GameMonkey Reference

译者: ring.of.the.c@gmail.com
http://shaohui.me

Thanks for Matt, Greg and GameMonkey:)

comments 注释

```
//  和 c++一样注释到本行末
/*
   和 c / c++ 一样的注释块
*/
```

注释块会被编译器忽略掉, 它的作用是给代码做注释或者临时使一段代码失效[调试脚本时常用]

变量和常量

GameMonkey 不像 c, Pascal 那样的强类型语言, 它更像是 Basic 语言.
GM 中的变量是以下类型中的一个:

```
null  -- 没有值, 这有类型
int   -- 32bit 的有符号整数
float -- 32bit 的浮点数
string -- null 结尾的 ansi 字符串
table  -- 数组/hash 容器
function -- 函数
user -- 用户自定义类型
```

GM 中, 变量名是大小写敏感的, __t0 和 __t1 保留做内部使用.
变量名 = [a..zA..Z] + [a..zA..Z_0..9]*
例子:

```
a = null;        // a 没有值
b = 4;           // b 是 int 类型
c = 4.4;         // c 是 float 类型
d = "hello";     // d 是 string 类型
e = table();     // e 是一个空的表
f = function() {}; // f 是一个函数
```

更多的例子:

```
a = ''SMID';        // a 是一个 int, 值为( 'S' <<24 | 'M' <<16 | 'I' <<8 | 'D' )
b = .23;            // b 是一个 float
c = 2.4f;           // c 是一个 float
d = 'c:\windows\sys' ; // d 是一个 string
```

语言和它的标准函数总是试图保留值, 然而不是保留类型. 具体规则是当不同类型的变量在一起运算时, 高级别的类型将被保留. 类型从低级到高级的顺序是: int, float, string.

例子:

```
print("hello" + 4);          // 输出: hello 4, 4 的类型被提高
print(2 + 5);                // 输出: 7, int 类型被保留
print(2.0 + 5);              // 输出: 7.0, 5 的类型被提高
print(sqrt(17.0));           // 输出: 4.1231, float 类型被保留
print(sqrt(17));             // 输出: 4, int 类型被保留
```

int 类型赋值的例子:

```
a = 179;            // 十进制
a = 0xB3;           // 十六进制
a = 0b0011001 // 二进制
a = 'BLCK';         // 字符转成 4 个 byte 分别赋予 int 的四个 byte 中
```

float 类型赋值例子:

```
b = 45.34;          // float 十进制
b = .345;           // float
b = 289.0;          // float
b = 12.34f;         // c 风格 float
b = 2.3E-3;         // 科学计数法
```

字符串赋值例子:

```
c = "c:\\path\\file.ext";       // 标准双引, 用\做转义字符
c = 'c:\path\file.ext';         // 和上面一样, 单引情况下, \不做转义字符用
c = "Mary says \"hello\"";   // 相当于'Mary says "hello"'
c = 'Chris' 's bike';           // 相当于'Chris's bike', 也就是说在单引内部表示单引的方法是连续两个单引
c = "My " "house";              // 相当于"My house"
```

基础类型可以使用标准内建库进行显示的转换, Int(), Float(), String()

例子:

```
a = 10;
b = a.String();   // 这样是最好的, 显示的调用类型转化函数, 返回转化后的值
b = "" + a;       // 这样不好, 赋值会将 a 的类型提升到 string, 但是效率底下
b = (10).String(); // 丑陋的
b = 10.String();   // 错误的, 编译不过, 因为编译器不认同这种语法
```

引用类型变量的可引用类型有 String, Function, Table, User. 当这些变量被赋值时, 并不发生 full copy, 而只是让变量指向具体的 obj

例子:

```
a = table("apple", "orange");    // a 是一个指向 table 的引用
b = a;                                   // b 现在和 a 指向同一个 table
b[1] = "banana";                  // 设置 b[1]
print(a[0], a[1]);                  // >> banana orange
print(b[0], b[1]);                  // >> banana orange
```

当一个变量被赋新值时, 该变量原来持有的值就有可能丢失掉了.

例子:

```
Oper = function(a, b){
     return a + b
};                      // Oper 现在指向一个函数
Oper = "hello";  // Oper 现在指向字符串, 原来的函数被丢失了
```

函数

语法: function(<params>) { <statements> };

一个函数体是一个值, 而函数是一个类型 {type = GM_FUNCTION, value=function...}

注意: 记住在将函数赋值给变量后面那个分号, 这是语法必须的

例子

```
// 将一个创建一个 rect table 的函数赋值给 CreateRect
CreateRect = function(posX, posY, sizeX, sizeY){
```

```
        rect = table(x=posX, y=posY, width=sizeX, height=sizeY);

        rect.Area = function() {return .width * height; };
        return rect;
};
myRect = CreateRect(0, 0, 5, 10);  // 创建一个用于描述 rect 的
table
area = myRect.Area();
// 可以用:代替.来隐式的传递一个 this 指针
Size = function(){
        return .width * .height;
};
s = myRect:Size(); // 调用时, myRect 会当做 this 指针传入 Size 中
```

作用域

和作用域有关的一些关键字, 语法:

```
global <variable>
Local <variable>
member <variable>
this
this.<variable>
.<variable>
函数中的变量.
```

默认情况下, 一个在函数中使用的变量就是这个函数的本地变量. 如果要声明一个全局变量, 需要使用 global 关键字. 访问成员变量必须通过 this 或者是使用 member 关键字声明它是一个成员变量. 在局部使用的变量可以用 local 关键字声明.

