init 函数中分配它们。

静态成员

静态成员的行为就像 struct 语法中的全局变量一样，只需在成员语法之前添加 static 关键字即可。 也可以有静态数组。

static 关键字也可以运用于结构的方法。

公共/私有结构

您可以在域范围内自由声明公共、私有结构以及结构变量

```
scope cool
    public struct a
        integer x
    endstruct

    globals
        a x
        public a b
    endglobals

    public function test takes nothing returns nothing
        set b = a.create()
        set b.x = 3
        call b.destroy()
    endfunction
endscope

function test takes nothing returns nothing
    local cool_a x=cool_a.create()
```

```
    set a.x=6

    call a.destroy()

endfunction
```

方法

方法（**method**）就像函数一样，不同之处在于它们与结构类型相关联。

普通的方法也与结构实例相关联（在本例中为“ this”）

再一次需要一个例子：

```
    struct point
        real x=0.0
        real y=0.0

        method move takes real tx, real ty returns noth
ing
            set this.x=tx
            set this.y=ty
        endmethod

    endstruct

    function testpoint takes nothing returns nothing
        local point p=point.create()
        call p.move(56,89)

        call BJDebugMsg(R2S(p.x))
```

```
    endfunction
```

this：表示当前实例指针的关键字，普通的方法是实例方法，这意味着它们是从已经分配的该类型的变量/值中调用的，它将指向该实例。 在上面的示例中，我们在方法内部使用此方法来分配 x 和 y，当调用 p.move()时，最终会改变 p 指向的实体结构的 x 和 y 属性。

method 语法:您可能会注意到方法语法与函数语法非常相似。

this 替代写法：您还可以使用一个 . 。 当您在实例方法中时。（可以这么写，set .member = value）

方法与普通函数的不同之处在于，可以在任何地方调用它们（全局变量声明时除外），并且您不一定可以在其中使用 waits，sync natives 或 GetTriggeringTrigger()（但是您可以使用任何其他事件响应）， 您也许可以使用它们，但这取决于多种因素，因此不建议您完全使用它们。在下一版本中，编译器在找到它们时甚至可能引发语法错误。

封装形式

封装是一种面向对象的编程概念，在该概念中，您可以授予结构成员的访问权限，换句话说，它对于结构是私有的还是公共的。

```
struct encap
    real a=0.0
    private real b=0.0
    public real c=4.5

    method randomize takes nothing returns nothing
        // 都合法:
        set this.a= GetRandomReal(0,45.0)
        set this.b= GetRandomReal(0,45.0)
        set this.c= GetRandomReal(0,45.0)
    endmethod

endstruct

function test takes nothing returns nothing
    local encap e=encap.create()

    call BJDebugMsg(R2S(e.a)) //合法
```

```
        call BJDebugMsg(R2S(e.c)) //合法
        call BJDebugMsg(R2S(e.b)) //语法错误


    endfunction
```

private 关键字只能在 struct 内声明时使用。 变量和方法都可以声明为公共或是私有。

默认的情况下，结构的所有成员都是公共的，因此 public 关键字不是必需的，所以，如果您希望该成员是私有的，您必须指定 private。

需要指出的是 **readonly** 关键字（只读）的存在，它允许结构外部的代码读取变量，但不能分配变量。目前，它是非标准的，因此您不应在公共发行版本上使用它。

静态方法

静态方法或类方法不使用实例，它们实际上类似于函数，但是由于它们是在结构内部声明的，因此可以使用私有成员。

```
    struct encap
        real a=0.0
        private real b=0.0
        public real c=4.5

        private method dosomething takes nothing return
s nothing
            if (this.a==5) then
                set this.a=56
            endif
        endmethod

        static method altcreate takes real a, real b, r
eal c returns encap
            local encap r=encap.create()
            set r.a=a
            set r.b=b
            set r.c=c
```

```
            call r.dosomething()
            //即使它是私有的，您仍然可以调用它，因为我们在
struct 声明的位置中
            return r
        endmethod

        method randomize takes nothing returns nothing
            // 全部合法:
            set this.a= GetRandomReal(0,45.0)
            set this.b= GetRandomReal(0,45.0)
            set this.c= GetRandomReal(0,45.0)
        endmethod

    endstruct

    function test takes nothing returns nothing
        local encap e=encap.altcreate(5,12.4,78.0)
        call BJDebugMsg(R2S(e.a)+" , "+R2S(e.c))
    endfunction
```

您可能会注意到，通常的 create()语法就像静态方法一样， 但是 destroy()却可以作为静态或实例方法

您可以通过声明自己的静态方法在结构声明内直接覆盖 create()方法，但您可能需要另一个静态方法来为该结构分配一个唯一的结构实例 ID，这是默认情况下添加所有结构的静态方法，即 allocate()， 这是一种私有方法。 当结构没有声明特定的 create 方法时，JassHelper 将在调用.create 时直接使用 allocate()进行实例 id 分配。

从 0.9.Z.1 开始，您还可以通过声明自己的方法来覆盖 destroy 方法。然后使用 deallocate()来调用常规的 destroy 方法。

```
struct vec
   real x
   real y
   real z
   //静态方法 create 必须返回该结构类型的值，create 可以有参数。
   static method create takes real ax, real ay, real az
 returns vec
         local vec r= vec.allocate()
         // allocate()是私有的，并且为该结构获取唯一的实例
ID
```

```
        set r.x=ax
        set r.y=ay
        set r.z=az

     return r
   endmethod
endstruct

function test takes nothing returns nothing
    local vec v= vec.create(1.0 , 0.0 , -1.0 )

    call BJDebugMsg( R2S(v.z) )

    call v.destroy()
endfunction
```

不带任何内容的静态方法也可以用作代码值。

```
    struct something
        static method bb takes nothing returns nothing
            call BJDebugMsg("!!")
        endmethod
```

```
    endstruct


    function atest takes nothing returns nothing
        local trigger t=CreateTrigger()
        call TriggerAddAction(t, function something.bb)
        call TriggerExecute(t)
    endfunction
```

析构处理

析构函数没有实际的语法，但是有一个规则，那就是，如果该结构具有一个名为 onDestroy 的方法，则在实例上发出.destroy()时总是会自动调用该方法。

当结构类型的实例在摧毁时，有清理的东西时，使用 onDestroy 很有用，这样可以节省时间，甚至可以使事情更安全。

```
struct sta
    real a
    real b
endstruct

struct stb
    sta H=0
    sta K=0

    method onDestroy takes nothing returns nothing
        if (H!=0) then
            call sta.destroy(H)
        endif
        if (K!=0) then
```

```
                call sta.destroy(K)
            endif
        endmethod
    endstruct
```

在上面的示例中，只需要销毁 stb 类型的对象，它将自动销毁 sta 类型的附加对象。

译者注：例如你的结构实例中有句柄类型的成员，最好在这里进行统一销毁处理。

结构初始化

通常在地图映射初始化的期间需要对结构的静态成员进行一些初始化，您可以使用 static onInit 方法使代码在映射初始化期间执行。

请注意，结构初始化是在所有库初始化函数之前执行的，如果您需要在其初始化之前执行库初始化函数，请使用库的初始化函数来做这些事情。

不同结构初始化程序之间的相对顺序取决于解析器在地图脚本中找到它们的位置，因此它们实际上还取决于库的顺序（库中的结构初始化方法将在需要它的其他库中的初始化函数之前运行。当然，也要看在之前域内的初始值设定项）。

```
    struct A
        static integer array ko


        private static method onInit takes nothing retu
rns nothing
        //申明为公共方法也是可以的
         local integer i=1000
            loop
                exitwhen (i<0)
```

```
            set A.ko[i]=i*2
            set i=i-1
        endloop
    endmethod
endstruct
```

接口

多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。

多态指同一个实体同时具有多种形式。它是面向对象程序设计（OOP）的一个重要特征。如果一个语言只支持类而不支持多态，只能说明它是基于对象的，而不是面向对象的。

因此，多态性是一个面向对象的概念，其中不同的对象类可能具有相同的行为，尽管具体的行为可能有所不同，但该行为具有相同的抽象名称。例如，一只蚂蚁和一个人都在奔跑，虽然它们是完全不同的对象，而且具体的奔跑动作也是以不同的方式实现的，但他们都具有奔跑这个抽象的行为概念。

接口就像是遵循一组规则的结构类型，并允许您在不确定接口子结构的确切类型的情况下调用结构方法的操作。

```
interface printable
    method toString takes nothing returns string
endinterface
```

```
struct singleint extends printable
    integer v
    method toString takes nothing returns string
        return I2S(this.v)
    endmethod
endstruct

struct intpair extends printable
    integer a
    integer b
    method toString takes nothing returns string
        return "("+I2S(this.a)+","+I2S(this.b)+")"
    endmethod
endstruct

function printmany takes printable a, printable b, prin
table c returns nothing
    call BJDebugMsg( a.toString()+" - "+b.toString()+"
- "+c.toString())
endfunction
```

```
function test takes nothing returns nothing
    local singleint x=singleint.create()
    local singleint y=singleint.create()
    local intpair z=intpair.create()

    set x.v=56
    set y.v=12
    set z.a=45
    set z.b=12

    call printmany(x,y,z)
endfunction
```

printmany 函数具有 3 个 printable 类型的参数，并且该函数并不确定这些参数对象是 printable 的哪种子结构类型，仅知道它们都是继承了 printable 结构的类型。

不过这些子结构都遵循了 printable 的接口规则，它们都具有 toString()的方法。因此，printmany 函数即使不知道参数对象的确切类型，但依然可以使用 toString()方法。

一个接口可以具有不限数量的方法，继承的子结构应当实现所有这些方法，当然，也可以实现其他额外的方法。

为接口声明 onDestroy 方法是非法的，您可以认为它是默认就被声明的，因此，直接就可以在接口类型的变量上使用.destroy()，并在需要时，也可以为其子结构申明 onDestroy 方法。

对于结构 singleint 和结构 intpair 来说，toString() 方法很明显是不同的，但当调用 printmany 函数时，它会为每个参数对象调用 toString() 方法，因此，以上示例代码的结果为：“56-12-（45,12）”。

接口也可以用变量来实现，在这种情况下，接口允许一些“伪”继承：

```
interface withpos
    real x
    real y
endinterface

struct rectangle extends withpos
    real a
    real b
```

```
    static method from takes real x, real y, real a, re
al b returns rectangle
        local rectangle r=rectangle.create()
        set r.x=x
        set r.y=y
        set r.a=a
        set r.b=b
        return r
    endmethod

endstruct

struct circle extends withpos
    real radius=67.0

    static method from takes real x, real y, real rad r
eturns circle
        local circle r=circle.create()
        set r.x=x
        set r.y=y
        set r.radius=rad
```

```
        return r
    endmethod

endstruct

function distance takes withpos A, withpos B returns re
al
    local real dy=A.y-B.y
    local real dx=A.x-B.x
    return SquareRoot( dy*dy+dx*dx)
endfunction

function test takes nothing returns nothing
    local circle c= circle.from(12.0, 45.0 , 13.0)
    local rectangle r = rectangle.from ( 12.3 , 67.8, 1
2.0 , 10.0)
    call BJDebugMsg(R2S(distance(c,r)))
endfunction
```

typeid 和 getType()

可以获取子结构实例的类型 ID，此 ID 是一个整数，每个整数对于该接口的子结构类型都是唯一的。

```
interface A
    integer x
endinterface
struct B extends A
    integer y
endstruct
struct C extends A
    integer y
    integer z
endstruct
function test takes A inst returns nothing
   if (inst.getType()==C.typeid) then
     //我们肯定知道 inst 实际上是 C 结构类型的实例
       set C(inst).z=5
     //注意转换类型操作符
   endif
   if (inst.getType()==B.typeid) then
       call BJDebugMsg("It was of type B with value "+I
```

```
2S( B(inst).y  ) )
   endif
endfunction
```

简而言之，.getType() 是一种获取接口子结构实例类型的方法。 .typeid 则是接口子结构类型的静态常量 id。

因此，在示例中，我们知道了给定 inst 参数的结构类型为 C，然后就可以对其进行类型转换和赋值。

typeid 的另一个功能是，它使接口具有一个构造函数的方法，该方法可以在给定正确的 typeid 的情况下创建一个新对象。

例如：

```
interface myinterface
    method msg takes nothing returns string
endinterface

struct mystructA extends myinterface
    method msg takes nothing returns string
        return "oranges"
    endmethod
```

```
endstruct


struct mystructB extends myinterface
    string x
    static method create takes nothing returns mystructB
        local mystructB m=mystructB.allocate()
        set m.x="apples"
        return m
    endmethod
    method msg takes nothing returns string
        return this.x
    endmethod
endstruct


struct mystructC extends myinterface
    string x
    //myinterface.create(...) 只能使用默认的构建函数或不带任何参数的自定义构建方法。如果在 myinterface.create (...) 中使用了 mystructC，则不会考虑以下声明。
    static method create takes string astring returns my
structC
```

```
        local mystructB m=mystructB.allocate()
        set m.x=astring
        return m
    endmethod

    method msg takes nothing returns string
        return this.x
    endmethod

endstruct

function test takes nothing returns nothing
    local integer T = mystructB.typeid
    local myinterface A
    set A=myinterface.create(mystructA.typeid)
    //这没什么用，因为和 mystructA.create()的作用相同
    call BJDebugMsg(A.msg())

    set A=myinterface.create(T)
    //这更有用，这里 A 变量创建了对应类型的对象...
    call BJDebugMsg(B.msg())
```

```
    set A=myinterface.create(122345)
    //使用无效值或 0 将使.create 返回 0（无对象）
    set A=myinterface.create(mystructC.typeid)
    //请注意，这里不会使用 mystructC.create，而只会使用
mystructC.allocate，而且可能会导致错误，如果发生这种情况，则
在调试模式编译时，会在游戏中显示错误的消息
    call BJDebugMsg(C.msg())
endfunction
```

