MUD 游戏的魔法系统设计

摘要

本次课题设计涉及的内容是 MUD 游戏场景。MUD 属于文字网络游戏，通过文字的相关描述告诉游戏者当前发生的状况。课题中需要设计相关的场景，以及课题的主要内容——魔法系统。

本文阐述了 LPC 语言的使用方法、数据类型等相关规则。从 LPMUD 的人物、房间、物品、战斗等方面入手，讨论了程序实现的具体思路和可能出现的问题。同时也提出程序中的不足之处。最后对游戏中出现的各种魔法效果，分析了它们是如何实现的。

课题设计中所用到的工具有 mudos，Uedit32 编译软件，文中出现的部分图片，来自 Uedit32 处理程序中的截图，用以帮助说明相关理论。

关键词：LPC　Mudos　MUD

ABSTRACT

This subject design is related to the MUD Game Scene. MUD is the online games of words. By the description, the net games will tell the players what is happening. About the subject, we design the game scene and the magic system.

This text expatiates on the rule and the date type of LPC. Though the figures、rooms、items and fights, we analyse the specific ideas of the program and the problem which maybe appears. At the same time, we also discuss the deficiencies. Finally, we analyse the effects and the methods of many magic in the game.

The tools that were used in the subject design involve Mudos、Uedit32(software of edit), some picture in the text is from Uedit32, in order to explain some theories.

Key Words: LPC Mudos MUD

第一章 绪论

1.1 课题背景和现状

MUD 指的是 Multi-User Dimensions，俗称“泥巴”。这里 mu*的意思是多用户游戏。MUD 是一种实时多用户网络游戏。是现代网络游戏的雏形。

在文字冒险游戏发展最初阶段，出现过 zork、advent 等，开发在老式 DEC 设备上的文字游戏，他们所支持的字符短，游戏内容不够丰富。而后出现了用 C++编写的 CMUD 网络游戏，在 C++构建的类上，创造了一个内容较为丰富，玩家间通信更为方便的虚拟世界。

鉴于商业运作的原因，MUD 虚拟世界需要不断地添加新的游戏元素。用 C++编写的 MUD 相对而言扩充性较差，遇到了发展的瓶颈。于是发展出了 LPMUD，所谓的 LPMUD 是使用一种类 C 的语言 LPC 所编写而成。游戏底层的 Driver 处理的速度有赖于用 C 语言编写的函数，而游戏中玩家所在的虚拟世界完全用 LPC 语言，并且可以调用底层的函数。但是虚拟世界几乎不与 Driver 有直接的联系，那么这样的一种形式实现了“即加即玩”的效果。

生活在过去 20 年中的每一位游戏编程人员对 MUD 都应该很熟悉，大家认为 MUD 是基于文字的劈杀游戏，玩家在游戏中到处乱跑，随心所遇的杀人。

从某种程度上来讲这个观点没错，但是 MUD 并不只现于劈杀。有些 MUD 是一些人群，这些人聚集在这个虚拟世界中，只是为了彼此通信。MUD 也未必总是基于文字的。事实上，几乎所有虚拟世界风格的游戏就如同魔术师帐篷下的秘密一样，他们都有相似的结构。使用图形客户程序解释 MUD 中发生了什么，也根本就不那么困难。

1.2 课题的主要研究内容

当今流行于网络的 MUD 游戏，从主要内容分为两种：侠客类与西游神话类。设计中运用的是后者的基础游戏构架，创建新的门派、魔法系统、游戏场景，同时为了新构建的游戏需要，对构架的某些部分做出调整。

为了达到设计目的，设计中的主要内容归纳如下:

1.熟练掌握 LPC 语言，并且可以利用其编写出较为复杂的效果。

2.掌握语言的同时，必须学习整个游戏的构架，避免设计作品游离于构架之外。

3.掌握游戏构架的同时，对其做出适当调整，实现新的魔法效果。

1.3 论文内容安排

第一章 绪论：课题背景与现状、课题的主要内容

第二章 LPC 程序设计：LPC 语言的特点、数据类型，编译错误讯息等

第三章 LPMUD 游戏：讨论人物、房间、物品、武器、战斗循环等的实现

第四章 MUD 中的魔法系统：讨论各种魔法的实现思路及整个魔法系统

第五章 总结与展望

第二章 LPC 程序设计

2.1 LPC 程序及其资料形态

2.1.1 LPC 程序的特点

LPC 编写程序的所写内容可以统称为物件(objects)。一般来说，运行一个程序的时候，是有开始和结束的。所有的程序开始执行的时候，进入主函数进行处理，程序执行完后就终止了。而 LPC 的程序不同，整个 mudlib 的 driver 系统运行的是用 LPC 编写出来的程序，这些程序在不同的时间和情况下被不断的调用，虽然都是运行程序，但是 LPC 的程序在 mudlib 中是不存在绝对的触发点和结束点的。

2.1.2 LPMud 的数据类型

Object：

是定义一个对象，具体是定义一个 NPC、一个物品、一个场景、一个运行于内存里的文件。实际上是一段由后面很多变量按一定运算方式组合在一起的程式。一个程序里制造成一件新的物品，则必须先定义一个变量，如:object obj，然后再 obj = new(xx) 将这个 obj 实际上就 clone 了出来，括弧里的 xx 代表它的文件绝对路径名。

Int：

表明定义的是一个整型，可以为正负或 0，定义出来的数字可以进行各种数字运算，但结果只保留小数点前的数字。

Float：

表示定义的是一个浮点型，对于使用的 MUDOS 来说，机器支持浮点值到小数点得后 7 位（精确度）。

String：

表明定义的是一个字符串型，字符串的长度在理论上是没有限制的。在 LPMUD 里，限于网络响应，一般是在编译 MUDOS 时，在 config.h 文件里进行设置与限制。对于字符串型变量的识别，有简单的区别标准：有用“”括起来的是 string 型，没有则看其是否整数而分辨为整数数值与浮点数值。因此在一些不严谨的语句中，如没有强制定义，也可将

int、float 与 string 区分出来。

A、set("number",783);------->int 型

B、set("number",78.3);------>float 型

C、set("number","783");----->string 型

D、set("number","78.3");---->string 型

同时 string 型可以相加，但决非数字意义上的运算，而是一种合并，例如上面的 C＋D 就是"78378.3"；

Array：

表明定义的是一个数组型， LPC 中的数组是在类型后面加 * 来表示数组，且不用预先定大小，但若有需要可以用 allocate(size)来固定大小。

如：a = allocate(10); 在固定了 SIZE 之后好处是可以任意用下标定位来对数组元素操作。

Mapping：

LPC 中利用率最高的散列型（映射型），可以用任意的数据类型来建立索引(如 string、object、int、array 等等)，而不是仅仅用整数。

例：mapping fam = (["a":2,"b":13,"c":2.333,"d":"一条小河","e":"158"]);

fam 里的 a、b 子变量是 int 型的，c 是 float 型的，d、e 是 string 型的。LPC 的说明文件里，a、b、c、d 被叫做“关键字”，而后面的 2、13、2.333、一条小河、158 被叫做“内容值”。

散列型的变量可以用 “变量名["关键字"]” 的形式进行调用，并可以用 “变量名["关键字"]=新内容值” 的方式进行赋值。例：fam["e"]的值是"158" ，如果 fam["e"]="400"，那么再次调用时：fam["e"]的值就是"400"。

2.2 LPC 函数

2.2.1 函数：

写出正确的 LPC 函数有三个部分:

1) 宣告 (declaration)

2) 定义 (definition)

3) 呼叫 (call)

宣告一个函数的格式如下: 返回值型态 函数名称 (参数 1, 参数 2, ..., 参数 N); 函数并不是由前往后连续执行的. 函数只有被呼叫时才会执行. 唯一的要求是, 一个函数的宣告必须出现在函数的定义之前, 而且也必须在任何函数定义呼叫它之前。举出 write_vals() 和 add()两个函数的例子，仅供参考:

```
/* 首先, 是函数宣告. 它们通常出现在物件码的开头. */
void write_vals();
int add(int x, int y);
/* 接着是定义 write_vals() 函数. 我们假设这函数将会在物件以外被呼叫. */
void write_vals()
{ int x;
/* 现在我们指定 x 为呼叫 add() 的输出值. */
x = add(2, 2);
write(x+"\n"); }
/* 最後, 定义 add() */
int add(int x, int y)
{ return (x + y); }
```

2.2.2 外部函数

外部函数是由 mud driver 所定义。很多函数如：this_player()、write()、say()、this_object()...等等，就是外部函数。外部函数的价值在于它们比 LPC 函数要快得多，因为它们是事先就以电脑可以直接读取和运行的二进制码的格式存在着。这些外部函数是早就被定义和宣告好的内容，需要时只需要直接呼叫调用它们就可以了。

创造外部函数是为了处理普通的、每天都需要使用到的函数呼叫、处理 internet socket 的输出与输入、其他用 LPC 难以处理的事情的完成和实现。它们是在 driver 内以 C 写成的，并与 driver 一起编译在 mud 开始之前, 这样它们执行起来会快得多。使用时需要知道两点::1) 它的返回值是什么 2) 它需要什么参数。

外部函数的详细资料及其形态,可以在 mud 中的 /doc/efun 目录找到（也可以通过 mudos 编译手册直接查询其功能）。不过因为每种 driver 的外部函数都不相同，所以无法

给出一个详细的类型表分类，不过可以通过「help」指令(视 mudlib 而定)找到详细的资料。

2.2.3 自定义函数

虽然在 object 的编译中，函数次序的先后是没有关系的，但是定义一个函数的程式代码的先后顺序却非常重要。当一个函数被呼叫时，函数定义中的程式代码按照出现的先后顺序执行。如函数 write_vals() 中，这个指令：

x = add(2, 2);

如果你想看到 write() 使用正确的 x 值，就必须把它放在 write() 呼叫之前。

函数要返回一个值时，由「return」指令之后跟着与函数相同资料型态的值来完成返回。在先前的 add() 之中，指令「return (x+y);」把 (x+y) 的值传回给 write_vals()并指定给 x。也就是说「return」停止执行函数，并返回程式代码执行的结果给呼叫此函数的另一个函数。另外，它将跟在它后面任何式子的值传回呼叫的函数。因此,要停止执行失去控制的无返回值函数，使用 return 就可以了。而后面不应加上任何东西。「return」传回任何函数的资料型态必须与函数式本身的资料型态相符合，这是一一对应的关系。

2.3 LPC 面向对象编程的继承

2.3.1 LPC 的继承性简述

继承性是面向对象程序设计最重要的特性，有别于传统设计中，每个项目程序都需要进行单独的开发和调试，从而造成工作量“大”、程序分布“散”、程序间“协同性差”等问题。继承性的这一机制有效地解决了这些，LPC 与 C++类似，同样需要利用程序的可重用性。在 C++中所谓的继承就是在一个已经存在的类的基础上建立一个新的类。比如“马”这个类，成为“基类”，而“公马”、“母马”、“幼马”则是他的“子类”或者“派生类”。