例子:

```
Access = function(){        // Access 是一个 local 变量, 它引
用着一个函数
        apple = 3;                  // apple 是函数的一个 local
变量
        global apple;               // 把 apple 声明成全局作用域

        local   apple;              // 把 apple 声明成局部作用
域
        member apple;           // 把 apple 声明成 this 的
member 变量
        this.apple;                 // 明确的访问 this.apple
```

```
    .apple                          // 隐式的访问 this.apple

};
```

例子:

```
a = 13;             // a 是一个 local 作用域变量
print(b);           // b 是 null
global b = function() {   // b 是一个全局作用域的变量, 类型是 GM_FUNCTION
        global c = 2;           // c 是一个全局作用域的变量

        d = 3;                  // d 是函数局部作用域变量

        {  if (c == 2)
            { local e = 3; }  // e 从这一刻开始成为函数局部作用域变量, 注意没有块作用域变量
        }
        print(e);           // e = 3
}
```

在查找一个变量时, 按照 local 和 parameters, 然后 global 的顺序查找.

成员变量有微妙的不同:

```
h = function() {   // h 是一个 local 变量
    global a = 3;   // a 是一个全局变量
    member n;       // n 可以从 this 被访问和创建
    d = 3;          // d 是函数局部作用域
    this.b = 3;     // b 是 member 作用域
    .b = .x + 1;    // b, x 都是 member 作用域
    print(b);       // b 是 null, 因为这里并没有 local 的 b
    print(n);       // 就像 print(this.n)一样, 因为上面显示声明过了
};
```

全局作用域中的语句.

```
x = 7;   // local
global x = 8; // global
a = function(y) {
      local x = 5;      // function local
      dostring(“print(x);”); // 这里打出 8, 和 lua 一样, dostring 总是在全局环境下编译运行的, 无法访问 function 的变量和
```

```
parameters
            };
```

变量可以是虚拟机全局作用域的，也可以是某个函数作用域的，或者是某个obj比如table的成员作用域的. 当执行一个文件或者是执行一个字符串的时候，和lua一样，文件或者是字符串被编译成一个无名的函数，所以默认情况下，其中最外层的未加特别申明的变量是该无名函数的函数作用域的.

this总是存在的. 它或者是null，或者是一个有效的值. 你可以传递this，或者是使用重载冒号操作符默认的this. 这一特性多用在创建诸如类似模板行为，那些地方的obj的操作往往只有run-time时才能确认. this也用在创建线程，例子:

```
obj:thread(obj.DoThings)       // 开始一个线程，并把obj作为this传递给它
obj:MakeFruit(apple, orange)  // 调用MakeFruit，并把obj当做this传给它
```

语法和操作符

```
!          Not 逻辑取反
~          每一个bit位取反
^          bit位使用与或 XOR
|          bit位使用或 OR
&          bit位使用与 AND
>>         bit位右移
<<         bit位左移
~=         bit位取反赋值
^=         bit位 XOR 赋值
|=         bit位 OR 赋值
&=         bit位 AND 赋值
>>=        bit位 右移 赋值
<<=        bit位 左移 赋值
=          赋值
'          单引     其中的字符会当做int值
"          双引     字符串(处理转义字符)
`          反引     字符串(不处理转义字符)
[]         方阔     用index取talbe元素
.          取table元素
:          给函数传递this
+          数学+
-          数学-
*          数学*
/          数学/
%          模取余
```

+=, -=, *=, /=, %= 数学运算并赋值
{} 界定语句块
; 语句结束标志
<, <=, >, >=, == 数学比较
&&或 and 逻辑 AND
|| 或 or 逻辑 OR

Tables 表

语法: table(<key> = <value>, ...);
table(<value>, …);
{<key>=<value>, …, };
{<value>, …, };

table 可以被同时认为是 array 和 map. 因为 table 中可以容纳 data 和 function, 所以 table 也可以被认为是 class, table 中也可以容纳别的 table, 这时它也已被认为是 Tree.

初始化 table 的例子:

```
fruit = table("apple", "banana", favorite= "tomato", "cherry");

fruit = {"apple", "banana", favorite="tomato", "cherry"};
```

这时, fruit 的样子就是:

```
fruit[0] = "apple";
fruit[1] = "banana";
fruit[2] = "cherry";
fruit["favorite"] = "tomato";  也可以写作是 fruit.favorite = "tomato"
```

可以注意到, fruit.favorite="tomato"并没有占据 element[2], 虽然它在逻辑上应该是 element[2]的位置, 但是它不是一个 index 索引成员, 是一个{key, value}成员.