如果您计划使用此功能，请始终谨慎溢出（0 返回值），并且子结构的构造函数最好没有参数。

还可以通过不返回任何值的 create 方法来声明接口：

```
interface myinterface
    static method create takes nothing
    method qr takes unit u returns nothing
endinterface

struct st1 extends myinterface
    static method create takes nothing returns st1
    //以上合法
```

```
            return st1.allocate()
        endmethod

        method qr takes unit u returns nothing
            call KillUnit(u)
    endmethod
endstruct

struct st2 extends myinterface
    integer k
    static method create takes integer f, integer k ret
urns st2
    //以上不合法
        local st2 s=st2.allocate()
            set st2.k=f+k*f
        return st2
        endmethod
        method qr takes unit u returns nothing
            call ExplodeUnitBJ(u)
    endmethod
endstruct
```

defaults 关键字

接口的方法可让您使用 **defaults** 关键字。**defaults** 关键字允许在实现子结构时，该方法可申明也可以不申明。 因此，如果该方法不存在于子结构中，则不会显示要求实现该方法的语法错误。但是实际上仍将默认实现一个空的方法。

defaults 后跟“nothing”或一个值 取决于方法的返回值类型（如果该方法不返回任何值则对于子结构来说，默认就是如此）。 **defaults** 仅支持常量值。

```
interface whattodo
    method onStrike takes real x, real y returns boolea
n defaults false
    method onBegin takes real x, real y returns nothing
 defaults nothing
    method onFinish takes nothing returns nothing
endinterface

struct A extends whattodo
```

```
//别忘了继承语句 extends...

    //当有人在 A 类型的 whatdo 上调用.onStrike 时，它将返回
false

    //我们可以添加 onBegin 方法，但不是被迫去添加的
    method onBegin takes real x, real y returns nothing
        //必须有可执行的
        //...代码
    endmethod

    method onFinish takes nothing returns nothing
        //必须有可执行的
        //...代码
    endmethod

endstruct

struct B extends whattodo

    //当有人在 A 类型的 whattodo 上调用.onBegin 时，它将什么都
```

不做

```
    method onFinish takes nothing returns nothing
        //必须有可执行的
        //...代码
    endmethod
endstruct
```

重载

JassHelper 允许您为结构声明操作符，然后这些操作符的内容将转换为方法调用，vJass 当前允许<，>，[]=数组赋值和[]获取数组的操作符。

此过程的正式名称（在 Wikipedia 和书籍中）是操作符重载（operator overloading）。对于 vJass，重载操作符是一种方法，在指定 operator 关键字后再指定操作符作为名称，但它是可以使用操作符来调用的方法。

一个例子顶一千字的说明：

```
    struct operatortest
        string str=""

        method operator [] takes integer i returns stri
ng
            return SubString(.str,i,i+1)
        endmethod

        method operator[]= takes integer i, string ch r
eturns nothing
                            set .str=SubString(.str,0,i
```

```
)+ch+SubString(.str,i+1,StringLength(.str)-i)
        endmethod


    endstruct


    function test takes nothing returns nothing
        local operatortest x=operatortest.create()
        set x.str="Test"
        call BJDebugMsg( x[1])
        call BJDebugMsg( x[0]+x[3])


        set x[1] = "."
        call BJDebugMsg( x.str)
    endfunction
```

在此示例中，我们重载了[]操作符，并为 operatortest 类型的对象赋予了新功能。

检查代码后，您可能会注意到我们将字符串作为了字符串数组来使用。

[]操作符需要 1 个参数（索引）且会返回一个值，而[] =操作符需要 2 个参数（索引和要分配的值）。

也可以将[]和[] =操作符声明为静态。 这可能会有一些用途，结构名称将被允许用作索引操作符。

对操作符重载有很多批评，因为它允许程序员编写没有意义的代码，请负责任地使用此功能。

您还可以重载 > 和 < ，请注意，只有通过声明 < 来声明您正在强行确定该结构类型的顺序关系的语法才能声明 < 。 因此，它会根据您的 < 声明自动生成 > 。

```
    struct operatortest
        string str=""

        method operator [] takes integer i returns stri
ng
            return SubString(.str,i,i+1)
        endmethod
```

```
        method operator[]= takes integer i, string ch r
eturns nothing
            set .str=SubString(.str,0,i)+ch+SubString(.
str,i+1,StringLength(.str)-i)
        endmethod

        method operator< takes  operatortest b returns
boolean
            return StringLength(this.str) < StringLengt
h(b.str)
        endmethod

    endstruct


    function test takes nothing returns nothing
        local operatortest x=operatortest.create()
        local operatortest y=operatortest.create()

        set x.str="Test..."
        set y.str=".Test"
```

```
        if (x<y) then
            call BJDebugMsg("Less than")
        endif
        if (x>y) then
            call BJDebugMsg("Greater than")
        endif
    endfunction
```

在该示例中，如果类型为 operatortest 的对象的 str 成员的长度大于另一个对象的 str 成员的长度，则认为该对象大于的另一个对象。

operator < 必须返回的是布尔值，并接受与结构类型相同的参数进行判断。

操作符从可读性上更为友好，这意味着接口也可以声明操作符，但有一个陷阱，那就是必须在接口中声明 operator < 而不带名称。 同样，当使用>或<来比较接口对象时，两个实例都必须具是相同的类型，否则该函数将在执行比较之前就会停止运行，如果在编译时启用了调试模式，还将显示警告消息。

```
    interface ordered
        method operator <
```

```
    endinterface


    interface indexed
        method operator [] takes integer index returns
ordered
        method operator []= takes integer index, ordere
d v returns nothing
    endinterface


    function sort takes indexed a, integer from, intege
r to returns nothing
        local integer i
        local integer j
        local ordered aux


        set i=from
        loop
            exitwhen (i>=to)
            set j=i+1
            loop
                exitwhen (j>to)
```

```
            if (a[j]<a[i]) then
                set aux = a[i]
                set a[i] = a[j]
                set a[j] = aux
            endif
            set j=j+1
        endloop

        set i=i+1
    endloop
endfunction
```

这是排序算法的接口。 现在，我们可以声明对内容进行排序的新结构类型：

```
struct integerpair extends ordered
    integer x
    integer y

    method operator< takes integerpair b returns bo
```

```
olean
            if (b.x==this.x) then
                return (this.y<b.y)
            endif
            return (this.x<b.x)
        endmethod
    endstruct

    type ipairarray extends integerpair array [400]

    struct integerpairarray extends indexed
        ipairarray data

        method operator[] takes integer index returns o
rdered
            return ordered( this.data[index] )
        endmethod

        method operator[]= takes integer index, ordered
 value returns nothing
            set this.data[index] = integerpair( value)
```

```
        endmethod
    endstruct
```

当然，这只是一个示例，使用的 quicksortd 逻辑方式。

操作符也完全支持文本宏。

您还可以声明一个自定义的== ，运作方式类似<。!=将自动转换为not(方法)。要进行指针比较,您将需要使用 integer(var1)== integer(var2)

我们可以使用自定义操作符做更多的事情

通过重载[]，>，<，您可以使方法模仿字段，以简化语法和保持抽象性。

```
struct X
    integer a=2
    integer b=2

    method operator x takes nothing returns integer
```

```
        return this.a*this.b
    endmethod

    method operator x= takes integer v returns nothing
        set this.a=v/this.b
    endmethod

endstruct

function test takes nothing returns nothing
    local X obj= X.create()

    set obj.x= obj.x + 4

    call BJDebugMsg(I2S( obj.x) ) //显示 8
    set obj.b=4

    call BJDebugMsg(I2S( obj.x) ) //显示 16
endfunction
```

您可以使用它来实现结构的只读字段：

```
struct X
    private integer va=2

    method operator a takes nothing returns integer
        return this.a
    endmethod
endstruct

function test takes nothing returns nothing
    local X obj= X.create()

    call BJDebugMsg(I2S( obj.a) ) //合法

    set obj.a=2 //不合法
endfunction
```

更重要的是，您可以使用它来实现需要额外分配的字段：

```
struct movableEffect


    private unit dummy
    private string rfx
    private effect uniteffect


    //...
        //(一些执行动作和创建的代码)
    //...


    method operator x takes nothing returns real
        return GetUnitX(this.dummy)
    endmethod


    method operator y takes nothing returns real
        return GetUnitY(this.dummy)
    endmethod


    method operator x= takes real value returns nothing
        call SetUnitX(this.dummy, value)
    endmethod
```

```
    method operator y= takes real value returns nothing
        call SetUnitY(this.dummy, value)
    endmethod


    method operator effectpath takes nothing returns st
ring
        return this.rfx
    endmethod

    method operator effectpath= takes string path retur
ns nothing
        set this.rfx=path
        call DestroyEffect( this.uniteffect)
        set this.uniteffect = AddSpecialEffectTarget(th
is.dummy, path, "origin")
    endmethod
endstruct


function moveRandom takes movableEffect me returns noth
ing
```

```
    set me.x= me.x + GetRandomReal(-50,50)
    set me.y= me.y + GetRandomReal(-50,50)
endfunction

function toFire takes movableEffect me returns nothing
    set me.effectpath ="war3mapimporte\\cutefireeffect.
mdl"
endfunction
```

注意：从 0.9.Z.1 开始，静态成员也支持此语法。

结构的继承

vJass 可以从先前已声明的结构中再构造一个结构。通过这种方式，您的新结构可以继承之前结构（之前的结构称为新结构的基本结构）的方法和变量成员，并且还可以与之前结构类型的实例一同使用。

一种将新的方法和变量成员添加到先前结构类型的方法。

```
struct A
    integer x
    integer y

    method setxy takes integer cx, integer cy returns n
othing
        set this.x=cx
        set this.y=cy
    endmethod
endstruct
struct B extends A
    integer z
    method setxyz takes integer cx, integer cy, integer
 cz returns nothing
        call this.setxy(cx,cy) //我们可以使用 A 的成员
```

```
        set this.z=cz
    endmethod
endstruct
```

在内部，B.allocate()实际上正在调用 A 的构造函数，而 B.destroy 也会调用 A 的解构函数。 如果基本结构具有自定义的 create 方法，则对其继承的结构必须使用 allocate()来进行创建。如果基本结构的 create 方法需要参数，则继承后的子结构 allocate()方法将需要相同的参数：

```
struct A
    integer x
    static method create takes integer k returns A
    local A s=A.allocate()
        set A.x=k
    return s
    endmethod
endstruct

struct B extends A
    static method create takes nothing returns B
```

```
        return s= B.allocate(445)
        //注意，B.allocate 需要与 A.create（）相同的参数
    endmethod
endstruct

struct C extends B
//是的，继承的子结构可以再次被继承
    static method create takes nothing returns B
         local C s=C.allocate()
// C 是 B 的子结构，以为 B 的 create 方法没有任何参数，因此这里也不带任何参数。
        set s.x=s.x*s.x
       //使用父结构的成员。
        return s
    endmethod
endstruct
```

如果一个结构声明 create 为私有的，则该结构无法被继承。 因为子结构不能使用父结构的私有成员。

在这种情况下，onDestroy 方法的行为很特殊，如果在上一个示例中，A，B 和 C 都有一个 onDestroy 方法，则销毁 C 的实例将依次按顺序调用 C.onDestroy()，B.onDestroy()和 A .onDestroy()。

可以扩展用于扩展接口的结构，在这种情况下，直接扩展接口的子级将被强制实现接口的方法，而子级则不会。 但是它们有可能再次替换它们。

```
interface myinterface
    method processunit takes unit u returns nothing
    method onAnEvent takes nothing returns boolean
endinterface

struct A extends myinterface
    method processunit takes unit u returns nothing
        call KillUnit(u)
    endmethod

    method onAnEvent takes nothing returns boolean
        return false
    endmethod
```

```
endstruct


struct B extends A


    method processunit takes unit u returns nothing
        //我们刚刚替换了 processunit 方法，
        //如果类型为 myinterface 的接口变量包含一个实例
        //键入 B，它将使单位爆炸。
        call ExplodeUnitBJ(u)
    endmethod
    //我们正在实现 processunit，但不必实现 onAnEvent
endstruct
```

如果使用 interface.create()，则必须同样注意带有参数的构造函数，如果声明的接口 create 不需要任何操作，则该接口的每个子项（，子子项等）都将受到此条件影响。

存根方法

什么是存根方法?

stub 即存根，比如支票存根，存根与支票对应，与支票有一对一的关系，保留有支票的部分信息等。stub method 也就是这种含义，本地不可能有远程对象的直接引用，所以就使用 stub 作为远程对象的替身，stub method 即远程的替身方法。

存根在面向对象的编程范畴是一个类，它实现了一个接口，但是实现后的每个方法都是空的。

它的作用是：一个接口有很多方法，如果要实现这个接口，就要实现所有的方法。但是一个类从业务来说，可能只需要其中一两个方法。如果直接去实现这个接口，除了实现所需的方法，还要实现其他所有的无关方法。而如果通过继承存根类就实现接口，就免去了这种麻烦。

存根方法可以简单地由子结构重写。 一个例子应该有帮助：

```
struct Parent


    stub method xx takes nothing returns nothing
        call BJDebugMsg("Parent")
    endmethod
```

```
    method doSomething takes nothing returns nothing
        call this.xx()
        call this.xx()
    endmethod

endstruct

struct ChildA extends Parent
    method xx takes nothing returns nothing
        call BJDebugMsg("- Child A -")
    endmethod
endstruct

struct ChildB extends Parent
    method xx takes nothing returns nothing
        call BJDebugMsg("- Child B --")
    endmethod
endstruct

function test takes nothing returns nothing
```

```
    local Parent P = Parent.create()
    local Parent A = ChildA.create()
    local Parent B = ChildB.create()
    //注意以上变量用的是'Parent'类型。
    call P.doSomething() //显示 'Parent' 两次
    call A.doSomething() //显示 'Child A' 两次
    call B.doSomething() //显示 'Child B' 两次
endfunction
```