LPC 继承的概念类似。游戏中存在人物、场景、物品（食物、钱财、武器、衣服）、技能、战斗，而如果对与每一事件 object 进行类 C 中 main()似的编写的话，从工作量和程序协同型角度来说，这样的方法是不可能实现一个完整庞大的游戏世界的。因为每一个物件除了它应该有的本质特性外，还有它们互相之间的关系，大多数情况下物件还会调用其他物件中的函数。而用一个精心构思过的程序（标准继承 room.c）来给这一类程序 room 继承的话，首先每一个 room 都从标准继承中继承了 room 本质具有的特性，其次在完成这

个物件的某些特殊程序后，也可以作为其与同类物件 object 的标准继承（标准继承 bank.c）。与 C++完全类似，利用了继承与派生的这一机制，来构造游戏大的构架。关于课题所涉及的 mud 的构架关系及程序调用流程会在游戏构架的章节中叙述。

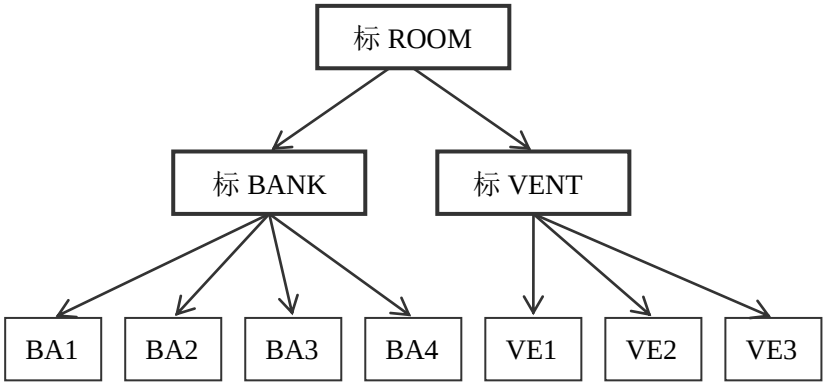


图 2-1 继承与派生

2.4 变数的处理

2.4.1 数值与物件

基本上，mud 里的物件都不一样的原因有两个：

1) 有的物件拥有不同的函数

2) 所有的物件都有不同的数值

现在，所有的玩家物件都有同样的函数。它们不一样的地方在于它们自己所拥有的数值不同(例如 id)。举例来说，名字叫做 mzjl 的玩家与名为 fyfbi 的玩家，他们各自的 name 变数值不同，一个是"mzjl"，另一个是"fyfbi"。所以游戏中量值地改变伴随着游戏中物件值的改变。函数名称就是用来处理变数的过程名称。例如：create()函数就是特别用来初始化一个物件的过程。函数之中，有些特别的事称为指令，指令就是负责处理变数的。

2.4.2 区域和全域变数

跟大多数程式设计语言的变数一样，LPC 变数可以宣告为一个特定函数的「区域」变数，或是所有函数可以使用的「全域」变数。区域变数宣告在使用它们的函数之内。其他

函数并不知道它们存在，因为这些值只有在那个函数执行时才储存在记忆体中。物件码宣告全域变数之后，则让后面所有的函数都能使用它。因为只要物件存在，全域变数就会占据记忆体。只有在整个物件中都需要某个值的时候，才要用全域变数。

2.4.3 复杂的运算式

有类似：i = ((x=sizeof(tmp=users())) ? --x : sizeof(tmp=children("/std/monster"))-1) 这样的复杂的运算式。

下面是对它的解释：

把外部函数 users() 传回的阵列指定给 tmp, 然后把此阵列元素的数目指定给 x. 如果指定给 x 的运算式值为真 (不是 0) , 就指定 x 为 1 并指定 i 的值为 x-1 的值. 如果 x 为伪, 则设定 tmp 为外部函数 children() 传回的阵列, 并指定 i 为阵列 tmp 的元素数目再减 1。这样的语句能提升程式的执行速度。

2.4.4 比较常使用的 LPC 运算写法

```
x = sizeof(tmp = users());
while(i--) write((string)tmp[i]->query_name()+"\n");
```

取代这样子的写法:

```
tmp = users();
x = sizeof(tmp);
for(i=0;i<x;i++) write((string)tmp[i]->query_name()+"\n");
```

2.5 LPC 运算符号

1) 算数运算符

+（加） -（减） *（乘） /（除）

%（整除求余） ++（自加） --（自减）

2) 关系运算符

>（大于） <（小于） ==（等于） >=（大于等于） <=（小于等于） !=（不等于）

3) 逻辑运算符

&&（逻辑与） ||（逻辑或） !（逻辑非）

4) 赋值运算符 （=及其扩展赋值运算）

5) 条件运算符（?:）

6) 指向成员运算符（->）

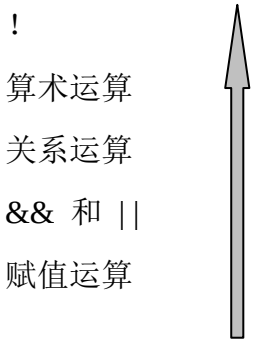


图 2-2 运算的优先级别

2.6 流程控制结构和语句

if(运算式) 指令;

if(运算式) 指令;

else 指令;

if(运算式) 指令;

else if(运算式) 指令;

else 指令

while(运算式) 指令;

do { 指令; } while(运算式);

switch(运算式)

{ **case** (运算式): 指令; **break**; **default**: 指令; }

介于这类语句的规则与 C 类似，所以论文中不予深入讨论其用法。(详：查阅 C 或 C++相关书籍)

2.7 常见编译出错信息

以某一文件为例（简述）：

1) 编译时段错误：/u/llm/npc/test.c line 13: parse error

parse error 一般表示错误出在基本的拼写上，多是像逗号、分号写错，或者是各种括号前后多写或漏写的情况，可以在提示的第 13 行或之前的几句里找错误；

2) 编译时段错误：/u/llm/npc/test.c line 13: Undefined variable 'HIY'

Undefined variable 表示有一些未曾定义、不知其意义的东西存在。后面跟着的就是这个不明意义的字串。这个错误有三种可能，一是将一些变量的词拼错。二是因为这个变量未曾定义或者根本就没有声明，三是这个变量是定义在一些继承文件里，但在这个文件里却没有继承；

3) 重新编译 /u/llm/npc/test.c：错误讯息被拦截: 执行时段错误：*Bad argument 1 to call_other()

一般是指这个文件里在调用其它文件里的函数或者是对象时发生错误了；

4) 重新编译 /u/llm/npc/test.c：错误讯息被拦截: 执行时段错误：F_SKILL: No such skill (froce)

属于拼写错误，系统无法找到 froce，应该是 force(游戏中的人物的气力)；

5) 重新编译 /u/llm/npc/test.c：编译时段错误：/u/llm/npc/test.c line 75: Type of returned value doesn't match function return type (int vs string)

表示在某一个函数里，返回值的类型与定义的不同，并指出是因为 string 与 int 的错误；

6) 重新编译 /u/llm/npc/test.c：编译时段错误：/u/llm/npc/test.c line 72: Warning: Return type doesn't match prototype (void vs int)

表示错在函数类型上了，是因为函数与前面的声明相冲突，一个是 int，一个是 void。以上是在 LPC 程序编译时经常出现的出错信息，并对其原因做了比较简易的解释。程序编译中会涉及更多的出错信息，可能关系到游戏构架底层的某些规则，或者是整个游戏中的文件相互间权限的问题。这些都需要对游戏构架有进一步的了解才能解决。

第三章 LPMud 游戏

在文字冒险游戏发展之初，可能只听说过 3 个游戏：zork、advent 和 dungen。游戏名字有些令人费解，这是因为他们是在老式的 DEC 数字设备上开发的，所支持的文件名最多只能有 6 个字符。当今定义 Mud 是指 multi-user dimensions，而本次设计所运行的 mud 是构架在 LPC 程序上的游戏，所以通常被称为 LPMud 游戏。

3.1 人物（NPC 及玩家）

人物是贯穿整个游戏世界的，MUD 也不例外的需要各类角色来充实故事情节，和增加游戏的可玩性。人物有各类属性比如：力量、健康、敏捷、经验点数、等级、生命点数、准确度、躲避率、攻击伤害等。大唐西游使用的 mudlib 中，与人物属性相关的 object 利用了继承与派生的机制，分工明确的实现了人物的各种机能，以及人物与外界互通的各种游戏效果。而每个个体所特殊具有的特性直接的可在该个体的程序里反映，相当的直观和便于修改。除了人物，房间、物品、甚至战斗技能也有类似构架。会在稍后的讨论中一一涉及。

3.1.1 NPC 人物

首先可以从 NPC 来分析，在所有 NPC 的 object 程序第一行都会译写 inherit NPC;（如图 3-1）这即是 npc.c 的标准继承程序，路径是 world\std\char\npc.c。

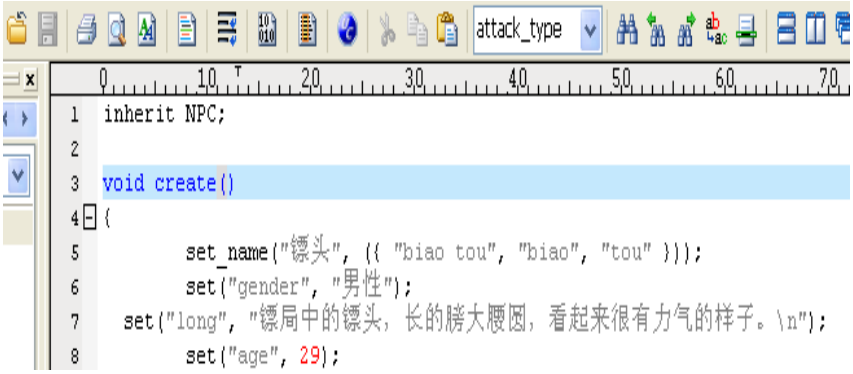


图 3-1 NPC 人物程序

Npc.c 的标准继承程序中，涵盖了 NPC 人物的一些操作属性，比如：携带金钱、装备的武器、是否接受挑战、在战斗情况下会以多少的几率来随机施法等。

```
 3  #include <command.h>
 4
 5  #define MAX_OPPENENT 4
 6
 7  inherit CHARACTER;
 8  inherit F_CLEAN_UP;
 9
10  object carry_object(string file)
11  {
28  }
29
30  object add_money(string type, int amount)
31  {
37  }
38
39  int accept_fight(object who)
40  {
76  }
```

图 3-2 Char 源继承中主要函数

从图 3-2 中可以明显的注意到 npc.c 只是另一标准继承文件的派生，或者说并不包含 NPC 人物的全部所需属性。原因很简单，NPC 与玩家所共同具有的“人”的属性才是源标准继承。它的路径为 world\std\char.c。