从表中取得元素的例子.

```
a = thing.other;      // other 是 table thing 中的一个成员
b = thing["other"]; // 相当于 b = thing.other
c = thing[2];          // c 取得了 thing 中的第三个 indexd 索引成员
index = 3;
d = thing[index];     // 用 int 做下标, 就可以把 table 当数组访问

accoc = "fav";
```

e = thing[accoc];　　// 用 string 做下标，就可以把 table 当 map 访问

注意，thing["Fav"]和 thing["fav"]是两个不同的东西，因为 GM 是大小写敏感的. 这样做设计上的考虑是:

1) 赋值可能是 string，也可能是任何类型的值.

2) 要做到大小写无关，底层需要一些额外的工作量，这会产生一定量的效率问题.

设置 table 中成员的值的例子.

```
thing.other = 4;
thing[3] = "hello";
```

嵌套表的例子:

```
matrix = { {1, 2, 3,}, {4, 5, 6}, {7, 8, 9,}, }   //
print("matrix[2][1] = ", matrix[2][1]);    // 输出"matrix[2][1] = 8"
```

关键字 if 和 else

语法:　if (<condition>) { <statements> }

或者 if (<condition>) { <statements> } else { <statements> }

或者 if (<condition>) { <statements> } else if (<condition>) { <statements> } else { <statements> }

例子:

```
foo = 3;
bar = 5;
if ((foo * 2) > bar){
     print(foo * 2, "is greater than", bar);
}
else{
     print(foo * 2, "is less than", bar);
}

// 输出: 6 is greater then 5
```

if 会计算条件表达式的值，并根据其结果的 true/false 来选择执行那一段语句.

if 在计算条件时，会像大多数语言那样，并且实现了短路求值，下面是一些例子:

```
if (3 * 4 + 2 > 13)    ==   if ( ( (3*4) + 2) > 13 )
if (3 > 0 || 0 > 1)    3 > 0 恒真，那么永远不会去对 0 > 1 求值
```

对 c 程序员的提示: 你不能把 condition 和一个单语句无语句块标示的 statements 写在同一行, 这是语法不允许的

例: if (a > 3) b = 4;　　// 错误, b = 4 必须被{}包起来

关键字 for

语法: for (<statement1>; <condition>; <statement2>) { <statements> }

例子:

```
for (index = 0; index < 6; index = index + 2){
        print("Index = ", index);
}
```

输出是:

```
Index = 0
Index = 2
Index = 4
```

for 语句的执行和大多数语言一样, 循序是

1. 执行 statement1
2. 执行 condition, 是 false 就退出 for
3. 执行 statements
4. 执行 statement2, goto 2

关键字 foreach

语法: foreach (<key> and <value> in <table>) { <statements> }

foreach (<value> in <table>) { <statements> }

例子:

```
fruitnveg = table("apple", "orange", favorite = "pear", yucky="turnip", "pinapple");
foreach(keyVar and valVar in fruitnveg){
    print(keyVar, "=", valVar);
}
```

输出是:

```
2 = pinapple
0 = apple
favorite = pear
1 = orange
yucky = turnip
```

注意到遍历 tale 的时候，它并没有按料想的顺序来输出. 事实上，这种顺序会在 table 中的元素填入和删除时发生变化.

在 foreach 的每次迭代过程中，key 和 value 将会作为循环体的 local 作用域变量. ~~在迭代过程中，对 table 执行删除元素操作是安全的，但是向 table 中新增元素和从 table 删除元素是~~[原文是: Although the foreach iteration is ‘delete safe’, the behaviour of adding and removing items from a table while iterating is undefined. 我理解不了 delete safe 和 removing items from a table]~~—~~请大家告诉我好的理解，我好改正

_

关键字 while

语法: while(<condition>) { <statements> }
例子:

```
index = 0;
while ( index < 3 ) {
     print("index =  ", index);
     index = index + 1;
}
```

输出:

```
  index = 0
  index = 1
  index = 2
```

while 结构先检查条件，如果条件为真就执行循环体并反复执行这一过程直到第一次检查到条件为假. 如果一开始条件就为假，那么循环体中的代码一次也不会执行.

关键字 dowhile

语法: dowhile (<condition>) { <statements> }
例子:

```
index = 0;
dowhile (index > 0) {
     print("index = ", index);
}
```

输出:

```
    index = 0
```

dowhile 和 while 不同, 它先执行循环体, 然后检测条件已决定是否要再次执行循环体, 循环体中的代码至少执行一次.