请注意，存根和接口之间存在着差异，首先，接口需要您遵循其定义，必须实现对应的方法。它们还允许.exists()。

Super 语句

当您继承一个结构时，可能该结构继承的是一个接口，或者您编写的方法是存根方法。 如果要调用父方法怎么办呢？不指定您要执行的操作是不可能的（否则，它将最终改为调用子级的方法）。

super 语句的意思便是强制调用父方法：

```
struct Parent


    stub method xx takes nothing returns nothing
        call BJDebugMsg("Parent")
    endmethod


    method doSomething takes nothing returns nothing
        call this.xx()
        call this.xx()
    endmethod


endstruct


struct ChildA extends Parent
    method xx takes nothing returns nothing
```

```
        call BJDebugMsg("- Child A -")
        call super.xx()
    endmethod
endstruct

struct ChildB extends Parent
    method xx takes nothing returns nothing
        call BJDebugMsg("- Child B --")
    endmethod
endstruct

function test takes nothing returns nothing
    local Parent P = Parent.create()
    local Parent A = ChildA.create()
    local Parent B = ChildB.create()
    call P.doSomething() //显示 'Parent' 两次
    call A.doSomething() //显示 'Child A|nParent' 两次
    call B.doSomething() //显示 'Child B' 两次
endfunction
```

动态数组

动态数组是可以动态实例化的数组，每个自定义数组类型的索引总数为 8190 个，这意味着大小为 100 的数组类型的实例总数为 81。

它们很容易声明和使用，并以某种方式与结构共享语法。

您只需在任何 function/struct 声明的外部添加一行：type <动态数组类型名称> extends <被继承的类型> array [<size>] 其中 size 可以是整数值或全局常量。

然后，您可以简单地以类似于结构的方式创建和使用它们，并使用[]操作符访问其索引，动态数组也具有静态常量的大小。

```
type arsample extends integer array[8]
function test takes nothing returns arsample
    local arsample r=arsample.create()
    local integer i=0
    loop
        exitwhen i==arsample.size //数组的大小
        set r[i]=i
        set i=i+1
```

```
    endloop
    return r
endfunction

function test2 takes nothing returns arsample
    local arsample r=test()
    local integer i=0
    loop
        exitwhen i==arsample.size
        call BJDebugMsg(I2S(r[i]))
        set i=i+1
    endloop
    return r
endfunction
```

您可以扩展任何类型的数组，甚至可以扩展自定义的类型（结构，接口，其他动态数组）的数组，从而使动态数组也成为动态数组，可以使用类似于矩阵的语法：

```
type iar extends integer array[3]
type iar_ar extends iar array[3]
```

```
function test takes nothing returns arsample
    local iar_ar r=iar_ar.create()
    local integer i=0
    local integer j
    loop
        exitwhen i==iar_ar.size //数组的大小
        set r[i]=iar.create()
        set j=0
        loop
            exitwhen j==iar.size
            set r[i][j]=j*i
            set j=j+1
        endloop

        set i=i+1
    endloop
    return r
endfunction
```

并且结构可以将这些数组作为成员，从而允许实例也带有数组成员（非静态）

```
type stackarray extends integer array [20]
struct stack
   private stackarray V
   private integer N=0

   method Create takes nothing returns stack
    local stack s=stack.create()
        set s.V=stackarray.create()
        if (s.V==0) then
            debug call BJDebugMsg("Warning: not enough
space for stack array")
            return 0
        endif
   endmethod

   method push takes integer i returns nothing
        if (this.N==stackarray.size) then
            debug call BJDebugMsg("Warning: stack is fu
ll")
        else
            set this.V[this.N]=i
```

```
            set .N = .N +1 //记住这个语法也是有效的
        endif
    endmethod

    method pop takes nothing returns nothing
        if (this.N>0) then
            set this.N=this.N-1
        else
            debug call BJDebugMsg("Warning: attempt to
pop an empty stack");
        endif
    endmethod

    method top takes nothing returns integer
        return .V[.N-1]
    endmethod

    method empty takes nothing returns boolean
        return (.N==0)
    endmethod
```

```
    method onDestroy takes nothing returns nothing
        call this.V.destroy()
    endmethod
endstruct
```

您可能会注意到，如果没有空间容纳新实例，则数组的 create 方法将返回 0。如果在调试模式下进行编译，它还会自动警告您!

动态数组默认具有 .size 成员，该成员使您可以轻松访问用于声明数组类型的大小。

数组成员

结构可以具有数组成员，但您还需要声明它们的大小。

```
struct stack
   private integer array V[100]
   private integer N=0

   method push takes integer i returns nothing
      set .V[.N]=i
      set .N=.N+1
   endmethod

   method pop takes nothing returns nothing
      set .N=.N-1
   endmethod

   method top takes nothing returns integer
      return .V[.N-1]
   endmethod

   method empty takes nothing returns boolean
      return (.N==0)
```

```
    endmethod


    method full takes nothing returns boolean
        return (.N==.V.size)
    endmethod
endstruct
```

在某种情况下，这是用于声明新的数组类型并使其成为结构体成员的语法，但不同的是，这会带来一些限制，这样是为了更多地优化和处理数组的分配和释放。

请注意，这大大降低了结构类型的实例上限，例如，我们只能有 80 个以上声明的 stack 堆栈对象实例（8190 除以 100）。

一个结构可能需要有尽可能多的数组成员，请注意，数组成员的大小是是设置实例上限时需要考虑的一项，因此，如果一个结构具有 2 个数组成员，一个数组大小为 4，一个数组大小为 100， 该结构将限制为 80 个实例。

与单独声明的动态数组类型相比，此方法的缺点在于，在将成员分配给您在其他场合中创建的另一个数组以及类似的事情时自由度降低了...优点是它速度更快，所需的代码更少。

作为动态数组，数组成员也可以使用.size 字段。

委托

到目前为止，我们已经看到了 vJass 得许多东西，接口、结构、结构的继承、操作符重载、动态数组……您可能会问自己，vJass 可能还会有另一种东西使我像 heck 一样困惑吗？ 别绝望！ 委托就是……

What the heck?

表示您不喜欢或不理解的烦恼……

委托（**delegate**）是一个奇怪的功能，委托只是为其填充内容的结构成员。该结构只是将工作委托给另一个对象，在这种情况下，和“方法”一样。它看起来没有意义，或者只是缩写，但是它可能非常有用，并且是继承有趣的替代方案。委托正在其他一些语言中使用，只需注意 Jass 并不是很动态，而 vJass 确实也继承了许多缺陷。 不管是好是坏，对于 vJass 而言，委托完全是一个在编译时的工作（所以并不够动态）。

委托完成结构的工作，这是一种非常简单的方法，而更为复杂的方法是，在编译过程中，如果 JassHelper 无法找到某个请求成员的方法，它将开始在结构体中查找该成员，如果它在其中一个委托中找到该成员，则将其编译为对委托成员的调用。

```
struct A
```

```
    private real x
    private real y

    public method performAction takes nothing returns n
othing
        call DestroyEffect( AddSpecialEffect("path\\mod
el.mdl", this.x, this.y) )
    endmethod
endstruct

struct B
    delegate A deleg

    static method create takes nothing returns B
        local B b = B.allocate()
        set B.deleg = A.create()
    endmethod
endstruct

function testsomething takes nothing returns nothing
    local B myB = B.create()
```

```
    call myB.performAction()

    //由于 performAction () 不是结构 B 的成员，因此 JassHelper 将检测出委托，它确实有一个名为 performAction 的方法，因此将尝试调用它，结果将与以下内容相同：

    call myB.deleg.performAction()

endfunction
```

一些注意事项：

- 您可以有多个委托，但相比规则来说这应该是例外。
- JassHelper 优先在该结构的委托之前先访问该结构的成员，这意味着如果两个结构和一个委托具有相同的成员，JassHelper 将始终考虑结构的委托。
- 在同一结构内的委托之间，作为申明指令的优先级是相同的。
- 您可以做很多奇怪的事情，例如委派给数组成员，在这种情况下，您甚至可以使用.size（）和[]。
- 你也可以毫无意义地将整数成员作为委托，这不会导致语法错误，但是它本身并不能真正做很多事情，因为整数没有成员。

- 现在，您不能用委托方法来满足接口的规则，例如：如果结构 B 正继承某个接口，需要一个名为 performAction 的方法，JassHelper 无法识别委托的方法，并会导致语法错误，将来可能会修订。
- 如果您不想代码出现错误，通常必须初始化委托。
- 您可以有一个私有的委托，只有在需要该成员的情况下，外部代码才能访问它。
- 如果尝试调用/使用委托的私有成员，则可能会出现语法错误，即无法在结构中找到它，而不是告诉您它是委托的私有成员。

Thistype 语句

thistype 关键字在结构内部代码中，等同于该结构的名称。

```
//下面的代码，
struct test
    thistype array ts
    method tester takes nothing returns thistype
        return thistype.allocate()
    endmethod
endstruct

//等同于：

struct test
    test array ts
    method tester takes nothing returns test
        return test.allocate()
    endmethod
endstruct
```

四. 模块化

模块就像代码包一样，您可以将其添加到结构中以获取额外的方法或成员等。

module，...，endmodule 用于声明模块，可用于将模块的成员复制到结构中。此可视为高级的文本宏。

模块中的方法可以调用属于该结构的方法/成员（可以是私有的），只需注意，如果该结构不具有对应的成员，则会弹出语法错误。用起来就像是您将模块的代码复制粘贴到结构之中。模块的私有成员对调用的结构将不可见，并且它们的名称不会与结构中的其他成员冲突，但是有一些例外：create 和 onDestroy 将以不同的方式进行处理。 您不能在模块中使用私有操作符（模块通常都是公共的 API，因此无论如何都不能将它们设为私有）。

从 JassHelper 0.9.Z.1 开始，模块内部的私有 onInit 方法将在使用该模块的每个结构初始化上执行一次。来自多个模块的 onInit 方法也可以与该结构的 onInit 方法共存。

```
//
// 声明模块，类似于结构声明
//
```

```
module MyRepeatModule
    method repeat1000 takes nothing returns nothing
        local integer i=0
        loop
            exitwhen i==1000
            call this.sub() //预期在 struct 中会使用的方法
            set i=i+1
        endloop
    endmethod
endmodule

// 看看这个结构:
struct MyStruct
    method sub takes nothing returns nothing
        call BJDebugMsg("Hello world")
    endmethod

    implement MyRepeatModule //添加模块

endstruct
```

```
function MyTest takes MyStruct ms returns nothing
    call ms.repeat1000() //将调用 ms.sub 1000 次
endfunction
```

您可以从模块内部再调用其他模块，在 implement 之后添加关键字 optional 将使它们成为可选的模块，即：如果找不到该模块，则不会显示任何错误，vJass 只会忽略该模块的调用。如果在结构中有已使用的模块，再次使用 implement 也将被忽略。

模块的内容遵循声明它的域/库中的范围规则（如果有）。

```
module MyOtherModule
    method uhOh takes nothing returns nothing
    endmethod
endmodule

//
// 声明模块，类似于结构声明
//
module MyModule
    //下一行添加一个不执行任何操作的成员 uhOh
    implement optional MyOtherModule
```

```
    //由于未声明 OptionalModule，因此将忽略下一行
    implement optional OptionalModule

    //此方法调用要求该结构具有 copy()方法
    static method swap takes thistype A , thistype B re
turns nothing
        local thistype C = thistype.allocate()
        //我们在内部，因此即使是私有的也可以实例化

        call C.copy(A)
        call A.copy(B)
        call B.copy(C)
        call C.destroy()

    endmethod

endmodule

struct MyStruct
    integer a
```

```
    integer b
    integer c

    //编写复制方法
    method copy takes MyStruct x returns nothing
        set this.a = x.a
        set this.b = x.b
        set this.c = x.c
    endmethod

    //自由的获取到了交换方法
    implement MyModule
    implement MyOtherModule //此模块已被 MyModule 包含，因此此行将被忽略

endstruct

function MyTest takes MyStruct A, MyStruct B returns nothing
    call MyStruct.swap(A,B) //现在有了该方法!
endfunction
```

五. 将函数作为对象

对于 vJass 的函数来说，它们可以作为对象，并具有两种方法可以进行调用：evaluate 和 execute，这两种方法都具有与函数相同的参数，而 evaluate 也具有其返回值。

将函数用作对象有许多优点， evaluate() 甚至允许您从函数声明上方的代码中调用该函数，execute 也一样，但能在另一个线程中运行该函数。

缺点是：evaluate() 一起使用的函数不应使用 GetTriggeringTrigger()（但您可以使用其他事件响应）或任何本地同步，evaluate() 不支持等待，并且 evaluate() 的速度比正常的函数调用要慢。

.execute()实际上比旧的 ExecuteFunc 要快，而在更高的版本中，它可能会更快。

evaluate()则比 ExecuteFunc 要快一倍，以后可能会更快。

对于具有参数的函数，在使用 ExecuteFunc 时必须使用全局变量来传递参数，.execute 和.evaluate 则可以直接传递参数。

```
function A takes real x returns real
```

```
    if(GetRandomInt(0,1)==0) then
        return B(x*0.02)
    endif
    return x
endfunction

function B takes real x returns real
    if(GetRandomInt(0,1)==1) then
        return A(x*1000.)
    endif
    return x
 endfunction
```