3.1.2 玩家与 NPC 人物基本属性

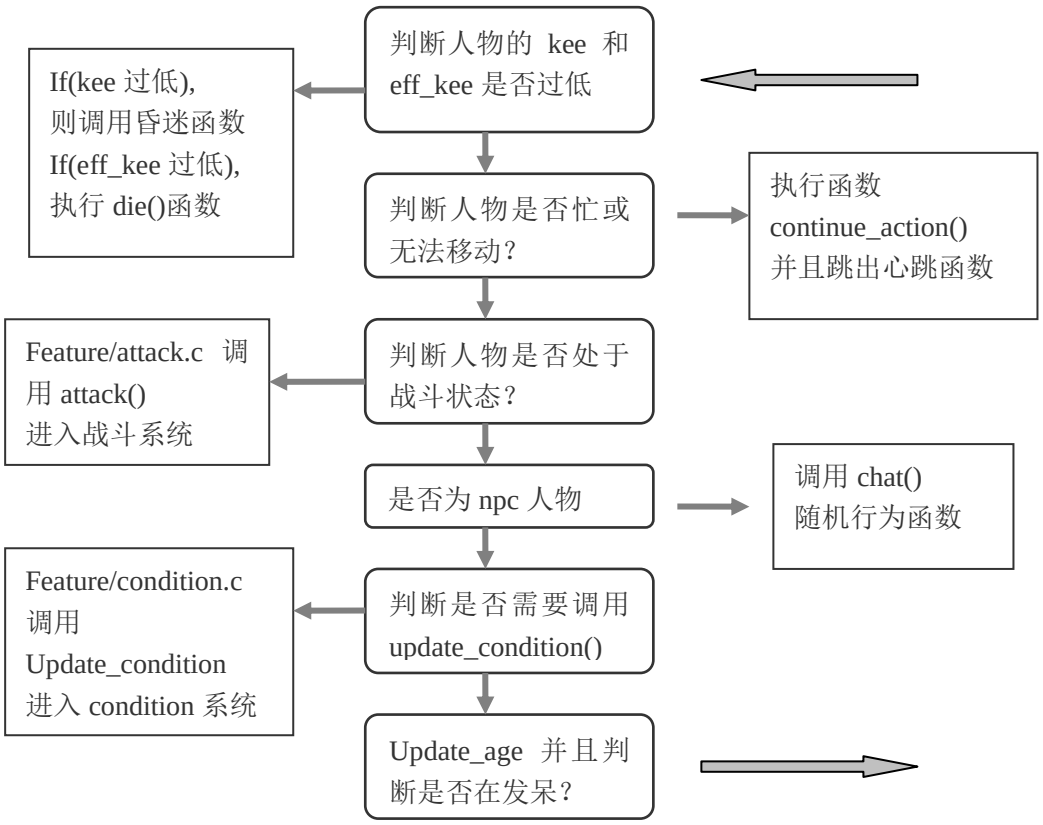


图 3-3 角色心跳循环处理流

下面给出标准继承 char.c 中的主要函数予以简单说明：

void create()： 从图 3-2 中可以看到，此函数是用于给继承它的 object 来创造 NPC 的各类生命数值的

int is_character()： 便于程序员判断人物是否是可活动用，是则返回整型

void setup()： create()函数创建完人物后，都会键入该函数，其作用是内存在读入 npc 时，触发 set_heart_beat()函数，从而启动人物的心跳特性 heart_beat()

void heart_beat(): 该函数是游戏中体现人物时间特性的函数(图 3-3)。

到这里人物的基本属性“年龄”分析完了。程序利用 mudos 提供的心跳计时，来完成对游戏整体人物群的无限循环操作（从 object 被读入服务端内存开始），在“一个程序文件的一个函数”中解决了生死判断，定时行为判断，战斗循环的触发，以及年龄的增长等问题。不过程序中牵涉到很多变量和函数，分别定义在构架的底层和表层中，这里的说明只是简易地解释了函数的功能，部分关键函数会在稍后的讨论中一一涉及（同样会涉及 heart_beat()函数）。

3.1.3 NPC 人物随机行为

回到 NPC 人物的设定中，游戏中的各类 NPC，有的是需要在街头小巷中走来走去商人，有的是需要一见面就说客套话的小二，还有的是需要在战斗中随机施法本派魔法的武馆小子，其实这些都是 NPC 随机行为中的一种。有效地利用这些随机行为功能便可以增加游戏的可玩性。

程序员经常在编写人物时，在 creat()函数中加入如下一段程序：

```
58
59      set("chat_chance", 30);
60      set("chat_msg", ({
61          (: random_move :)
62      }));
63
64  }
```

图 3-4(a)

```
74  set("chat_chance_combat", 90);
75  set("chat_msg_combat",
76  ({
77      (: cast_spell, "zhenhuo" :),
78      (: cast_spell, "qiankun" :),
79  }) );
```

图 3-4(b)

这样那个 NPC 便具有随机行为功能，图 3-4(a)中的 NPC 具有 random_move()函数提供的随机移动功能。当然也可以换其他函数，来实现比如施放魔法等功能。这里就图 3-4(a)，讨论 NPC 随机移动所涉及的程序及相关流程。

下面给出 char()函数相关程序：

```
int chat()
{
  string *msg;
  int chance, rnd;

  if( !environment() ) return 0;

  if( !chance = (int)query(is_fighting()? "chat_chance_combat": "chat_chance") )
    return 0;

  if( arrayp(msg = query(is_fighting()? "chat_msg_combat": "chat_msg")) ) {
    if( random(100) < chance ) {
      rnd = random(sizeof(msg));
      if( stringp(msg[rnd]) )
        say(msg[rnd]);
      else if( functionp(msg[rnd]) )
        return evaluate(msg[rnd]);
    }
    return 1;
  }
}
```

图 3-5 随机行为函数 chat()

这里可以清楚地看到，提取人物预先设置的数值（”chat_chance_combat”对应的值）、以及函数数组。然后对读取数组关键词并进行判断，如果是程序名则直接调用 evaluate()启动该程序函数。如 random_move()程序。（random_move 等 npc 随机行为程序同样译写在 npc.c 继承中）

人物的随机行为函数 chat()的触发，显而易见应该译写在心跳函数 heart_beat()中。这一点也在前面的心跳流程途中给出过。那么可以归纳随机行为的流程如下：

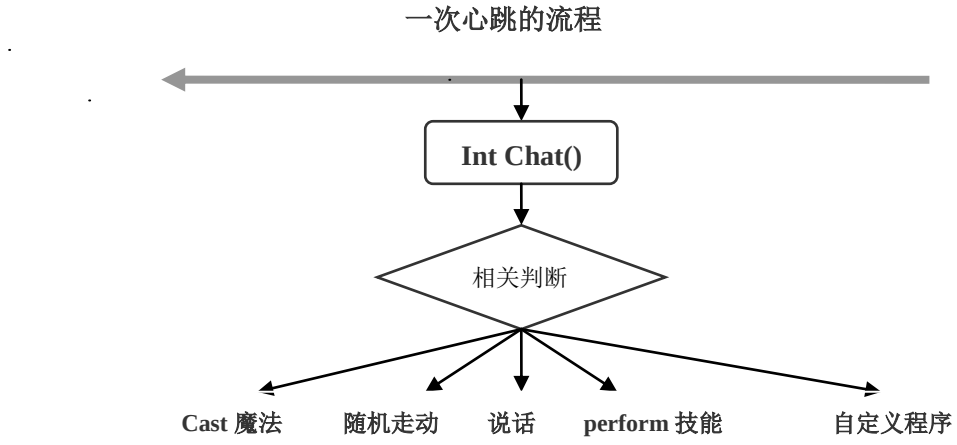


图 3-6 随机行为流程

Cast 魔法函数：　　void cast_spell(string spell)

Perform 技能函数：　int perform_action(string skill, string action)

从图 3-6 中可以看出心跳机制的重要性，包括了整个游戏人物设计的所有基本特性，通过一系列条件判断从而触发相关处理函数，本论文中还会涉及多个相关流程，会具体做

出说明。到这里对 NPC 的基本属性分析完，对于人物的随机魔法行为会在后面的魔法系统中具体讨论。

3.2 房间（Room）

Mud 游戏中玩家所在的空间，即称为 Room（房间）。房间是游戏运行必不可少的部分，玩家进入游戏直接接触的就是房间，并且直观地看到房间的描述。但房间不可能千篇一律的光是描述，而没有实际的功能。所以程序为了表层编译的简便，把所有房间所具有的基本属性都设计在标准 Room.c 中。如开门关门等，而表层程序的编译则只须负责房间的特殊属性和描述。

3.2.1 表层 room 文件

```
5  void create ()
6  {
7    set ("short", "南城客栈");
8    set ("long", @LONG
9
10 这家客栈紧靠长安城中心，所以生意非常兴隆。兼之价钱低廉，更是
11 成了外地游客落脚的首选。露天摆了七八张桌子，三教九流人等在此
12 进进出出，络绎不绝，几个跑堂的小二来来回回招待着四方来客。二
13 楼上是雅厅，后面是客房。
14 LONG);
15   set("exits", ([ /* sizeof() == 4 */
16
17 ]));
18
19   set("objects", ([ /* sizeof() == 2 */
20
21 ]));
22   setup();
```

图 3-7 表层 room 程序

如图 3-7 中便是一般房间文件设计中的程序语句。Short、long 用来描述当前文件到底是一个怎么样的房间。Exits 表示房间与别的的房间的连接处，或者称为方向出口。Objects 则是设定在这房间中的 NPC 人物或场景物品（系统启动或刷新时辨认）。最后触发标准继承函数中的 setup()。以上这些是一个基本房间的程序语句，游戏服务端便可以读出这些特性。

3.2.2 Room.C 标准继承

编译任何房间都需要继承 Room.c（inherit ROOM），在每个普通房间的 create()中触发 Room.c 的 setup()函数，其功能是实现游戏中自动更新系统。房间就其本身来说应该不具备反映时间的特性，但是考虑到游戏中自由移动的 NPC、被玩家杀死的 NPC 等，所以需要有更新的机制来实现一定时间后游戏的复原。这也是这个继承文件的主要功能。