关键字 break, continue, return

break 的例子:

```
 for (index = 0; index < 4; index = index + 1) {
        if (index == 2) {
             break;
        }
        print(“index =”, index);
 }
```

输出:

```
   index = 0
   index = 1
```

continue 的例子:

```
    for (index = 0; index < 4; index = index + 1) {
          if (index == 2) {
               coutinue;
          }
           print("index = ", index);
   }
```

输出:

```
   index = 0
   index = 1
   index = 3
```

return 的例子:

```
Add = function(a, b) {
      return a + b;
 }
print("Add res = ", Add(4, 5));
```

输出:

```
      Add res = 9
```

```
 Early = function(a) {
      if (a < = 0) {
           print(“Dont want zero here.”);
           return ;
       }
       print("Above zero we handle.");
   }
  Early(-2);
```

输出:

```
   Dont want zero here.
```

break 和 continue 用来退出或者是忽略 for, while, dowhile, foreach 循环. break 会使执行流跳出循环语句块, continue 导致终止本轮迭代, 并立即进行下一轮迭代, return 不光是能跳出循环, 它是直接跳出当前函数.

关键字 true, false, null

在 GM 中, true 和 false 分别表示 0 和 1. 除此之外没有别的意义. 注意, 和其他语言不太一样的地方

例子:

```
a = 3;
if (a == true) {
      print(a, "==", true);
 }else{
      print(a, "!=", true);
 }
```

输出: 3 != 1

null 是一种类型, 而不是一个值, 它通常用来表示错误. 当它和其他类型混用时, 它的类型会被提升, 值为 0. 当一个变量被声明但没有赋值时, 这个变量就是一个 null. 对于 table 中的元素, 如果被赋值为 null, 就表示这个元素被从 table 中删掉了.

例子:

```
local var;
if (var) {  // var 声明了但没赋值, 所以是 null, 这里类型提升了, 值为 0 == false
    print("var was initialised or non zero : ", var);
} else {
    print("var is zero or null : ", var);
 }
```

输出: var is zero or null : null

---------- 分割线 ----------

Thread

1. int thread(function a_function, …)
   创建一个线程.
   a_function 是线程的执行体
   ... 是传给 a_function 的参数
   该函数返回一个 thread id, 控制和查询线程都必须通过这个 id 来进行.

2. void yield()
   调用该函数导致当前执行体让出对 GM 虚拟机的控制权.

3. void exit()
   调用本函数导致当前执行体立即退出.

4. void threadKill(int a_threadId)
   kill 掉指定的线程, 被 kill 掉的线程不能再次运行.
   a_threadId 是要 kill 的线程的 id, 由 thread()函数返回.
   如果调用 threadKill()将导致当前线程被 kill 掉.

5. void threadKillAll(bool a_killCurrentThread = false)
   kill 掉所有的线程, 参数为 false 的话 kill 掉出自己外的所有线程, 否者连自己也 kill 掉.

6. void sleep(float a_time)
   停止当前执行体指定的秒数.

7. int threadTime()
   返回当前线程执行的总时间, 单位是毫秒.

8. int threadId()
   返回当前线程的 id

9. table threadAllIds()
   用一个 table 返回所有的线程 id.

10. void signal(var a_event)
    引发一个事件. a_event 是任意类型的, 表示一个事件.

11. void block(var a_event, ...)
    让当前线程阻塞在一个或多个事件上. 只到事件发生, 该线程才能被转化为可执行的.

States

在游戏编程中, 常使用状态的概念来描述一个游戏实体的行为, 就是常常说到的有限状态机. 在 GM 中, states 允许一个线程结束后马上丢弃这个线程的栈并跳到另一个执行点开始执行.

1. void stateSet(function a_function, …)
   设置当前执行线程的新的状态函数.
   a_function 表示要执行的状态函数.
   ... 表示要传给状态函数的参数.

2. function stateGet()
   获取当前执行的状态函数. 如果之前没有调用过 stateSet, 那么将返回 null.

3. function stateGetLast()
   获取当前状态的前一个状态, 可以用来得知变迁信息.

4. void stateSetExitFunction(function a_function)
   设置一个函数, 该函数将在状态变迁时调用. 可以用来在进入下一个状态前本次状态函数本身的一些清理工作, 如果没有下一个状态, 那么这个函数不会被执行.

System

1. void debug()
   使调试器在这里产生一个断点.

2. void assert(int a_condition)
   检查 a_condition, 它必须是非 0 值, 否者产生一个异常然后退出线程.

3. int sysTime()
   返回虚拟机执行的实现, 单位是毫秒.

4. int doString(string a_script, int a_executeNow = true)
   编译并执行 a_script 中的脚本
   a_executeNow == true 的话, script 将马上被执行, 然后 doString 函数才返回, 否者返回新建的 thread id.
   实质的步骤是:
   1. 把 a_script 编译成一个函数 func
   2. 调用 id = thread(func)
   3. if a_executeNow == true
          yield()
      else
          return id

5. int typeId(variable a_var)
   返回 a_var 的类型值

6. string typeName(variable a_var)
   返回 a_var 的类型名字

7. int typeRegisterOperator(int a_typeid, string a_opName, function a_func)
   给指定的类型注册一个操作
   a_typeid 目标类型
   a_opName 操作名
   a_func 现实的操作函数
   返回 1 成功, 0 失败.
   a_opName 的取值: getdot, setdot, getind, setind, add, sub, mul, dov, mod, inc, dec, bitor, botxor, shiftleft, shiftright, bitinv, lt, gt, lte, gte, eq, neq, neg, pos, not

8. int typeRegisterVariable(int a_typeid, string a_varName, variable a_var)
   给指定的类型注册一个变量, 使用(type).varname 的方式就可以获得这个变量
   a_typeid 目标类型

a_varName 要加的变量名
a_var 要加的变量
返回 1 成功, 0 失败.