这两个是相互递归的函数，对于普通的 Jass，这会产生语法错误，为了防止出现此问题，您可以使用 evaluate：

```
function A takes real x returns real
    if(GetRandomInt(0,1)==0) then
        return B.evaluate(x*0.02)
    endif
    return x
endfunction
```

```
function B takes real x returns real
    if(GetRandomInt(0,1)==1) then
        return A(x*1000.)
    endif
    return x
endfunction
```

假设您需要等待x秒钟后摧毁某个特殊效果,您可以使用计时器来实现,但是由于我们可能需要正常的等待,但我们又不想停止执行销毁此效果的函数,因此我们需要一个新线程:

```
function DestroyEffectAfter takes effect fx, real t ret
urns nothing
    call TriggerSleepAction(t)
    call DestroyEffect(fx)
endfunction

function test takes nothing returns nothing
    local unit u=GetTriggerUnit()
    local effect f=AddSpecialEffectTarget("Abilities\\S
pells\\Undead\\Cripple\\CrippleTarget.mdl",u,"chest")
    call DestroyEffectAfter.execute(f,3.0)
```

```
    set u=null
    set f=null
endfunction
```

注意：此功能目前仅限于地图脚本中声明的函数，您不能将其用于 common.j 的本地函数，或 blizzard.j 函数。

函数对象的.name 成员将返回一个包含函数编译后名称的字符串，当您想在域内使用 ExecuteFunc 之类的函数时，此字符串会很有用。

```
scope test
    public function xxx takes nothing returns nothing
        call BJDebugMsg(xxx.name) //将显示“ test_xxx”
    endfunction
endscope
```

函数接口

如果函数是对象，那么我们也可以为它们提供接口。

函数接口的语法为：

interface 接口名称 **takes** （参数） **returns** （返回值）

它实际上类似于函数声明。

可以使用上面定义的 execute() 和 evaluate()调用函数接口类型的变量/值：

要分配给函数接口类型的变量，您首先需要获取函数的指针。如果您假设在地图脚本中找到的每个符合函数接口参数、返回规则的函数都是其静态成员，那么获取它们的语法是可以理解的：

```
function interface Arealfunction takes real x returns r
eal


function double takes real x returns real
    return x*2.0
endfunction
```

```
function triple takes real x returns real
    return x*2.0
endfunction

function Test1 takes real x, Arealfunction F returns re
al
    return F.evaluate(F.evaluate(x)*F.evaluate(x))
endfunction

function Test2 takes nothing returns nothing
    local Arealfunction fun = Arealfunction.double //获
取函数指针的语法

    call BJDebugMsg( R2S(  Test1(1.2, fun) ))

    call BJDebugMsg( R2S(  Test1(1.2, Arealfunction.tri
ple ) )) //也可以这样...
endfunction
```

在此示例中，我们实际上具有作为函数变量的参数。 您还可以将函数的指针类型进行转换（请参见下面的内容），使其指向整数，然后返回其起源的接口函数。

也可以在不键入接口名称的情况下获得函数指针，请注意，这将不允许您按照接口的声明来验证函数，但此方法更简单：

```
function double takes real x returns real
    return 2*x
endfunction

function square takes real x returns real
    return x*x
endfunction

function interface realfunc takes real x returns real

//在实数变量上重复三次相同函数的调用!
function repeater3 takes real x, realfunc F returns rea
l
    set x=F.evaluate(x)
    set x=F.evaluate(x)
```

```
    set x=F.evaluate(x)
    return x
endfunction

function test takes nothing returns nothing
 local real x = repeater3( 2.0, double) //注意,我们只是在使用函数的名称，就好像它们是值一样。
 local real y = repeater3( 2.0, square)

   //结果是 x = 16 和 y = 256。
   //说明：第一个等效于：
   //    set x=2
   //    set x=2*x
   //    set x=2*x
   //    set x=2*x

   //第二个是：
   //    set y=2
   //    set y=y*y
   //    set y=y*y
   //    set y=y*y
```

```
 // 是的，函数接口使您可以像使用其他变量一样使用函数。
endfunction
```

Typecast 类型转换

目前，将结构类型的值分配给整数变量/或任何其他结构类型的变量，编译器将更改该引用的类型。

请注意，有时使用变量可能会变得乏味，甚至会产生不必要的开销，这就是添加类型强制转换操作符的原因。

它的语法看起来类似于一个函数，但是函数的名称其实是自定义类型的名称。不久的将来也会支持本地类型。

```
interface wack
    //...一些声明
endinterface

struct wek extends wack
   integer x
    //...更多的声明
endstruct

function test takes wack W returns nothing
    //您确定 W 属于 wek 类型，可以通过以下方式强制转换该值：
    local wek jo = W
```

```
    set jo.x= 5 //搞定

    //但是有时候创建变量对于虚拟机来说是太多的工作，您只需要访问一次

    set wek(W).x=5 //这样也可以
endfunction

type anarrayofdata extends integer array [6]

function getdata5 takes unit u returns integer
    //对类型为 anarrayofdata 的对象的引用已保存的单位自定义值
    return anarrayofdata(GetUnitUserData(u))[5]
endfunction

function setdata5 takes unit u, anarrayofdata x returns
 nothing
    // 在这里我们做相反的事情，请注意，integer 可以用作转换结构和动态数组类型的运算符。
    call SetUnitUserData(u,  integer(x))
endfunction
```

六. 将方法作为对象

方法可以用作为.execute()/.evaluate()函数的对象，也可以用作允许访问名称字段的对象。该.name 字段将返回编译后赋予方法的函数名称。

如果要在基于 ExecuteFunc 的系统上使用结构的静态方法，此功能特别有用。

```
struct mystruct
    static method mymethod takes nothing returns nothin
g
        call BJDebugMsg("this works")
    endmethod
endstruct

function myfunction takes nothing returns nothing
    call ExecuteFunc(mystruct.mymethod.name) //兼容
ExecuteFunc
    call OnAbilityCast('A000',mystruct.mymethod.name)
    //例如，施法者系统的 OnAbilityCast 需要一个函数名称
endfunction
```

方法是否存在

方法可以使用的另一个字段是 exist 字段，它是一个布尔字段，如果方法已经被声明，则返回 true，否则返回 false。 在大多数情况下，它是正常运作的，唯一可能错误的情况是，该结构正在继承接口方法的默认值。

```
interface myInterface
    method myMethod1 takes nothing returns nothing
    method myMethod2 takes nothing returns nothing
endinterface

struct myStruct
    method myMethod1 takes nothing returns nothing
        call BJDebugMsg("er")
    endmethod
endstruct

function test takes nothing returns nothing
    local myInterface mi = myStruct.create()

    if( mi.myMethod1.exists) then
```

```
        call BJDebugMsg("Yes")
    else
        call BJDebugMsg("No")
    endif
    if( mi.myMethod2.exists) then
        call BJDebugMsg("Yes")
    else
        call BJDebugMsg("No")
    endif
    /*
    输出:
     yes
     no
    */
endfunction
```

七. 数组结构

有时，您想要一个结构类型的全局数组，只是为了拥有我们大家都非常喜欢的字段语法，它可能比预期的要复杂，例如，您必须手动初始化数组的所有索引……

另一个问题是，当您不想使用.allocate()和.destroy() 时，您希望拥有自己的分配方式。

数组结构是一种小的语法增强功能，等效于结构类型的数组，您将能够为每个索引使用成员，而不必担心.create()。

```
//数组结构很难解释，但通过示例应该容易理解
//声明数组结构的语法
struct playerdata extends array
    integer a
    integer b
    integer c
endstruct

function init takes nothing returns nothing
    local playerdata pd

```

```
    set playerdata[3].a=12  //修改玩家 3 的字段。
    set playerdata[3].b=34  //注意它的行为就像一个全局数组
    set playerdata[3].c=500

    set pd=playerdata[4]
    set pd.a=17     //修改玩家 4 的字段。
    set pd.b=111    //是的，这也是有效的。
    set pd.c=501
endfunction

function updatePlayerStuff takes player p returns nothi
ng
    local integer i=GetPlayerId(p)
    set playerdata[i].a=playerdata[i].b
endfunction
```

数组结构的某些问题：

- 不能声明变量成员的默认值（根据成员的类型，它们将自动为零，null 或 false）
- 不能使用.allocate 或.destroy
- 不能声明 onDestroy（将毫无意义）

- 不能有数组成员。

请注意，默认值和数组成员的问题可能会在下一版本中修复。

请注意，您可以使用操作符声明来覆盖 get []操作符，在这种情况下，为了能够使用 ID，您可以使用 typecast 操作符，例如 playerdata（4）来获取实例。如果您不理解最后一段，请不要担心，您可能根本不需要知道这一点。

八. 键值

键值 key 是一种特殊的 vJass 数据类型，旨在生成一个唯一的整数常量，它主要用于生成魔兽争霸 3 哈希表句柄类型的密钥。

每当使用 key 类型声明变量时，都会为其分配一个唯一的整数。如果需要，可以添加 constant 关键字以提高可读性。

```
scope Tester initializer test


    globals
        key AAAA
        private  key BBBB    //是的，这只是另一种数据类型，因此您可以
        public   key CCCC    //申明为公共或私有的...
        constant key DDDD    //也可以正确地将其描述为常量 (非必须)
    endglobals


    private function test takes nothing returns nothing
        local hashtable ht = InitHashtable()
        call SaveInteger(ht, AAAA, BBBB, 5)
```

```
        call SaveInteger(ht, AAAA, CCCC, 7)
        call SaveReal(ht, AAAA, DDDD, LoadInteger(ht,AA
AA, BBBB) * 0.05 )
        call BJDebugMsg( R2S( LoadReal(ht,AAAA,DDDD) )
)
        call BJDebugMsg( I2S(BBBB) ) //将显示两个数字，并
且
        call BJDebugMsg( I2S(CCCC) ) //数字会有所不同...
    endfunction

endscope
```

九. 储存增强

介绍

由于 Jass 在数组大小方面有其局限性，这个局限性也会直接影响结构实例的上限，因此，vJass 也会在很多方面受其影响。

有一种方法：可以将多个 Jass 数组组合在一起，用以虚拟地增加数组的上限。

通过使用存储增强语法，就可以做到这一点。但是也会做出许多的牺牲：

- 通常只需要数组查找的操作将需要调用函数，在 Jass 中，与数组查找相比，函数调用的速度非常慢。
- 该函数本身需要执行一些额外的操作，这些函数执行的操作次数取决于（index_limit / 8191）。
- 如果在很多对象上使用非常大的索引上限，则脚本的大小会大大增加，例如您的结构具有 20 个字段并且使用的索引上限为 60000，则编译后的脚本将需要 40 个新函数，每个函数可能都比 16 行要多一点。

某些 Jass 应用程序将需要更多空间，这意味着您必须使用储存增强以防万一。

如果您实际上并不需要更多空间，由于上述缺点，建议不要使用这些增强。

但您可能需要也可能不需要，可以使增强功能的使用作为可选，因为如果使用大小增强程序的语法指定的数组实际不大于 8191（或者在结构的情况下为 8190）将不会发生效用。

储存增强可以提供大约 408000 的数组空间索引上限，如果需要更多空间，请提出要求，但请说明清楚你要做什么需要如此大的空间……我很有兴趣了解一下……

数组大小

全局数组变量可能有时候需要更多的索引空间，jasshelper 引入了数组大小的语法，它有两个用途：它将允许您请求更多的空间，并且还允许在全局数组上放置一个.size 字段。

```
globals
    integer array myArray [500]
endglobals
function test takes nothing returns nothing
    local integer i=0

    call BJDebugMsg(I2S(myArray.size)) //输出 500

    loop
        exitwhen i>=myArray.size
        set myArray[i]=i
        set i=i+1
    endloop
endfunction
```

当然，您也可以绕开数组 8191 的大小限制：

```
globals
    integer array myArray [9000]
endglobals
```

您也可以使用常量:

```
globals
    constant integer Q= 60000
    integer array myArray [Q]
endglobals
```

您页可以在结构的静态数组成员上使用它。

```
static integer A[10000]
```

二维数组

数组大小改进的好处就是又拥有了二维数组的能力，n 维数组尚未实现，如果您真的非常需要它，请与我联系。

由于 vJass 是在 Jass 之上实现的，因此 vJass 中的二维数组只是变相的普通数组，它使用乘法技巧将二维索引转换为一维索引。 声明这些数组的方法是：<type> 数组名 [width] [height]，请注意，数组的实际大小为 width * height，该大小与普通数组的大小存在相同的限制，不能超过 40800，并且如果此大小大于 8191，则将使用较慢的函数调用。

字段 size 将返回我刚说的实际大小，字段 height 和 width 返回我们声明的数组大小。与之前一样，需要用常量来定义 height 和 width。

```
globals
    integer array mat1 [10][20]

    constant integer W=100
    constant integer H=200
    integer array mat2 [W][H]

endglobals
```

```
function test takes nothing returns nothing
    local integer i=0
    local integer j=0
    local integer c=0
    call BJDebugMsg(I2S(mat1.size)) //显示 200 (10 * 20)

    //填充数组:
    loop
        exitwhen (i==mat1.width)
            set j=0
            loop
                exitwhen (j==mat1.height)
                set c=c+1
                set mat1[i][j]=c
                set j=j+1
            endloop
            set i=i+1
        endloop
        call BJDebugMsg( I2S( mat1[0][1]  )  ) //显示 2
        call BJDebugMsg( I2S( mat2.width) ) //显示 100
   endfunction
```

具有更多索引空间的结构

我们得到了一个结构 X：

```
struct X
    integer a
    integer b
endstruct
```

由于某些原因，对于我们来说 8190 个实例上限还不够，我们需要 10000 个实例！所以：

```
struct X[10000]
    integer a
    integer b
endstruct
```