3.2.3 游戏中房间的更新

游戏中内存读入房间物件时，便从 creat()中的 setup()触发了 reset()函数，下面对该函数进行逻辑上的梳理：

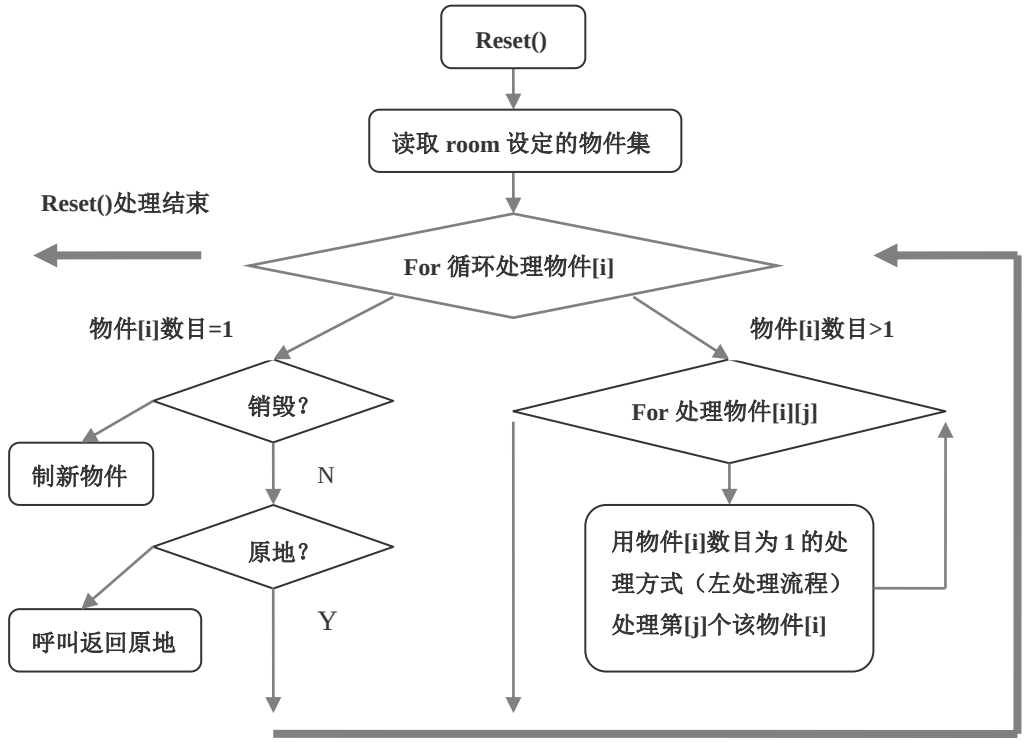


图 3-8　房间更新流程

如流程图 3-8 中的制造新物件和呼叫返回分别调用了 **make_inventory** 和 **return_home** 这两个函数，功能如下：

Object make_inventory(string file)：送入字符串格式路径 file，复制一个新的 object 放入该房间。（定义在 room.c 中）

Int return_home(object ob)：使用函数物件返回 ob 中，物件如果未能返回则函数返回值为 0。（定义在 npc.c 中）

在 reset()被调用时，程序就会根据房间物件原始设定的物件映射集(mapping)来循环处理。逐个地查找物件是否还在 mud 游戏中，如果已经不存在了，那么通过调用复制函数制造出新的物件。如果物价随机到了别处，那么调用 return_home()调回物件（如果调回失败，则对这个房间放上再次调用的记“no_clear_up”）。其中调用 return_home()是把该房间物件作为参数传送过去。

整个程序的处理中，很明显地可以观察到一点：如果房间设定物件集中某物件是一个物品(item)，并且处理中判断它并没有被 destruct（则不调用复制函数）。结果是该物件永远无法刷新出来。因为后续判断中用到的 return_home()是 NPC 才具有的函数，而物品(item)调用该函数返回的一定是 0。所以这段程序表明：当一样物品(item)在被一个玩家拿走后，这个地方是不能重新出现这个物品的。同时从这里可以知道游戏设定中的更新和程序员指令 update 是有本质上的区别的，reset()更新房间物件，并不会消除本来不属于房间的物件，而只是让房间应该有的物件重新返回或复制出来（所以玩家通常会在一个地方发现刚才还没有的 NPC 一下子被刷了出来），update 涉及的则是一个资源清除和重新读入的概念。

在 room.c 的开头处继承了 F_CLEAN_UP，其实是整个游戏的资源清除函数。Mudos 为了节约内存的损耗，对于每个占用内存的对象都有资源清除的处理。如果相当长时间没有被其他程序参考到，mudos 会自动调用 clean_up()函数，进行处理判断。判断中会访问物件的“no_clean_up”数值，也就是 reset()程序开始部分对房间物件设置(set)为 0 那个标记。设置为 0 则每次调用会清除(destruct)该房间（前提是判断出该房间中的物件没有一个是玩家）。

标准房间继承文件的主要部分是 reset()函数，除了 reset()函数还可以看到一些关于房间门处理的函数，用于一般房间编译时调用。比如该房间需要一扇用 open 命令才能开的门，那么在程序中可以直接写入 create_door()这类函数。玩家通过命令从一个房间移动到另一个房间时(go north)，调用的是 valid_leave()这个函数。如果程序员需对某一个房间做一些特殊处理的话，在那个房间的编译程序中加入 valid_leave()函数后，编写一些特殊的功能，最后继承底层的::valid_leave()即可完成编译。以上是对标准 room 继承的基础特性分析。

3.3 物品（Item）

人物和房间中必定携带着某些物品物件。MUD 游戏中的物品物件分为武器、防具、衣裤、书籍、法宝、药物等。从具体的物件来说每个物品必须和人物房间一样，有区别于其他的名称或 id。其次对其单位的描述设定，比如说椅子的单位肯定是把(set(“unite”, “把”))。借着是最终的物品属性，就是价格属性。游戏中的交易价格读取的是物品的 value 映射数值。

3.3.1 表层 item 文件

```
10  void create()
11  {
12      set_name("琥色毫毛", ({ "amber hair", "amberhair", "hair" }) );
13      set_weight(100);
14      if( clonep() )
15        set_default_object(__FILE__);
16      else {
17        set("long", "一把琥珀色的毫毛，可以用来变成衣服（transform）。\n");
18        set("unit", "把");
19        set("value",10000);
20        set("material","hair");
21        set("armor_prop/armor",1);
22      }
23      set("no_drop",1);
24      set("no_get",1);
25      set("no_sell",1);
26      setup();
27  }
```

图 3-9 表层 item 程序

如图 3-9 中可以观察到，可以对物品进行一些特殊的设置，类似的功能有不能被丢弃(no_drop)、不能被拾起(no_get)，不能被交易(no_sell)等。这些映射变量数值的设置在物件物品上作上记号后，cmds/std/下的某些命令程式会自动访问它们，已实现如上这些游戏效果。

3.3.2 Item 标准继承及相关底层

Std/item.c 是物品的标准继承文件，就文件本身并没有规定很多函数，程序中只是对物件程式做了权限的设定。而后便继承 feature/move.c 和/name.c 。

在前面分析的人物和房间继承文件都继承了这两个文件，那么文件程序的作用就非常明显了。Name.c 对表层物件 creat()里一系列出现过的设定进行了定义。如 id&name、short&long 描述等。Move.c 则相对函数较多，对于房间编译了规定房间容量的函数 set_max_encumbrance()，对于物体设定有关于重量的 set_weight()函数。对于玩家或 npc 定义有 ride()相关的函数，函数表示玩家有坐骑等提高轻功的游戏物件(在 lpmud 中是对人物身上的一个”apply/dodge”变量进行处理。其中 dodge 表示轻功，有战斗底层读取)。既然有 set 的相关函数，必定对应有 query 相关的访问函数，他们同样被定义在 move.c 文件中。Move.c 中最重要的函数是 move(string file/object ob)。最常见的房间连接后玩家能够用 go 命令移动便是通过该函数得以实现。玩家与物品之间的关系（拿起，丢下）同样被定义在 move()里。所以通常程序员用的 move()可以对任何物件使用的，原因是不管 npc、玩家、房间还是物品都继承 F_MOVE。

另外考虑到物品既然继承 move 文件，那么它必定也可以使用关于容量等函数。也就是在物品内还可以放入物品或人物。这样的继承明显是合理的，实现了一些比较特别的功能，比如“紫荆红葫芦”可以把玩家装入其中、储藏箱可以储藏玩家不常用的物品等。提高了 mud 的游戏性。

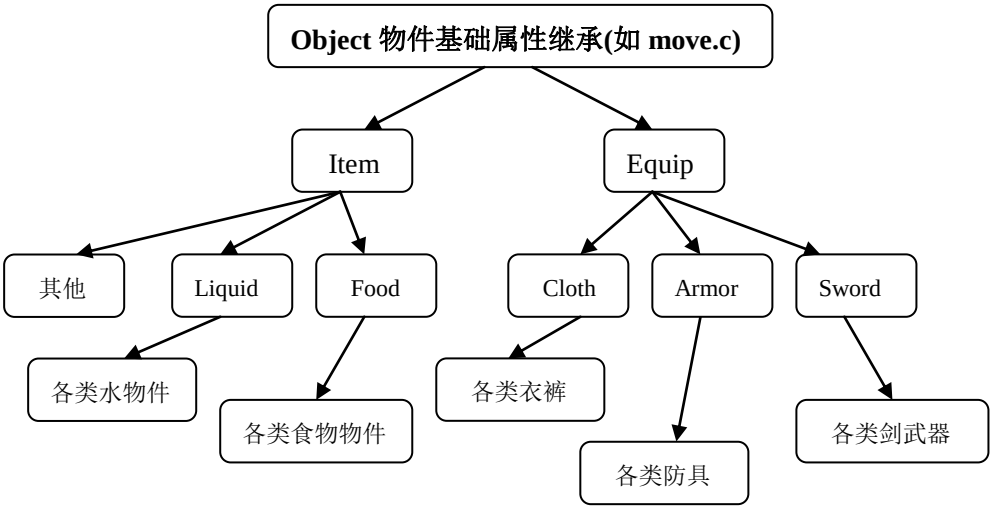


图 3-10　游戏中的继承机制

分析到这里并没有涉及具体物件的一些属性。比如水和食物的属性，武器的属性，衣服鞋子的属性等。LPMUD 并没有把所有的函数一起饼入 item.c 中，而是对物件的种类做了分类后编写标准继承文件(图 3-10)。

在 LIQUIED 和 FOOD 的继承中直接加入 add_action(“xxx”,”eat/drink”)语句，那么继承他们的各类水和食品不必一个个的编写命令，而统一的可以使用命令。各自文件下就可以编写一些特殊的函数，比如吃了某些食物后中毒、身体隐身、来到了某些地方等。是具体某个物件的程序可读性提高。

3.3.3 装备继承及相关

如图 3-10 右半部分，装备包括了游戏中的武器，防具，各类衣服裤子等。它们的继承机制并不是直接与 item.c 相关。而是由另一个文件 equip 来负责继承，接着再次派生出作用不同的类，武器相关的派生类定义在 std/weapon/下，防具相关的派生类定义在 std/armor/下。

在具体的某种武器或者某种衣服中，除了有最基本的描述定义外，最主要是对该物品的一个武器（防具）附加性能的定义，常见的类似"armor_prop/dodge"的描述（表示该物品可以附加轻功多少值）。如果需要对武器做某些特殊动作的话，也可以直接在该物件文件

下编译自定义函数。这些情况也相当常见。

派生类继承文件（如 sword）主要是对这一类的武器（防具、衣服）设定某些参数，当人物在打斗或防御时，相关的底层函数会调用这些参数，从而输出与该物件相符的一些描述或特性。由于这个文件是会被一个大类的很多文件继承，那么如果需要这个类的文件都使用某个命令的话（如衣服可以 tear），便可以直接定义在该文件中，从而很大程度减轻程序员的工作量。