9. int sysCollectGarbage(int a_forceFullCollect = false)
如果当前内存使用量超过了指定的内存使用量, 那么执行垃圾回收.
a_forceFullCollect 如果垃圾回收可用的话, a_forceFullCollect=true 将马上开始执行
返回 1 表示垃圾回收执行了, 其他情况返回 0.

10. int sysGetMemoryUsage()
返回当前的内存使用量, 单位 byte.

11. void sysSetDesiredMemoryUsageHard(int a_desired)
设置内存使用量硬限制. 在垃圾回收时会根据这个值来决定要不要执行一次完整的回收.
a_desired 内存使用硬限制, 单位是 byte.

12. void sysSetDesiredMemoryUsageSoft(int a_desired)
设置内存使用量软限制. 在垃圾回收时会根据这个值来决定是否开始增量回收. soft 值必须小于上面的 hard 值, 谢谢
a_desired 内存使用软限制, 单位是 byte.

13. void sysSetDesiredMemoryUsageAuto(int a_enable)
开启或者关闭在接下来的垃圾收集中是否能自动调整内存限制.
a_enable 1 开启 0 关闭

14. int sysGetDesiredMemoryUsageHard()
获取内存使用量硬限制, 单位是 byte. 注意, 这个值是用在开始一次完整的垃圾回收前的检测.

15. int sysGetDesiredMemoryUsageSoft()
获取内存使用量软限制, 单位是 byte. 注意, 这个值是用在开始增量垃圾回收前的检测.

16. int sysGetStatsGCNumFullCollects()
返回虚拟机执行完整垃圾回收的次数.

17. int sysGetStatsGCNumIncCollects()
返回虚拟机执行增量垃圾回收的次数. 注意在 restart 的那一次回收中, 这个值会+2.

18. int sysGetStatsGCNumIncWarnings()

返回 GC 或者 VM 因为配置的问题[soft 和 hard 限制]而导致的警告的数量. 如果这个数庞大而且急速增加, 那么 gc 的软硬内存限制应该重新配置以获得更好的性能表现. 这些警告的出现一般意味着 gc 次数过于平凡或不足. 如果这个值很小, 或者是增长很慢, 那么不用去担心它. 可以阅读介绍 GM 的 gc 的文档[翻译完这个, 我就翻译 GM gc 的文档]来获取关于 gc 话题的很多信息. 我们将在以后的版本中改进这个函数, 以便让它的意义很明确易懂.

表操作

1. int tableCount(table a_table)
   计算 table 中元素的个数.

2. table tableDuplicate(table a_table)
   返回传入 table 的一个副本.
   我测过了, copy 的深度就是 a_table 的元素这一层, 比如

```
t1={a=9}; t2={uu=t1, b=45};
t3 = tableDuplicate(t2);
t3.b = 78;
t3.uu.a = 80;
print(“t2.b = ”, t2.b);  // t2.b = 45
print(“t3.b = ”, t3.b);  // t3.b = 78
print(“t2.uu.a = ”, t2.uu.a); // t2.uu.a = 80
print(“t3.uu.a = ”, t3.uu.a); // t3.uu.a = 80
```

   啥内涵大家一看就明白了

----------华丽的风格线----------

绑定 C 函数到 GM 脚本中

C 函数可以绑定到类型上, 也可以绑定到一个全局变量.
一个可以绑定到 GM 中的 C 函数的原型是:

```
int GM_CDECL gmVersion(gmThread* a_thread)
  a_thread->GetNumParams() 可以获得参数的个数
  a_thread->Param*()              获取各个参数
```

a_thread->GetThis()　　访问 this
a_thread->Push*()　　向脚本中返回值

还有一些有用的宏和简写的功能函数.