不要与实例上限的改进相混淆，它只是对索引空间的改进，除非包含数组成员，否则这两个术语通常是等效的：

```
struct X[10000]
    integer a[2]
    integer b[5]
endstruct
```

此结构的实例上限为 2000

您不能在继承其他结构或接口的结构上使用索引空间增强。

```
struct X[10000] extends Y //不好
    integer a[2]
    integer b[5]
endstruct
interface A[20000] //可以
    method a takes nothing returns nothing
endinterface
struct B extends A
    method a takes nothing returns nothing
        call BJDebugMsg("...")
    endmethod
endstruct
struct C[20000] //可以
    integer x
endstruct
struct D extends C
    integer y
endstruct
```

请注意，A，B，C 和 D 的索引上限为 20000。

对于数组结构，这有点不同，因为如您所见，您不能使用[]存储大小说明符，并且不能同时继承。 从 0.9.E.1 开始，在 array 关键字后指定大小可以使用具有增强存储功能的数组结构：

```
struct aBigOne extends array [ 20000]
   integer a
   integer b
   integer cendstruct
function meh takes nothing returns nothing
   set aBigOne[19990].a = 12
endfunction
```

具有更多索引空间的动态数组

动态数组已经使用[]指定每个实例的大小，但是如果要指定最大存储空间怎么办？ 我被迫添加一个逗号：

```
type myDyArray extends integer array [200]
//大小为 200 的普通动态数组类型，最多 40 个实例

type myDyArray extends integer array [200,40000]
```

```
//增强型动态数组类型，数组大小 200，最多 200 个实例
```

十. Jass 语法扩展

冒号

这是一个新的操作符，基本上允许您以不同的方式使用[]，将其称为反向[]。有时，脚本要求的逻辑顺序不同，可能会有有一定的意义。

```
function test takes nothing returns nothing
    local integer a=3
    local integer array X

    set X[a]=10 //这两个语句都执行相同的操作
    set a:X =10

    set X[a] = X[a] + 10 //一样.
    set a:X = a:X +10

    set X[3]=1000
    set 3:X =1000 //这是无效的语法，抱歉，仅能在变量之类的东西上使用。
endfunction
```

换行注释

这些是典型的 /* … */ 注释，可用于注释掉不一定以换行符结尾的代码块。 然后，这些注释会从地图脚本中删除。您还可以嵌套这些注释并做一些有趣的事情...

```
/* 换行注释示例
它们比正常的注释有用
真的!
 */
function test takes nothing returns nothing
    call Something( /*5*/ 66) /*我们暂时将 5 注释掉，然后将其替换为 66 */

    /*
    call Something( /*5*/ 66)
    */

    // 那里的注释包含另一个换行的注释... /*
    call BJDebugMsg("Notice how the previous comment st
art was ignored" + /*
    */+"because it was inside a 'normal' comment "+/*
    */+"Also notice how we made the parser skipped the
```

```
previous "+/*
    */+"line breaks because they were inside a comment"
+/*
    */"These comments do not count if they are /*在字符
串内*/ ... ")

endfunction
```

十一. 文本宏

让我们感受一下，有时候我们想要将非常复杂的内容添加到 Jass 中，但实际上，我们真正需要的只是自动的文本复制+粘贴+替换。

因此 vJass 支持文本宏功能，在很多情况下都非常有用。

语法很简单, //! textmacro 宏名称 [takes 自变量 1, 自变量 2, ..., 自变量 n] 然后 //! endtextmacro 结束. 还有 runtextmacro 开始运行宏。一个例子之后更容易理解：

```
    //! textmacro Increase takes TYPEWORD
    function IncreaseStored$TYPEWORD$ takes gamecache g
```

```
, string m, string l returns nothing
        call Store$TYPEWORD$(g,m,l,GetStored$TYPEWORD$(
g,m,l)+1)
    endfunction
    //! endtextmacro

    //! runtextmacro Increase("Integer")
    //! runtextmacro Increase("Real")
```

该示例的结果是：

```
    function IncreaseStoredInteger takes gamecache g, s
tring m, string l returns nothing
        call StoreInteger(g,m,l,GetStoredInteger(g,m,l)
+1)
    endfunction
    function IncreaseStoredReal takes gamecache g, stri
ng m, string l returns nothing
        call StoreReal(g,m,l,GetStoredReal(g,m,l)+1)
    endfunction
```

需要$$分隔符，因为替换标记可能需要与其他符号一起使用。

请注意，字符串和注释不受文本替换的保护。 因此，如果匹配任何以$$分隔的参数，将始终被替换。

textmacros 可以不需要参数，在这种情况下，您只需删除 Takes 关键字即可。

```
//! textmacro bye
call BJDebugMsg("1")
call BJDebugMsg("2")
call BJDebugMsg("3")
//! endtextmacro

function test takes nothing returns nothing
    //! runtextmacro bye()
    //! runtextmacro bye()
endfunction
```

文本宏仅为该语言增加了很多虚拟的功能性用法，如以下典型的用例：

```
    //! textmacro GetSetHandle takes TYPE, TYPENAME
        function GetHandle$TYPENAME$ takes handle h, st
ring k returns $TYPE$
            return GetStoredInteger(udg_handlevars, I2S
(H2I(h)), k)
            return null
        endfunction
        function SetHandle$TYPENAME$ takes handle h, st
ring k, $TYPE$ v returns nothing
            call StoredInteger(udg_handlevars,I2S(H2I(h
)),k, H2I(v))
        endfunction
    //! endtextmacro

    //! runtextmacro GetSetHandle("unit","Unit")
    //! runtextmacro GetSetHandle("location","Loc")
    //! runtextmacro GetSetHandle("item","Item")
```

开发上的时间已大大减少

文本宏和域/库可以通过某种静态面向对象的方式成为最好的朋友：

```
    //! textmacro STACK takes NAME, TYPE, TYPE2STRING
        scope $NAME$
            globals
                private $TYPE$ array V
                private integer N=0
            endglobals
            public function push takes $TYPE$ val retur
ns nothing
                set V[N]=val
                set N=N+1
            endfunction

            public function pop takes nothing returns $
TYPE$
                set N=N-1
```

```
                return V[N]
            endfunction


            public function print takes nothing returns
 nothing
             local integer a=N-1
                call BJDebugMsg("Contents of $TYPE$ sta
ck $NAME$:")
                loop
                    exitwhen a<0
                    call BJDebugMsg(" "+$TYPE2STRING$(V
[a]))
                    set a=a-1
                endloop
            endfunction
        endscope
    //! endtextmacro


    //! runtextmacro STACK("StackA","integer","I2S")
    //! runtextmacro STACK("StackB","integer","I2S")
    //! runtextmacro STACK("StackC","string","")
```

```
function Test takes nothing returns nothing
    call StackA_push(4)
    call StackA_push(5)
    call StackB_push(StackA_pop())
    call StackA_push(7)
    call StackA_print()
    call StackB_print()
    call StackC_push("A")
    call StackC_push("B")
    call StackC_push("C")
    call StackC_print()
endfunction
```

提示 1:您可以使用类似于 library_once 的方式使用 textmacro_once。

提示 2：如果：//! runtextmacro optional textmacroname(args)，如果 textmacro 不存在，则 textmacro 行将不会引起语法错误。

十二. 钩子

在大多数情况下，有些函数我们是无法控制的，例如本地接口函数和 blizzard.j 中的函数。 “hook”关键字能够使我们追踪并且检测到它们。

使用 hook，native / bj 函数的名称、函数或静态方法的名称，您将能够追踪到何时调用了该函数并捕获为其提供的参数。

```
function onRemoval takes unit u returns nothing
    call BJDebugMsg("unit is being removed!")
endfunction

struct err
    static method onrem takes unit u returns nothing
       call BJDebugMsg("This also knows that a unit is being removed!")
    endmethod
endstruct

hook RemoveUnit onRemoval
hook RemoveUnit err.onrem //正常运作
```

在某个地图的测试代码中，，查看下调用 RemoveUnit 时会发生什么。

注意：现在有一些限制，如果某个 bj 函数调用了 native / bj 函数时，钩子将不起作用。

十三. 注入

某些高级用户可能在使用世界编辑器时，希望对地图脚本进行更多控制，即制作自己的主函数或配置函数，在注入时，预处理器允许替换此类函数。

语法是：//! inject main/config (...)　//! Endinject

例子：

```
//! inject main
    //一些函数调用可能会在这里
    //将 vJass 初始化放置在此处，注意，结构优先被初始化，然后是库初始化
    //! dovJassinit

    //其他的调用可能会在这里

    call InitCustomTriggers()
    //也许您想使用 WorldEditor 的该功能...
//! endinject
```

dovJassinit 预处理器可能非常有用，只有在自定义 main 中没有对 InitBlizzard 调用或需要控制结构和库的初始化位置时才有必要。

//! inject config 工作方式基本相同，只是对于这种情况没有//! dovJassinit。

十四. 从 SLK 文件加载结构

可以从 slk 加载，并转换为要添加到地图的脚本代码，特别是加载给结构。当系统使用结构来存储数据时，会非常有用，因为 slk 是表格式，它可以节省不少的工作量。

加载 slk 文件的预处理器是//! loaddata "path.slk"。文件路径参数遵循导入外部脚本文件的规则。

SLK 文件

SLK 文件要求在（第 1 行，第 1 列）处具有结构名称，然后得在第一行包含其字段的名称，接下来的行包含一个键值[key]和在那里指定字段名称的值。

stname	this	is	just	an	example
1	2	3	4	5	6
7	8	9	10	11	12

在此示例中，要加载的结构类型名称为 stname，将要加载的实例键值为 1 和 7，其余为有关字段和值的信息。

结构类型

要使用的结构类型（在示例中为 stname）需要一个 getFromKey 的静态方法，该方法返回该结构类型的值。

getFromKey（）会简单地转换一个键值并返回一个等效的结构，这是因为希望数据结构与其他东西相关。（大多数情况下，[其他东西]会是对象 ID。）

对于此示例，getFromKey 必须使用一个整数参数。

结构类型还需要在 slk 文件中声明字段，slk 不会强制列出结构类型的所有字段。

因此，我们需要定义以下结构：

```
struct stname
    integer this
    integer is
    integer just
    integer an
    integer example
```

```
    static stname array values


    static method getFromKey takes integer i returns no
thing
        if (stname.values[i]!=0) then
            set values[i]=stname.create()
        endif
        return stname.values[i]
    endmethod
endstruct
```

我们从上一个示例加载了 slk，加载的初始化代码实际为：

```
    set s=stname.getFromKey(1)
    set s.this=2
    set s.is=3
    set s.just=4
    set s.an=5
    set s.example=6
    set s=stname.getFromKey(7)
    set s.this=8
    set s.is=9
```

```
    set s.just=10

    set s.an=11

    set s.example=12
```

注意，此代码在结构和库初始化之后运行

有关更实际的解释，请查看 JassHelper 发行版中包含的 slk 演示。

十五. 代码的调试

Jass 包含一个 debug 关键字，该关键字可以被正确编译，但会使其后的代码被忽略。 似乎此关键字用于调试目的。

现在，我们可以利用此隐藏的 Jass 功能。 JassHelper 在调试模式下保存时，只会删除 debug 关键字，从而使其后的代码内容生效，如果未启用调试模式，JassHelper 会删除以 debug 开头的行。

```
function Something takes integer a returns nothing
    debug call BJDebugMsg("a is "+I2S(a))
    call KillNUnits(a)
endfunction
```

如果我们在以调试模式保存，则每次调用此函数时都会看到“ a is value”，否则它仅以静默方式调用 KillNUnits。

您也可以使用常量布尔值 DEBUG_MODE，该值设置为 true 或 false 取决于是否启用调试模式。

在脚本中加入#define DEBUG_MODE true 可以手动开启调试模式。

或者使用#define BJDebugMsg(s) DoNothing() 将所有BJDebugMsg 调试打印函数的内容替换成 DoNothing() 。

十六. JassHelper 功能

避免局部变量重影

从版本 0.9.K.0 开始，局部变量重影就是 vJass 语法解决方案的一部分，不像 Jass 语法，始终存在着问题。在 1.24 补丁之前，甚至不能重影超过一个变量。即局部变量名和全局变量名相同……

这些问题显然已在 1.24 补丁中修复，但是依旧没有达到可以帮助 GUI 用户利用局部变量技巧的级别。问题仍然存在，并且这类问题最终可能会在脚本中无法预测的地方导致某些问题。

如果正确使用了 vJass 代码并且运用的域范围也正确，那么应该不会出问题。 但是，当从多个域中调用代码时，其中一个代码运用的域可能就不正确了，这就有可能导致与局部变量发生冲突，从而使您的地图在游戏中神秘地崩溃。 Jass 对变量重影的缺陷本质上是不可预测的。 因此，我决定在 JassHelper 中添加一个小的预处理片段，以修正局部变量重影。 这意味着函数中的局部变量现在可以与声明的任何全局变量（甚至是您不知道的全局变量）具有相同的名称，而不会引起进一步的问题或是未知的、不可预测的错误。 尽管 vJass 代码允许变量重影，但仍将其编译为没有重影的 Jass 代码（会添加一些 l__前缀到有冲突的局部变量）。

return bug 修复程序

注意：默认情况下，新功能是禁用的，因为问题已由 1.24b 补丁修复。

1.24 补丁给 Jass 使用者们带来了毁灭性的打击，这要归功于一些新的暴雪 bug，带有多个 return 语句的函数语法会发生错误或者在运行中游戏直接崩溃，从而有效地使无辜的地图也无法运行，即使它们并没有使用臭名昭著的“Return Bug”。 这就是原因，我必须为 JassHelper 添加一个快速修复程序，该编译器 mod 会将具有多个返回值的函数替换为两个单独的函数，这样可以避免误报。