武器衣服等所谓的装备其实在玩家看来只是一些描述，而实际意义应该是提升他们在游戏中的可战斗性和可防御性。这些性质必定体现在玩家身上的某些参数发生变化。源继承文件 EQUIP 就是处理游戏中战斗参数的一个文件。

int wear() ：对衣裤鞋类的函数，用于 add 物品的物价属性。

int wield() ：对武器的函数，作用同上。

int unequip() ： 在某件物件（包括武器和衣服）被脱下后，还原玩家的初始数值。

以上三个函数就是 Equip 的主体部分，它们的处理方式类似，下面就其中的主要处理过程给出图示。

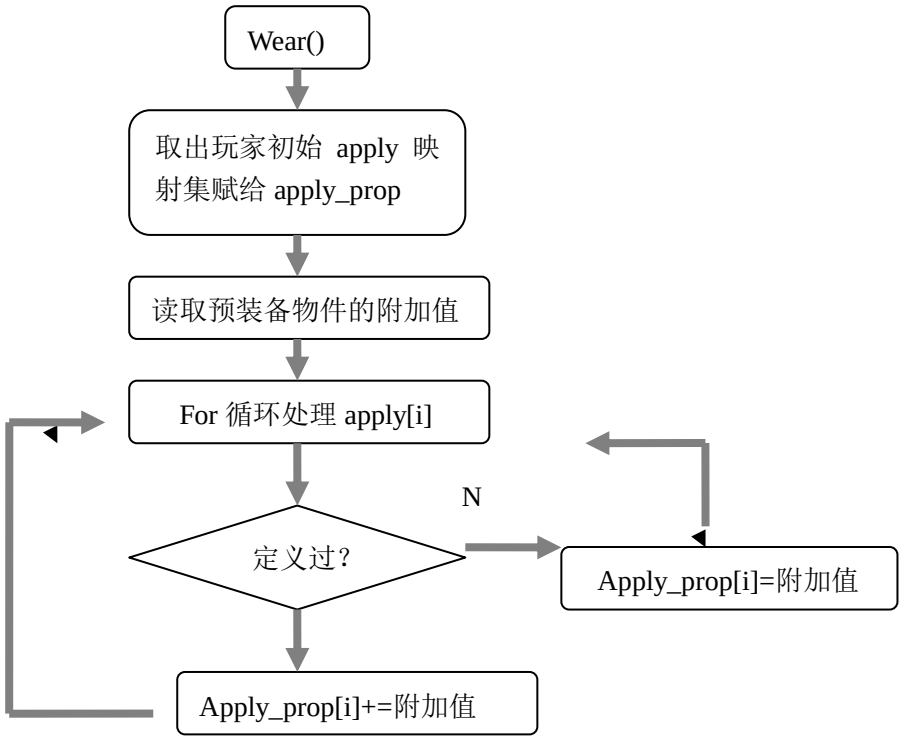


图 3-11 wear()函数处理流程

分析了图 3-11 后可以明显地看出各类物品物件中定义的”apply/xxx”的作用了，对于 wield()中的处理，是用来读取武器的相关特性”weapon_prop/int”，然后做相同的处理流程。

Unequip()的处理方式正好与它们相反，把持有武器的附加值从身上减去，恢复本来的数值。

3.4 战斗循环及 Condition 系统

如今在网络上运行的游戏，战斗这个游戏元素是必不可少的部分。MUD 是现代网络游戏的雏形，战斗系统肯定也是 MUD 中最主要的组成部分。一个战斗系统编译的好坏直接决定了该 MUD 运营的成败。

3.4.1 战斗循环的实现

在本次毕业设计中，MUDLIB 中的战斗系统建立在人物的心跳机制上。直接利用 MUDOS 提供的时间机制来调用战斗，很好的控制了战斗中的循环间隔，避免了另外译写复杂的程序来计数计时。对于一个好的游戏，战斗系统应该是完备的。如下是关于战斗循环系统的流程：

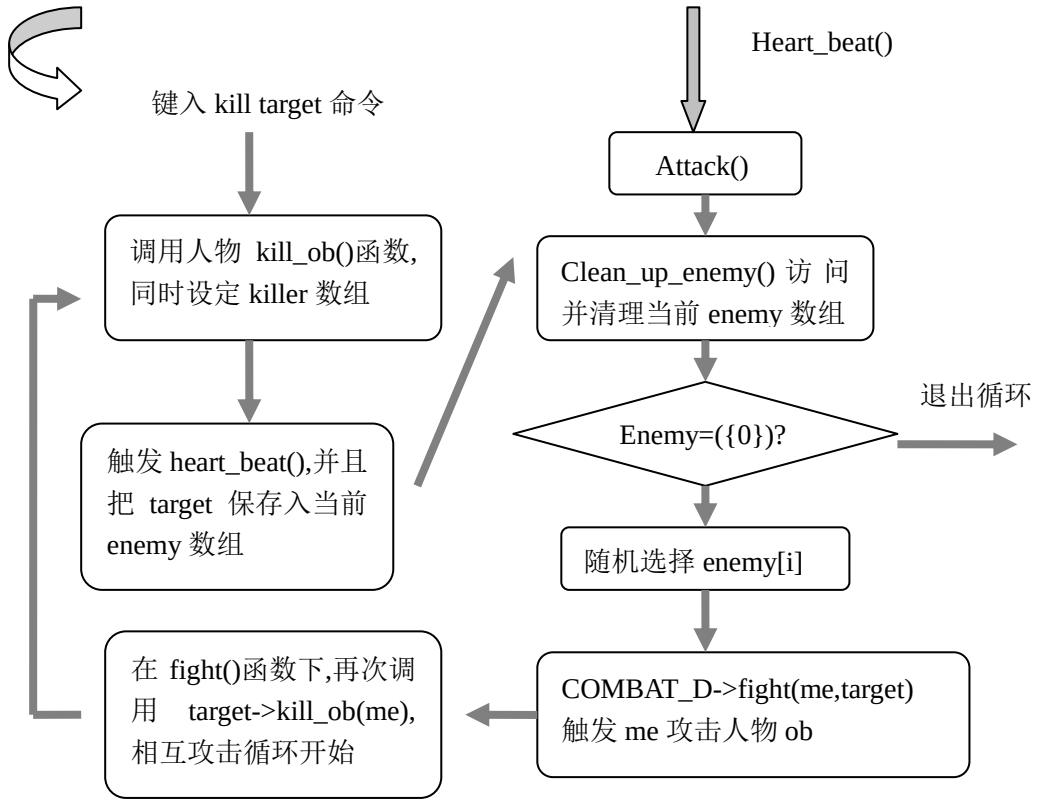


图 3-12　战斗中相互攻击流程

流程以玩家键入命令作为切入口，当玩家键入命令后，整个循环开始工作。在分析 char.c 时提到它继承了底层的很多继承文件，战斗是其中的一个文件(ATTACK)，Kill_ob()与 fight_ob()定义在该文件中。触发 kill_ob()时，同时把敌人的 id 保存入 killer 数组，以便敌人在遇到 me 是自动进行进攻。然后调用 fight_ob()时，直接设定启动心跳，并且重置或增加当前 me 的 enemy 数组。

心跳函数触发中，判断出 me->is_fighting()返回值为 1，便直接触发继承文件中的 attack()函数。如流程右半侧，在决定是否进攻时还需要做进一步的判断。原因很简单，从刚才的键入命令到触发心跳的过程中，target 可能已经和 me 不在同一环境中，甚至有可能 target 已经断线或被别人杀死。那么就这样直接进入攻击函数显然是不合理的。重新调用 clean_up_enemy()处理当前的 enemy 数组，如果刚才被保存入数组的 target 已经 destruct 或离开的话，清除 enemy 数组中的该元素。（当 me 的 enemy 大小为 0，该点就是战斗循环的自动出口）接着利用 select_oppenent()对该数组中的 target 进行随机选择，编译的程序语句限定了 MUD 游戏中 1 次战斗只能攻击到 enemy 的前四位 target，即告诉高级玩家不要同时招惹 4 个以上的敌人。

被选择的 target 会在 combatd.c 中被 me 执行 fight()函数，这样便是一个玩家攻击另一个玩家基本流程，那么让 target 反击很明显地就应该译写在 fight()中，让 target->kill_ob()已实现一个关于 target kill me 的新的循环。互相攻击效果到这里位置分析结束。从程序的流程里也可以观察出战斗中施放魔法，吃药水等并不会影响攻击。

3.4.2 Condition 系统

Condition 系统与战斗系统相同利用了心跳函数，提供了游戏中定时触发某种现象的方法。Condition 的基本思路，是在物件上设定某种记号和数值，这些记号会通过 save()保存入玩家的档案中。然后通过系统的心跳，每一次心跳会根据相关判断来执行一次固定的 condition 函数。游戏中通常会出现的效果是中毒，坐牢等，函数每次被执行就会减少一个点数，最后为 0 时便跳出该效果。**完整的 condition 包括 3 个部分**：首先是触发 condition 的程序，如前面所提到的该效果利用心跳触发，那么可以在心跳函数 heart_beat()中找到如下程序语句：

If(tick--)　return; Else tick = 5+random(10);　而后执行 update_condition();

其中 tick 是 char.c 中的一个全局变量。Heart_beat()每次被触发到该段语句，tick 减少 1，直

至为 0 时必然触发 update_condition()函数。其次是 update_condition()，这是 condition 的主题部分。下面给出图示 3-13