C 函数的返回值描述如下:
GM_OK　函数执行成功
GM_EXCEPTION 函数执行失败, 在函数运行的 thread 中产生一个运行时异常
当然函数也可能返回一些控制 thread 行为的值, 比如 GM_SYS_SLEEP, GM_SYS_YIELD, GM_SYS_KILL, 这些值可以让脚本的高级用户实现和修改虚拟机的行为. 用户拥有强大的控制权, 可以更高效的实现参数变量, 重载函数(通过支持不同类型的参数), 检查错误或无效的输入.
一个 GM 操作符绑定函数的原型是:

```
void GM_CDECL dunc(gmThread* a_thread, gmVariable* a_operands)
```

a_operands[0] 是左参数
a_operands[1] 是右参数
a_operands[0] 同时也是返回值

如果操作符函数不能执行该操作(比如错误的参数类型等), 就把 a_operands[0]置为 null
对于二元操作符来说, 比如 O_MUL, 调用操作符函数时将选择两个参数类型较高的参数的绑定函数. NOT 是一个一元操作符(这时将使用 a_operands[0].m_type 的绑定函数). 这一点和 c++是不一样的, 在 c++中, 如果你创建了一个类 Vec3, 那么 Vec3 * float 的运算就需要重载一个*操作符, 而 float * Vec3 需要重载一个全局的友元函数. GM 这样做是为了降低原生类型的处理代价和易于用户定义类型的扩展. 所以原生的 int 和 float 类型不需要在意那些比他们高级的类型, 但是用户自定义类型例如 Vec3 可以很有弹性的和低级类型一起工作, 它的绑定函数将被调用.
可能发生冲突的地方就是当用户自定义类型之间发生运算时, 如果用户知道注册的顺序的话, 他们可以依据这个来编码, 否者可能要实现同样的操作符函数来保证不会发生因为注册顺序而导致的问题. 两个用户类型可以给一个操作符绑定同样的操作符函数, 这样可以避免不必要的重复.

例子 1, 实现一个可以注册到 GM 中的 C 函数, 比较简单, 不写注释了

```
// int GetHealth(GObj* a_obj, int a_limit)
int _cdecl GetHealth(gmThread* a_thread) {
        GM_CHECK_NUM_PARAMS(2);
        GM_CHECK_USER_PARAM(GObj::s_scrUserType, userObj, 0);

        GM_CHECK_INT_PARAM(limit, 1);
        Gobj* gob = (Gobj* )userObj->m_user;
```

```
            if (gob->m_health > a_limit) {
                gob->m_health = a_limit;
            }
            a_thread->PushInt(gob->m_health);
            return GM_OK;
        }
```

例子，向 GM 中导入一个函数，使得在 GM 中可以使用 sqrt(56)或者 sqrt(67.8)，过程比较简单就不写注释了

```
int __cdecl gmfSqrt(gmThread* a_thread) {
      GM_CHECK_NUM_PARAMS(1);
      if (a_thread->ParamType(0) == GM_INT) {
           int intVal = a_thread->Param(0).m_value.m_int;
           a_thread->PushInt((int)sqrt(intVal));
           return GM_OK;
       } else if (a_thread->ParamType(0) == GM_FLOAT) {
            float floatVal =
a_thread->Param(0).m_value.m_float;
            a_thread->PushFloat(sqrtf(floatVal));
            return GM_OK;
      }
      return GM_EXCEPTION;
   }
static gmFunctionEntry s_mathLib[] = {
      {"sqrt", gmfSqrt}, };
machine->RegisterLibrary(s_mathLib, sizeof(s_mathLib) /
sizeof(s_mathLib[0]));
```

例子，为 String 类型加上一个 Compare 操作的演示，使得可以在 GM 中使用 "hihi".Compare("hihi") ，因为比较重要，给出完整代码.

```
#include "gmThread.h"
int GM_CDECL gmfStringCompare(gmThread* a_thread)
{
       GM_CHECK_NUM_PARAMS(1);
      // Compare 的参数必须是 string，因为这个函数预期将进行字
符串的比较
      if (a_thread->ParamType(0) == GM_STRING)
      {
            // 获取调用 Compare 的变量
            const gmVariable* var = a_thread->GetThis();

            // 这个变量一定也是一个 string
            GM_ASSERT(var->m_type == GM_STRING);
```

```
                    // gm str ----> c str
                    gmStringObject* obj =
(gmStringObject* )GM_OBJECT(var->m_value.m_ref);
                    const char* thisStr = (const char* )*obj;
                    const char* otherStr = a_thread->ParamString(0);

                    // 具体的操作
                   a_thread->PushInt(strcmp(thisStr, otherStr) ? 0 : 1);

                    return GM_OK;
              }
              return GM_EXCEPTION;
       }

       static gmFunctionEntry s_stringlib[] = {
              {"Compare", gmfStringCompare},
       };

       int main(int argc, char* argv[])
       {
              // 先创建虚拟机
             gmMachine machine;

             // 注册到虚拟机
            machine.RegisterTypeLibrary(GM_STRING, s_stringlib, 1);

            // 好了可以用了:)
            machine.ExecuteString("print("res = ",
\"hihi\".Compare(\"hihi\"));");
            getchar(); // Keypress before exit
            return 0;
      }
```

程序执行结果是输出 res = 1

从C中调用GM脚本

从C中调用GM脚本时使用gmCall辅助类会让整个事情变得很简单, 下面就是一个例子:

```
#include “gmCall.h”  // 要使用 gmCall 就必须包含这个头文件
```

```
            gmMachine machine; // 初始化一个 GM 虚拟机
            // 我们要调用的函数是: global Add = function(a, b) { return
a + b; };
            gmCall call;
            int resultInt = 0;
            if (call.BeginGlobalFunction(&machine, “Add”)) {
                 call.AddParamInt(3);
                 call.AddParamInt(5);
                 call.End();
                 call.GetReturnedInt(resultInt);  // 取结果
             }
```

警告: 如果你从函数中返回一个 string, 那么你就马上使用它, 或者是把它 copy 出来, 不要长期的持有这个指针. 因为这个字符串不会一直有效, 说不定在下一轮的垃圾收集中就把它回收了, 这样的话, 你再次使用它的指针时就很危险了.