副作用和局限性

此修复程序仅是临时措施，当暴雪修复此可怕的错误时，或者在使函数避免出现多个 return 语句的情况下（无论发生什么情况），都是安全的。 该项技术有点愚蠢，对于每个可疑的函数，它都将创建一个新的虚拟对象，该虚拟对象不返回任何内容，并且仅在返回之前分配一个全局变量，然后旧函数实际上将调用此虚拟函数。 简而言之，这意味着通过添加额外的函数调用会使某些函数（特别是具有返回值和多个返回语句的函数）的调用速度稍慢。

此方法不适用于递归函数（因为仍然无法从其声明上方调用函数）。 如果函数是递归的并且具有多个 return 语句，则可能会导致编译器错误。在这种情况下，最好修改该函数，因为它很有可能也是 return bug（它很可能导致您的地图无法在 1.24 补丁运行）。 为了防止这种情况，JassHelper 将尝试检测递归，并在可能的情况下进行处理。 但是 JassHelper 目前无法修复某些递归模式。

启用修复程序

必须手动启用此功能，因此，如果由于某种原因需要使地图专门与补丁 1.24 配合使用，或者如果您认为 return bug 仍在影响您的地图，则可以启用该功能。

为了启用返回修复程序，请打开配置文件 JassHelper.conf。

如果有 [noreturnfixer] 标签，则将 [noreturnfixer] 替换为 [doreturnfixer]。

如果 JassHelper.conf 中没有对应标签，添加[doreturnfixer]标签即可。

导入外部脚本文件

JassHelper 还包括像 WEHelper 一样的导入预处理符，它允许您导入外部的文件。

用法是 //! import "scriptfile" ，可以使用固定路径或相对路径。

例如： //! import "c:\x.j" 将在地图脚本插入对应文件的脚本。 它实际上就是在编译文件之前将文件的脚本内容粘贴到那里一样，因此该文件中可以包含全局变量，库，textmacros 等。它甚至也可以在内部使用 //! import

例如：

//! import "x.j"

使用相对路径：

//! import "subfolder\f.j"

当路径相对时，JassHelper 会在地图的文件夹或 JassHelper 的配置中查找文件夹对应的脚本文件。

也可以使用绝对路径：

//! import "d:\f.j"

注意：如果您在同一文件名上导入两次或多次，该命令将被忽略。

Grimoire 版本使用配置文件 JassHelper.conf 来确定查找哪一个文件夹，WEHelper 版本则带有一个对话框，可用于查找文件夹的设置，并且 WEHelper 版本还会自动将 wehelper 的导入路径添加到文件夹查找列表。
WEHelper 的内部导入预处理有可能覆盖 JassHelper 的导入，因为默认情况下是在 JassHelper 之前调用它，如果要使用 JassHelper 的导入，则必须先禁用 WEHelper。

Grimoire 的 jassehlper.conf 已将 grimoire 的 Jass 文件夹确定为导入脚本的位置，您可以在此放入所需的任何文件。

但请注意，这里有一个导入的优先级，如果在地图的文件夹中查找到对应的文件，则无论该文件是否存在于 JassHelper.conf 配置的另一个文件夹中，都将优先导入该文件。

使用地图的文件夹存在一定的问题，因为 testmap 将地图保存在其他路径中。 因此，建议您为此定义绝对路径来进行导入。

当然，最好的解决方法就是在测试之前保存一下地图。

从 JassHelper 0.9.C.0 开始，已经导入的文件所使用的相对路径也将尽可能支持文件的路径。 请注意，与当前文件位于同一文件夹中的文件优先级最高。

Zinc

JassHelper 支持两种脚本语言，即 vJass 和 Zinc。 vJass 是 Jass 的扩展，而 Zinc 是不太冗长的语言，但结构和 Jass 的兼容性不强。

有关 Zinc 语言及其语法的更多信息，请参阅其他的文件，Zinc 手册：

http://www.wc3c.net/vexorian/zincmanual.html

编译忽略

//! novJass 和 //! endnovJass 预处理符可以使JassHelper完全忽略它们之间的代码。

```
function VerifyvJass takes nothing returns nothing
    local boolean b=true
    //! novJass
        set b=false
    //! endnovJass
    if(b) then
        call BJDebugMsg("You got vJass")
    else
        call BJDebugMsg("Where's vJass?")
    endif
endfunction
```

注意：

如果该代码由 vJass 编译器解析，它将删除 //! novJass 内部的内容！

如果该函数只是保存在普通的 World Editor 中，则它将忽略 //! novJass 标签（因为它会认为是注释），因此仍会使用其中内容。

脚本优化

由于 JassHelper 中提供了 0.9.A.0 版本的脚本优化功能，因此当前默认情况下处于启用状态。要禁用优化功能，您可以启用调试模式或使用 --nooptimize 参数（JassHelper.exe）。

目前，唯一可用的优化是函数内联。后续将添加更多的优化，包括 wc3mapoptimizer 的某些改进版本。

函数内联

函数内联将查找可以内联的函数调用，然后将其调用转换为直接使用函数内容。 为了不破坏地图的正常运行，仅在少数情况下才执行此操作。内联的示例如下：

```
function MyFunction takes integer a, integer b returns integer
    return myarray[a]*b
endfunction
function MyOtherFunction takes nothing returns integer
    return MyFunction(3,4)
endfunction
```

```
//变成:
function MyOtherFunction takes nothing returns integer
    return myarray[3]*4
endfunction
```

内联很重要，因为它可以减少函数调用的次数，并使地图脚本的某些部分更快，同时，它还使您可以编写可读性更高的代码。

外部工具

JassHelper 允许//! 外部预处理器，使用 JassHelper 编译地图后，您可以在地图上调用其他工具，因此这些工具可以与 grimoire 和 WEHelper 一起使用

预处理器是//! external EXTERNAL_TOOL_NAME [外部参数]

EXTERNAL_TOOL_NAME 必须与配置中的工具名称匹配（wehelper 插件中的对话框或 grimoire 版本的.conf 文件）

外部名称后面的文本是可选的，并作为命令行参数提供给工具。

如果要在 stdin 中指定要发送到外部工具的输入，也可以使用 externalblock 预处理器：

```
//! externalblock EXTERNALNAME ARGUMENTS LIST
//! i These lines will get send to
//! i The tool as stdin
//! i The i and the space after the i are ignored.

    //注释和空格不要出现在 stdin 中
```

```
//! i

//! i That one up there was just an empty line to send to the program

//! endexternalblock
```

接下来是程序员的一部分：

什么样的工具？

为了使工具与外部预处理器正确的进行配合，它必须具有非常特定的行为。

首先，JassHelper 会将这些参数传递给工具

- 地图文件所在路径。
- 目录路径的标记化链（c:/; d:/ somepath; c:/ anotherlocation/）这些是 JassHelper 使用的查找路径，为了解决某些问题，\字符用/代替。
- 外部预处理器中的其余行。（该工具可能有也可能没有更多参数）

然后，如果成功，该工具必须使用 0 代码，如果返回不等于 0 的数字，则会向 stdin 和 stderr 写入一条错误消息。

换行修复

作为此项目所需的解析的副作用，它还将 Jass 字符串文字内的换行符替换为正确转义的\ n，这还解决了 PJass 在存在换行符的字符串时给出错误行号的问题。

命令行

如果将 JassHelper.exe 与 sfmpq.dll 和 pjass 结合使用，则可以在没有编辑器黑客的帮助下编译地图，如果您使用的是 Linux（WINE 允许您使用 WorldEditor 和 JassHelper，但不能使用 grimoire），则可能需要了解以下。

基本命令行语法为：

```
JassHelper.exe <path_to_common.j> <path_to_blizzard.j> <path_to_map.w3x>
```

这将使 JassHelper 处理源地图，并使用新编译的脚本更新地图。 您可以从 war3patch.mpq 中的 scripts 文件夹中提取 common.j 和 blizzard.j。

如果将四个文件参数传递给程序，而不是三个参数，则行为将发生变化：

```
JassHelper.exe <path_to_common.j> <path_to_blizzard.j> <path_to_mapscript.j> <path_to_map.w3x>
```

它将忽略地图的脚本文件，而是将给定的脚本文件视为 vJass 源。 由于这 3 个文件的语法从地图中删除了原始 vJass 源代码，因此此方法更加有用，您可以通过从编辑器中导出地图脚本来生成源地图。 （**提示：**

结合使用//! import 和//! novJass 到 JassHelper 和 World Editor 的命令行)

当然，这还不够灵活，因此 JassHelper 支持可以在 blizzard.j 路径之前放置的几个命令选项：

- **--debug**：此标志将使 JassHelper 在调试模式下进行编译。 并且还会开启**--nooptimize**。
- **--nopreprocessor**：如果由于某种原因您只想检查普通的 Jass 语法，并使用 JassHelper 作为代理来调用 PJass，则可以使用此选项。
- --nooptimize：禁用优化，请参阅脚本优化部分以获取更多信息。
    - **--scriptonly**：注意，它迫使您提供四个文件：此命令更改了下一个参数的行为。

JassHelper.exe --scriptonly <path_to_common.j> <path_to_blizzard.j> <path_to_input.j> <path_to_output.j>

- 此语法不需要提供任何映射，只会评估 path_to_input.j 文件并在必要时显示语法错误。 如果编译成功，则 JassHelper 会将输出脚本 path_to_output.j 写入您提供的文件路径。

- **--warcity** : This setting will automatically turn --scriptonly on, it makes JassHelper evaluate the input script file as if it was a "warcity script file", WarCiTy is a program that converts a map's custom trigger data into a special sort of .j script. This setting will simply make JassHelper evaluate only the //! import and //! novJass preprocessors, it will also prevent the adition of guide comments specifying it just imported the file. There is no syntax checking feature for this mode.
- 此设置将自动打开，它使 JassHelper 像对待“ warcity 脚本文件”一样评估输入脚本文件，WarCiTy 是一个程序，可将地图的自定义触发数据转换为特殊的.j 脚本。此设置将仅使 JassHelper 评估 //! import 导入和 //! novJass 预处理程序。 此模式没有语法检查功能。
- **--zinconly** : 此设置将自动打开 scriptonly，如果它是单个 zinc 文件，则 JassHelper 会评估输入的脚本文件。然后，它将在 zinc 阶段之后输出已编译的 vJass 代码。
- **--macromode**：和 warcity 一样，也可以评估 textmacros.
- **--about**：仅显示“**about**”对话框（您想知道正在使用的 JassHelper 的版本吗？）请注意，它会忽略提供的文件参数。
- **--showerrors**：显示前一个语法错误（不进行编译）。

- **cliJassHelper.exe**：cliJassHelper.exe 的行为与 JassHelper.exe 完全相同，但它不使用 Windows 的 GUI（尽管它仍然是 Windows-WINE 应用程序），有时它可能很有用（例如，如果您想从 ssh 使用 JassHelper ），它只会将内容输出到 stdout，如果由于某种原因 stdout 无法正常工作，它将输出到 work 文件夹中的 stdout.txt。

更新

除非另有说明，否则更新 JassHelper for newgen pack 的是最简单的方法。

卸载

- WEHelper 插件：将 JassHelper.dll 从插件文件夹中移出/删除
- Newgen pack/grimoire:我认为删除 JassHelper 文件夹就会使它卸载。
- From your computer: JassHelper 不使用注册表，因此您只需删除包含它的文件夹。

团队和感谢

- Gold 解析系统可协助进行一些最复杂的解析： http://www.devincook.com/goldparser/
- Pipedream：在语法设计方面提供了大量帮助。并为破解 WE 做出了巨大贡献。
- weaaddar：结构和数组的使用提供了聪明的分配方法。
- Zoxc：制作 WEHelper。
- Vexorian：将他心中所想制作成了该编译器。
- ZergLeb：帮助我修复了一个很邪恶的 bug
- Grim001：Bug 反馈
- Anitarf：让我实现了很多他甚至不用的功能。帮助查错。
- Rising_Dusk：如果没有他，就不会添加委托功能。
- StealthOfKing : Helped fix SLK issues
- StealthOfKing：帮助解决 SLK 的相关问题
- rain9441：发现了结构在 onDestroy 的大量错误。
- Captain Griffen, Here-b-Trollz 等人提供了 BUG 反馈
- Alexander244：由于过多的使用 loaddata，因此生成的函数对于 PJass 而言太大。
- Litany：提供意见。
- Flame_phoenix：为了让我弄清楚二维数组的重要性。
- C2H3NaO2：Bug 反馈

- Av3n：我可以根据他提供的脚本文件修复 BUG
- Zoxc and Deaod：帮助解决一些与 blizzard / common.j 相关的问题。
- MindworX: Bug 反馈
- http://www.wc3c.net 提供相关链接
- 我使用 gvim 生成了大部分文件 http://www.vim.org/
- 适用于 eXecutables 的 Ultimate Packer：版权所有（c）1996-2002 Markus Oberhumer 和 Laszlo Molnar：http://upx.sourceforge.net（老实说，我已经不再使用它了）

更新日志

- 0.A.0.0
  - . or this. are not required anymore to use members. Note that this may cause issues if for some (incredibly weird) reason you try to use global variables from a method of a struct that has variables of the same name. To disable this feature, you can add [noimplicitthis] to JassHelper.conf.
  - Improved the syntax error when you place a function inside a struct.
  - Code values might get implicitly casted to boolexpr in some occasions, specifically, when using them as arguments for natives/bjfunc that take boolexpr. More cases will get added when type safety gets on its way for more stuff...
  - Zinc: Added anonymous functions, but they cannot use locals from their parent (yet).
  - Zinc: Fixed a crash that could happen when the zinc input is much smaller than the vJass output.
  - Zinc: Fixed a couple of missing ; mistakes in the examples.
- 0.9.Z.5

- Added externalblock.
- optional textmacros work.
- static ifs support elseif.
- static ifs support a struct's static constant booleans.
- Missing thens in static ifs are reported as a syntax error.
- Comments inside static ifs are deleted correctly when the condition is false.