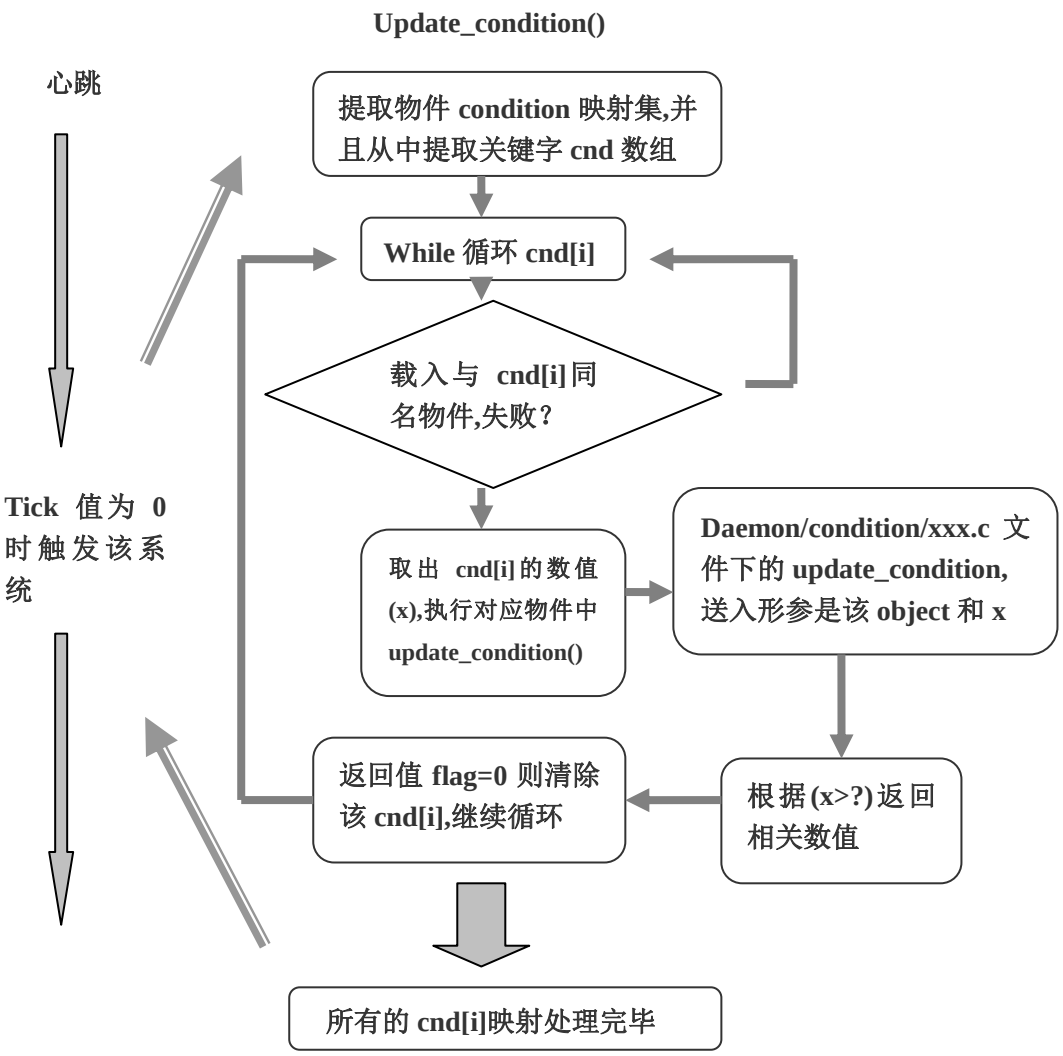


图 3-13 condition 系统的实现流程

图 3-13 流程中包含了完整 condition 的 3 个部分，流程最右侧的部分译写在 daemon/condition/下的，是 condition 的第 3 个部分。程序员根据自己需要在其中编写一些定时行为。

接着就流程做出简易说明，通常在某些魔法的编译中会译写 apply_contion(“poison”,10)执行这样一段，在 target 身上添加一个 condition 映射（如果已经有该 poison 元素，则修改其对应的数值。游戏中一个玩家已经中毒，可以再对其施放魔法延长中毒效果，这样也是符合游戏的合理性的）。在心跳函数每次执行中 tick--，直至为 0 时启动继承文件 CONDITION 中的 update_condition()。提取该玩家的 condition 集并且列出其中关键字 cnd 的数组，循环处理每一个 cnd[i]。（游戏中 daemon/condition/下的各种文件的文件名与其对

应的 cnd[i]是同名的。比如 condition 中是 poison，则有 daemon/condition/poison.c 文件）找到对应文件后，执行文件里的定时行为并且返回相关的值。当该返回值为 0 时清除 cnd[i]，进入下一个关键字 cnd[i+1]处理流程。全部处理完跳出 CONDITION 循环。整个流程表示 1 次定时行为的触发，这样不断往复便可以实现如中毒，坐牢，运功等效果。

在 Condition 继承文件中除了 update_condition 函数还编写了相关的一些函数，这里简要说明其功能：

Apply_condition(string cnd,mixed x)： 对物件的 condition 集做添加处理，根据 x 的正负值可以添加或减少 cnd 对应值。

Query_condition(string cnd)： 访问物件 condition 集中名为 cnd 的对应值。如果不存在则返回 0。

Clear_condition()： 该函数用于 die()调用，玩家一旦死亡，清除身上所有的 condition 集。

本节具体的介绍了设计中运用到的战斗循环系统和 Condition 系统，并且对实现它们的流程做了分析，从中可以得到许多对魔法系统编译的启迪。扩大了编写魔法的具体思路，从不同的系统着手，魔法的效果和真实性会越发增强。这点会在稍后的章节中提到并作分析。

第四章 MUD 中的魔法系统

前面章节讨论战斗循环和 Conditon 系统后，该章节将对魔法系统做出分析和论述。游戏中各个门派下的魔法都是直接译写在 cast()函数中，其中比较多的是调用 Combat_d 和 Spell_d 这类文件。设计中同样也会利用它们，来实现一些常用的打斗效果。

4.1 Cast 魔法系统

本节开始讨论游戏的魔法系统，魔法系统本身并不与魔法所实现的效果相关，但是构架一个好的魔法系统是有其意义的。游戏中玩家不论施放任何一种魔法都会键入 cast+魔法名字，也就是说玩家只要记住 cast 一个命令。其次游戏中的各门各派的魔法不同，如果以每个魔法一个命令的形式，那么势必要在每个 main()程序中对大量的门派进行过滤。一方面编写程序的工作量大容易遗漏，另一方面如果增加一个门派种类，那么必须对大量编写完整的程序进行修改添加，从工作量上考虑明显是不切实际的。所以魔法必须构架在一个比较完整机制上，从而每个表层魔法文件只需针对其效果编写程序，大大提高了程序的可

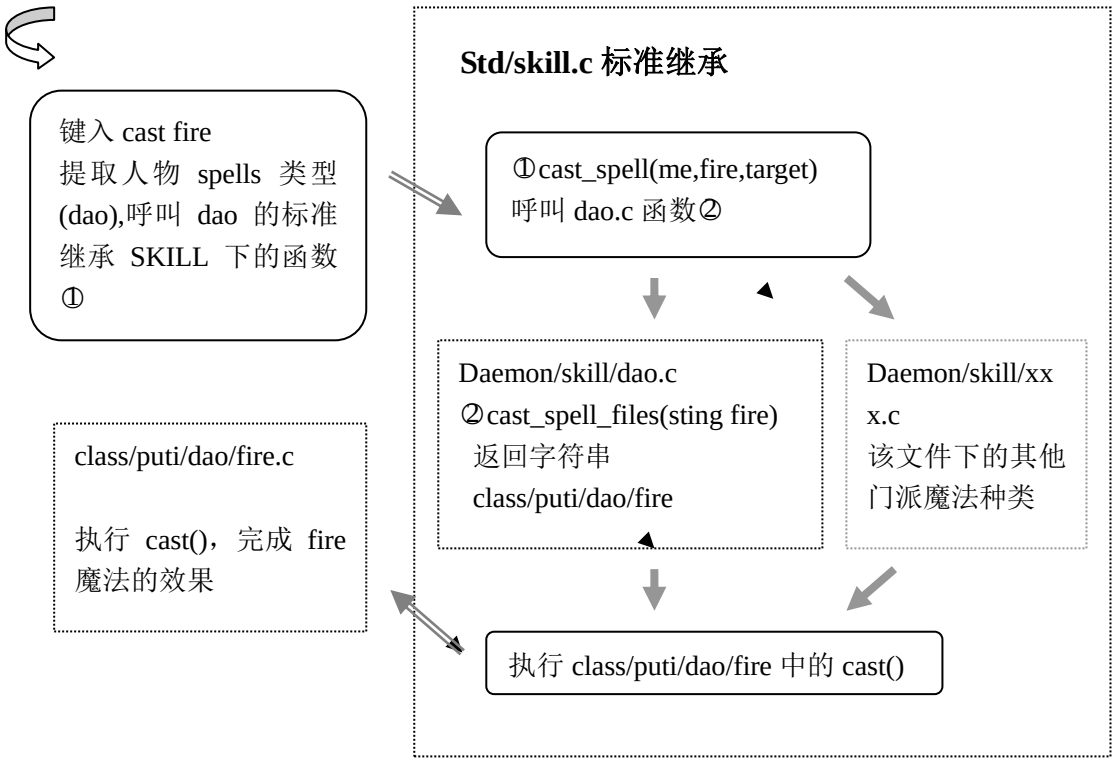


图 4-1 cast 系统的流程

读性。如需对某个魔法交换到另一门派下，也不用对其中的语句作任何修改。程序的灵活性是显而易见的。

图 4-1，在 cast.c 文件的主程序 main()中调用 cast_spell 函数，该函数定义在 Skill 文件中，其作用是启动继承它的每个门派魔法文件(daemon/skills/xxx.c)，比如方寸上使用的魔法类别是道术(dao)。这些类别文件的关键作用是寻找该门派魔法的路径（cast 中需要保存玩家的魔法，原因是其与类别文件同名，省去了复杂的语句），并且作为字符串返回给 cast_spell()处理，如果没有找到该文件或者说文件的大小为 0，那么此次 cast 失败。否则按照路径呼叫 fire 中的 cast()实现 fire 的魔法效果。值得注意的是，在 cast()中的返回值是有其作用的，按照流程图反向传回，该值在 cast.c 文件中被赋给了 nocast 变量。

nocast=(int)SKILL_D(spells)->cast_spell(me, spl, target)；

用于对玩家计算两次魔法的间隔时间，同时防止了魔法被无限施放的问题。

在前章节中提到的 NPC 随机行为是可以施放随机魔法的，其中大致原理和流程图 4-1 相似，只是进入魔法系统的入口有所改变，NPC 是不可能通过实行 cast 命令来施放魔法的。所以 mud 在 NPC 的人物继承中编写了一个 cast_spell()函数，它的作用是提取 NPC 的魔法种类，并且调用 SKILL。这样 NPC 就进入了如上述施放魔法的流程。

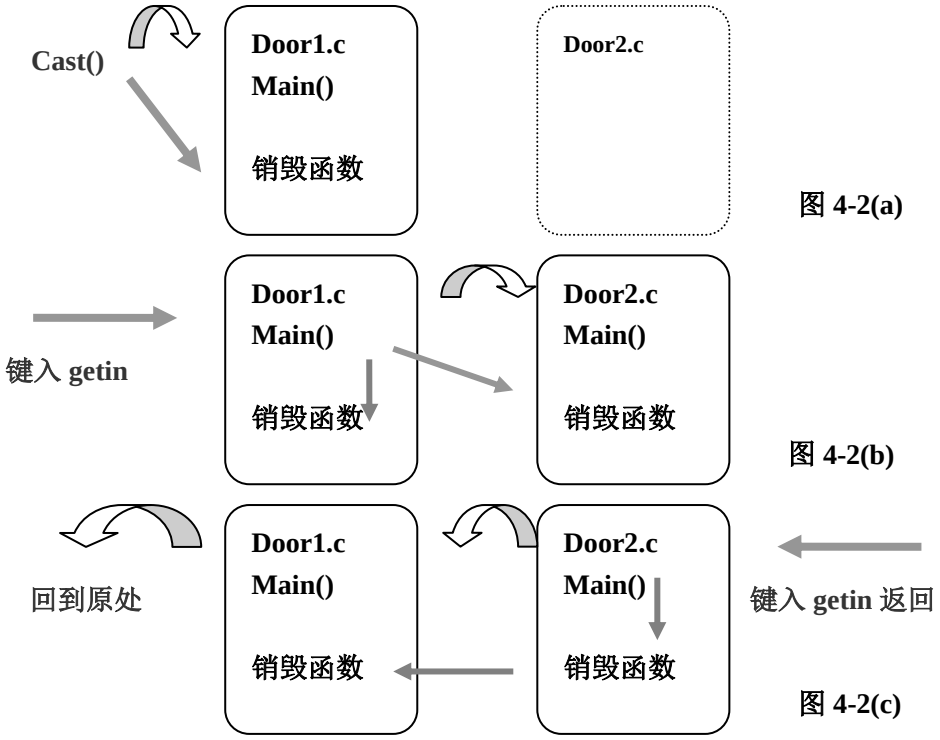
游戏中魔法所对应的基本技能是 spells，除了这些人物也具有各种武功，而武功对应的是内力 force。由于各个门派的内力武功不同，所以内力使出的 perform 技能也利用了和魔法系统同样的构架原理。此文中不再做详述。