游戏对象扩展

我怎样才能扩展 GM, 向它中添加一个我自己定义的类型, 就像 table 那样子.

怎样在 GM 中表达一个 game obj?

下面的代码就是完整的将 GameObject 类型导入到 GM 中, 包含创建, 访问, 内存管理的各个方面

```
        struct GameObject {
             gmTableObject* m_table;    // 容纳表功能
             gmUserObject*  m_userObject;
             static gmType s_typeId; // 存储自己定义的类型
         };

         gmType GameObject::s_typeId = GM_NULL;
         #if GM_USE_INCGC
         static bool GM_CDECL GCTrace(gmMachine* a_machine,
gmUserObject* a_object, gmGarbagCollector* a_gc, const int
a_workLeftToDo, int& a_workDone) {
             GM_ASSERT(a_object->m_userType == GameObject::s_typeId);
             GameObject* object = (GameObject* ) a_object->m_user;
             if (object->m_table)
                    a_gc->GetNextObject(object->m_table);
             a_workDone += 2;
             return true;
```

```
        }

        static void GM_CDECL GCDestruct(gmMachine* a_machine,
gmUserObject* a_object) {
             GM_ASSERT(a_object->m_userType == GameObject::s_typeId);

             GameObject* object = (GameObject* )a_object->m_user;
             object->m_table = NULL;
        }
       #else
       // 垃圾回收的标记函数
       void GM_CDECL GameObjectMark(gmMachine* a_machine, gmUserObject*
a_object, gmuint32 a_mark) {
             GM_ASSERT(a_object->m_userType == GameObject::s_typeId);

             GameObject* obecjt = (GameObject* )a_object->m_user;
             object->m_table->Mark(a_machine, a_mark);
        }
        // 垃圾回收的回收函数
        void GM_CDECL GameObjectGC(gmMachine* a_machine, gmUserObject*
a_object, gmuint32 a_mark) {
              GM_ASSERT(a_object->m_userType == GameObject::s_typeId);

              GameObject* object = (GameObject* )a_object->m_user;
              object->m_table.Destruct(a_machine);
              delete object;
         }
        #endif

        // 设置一个用来描述类型的字符串以便在调用"AdString"得到它
        static void GM_CDECL AsString(gmUserObject* a_object, char*
a_buffer, int a_bufferLen) {
              GM_ASSERT(a_ojbect->m_userType == GameObject::s_typeId);

              GameObject* object = (GameObject* ) a_object->m_user;
              char mixBuffer[128];
              sprintf(mixBuffer, “GameObject Cptr = %x”, object);
              int mixLength = strlen(mixBuffer);
              int useLength = GM_MIN(mixLength, a_bufferLen - 1);
              GM_ASSERT(useLenght > 0);
              strncpy(a_buffer, mixBuffer, useLength);
              a_buffer[useLengrh] = 0;
        }
```

```
        // get dot 操作符用来访问 table
        void GM_CDECL GameObjectGetDot(gmThread* a_thread, gmVariable*
a_operands) {
              GM_ASSERT(a_operands[0].m_type == GameObject::s_typeId);

           gmUserObject* user =
(gmUserObject* )GM_OBJECT(a_operands[0].m_value.m_ref);
              GameObject* object = (GmaeObject* )user->m_user;
              a_operands[0] = object->m_table->Get(a_operands[1]);
         }

        // set dot 操作符用来访问 table
        void GM_CDECL GameObjectSetDot(gmThread* a_thread, gmVariable*
a_operands) {
               GM_ASSERT(a_operands[0].m_type == GameObject::s_typeId);

           gmUserObject* user =
(gmUserObject* )GM_OBJECT(a_operands[0].m_value.m_ref);
               GameObject* object = (GameObject* )user->m_user;
               object->m_table->Set(a_thread->GetMachine(),
a_operands[2], a_operands[1]);
        }

        // 从脚本中创建 GameObject
        // 注意: 游戏中像这样直接创建对象实体并不常见, 还有, 可能并不
像保存对象的 c 指针, 取而代之的是保持一个 32bit 的 UID 来代表对象, 在使用
的时候通过 UID 来查询和验证对象
        int GM_CDECL CreateGameObject(gmThread* a_thread) {
               GameObject* object = new GameObject();
               object->m_table =
a_thread->GetMachine()->AllocTableObject();
               object->m_userObject = a_thread->CreateUser(object,
GameObject::s_typeId);
               return GM_OK;
        }
        // 获取一个 object, 这种情况下的 obj 在 c 和脚本中是一对一的.
        int GM_CDECL GetGameObject(gmThread* a_thread) {
               GameObject* foundObj = NULL;
               GM_CHECK_NUM_PARAMS(1);
               if (a_thread->ParamType(0) == GM_STRING) {
                      // todo: foundObj =
FindByName(a_thread->ParamString(0));
                      // 如果找到的话
                      a_thread->PushUser(foundObj->m_userObject);
```

```
                a_thread->PushNull();
                return GM_OK;
            }
            else if (a_thread->ParamType(0) == GM_INT) {
                // todo: foundObj =
FindById(a_thread->ParamInt(0));
                // 如果找到的话
                a_thread->PushUser(foundObj->m_userObject);
                a_thread->PushNull();
                return GM_OK;
            }
            return GM_EXCEPTION;
        }
        // 注册函数
        gmFunctionEntry regFuncList[] = {
            {"GameObject", CreateGameObject},
        }
        // 向虚拟机注册类型, 假定虚拟机已经构造好了
        // 1. 注册新类型
        GameObject::s_typeId = machine.CreateUserType(“GameObject”);

        // 2. 注册垃圾回收要用到的
        #if GM_USE_INCGC
            a_machine->RegisterUserCallbacks(GameObject::s_typeId,
GameObjectTrace, GameObjectDestruct, AsString);
        #else
            a_machine->RegisterUserCallbacks(GameObject::s_typeId,
GameObjectMark, GameObjectGC, AsString);
        #endif
        // 为类型绑定 get dot 操作
    machine.RegisterTypeOperator(GameObject::s_typeId, O_GETDOT, NULL,
GameObjectGetDot);
        // 为类型绑定 set dot 操作
     machine.RegisterTypeOperator(GameObject::s_typeId, O_SETDOT, NULL,
GameObjectSetDot);
       // 注册函数
        machine.RegisterLibrary(regFuncList, sizeof(regFuncList) /
sizeof(regFuncList[0]));
```