- 0.9.Z.4
  - Reversed the deprecation of automatic method TriggerEvaluate, you may enable the syntax error adding [forcemethodevaluate] to JassHelper.conf.
  - Added a syntax error when . and other unsupported operations are used in static ifs.
  - Added a --zinconly command line argument that will just compile a zinc file into vJass code.
  - Fixed a probable error with --macromode removing the first line of code.
  - You may now use the identifier DEBUG_MODE as a constant boolean that is true if and only if debug mode is on.

- 0.9.Z.3
    - Correct operator precedence in structs phase, this change is not noticeable unless you had an overloaded == operator.
    - structs phase's "Syntax Error" message is now slightly more detailed.
    - You may now use a pair of decorative parenthesis in static ifs.
    - custom operators used from above their declarion will once again use .evaluate automatically. (as it is not possible to do it manually).
    - Hopefully fixed issues regarding using deallocate on child structs.
    - Fixed a bug with deallocate requiring evaluate for no reason.
    - Fixed a syntax error regression that happened when calling .destroy from above onDestroy.
    - Zinc: When an if is all that is inside an else's contents, it is translated into elseif.
    - Zinc: Compiler will try its best to keep comments, though it might place them in awkward positions...

- Zinc: Fixed a z.2 regression that made Zinc eat up parenthesis...

- 0.9.Z.2

  - Fixed a crash related to ==.
  - Fixed a bug that for some reason required slks to have more than 2 columns, instead of more than 1...
  - If a SLK value for a boolean member is 0 or 1, JassHelper will convert it into true or false.
  - Added runtextmacro optional .
  - .evaluate() is mandatory on methods if you want to call them from above their declaration

    ```
    to disable this new syntax error, add a [automethodevaluate] option to JassHelper.conf
    ```

  - Zinc: Make grammar allow negating a negation.
  - Zinc: A while's condition is compiled neatly, being able to avoid unnecessary not operators.

- 0.9.Z.1

  - Important: Removed virus that sneaked into some hidden folder inside the source tree.
  - destroy can get replaced inside a struct.

- Added a deallocate method that works like allocate
- Added static versions of the name and name= operators.
- Added == overloading (and != for that matter)
- Zinc: add operator== to grammar.

- 0.9.Z.0
  - Added static ifs
  - Added optional library requirements
  - Added Zinc.

- 0.9.K.0
  - Variable shadowing is now part of correct vJass syntax. Compiler guarantees (or at least should) that there won't be global-local conflicts in the compiled jass code.

- 0.9.j.2
  - Fixed a memory out of bounds error related to some bugged native declaration usage.
  - Fixed issues with undeclared variables and also array member leaks when you use array members on an interface and do not have onDestroy declared on one of its children.

- Fixed a bug with methods called the same as any function not being callable.

- 0.9.j.1
  - Removed returnfixer from the default, you may still toggle it on but it is not necessary anymore.
  - Fixed some crashes in windows with non-cli JassHelper.

- 0.9.J.0
  - JassHelper now comes with a phase that will do its best to fix return bug false positives at the cost of an extra function call. This is meant as a temporary fix while blizzard fixes the bugs caused by patch 1.24 , or you update your functions to avoid multiple return statements. This phase can be disabled through the config file. Check the updated manual for more info.

- 0.9.I.2
  - Fixed a bug with function hooks not working correctly if nothing else related to function interfaces is used by the map.

- Fixed a bug with function hooks not working correctly at all most of the time.
- The manual no longer wrongfully states that the order of execution of onInit methods is undefined (as it turns out it isn't).

- 0.9.I.1
  - Fixed a crash related to extends and array members.
  - Fixed some chance that cliJassHelper would attempt to create window.
  - Fixed a bug with stub keywords causing childless struct not to call onDestroy correctly.
  - Added hooks.
  - It is more likely that methods using evaluate will not use TriggerEvaluate if they only call natives/blizzard.j functions.
- 0.9.I.0
  - JassHelper can now support native declarations around the map script, and move them to the top of the script.
- 0.9.H.3

- Added GetHandleId, StringHash, the gamecache Get and HaveStored natives and the hashtable Load and HaveSaved natives to the list of natives that do not modify the state. This means that functions that directly or indirectly call these natives are more likely to get inlined.
- Fixed a small typo bug inside a comment of sample JassHelper.conf .

- 0.9.H.2
  - Fixed a freeze and out of memory bug with mass storage arrays/structs/dynamic arrays when the storage size was greater than 8191*13.
  - The mass storage arrays/structs/dynamic arrays picker functions will now take slightly less lines of code,

- 0.9.H.1
  - Fixed an off-by one error in the mass size arrays/structs/dynamic array code that caused various issues on boundary cases.
  - JassHelper.conf can now determine the command line arguments given to the Jass syntax checker.
  - Added key

- 0.9.H.0
  - Added block comments.
  - If call InitBlizzard() is not found in the main function, JassHelper will add the initializing code to the end of the function, instead of raising an error
- 0.9.G.3
  - Fixed a crash when there were empty lines on some methods.
  - Fixed a bug that prevented array structs from having static array members.
  - Fixed private delegates/constants not working correctly inside modules (and possibly causing further bugs)
  - Fixed problems related with cliJassHelper not working in windows' cmd.exe.
  - If for some bizare reason, there's no stdout assigned for cliJassHelper, it will automatically send its output to stdout.txt.
  - GetUnitUserData is now considered a non-state changing function by the inliner, which should increase the chances of functions that use it to get inlined.

- 0.9.G.2
  - Fixed a regression introduced in G.0 that caused various issues with function interfaces.
  - Fixed "operators that return self" adding a chance to cause odd syntax errors, stack overflows and access violations.
- 0.9.G.1
  - Modules' private members are now truly private, unlike the other members, they are not visible to the calling struct, and their names will not collide with other names declared in the struct.
- 0.9.G.0
  - Added Modules and thistype.
  - Will now call the optimize phase after PJass, as it was always intended, this should prevent some crashes during optimizations.
  - Functions can now use .name in a similar way to methods.
  - Fixed a bug that made child structs ignore the parent's storage size if a constant was used.

- Fixed a crash when the user attempts to assign a static array member.

- 0.9.F.7
  - $ Is now supported as hexadecimal prefix in integers - turns out wc3 always did.
  - You can now tweak JassHelper.conf to set a different Jass compiler (i.e. change pjass.exe requirement into foojassc.exe).
  - Fixed bugs related with displaying errors in blizzard.j / common.j in an unusual JassHelper setup is unusual like newgen's - several situations in which it would fail to show the correct file in the syntax error window/report have been fixed.
  - cliJassHelper will now specify the script file in which the errors were found.

- 0.9.F.6
  - JassHelper will avoid using TriggerEvaluate when the evaluated function/method does not contain function calls.
  - function interfaces will consider all custom types as integers when comparing for validity, later it will have some

type safety (i.e. you will not be able to use a integer in place of a struct, but you would be able to do the opposite) but for now it is type unsafe, use with care.

- member declarations with odd characters between the type and the name are now reported as syntax errors.
- static 2d arrays used to calculate their storage space incorrectly which caused issues like them not using get/set functions correctly, this has been fixed.
- It is not anymore possible to declare a member variable with the same name as a method operator.
- Fixed a documentation bug in the section about interfaces.

- 0.9.F.5

  - Old OS/X line breaks are now supported in input .j files, this would most likely be useless to everyone unless a bugged text editor saves the file using those...
  - Fixed a crash bug introduced in 0.9.F.4 related with structs that extend interfaces/stub methods and don't *override* the method.

- 0.9.F.4

- stub on a childless struct is not going to cause a syntax error anymore.
- The getType() method can be called on any struct instance.

- 0.9.F.3
  - More stub related bug fixes.
- 0.9.F.2
  - Fixed a bug when using super on methods that had arguments.
  - Fixed a bug with bigarray.size using an undefined type which caused some type comparisons errors.
- 0.9.F.1
  - Fixed some bugs with stub methods on structs that extended interfaces.
  - cliJassHelper?
- 0.9.F.0
  - Fixed the return bug detector, there will not be false possitives, which means more functions will be inlined.
  - Added stub methods.

- Added super.

- 0.9.E.1

  - Fixed issue with operator priority in result code of using 2D arrays.
  - Fixed compile error caused by .execute on static methods with arguments.
  - Fixed an usual chance to incorrectly inline return bug exploiters.
  - Single-line return bug exploiters now recognized as non-state changing functions by the inliner (Increases chance to inline certain functions).
  - dynamic array declarations now report garbage code after the end of the declaration.
  - Fixed a bug with scope initializers making a next library unable to have nested scopes.
  - Array structs can now have a max size specifier after "array".
  - Fixed a readme bug, it incorreclty stated the command line arguments order.

- 0.9.E.0

- Added --macromode.
- Added method.exists.
- Added 2D global arrays (and 2D static array members)
- Extra text after certain array size declaration is not ignored anymore (a correct syntax error now appears)
- Fixed certain readme bugs.

- 0.9.D.3
  - Array structs now work as intended.
  - Fixed yet another regression with method calls.
- 0.9.D.2: -Fixed a [list out of bounds] error when using .method() syntax.
- 0.9.D.1
  - Non-static methods that take no arguments are possible to be called again i.e: .destroy().
  - Delegate cycles will now cause a crypting syntax error, which is better than JassHelper overflowing the stack.
- 0.9.D.0
  - Added delegate.
  - Added functionname for function pointer values.

- Added a way to get the return value of .name= and []= assignment operators.
- Added extends array.

- 0.9.C.1

  - Fixed a problem caused by waits inside library initializers, they prevented other library initializers from running.
  - SLK cells with either - or _ as ignored, whitespace inside these cells is also ignored.
  - Fixed syntax errors caused by international compatibility issues in certain setups.

- 0.9.C.0

  - Fixed a crash that happened if you attempted to assign a method. (set x.create=2)?
  - Fixed a bug that made children struct cause syntax errors if the parent uses extra space.
  - Fixed a conflict between --nopreprocessor and using script arguments (the script used to be ignored)
  - Fixed a conflict between --nopreprocessor and --scriptonly though it would be nonsense to use both simultaneously anyway...

- Inline phase no longer gets confused by control characters in strings.
- Inline phase no longer gets confused by control characters in strings (Both of these were meant to handle those things correctly, but there were small bugs in certain functions that made the provisions fail)
- Added the colon operator.
- Improved the syntax error caused by getFromKey not taking a single argument
- Fixed a bug with the SLK parser that made it unable to parse SLKs with UNIX (normal) linebreaks.
- Certain loaddata syntax errors will now also point to the location of the related struct/member.
- Cells with - are now ignored (that would have created bad syntax anyway).
- Compatibility with yet another odd quote sequence used by openoffice in SLKs.
- Given how absurdly hard and tool-dependent it is to use " in a SLK cell, JassHelper will now automatically add "" to cells in columns for string-typed cells that don't have them.

(unless it is - or an empty cell, in which case it will use the default value, which is probably "" anyway)

- Similarly, If a column's field is of integer type, it will add '' automatically to non-integer values of length 4 or 1 inside cells.
- Code generated by loaddata is now split in function batches of 100 loaded structs each (each using its own thread), long functions cause PJass errors and might also cause thread crashes...
- If you use //! import from an imported file, you are able to use relative file paths based on the importing file's location.
- Backwards incompatibility: I hope no one was using variables in SLK cells. It should still work if the type is not string or integer, if the type is integer, variables would still work provided their name length is not 4 or 1.

- 0.9.B.1

  - Once again struct members can be initialized.

- 0.9.B.0

  - Fixed a bug with the first JassHelper phase which used to cut the last line of an input file potentially causing

problems if WE decides not to print a last empty line (which apparently happens sometimes).

- Fixed a chance for access violation after the structs phase finishes.
- JassHelper no longer ignores extra code after a struct member declaration (It now shows an error).
- JassHelper no longer ignores extra code after a local variable declaration (It now shows an error).
- Structs now allow sized static array members.
- Added //! novJass and //! endnovJass.
- Added --scriptonly and --warcity
- global variable addition order is now affected by library requirements.
- Fixed a couple of "&aquot;" bugs in the readme file.
- The readme file comes with description about JassHelper's command line options.

- 0.9.A.0

  - Added inline phase and --nooptimize

- 0.9.9.B

- Fixed a bug that would cause syntax errors when two or more scopes got initializers.
- Maximum extended array size limit increased to 409550.
- Dynamic arrays allow sizes smaller than array's storage limit / 8.

- 0.9.9.A
  - Fixed yet another bug that prevented f__arg_this from being created.
  - Scopes now use normal calls instead of ExecuteFunc for initializers.
  - Updated readme, scope initializers are mentioned, made it aware //! for libraries and scopes now cause a syntax error.

- 0.9.9.9
  - Hopefully fixed onDestroy issues with extends and similar.
  - Fixed memory corruption when using [] on structs/interfaces to increase index limit.
  - Added big sized global arrays.
  - Library declarations more likely to survive comments.
  - Fixed hang outs related to wrong use of extends.

- Fixed bug with defaults on methods that return custom types.
- Scopes can have initializers.