通过前面的分析，确定魔法系统的构架需要利用 windows 下已经建好的文件夹。每个编写好的魔法只需按门派种类放在对应的地方，有魔法系统读取即可。并且在运行中出错时，可以快速的查出问题处于那个流程中。

4.2 时空门

MUD 中没有类似 DIABO 回城卷轴效果的物品或魔法，该魔法的效果顾名思义是让玩家在一处使用魔法后，能够回到客栈等地方购买所需补充物品，然后继续回到原处作战或者完成游戏人物。

下面是讨论该魔法实现的一个基本思路：



Cast 主程序中会根据玩家当时所在环境做出判断，并决定是否可以释放魔法。那么在可以释放魔法的环境中，该时空术就可以起到效果。时空术中需要对几种情况分别进行考虑来编写函数，图 4-2(a)中，刚释放魔法后玩家环境中多出一个文件 Door1，同时考虑到玩家释放了魔法后有可能不进入 Door，那么需要同时呼叫 Door 中的定时销毁函数。在玩家键入 getin 后如图 4-2(b)，复制出另一扇 Door2 放置在预先设置的地点，并且让玩家移动至那里。在该种情况下应该取消先前 Door1 中的销毁函数，并且应该对其重新设置销毁时间，于此同时呼叫启动 Door2 的销毁函数。作用是如果玩家并没有回去的意思，那么在设定时间过后两边的时空连接都会消失。图 4-2(c)中，当玩家在销毁前，再次键入 getin 表示返回时，通过 main()函数中的处理把玩家移动回去，并且重新设定销毁的时间（此次设定时间较短）。

与释放该魔法玩家同一环境的其他玩家，是无法调用时空术主程序 main()的，在 cast()程序中对 door1 做了标记，把该玩家的 id 保存入它的映射变量中。这样做法有其两种好处，首先防止其他玩家输入 getin 后进入 main(); 其次是如果在同一房间两个玩家都是用了这一魔法，那么每扇 door1 会自动辨认施放他的玩家。这样的做法也优于把标记做在玩家身上，因为如果标记物品 id 给玩家，那么两扇门的 id 是相同的，也就失去了意义。如果以 object 的形式直接标记给玩家，那么当用户众多时，对于服务器端的负载是相当大的。

为了增加游戏性，该魔法对其他玩家键入 getin 做了另外的出口，起到惩罚的作用。比如让其进入一个迷宫、中毒等效果，这里不详细叙述。

4.3 附体术

现存的 MUD 游戏中，多数魔法建立在对底层魔法攻击文件(spelld.c)的利用，和对人物伤害继承文件(damage.c)中 receive 函数的运用。总体上来说，都是对 target 进行攻击伤害或者中毒伤害等，并没有出现过类似附体术这样的魔法效果。附体术顾名思义的是控制其他玩家，在其游戏中的描述是施法玩家进入对手体内，通过一些小任务后。能够执行命令来操控对手的行为。

一个完整的附体术魔法效果有三个部分组成，其中一个是循环检查函数，该函数的作用在极短的间隔时间里，循环地检查被附体的 target 的各项数据。首先判断他是不是还在线，判断他是否已被杀死，借着判断他是否血量过低。这些判断都是必须得，缺少了这些判断此魔法就失去了合理性。

既然成为附体术，那么对 target 的部分状态施法者应该能够掌握，target 移动，交战，并且交易时都会触发 watch_out()函数，该函数最初的设定只是观察玩家当时所在环境，但是考虑到战斗时需告诉体内 target 正在交战，所以对 target 的 enemy({})进行了处理，并且

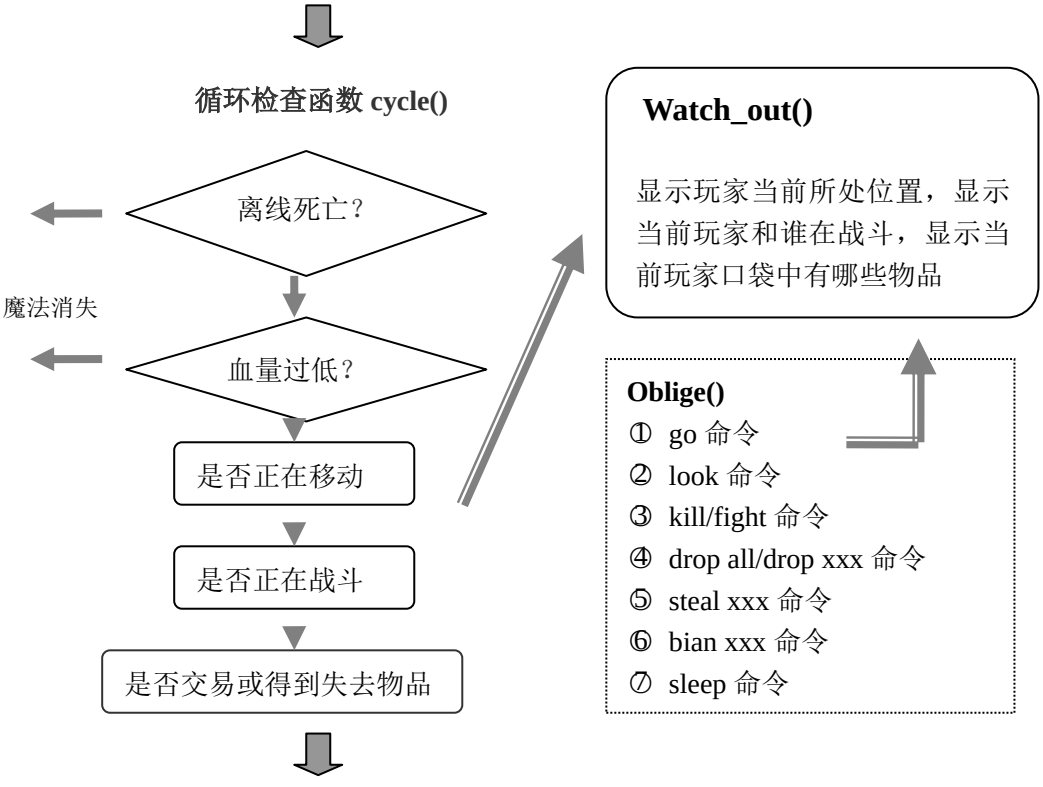


图 4-3 附体术具体流程

通过 for 循环添加入需要输出的显示。然后是关于 target 的物品交易问题，交易中出现的现象是金钱数减少，并且在包裹中出现相对应 value 的物品。当检查到物件数与最初保存映射集中的物件数量不一致时，自动把多出（缺少）物品的 value 和缺少（多出）的钱做总体对比。如果符合需显示给施法者相关信息，否则判定金钱与物件间没有直接关系，但同样需显示物件信息。

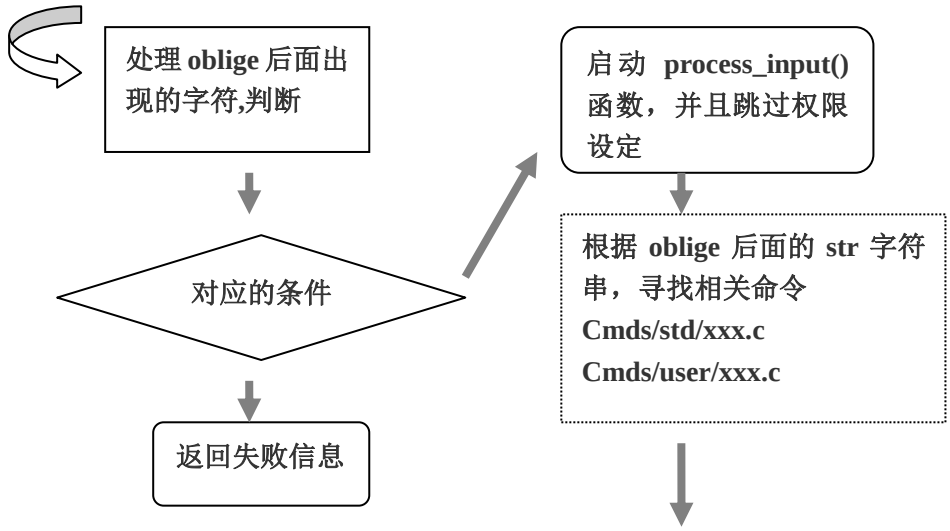


图 4-4　oblige()的处理方式

然后是讨论该术中最重要的功能，就是控制功能。施法者从 zmud 中输入 oblige go north 等的命令，target 则按照命令执行，同时施法者会显示相关信息或调用 watch_out()。那么关键的问题就落在了这些字符串是如何启动 target 做动作的。在 feature/alias.c 文件中的 process_input(string str)函数，就是玩家输入字符串命令的处理，但是表层的一些文件是无法直接调用该函数的，那么需对该函数做部分修改，设计中使用的方法是为玩家标记上一个映射变量，让函数运行中自行判断，并且跳过文件间权限的设定，直接处理字符串。

到此为止该魔法的主体思路讨论完了，鉴于魔法的合理性和游戏性，在施法者控制 target 前需经过一番考验，设计中设置了五脏六腑及其中的一些 NPC 关。在五脏六腑中打斗是会伤害到 target 的，但是施法者为控制 target 是一定要打这一仗的。但也有可能初级玩家并不知道其中的一些命令，所以在一些任务完成后会有相关命令的现实。在五脏六腑中，玩家如果什么都不做是会被胃酸冲出体内的。

4.4 封印术及召唤术

考虑到游戏中并没有领养宠物这一概念，所以借鉴了日本等游戏中的一些思路。但是

并没有完全类同的就是宠物，该宠物是转为战斗设计的，游戏中的特定场景才能利用封印术捕捉该类宠物。考虑到封印术是否可以对 NPC 或玩家有某些效果后，设计了封印术对玩家起到你死我亡的作用，但是为了控制保护游戏中的高级 NPC，该术对 NPC 没有任何效果。

这里开始讨论封印术主体实现方法：魔法启动前该施法玩家身上必定被设定过一个空的数组，用来给召唤术访问。数组中存储的是玩家曾因封印过的宠物。在程序的处理中，必须避免同一种类宠物(id)不重复再数组中存储。设计中利用了 mudos 提供的 member_array()函数，提取出已经存在该 id 的序数来做判断处理。每一次对不同宠物使用封印术会扩大数组。