虚拟机的回调

如果一个应用程序拥有它自己的 gmObject, 它必须让垃圾回收器知道这个对象正在被使用. 要做到这一点, 你需要在虚拟机回调中捕获 MC_COLLECT_GARBAGE 消息, 这个回调发生在垃圾回收器开始扫描根时. 一个让 gc 正确管理 c++持有的 gmObject 的代换方案是使用 gmMachine::AddCPPOwnedGMObject() 和 gmMachine::RemoveCPPOwnedGMObject(). 第三种方法是使用 gmGCRoot<>指针来实现.你可以通过查阅 GM gc 的文档来获取更多这方面的知识.

应用程序可能希望在 thread 创建和销毁时执行一些动作, 比如管理这个 thread 专有的 object 等. 此外, 应用程序可能希望将 thread 的异常信息导入到 error stream 中. 下面是一些列子.

```
// 假定你在别处已经建立了虚拟机, 现在你要注册 callback 函数
gmMachine::s_machineCallback = ScriptCallback_Machine;
bool GM_CDECL ScriptCallback_Machine(gmMachine* a_machine,
gmMachineCommand a_command, const void*a_context) {
        switch (a_command) {
            case MC_THREAD_EXCEPTION: {
                // dump 异常信息导到标准输出
                bool first = true;
                const char* entry;
                while ( (entry =
a_machine->GetLog().GetEntry(first)) ) {
                    fprintf(stderr, "%s", entry);
                }
                a_machine->GetLog().Reset();
            }
            break;
            case MC_COLLECT_GARBAGE": {
        #if GM_USE_INCGC
            gmGarbageCollector* gc = a_machine->GetGC();
            // 对于所有的 c 拥有的 obj
            //  gc->GetNextObject(obj);
        #else
            gmuint32 mark = *(gmuint32 *)a_context;
            // 对于所有的 c 拥有的 obj
            // if (object->NeedsMark(mark)) {
            //        object->GetTableObject()->Mark(a_machi
ne, mark);
            // }
        #endif
            }
            break;
            case MC_THREAD_CREATE: {
                // 线程创建时的回调
```

```
                }
                break;
                case MC_THREAD_DESTROY: {
                        // 线程销毁时的回调
                 }
                 break;
        }
         return false;
}
```

翻译后记

前前后后翻译了一周多，终于算是告一段落了，通过翻译，增加了对 GameMonkey 的一些理解.

因为它是从 lua 发展起来的，有很多概念有一些像，但是经过仔细的观察和研究代码，发现 GM 除了借鉴了一些 lua 的概念，从实现上和 lua 是完全不一样的. 比如元方法的实现等等. GM 使用起来将会感觉更加复杂，有很多问题都需要去 coding 解决，而不像 lua 那样美丽:）但是从另外一方面来讲，GM 的确是给了程序足够的控制力，的确称的上是一门面向程序员的语言.

翻译的比较匆忙，有什么错尽管指出:）谢谢