- 0.9.9.8 (test release)
  - Fixed a bug with f__arg_this not being declared when required if extends is involved.
  - Fixed a crash related to misplaced endscope inside a library.
  - Fixed a bug with array members in child structs, possibly "leaking", which means they did not get recycled properly and the struct would eventually malfunction.
  - Can declare methods to replace variable access and write operators (method operator name and method operator name=)
  - "member is private" syntax error now also shows the name of the involved struct.
  - Access violations will report a related line of code if possible
  - Can setup max index space on dynamic arrays and structs (type arrayname extends typename array [instancesize,

spacerequired])(struct name[spacerequired])(interface name[spacerequired]

- In order for last feature to work I have to modiffy plenty of things, expect an unstable version, releasing it so I could get free testing...

- 0.9.9.7

  - Fixed a bug with methods, pseudo inheritance and interfaces that is just too hard to explain.
  - Fixed a bug with third generation (and above) child structs and array members.
  - Scopes and libraries may now use numbers in their names (Still not _).
  - External command line length limit extended to 1000.

- 0.9.9.6

  - Fixed a bug with onDestroy if it contains calls to methods from other structs and is used on an struct extending another struct.
  - [] and []= operators can also be declared as static.
  - Order of addition of libraries that don't extend each others is now sorted by name.

- empty interfaces or onDestroy methods are assigned to null rather than not assigning their arrays, it probably was all right but this sounds better.

- 0.9.9.5

  - Fixed more bugs related to onDestroy and extends.
  - Fixed various bugs relating to syntax errors and library declarations.
  - Fixed a certain line off-set present for syntax errors after files are imported.
  - Fixed a readme bug with the changelog.

- 0.9.9.4

  - Fixed a bug with static methods that return custom types and require evaluate.
  - Fixed plenty of bugs related to "running" undeclared textmacros.
  - Fixed plenty of bugs related to onDestroy methods when inheritance or interfaces are involved.
  - Fixed an access violation crash when the [] operator is used on methods

- Fixed probable (local-not-set-to-null) memory leak with function interfaces, methods called from above their declaration, evaluate, execute and function interfaces
- Long external calls will now popup an error instead of crashing JassHelper.
- Added onInit method support for structs.

- 0.9.9.3

  - Fixed a bug with return statements in interface functions that return nothing
  - Fixed a crash with --nopreprocessor that introduced a long ago.

- 0.9.9.2

  - Fixed a major bug causing desyncs (All functions generated by JassHelper that are passed to Condition() will have a boolean return value)
  - struct.typeid now returns an accurate integer not dependant on parent ids and is replaced by a constant added to the map script instead of a single number.

- 0.9.9.1

- Fixed a bug with static methods that return custom types and required evaluate mode, causing some pjass parse errors.
- Structs may now extend other structs.
- Interfaces are now allowed to declare a rule so that constructors of the interface's children do not declare a create method that takes arguments.
- interface.create() will now return 0 when there is attempt to call the private allocate() method.

- 0.9.9.0
  - Fixed a critical error causing infinite loops when there were external commands used for grimoire version.
  - Improved Wine compatibility again, should work with more versions of Wine including the newest.
  - Grimoire version's compile error window used to have a chance to look akward under certain windows UI settings, this problem is fixed.

- 0.9.8.9
  - Documented inject.

- Added information about problems related to sync natives and methods/evaluate().
- Improved the error message given if InitBlizzard() is not present in the main function.
- New command line option to use an external war3map.j instead of the one found in the map.
- Improved compatibility with WINE, at least on the recent WINE versions I have tested JassHelper.exe and it can compile a map fine.
- Added some syntax errors for maluse of dynamic arrays or array members
- Fixed issues with commented-out/incomplete return statements inside certain methods or functions.
- private or public are ignored when using on scope declarations, used to silently cause other bugs before.
- Made usage of public and private more strict to prevent bugs (Instead there would be syntax errors)
- Added: keyword
- [scope symbol redeclared] syntax error will now also specify the first declaration of the symbol.

- Made certain statements more strict, some used to allow extra text after the statement (scope or globals for example)
- Made a syntax error related to libraries more understandable.
- Struct member initializers may now use . syntax (it used not to parse them)
- Added a .name field for static methods, returns the string of the generated function name.
- interfaces can use .create if given a typeid value.

- 0.9.8.8

  - Fixed plenty of issues with parsing of runtextmacros and handling of syntax errors in textmacros.
  - Child structs may override the initia default values of members of the interface.
  - structs allocation needs one less array and is a littler faster.
  - Improved error message given when someone makes an attempt to make an struct that extends an struct.
  - Can use .execute() on methods.

- 0.9.8.7

- Fixed a bug that caused issues with functions that returned custom types.

- 0.9.8.6
  - Fixed a bug with scope private/public members used inside an struct.
  - Optimized performance of struct stage.
  - Compiler will now prevent name conflicts and rais errors if it finds them, these prevent bugs later but it is possible that an old name conflict in your map survived for a lot of time and this new JassHelper version will popup a previously unheard error for that map.
  - Undeclared InitTrig functions are ignored instead of poping syntax errors, this allow for cleaner usage of the trigger editor with libraries.

- 0.9.8.5
  - Added defaults keyword for interface methods.
  - Fixed some bugs with the installer for WEHelper, should be easier to install to 1.8 although you still have to specify the path.

- Fixed a bug that made JassHelper unable to recognize hex integers if they had lower case letters.
- Formatted the manual a little.

- 0.9.8.4

  - Fixed a bug with interface extending structs with multiple array members.
  - Fixed a bug with international characters inside strings.
  - Fixed some scoping issues, locals are now handled correctly by the structs convertor.
  - Added getType() and typeid for interfaces.

- 0.9.8.3

  - Fixed a possible compiler crash.
  - Fixed a bug that made function interfaces unable to have arguments of custom types.
  - Modified the way grimoire version works in many ways, just replacing the executable will not work. A new version of newgen pack is released which you should update, else you may also have to wait for a new grimoire version.

- 0.9.8.2

- Fixed a 0.9.8.0 bug that prevented dynamic arrays from being used.

- 0.9.8.1
  - Fixed a bug with function evaluate/execute methods using wrong variables.

- 0.9.8.0
  - Structs now come with an internal private static method called allocate that does what .create used to do.
  - If no correct static method create is declared within an struct body, a default one which calls .allocate is added.
  -
  - Functions are now objects, you can call methods evaluate and execute on function names no matter the position of the function declaration, execute() runs the function in another thread.
  - Added function interfaces, this allows function variables and other fun stuff.

- 0.9.7.4

- Fixed bugs with array members in structs that extend interfaces.
- Fixed syntax error bug with calling .destroy above an onDestroy declaration.
- Improved performance of interfaces (reduced a function call when calling a method, and improved constructor performance).
- Also improved performance of methods when called from above their declaration

- 0.9.7.3

  - structs may now have array members.
  - Constant integer variables may now be used for size of dynamic array and array member declarations.

- 0.9.7.2

  - <, > comparissons between zero and an struct type are allowed again.
  - Fixed logic flaw in implementation of < operator for interfaces that made them unable to use it.

- Added SCOPE_PREFIX and SCOPE_PRIVATE, assist to use ExecuteFunc / real variable events on scope private/public members, and are also useful for debugging.
- Public InitTrig function is now translated to InitTrig_ScopeName instead of ScopeName_InitTrig (makes some stuff way easier, specially for JESP spells)
- Grimoire version now writes logs in a logs subfolder as compliance with newest grimoire version.
- Fixed multiple bugs related to handling SLKs saved by certain versions of MSExcel

- 0.9.7.1
  - Fixed a major bug introduced on 0.9.7.0 that made interfaces useless.

- 0.9.7.0
  - Fixed bug with big string literals.
  - Fixed major bug with the way dynamic array indexes are handled.
  - Added an integer() typecast operator (for structs and dynamic arrays only, we'll soon have an actual typecast operator for native types).

- Added operator overloading for [] (both get and set) and >
- Fixed a bug with some syntax error showing extra, debugging information that was supposed to be removed
- Structs extending an interface no longer have to be declared after the interface
- Will now show a proper syntax error if a method derived from interface is declared as static, instead of generating bugged code.

- 0.9.6.3

  - Fixed requirement of whitespace before [in dynamic array declarations.
  - Fixed some issues with nested methods causing misleading syntax errors.
  - Fixed some problems with local variables/arguments with the same name of previous local variables/arguments that were of custom types while the new ones were not (causing some confusion/odd syntax errors).
  - Fixed a probable issue with dynamic array indexes
  - Added instructions about how to update JassHelper in newgen pack

- 0.9.6.2
  - Fixed syntax errors that could appear if there were special characters in strings or comments.
- 0.9.6.1
  - (WEHelper only) fixed a bug that made worleditor unable to ever finish compiling.
- 0.9.6.0
  - Added dynamic arrays (type name extends anothertype array [size]).
  - Added typecast operators (for struct types, soon we will have them for all the types).
  - It might now detect some few syntax errors before the pjass stage (prevents confusion when there are errors in code that is already compiled by JassHelper)
  - Instances might also call static members
  - Fixed a bug with the readme's html
- 0.9.5.2

- Fixed a bug with struct methods that had more than one argument of different types.
- JassHelper now repeats its process after external tools are executed.
- Added an interfaces demo.

- 0.9.5.1
  - Fixed a bug that could cause [Undeclared variable f__arg_this] pjass errors when saving
  - Fixed a bug with //! import not importing the last line of the file.
  - Fixed a bug with //! import not being able to import a file if quotes weren't used for the path and there were comments after the command (may happen if import is the last line of a world editor [trigger]])

- 0.9.5.0
  - Can now convert slk files to struct assignments with the //! loaddata preprocessor.
  - Added the //! external preprocessor which allows you to configure JassHelper to run command line tools, the way the command line tools have to work is very specific so if

you should check out the manual if you are interested in making them.

- WEHelper's plugin has now dialogs to configure lookup folders and external tools. Grimoire's mapcompiler can take advantage of a .conf file.
- Grimoire mapcompiler is now able to run wewarlock once configured correctly.
- JassHelper's import can now be used in WEHelper if WEHelper's is disabled. The advantage you can get from it is the ability to configure the lookup folders.
- Fixed some terrible typos in the interfaces explanation of the readme

- 0.9.4.4

  - Fixed a compiler crash when there was an struct (something) extends (something else) when the parent struct name wasn't declared yet.
  - Fixed a chance for the struct usage to generate game-crashing code.

- 0.9.4.3

- Fixed a bad bug that could cause access violations if textmacros are used extensively
- Fixed a bug with public/private members not being replaced accordingly on lines that had a / in them.
- Grimoire version allows relatives paths for //! import , you can specify where to look for files in the newly set mapcompiler.conf file

- 0.9.4.2
  - It is again safe to call JassHelper twice. (Fixes some issues with testmap and WEHelper)
  - Grimoire version now includes a beta of //! import , use //! import on complete paths only (for example: //! import c:\goo.j)

- 0.9.4.1
  - Documented 0.9.4 features.
  - Fixed a bug with static methods on interface extending structs causing PJASS errors.
  - Fixed a bug with static methods with no arguments having a chance to cause compile errors that to make matters worse were not detectable by PJASS.

- Fixed a bug with default values being ignored on structs that extend interfaces.

- 0.9.4

  - structs can now have methods.
  - Added interfaces
  - Added //! inject
  - Again, documentation of new features would take a while
  - Compiler for grimoire now got a progress bar and uses SFMPQ.dll directly instead of mpqutils, which should be faster and also be compatible with a later version of grimoire which will remove mpqutils and replace them with mpq2k.

- 0.9.3

  - Fixed multiple bugs in compatibility between scopes and structs.
  - Fixed a minor issue with the grimoire compiler.
  - Updated some sections of the manual, added more info about structs.

- 0.9.2

- Fixed grave bugs probability when many structs were used
- Destroying the 0 struct will do nothing instead of sending it to the recycle stack.
- Include compiler to be used by grimoire's wehack.dll.
- Documented 0.9.0 aditions.

- 0.9.1

  - Fixed a wrong syntax error when comments with triple / were present
  - Fixed a syntax error caused by having dot characters inside comments
  - The new additions to the syntax are not documented yet

- 0.9.0

  - Fixed some wrong instructions that could end up causing access violations
  - Fixed a bug with some textmacro declaration errors giving the wrong line number.
  - Fixed a bug that made libraries unable to have child scopes unlike what the documentation said.
  - Fixed a bug with private/public that made it unable to rename identifiers correctly after single / characters.

- Added library_once
- Added textmacro_once
- Added structs, dynamically allocated object types. Seriously.
- The new additions to the syntax are not documented yet

- 0.8.0
  - Added //! textmacro support.
  - Fixed a bug with private/public that didn't process global variables correctly if they were initialized and had = stuck to the name.

- 0.7.0
  - Better handling of some syntax errors.
  - Nested scopes are now legal.
  - Added the public keyword.
  - Cut the file size.

- 0.6.1 : Added installer for the WEHelper plugin.
- 0.6
  - Added private keyword and //! scope.
  - Various optimizations, specially for the plugin edition.

    - JassHelper is called again after WEWarlock, so you can use features like debug or private in files called by //! require

- 0.5.2: For WEHelper 1.5.2
- 0.5

    - Fixed wrong error messages in the case of unclosed strings causing issues.
    - Fixed a bug that made debug cut the last character

- 0.4

    - Fixed a bug that made this preprocessor unable to recognize globals//comment and endglobals//comment.
    - Removed the progress bar from WEHelper plugin
    - Added WEWarlock support to WEHelper plugin (Can call the WEWarlock compiler, and you can use wewarlock's features in your map by just saving.).
    - For WEHelper 1.5.1

- 0.3

    - initializer will now call the init functions AFTER call InitBlizzard() allowing to use blizzard globals on init functions.

- requires and uses also work in the place of needs
- For WEHelper 1.5

- 0.2: (initial public release)