借着是讨论召唤术，在前几章讨论的魔法系统里，命令 cast.c 文件中，对玩家在 zmud 中输入的 target 已经做了处理，让其输入时的字符串形式，变为 object 类型（说明表层魔法文件中的 cast()函数，它的形参已经被定为 object 而能是别的类型）。那么如果利用 cast()来写召唤魔法显然是不合理的，原因是：施法者在数组中包含的数据类型是字符串型。所以设计中决定编译一个新的命令来实现该效果。该魔法如果只输入召唤，则可以知道玩家当前的召唤状态。在检查完当前状态后，可以再召唤后加上召唤宠物的 id，那么实现对该宠物的召唤。

宠物与玩家间的对战和消失必定一些在每个宠物自身的文件中。如果玩家在战斗状态，那么玩家的 enemy({})必定有对手物件，利用这些，在宠物中编写 invocation()函数，函数的主要功能是对当前环境的敌人做一个 for 循环扫描，利用 kill_ob(enemy[i])函数，自动把对手加入宠物的 enemy({})进行对战。

当战斗状态消失时，利用底层提供的 heal_up()，实现宠物自动消失的功能。整个过程中不需要玩家对宠物进行操作，宠物会自动 follow 玩家(set_leader(me))，这些都符合游戏的合理性。

4.5 变鸡术

该魔法效果的思路来自于当前一些战略游戏，施法把强大的对手变成一只小鸡，小鸡会在一定得时间后恢复人形。变形的时间有计算程式比较双方等级差别来决定。既然是把对手变成小动物，那么在人物属性方面必须有所变化。由于人物的技能等级是由 set_skills 设定的，一旦改变在恢复人形后必定无法恢复技能值。所以必须构造出一个函数，既可以

用于保存技能值，又可以恢复进技能值。

本设计中构造了 copy_ren(object me, object ob)这样一个函数，函数的主要功能是对 me 物件的技能和属性进行读取，并且利用 for 循环逐个保存到一个映射集 giveme({})中，同时让该 ob 物件的技能属性，for 处理复制给 me。同时包括了 ob 的 name 和相关描述。这样在一个魔法释放后，对手的技能值大幅度降低，并且出现很多原有的技能消失的情况。规定时间过后调用返回函数 copy_ji(object ob,object me)，逐个删除之前复制过来的各种数值，并且清理 apply 映射。借着把映射集 giveme({})中的技能和其值返回给对手，实现了复原的效果。整个魔法充分利用了映射集这个数据类型。避免了利用数组类型后，需要两次提取关键词和数值的做法。提高了编程的效率。

4.6 火墙与火遁术

火墙术所达到的目的是对当前环境的生物进行封锁，让其中中术的物件在固定时间内无法离开此处。但是既然由此效果，必然需要平衡一下游戏性。所有中术的玩家或 npc 都会攻击实术者。所以该种法术转提供给高级玩家使用。游戏任务中经常会出现一些对手，他们在战斗时时常逃跑，虽然用 follow 命令能够跟踪他们，但是对战斗中的效果是大打折扣的。那么这样一个魔法对于高级玩家用于追杀，是极其有用的。

在用 for 处理当前环境的每一个玩家时，在 for 中同时启动 call_out 函数呼叫循环处理函数 cycle()。这样做是有好处的，Cycle 函数是一个对单个 object 做处理的函数，根据当前传入 object 进行处理判断。也就避免了环境中所有玩家定时游戏效果一样的不合理现象。另外循环处理函数利用 call_out 自身进行判断循环，决定对手的伤害值和相关移动数据。

火遁术是一个单体魔法，就是只能对一个对手施放，该魔法主要注重于对游戏效果的描述。施放该术后并不会立即产生效果，根据对方玩家与自身等级的差距计算呼叫函数的时间。函数根据想关判断决定是否需要伤害，并且决定是否会产生一些附加效果，比如对手是一个高级玩家，那么有相当高的几率出现反击。

4.7 残废术

残废术是一个针对武器效果所编写的魔法，在前面几章中讨论过武器给人物带来的附加属性的变化，其实武器本身的攻击力会影响战斗的变化。在 combated.c 中的 do_attack 函数里，武器是 4 项参数之一，在程序中需要访问对战中人物的武器，并且获取它的攻击

力来做判断处理。有无好的武器间接地影响着每次攻击的结果，所以设计中对该魔法进行了考虑。魔法针对魔术师门派不擅长战斗技能（如剑术、刀术等）做了设计，让施法者有短暂操控敌人武器的效果。等级越高的玩家操控敌人武器的时间必定越长。

该魔法会有两种效果：

1.如果施法者手中有武器，那么对手武器掉落。

2.如果施法者手中没有武器，则让他短时间操控对方武器。

显然这两种效果是符合游戏平衡性的。在程序的实现方法上，需要把对手的武器映射标记删除 target->delete_temp("weapon")，这样做是让 combate.c 中读取不到对手的武器。但是如果光删除标记，人物的属性仍旧是施法前的状态，也就违背了该术的意义。所以需要对人物的属性值处理，那么就利用装备章节讨论的函数 unequip()，来删除对手的附加属性。一定时间后，同样利用 wield()函式和 set_temp(“weapon”)做反处理，恢复对手原始状态。

4.8 洪水术

该魔法是利用水来让对手从自己面前消失，程序中根据当时玩家所在地形做出相关处理，然后判断应该把对手冲去哪里。在讨论房间的章节中，提到每个普通房间都设有”exits”这个映射集，集中的关键字数组表示该房间有几个方向的出口，对应的字符型表示出口对应的文件路径。那么洪水术正好利用了这样个一映射集，首先取出当前方面的 exits，而后取出关键字数组并且利用 exits(key[i])读取目的文件路径，考虑到游戏的合理性，并不能在无魔法无战斗的房间战斗，所以需要有一系列判断机制进行过滤，最后根据结果把对手冲向某个房间。

考虑到游戏性，洪水术一定会涉及到当前房间的每一个人，那么上述讨论的实现方法，应该译写在一个 for 循环中。并且被洪水冲走是肯定需要昏迷效果的，所以每次单个处理完玩家，应该实行 target->unconcious()。以上这些是实现该魔法的基本思路。

4.9 宝箱术

既然是魔术师门派，必定需要会变魔术。为门派定做了一个宝箱术，宝箱术与传统理解上一样，其中会变出 Mud 中的各种宝物。同样其中也会变出女人来。如果是男玩家同意女人跟随自己，那么时不时地会发现包裹中有东西消失。宝箱中出现的宝物同样有可能是假货。

在编写程序时，考虑到宝箱被置放在一个环境中，那么施法者和其他玩家都可以触发 add_action 中的命令，那么需要对其他随意使用命令的玩家进行处罚，游戏中用了一个类似黑箱的效果，把玩家丢如其中，让其只具有一个指令 out，但是 out 却又无法实现任何实际功能，只是返回给玩家类似“失败”字符串。当道一定时间后，函数检查如果玩家还在线，并且在宝箱中，那么自动丢出玩家。

4.10 游戏场景及任务

游戏的任务设计主要是围绕创立的魔术师门派和门派下的各类魔法进行的，进入大唐西游的游戏世界后，在长安城附近经常会遇见一个名为怪老头的 NPC，该 NPC 是为这个门派所创立的祖师级人物。首先玩家应该能够从老头那学到魔术师的各类法术，这里就需要有拜师的继承机制。也就是说游戏中的一些厉害的 NPC 都可以收徒弟，并且徒弟可以学到他们相同等级的魔法技能。其实这样一个游戏效果取决于程序中的一个自定义函数 attempt_apprentice(object ob, object me)，在这个怪老头的文件中，编写了该函数对收徒的条件加以删选，并且对拜师成功的玩家赋以门派的映射集。既然有拜师的函数，必定也需要一个退出门派的函数 expell_me(object ob)，函数的主要目的是清除玩家的映射集，但是其中可以自定义其他功能。在魔术师门派下如果想背叛的话，就会被剥夺相当多的经验点数以及失去一些先天的属性。这样是为了提高游戏的合理性。

NPC 怪老头的编写也集成了许多其他功能。魔术师魔法中的附体术是一个控制对手行为的可怕魔法，虽然成功的几率很低，但一旦附体后是不能摆脱的。怪老头具有解除这种魔法的功能。并且在游戏中，如果被附体的玩家靠近老头，他会自动上前询问是否要解毒，当玩家同意他时，可能会因为所带金钱不够而解毒失败；或者虽然解毒成功，但包裹里的所有值钱物品扫荡一空。考虑到被附体的可能是本派弟子，所以该老头如果碰见这样的情况会传授他驱魔大法（课题设计中的另一种魔法效果），根据魔法的名字可以知道它大概的功能。但是光会给自己驱魔治疗肯定是远远不够的，所以必须对魔法的功能进行扩充，是它对任何一个人都可以该效果。这样游戏中的相关交易必定悠然而生，原因很简单：怪老头的治疗费用昂贵，而玩家可以收取部分金额施展驱魔。

老头是一派的掌门，那么必定存在高级玩家想 kill 它，根据这样一个游戏逻辑而编写了 die()函数。当情况发生时会同时出现许多护法卫士保护老头，保护的功能是由编写在卫士文件里的 invocation()函数完成的。怪老头还会根据门派弟子的需要，毫不费力地送他们

去任务地点。对于这样一个游戏功能，主要是针对游戏任务的。

在游戏场景设计时，必不可少的元素就是迷宫。那么下面会简单分析几种迷宫的实现：

首先是物理连接的迷宫，在讨论 room 文件时提到的房间映射 exits，是连接各个房间的映射集，根据对应路径寻找房间。物理迷宫利用混乱匹配房间路径便可以简单的达到迷宫效果。但这样的迷宫并不能在增加游戏性。并且多出了一大堆类同的简单文件。

其次是利用延迟呼叫来实现迷宫效果，这样的房间，exits 映射中所有的文件路径都是它自己，那么玩家怎么走都是在该房间中。但当呼叫函数时间到并触发时，利用 me->move(“xxx”)让玩家移出迷宫。

第三种迷宫是功能最多的一种。同上述第二个迷宫一样，它的映射集同样都是自身。在玩家每次键入命令离开该房间时，启动 valid_leave()函数，函数的功能可以是任何自定义的效果，设计中多次利用了这样的模式。比如在大沙漠中玩家肯定是需要消耗喝水的，在多次移动中，玩家的水分必将减少，时不时地脱水最后会导致昏迷 unconcious()。迷宫的作用是让玩家在其中迷路，在 valid_leave()中采集玩家命令，当错误方向命令过多时会对其惩罚，并且传送到错误的位置。这里概要的说完了迷宫的三种实现方法。

回到任务的设计中，当玩家通过各种考验任务来到 NPC 使者那里时，它会根据玩家之前的任务完成情况提出相关问题，或者是决定试一试你的武功。程序文件中的函数主要完成三个目标，首先是处理当前玩家的各种映射集，并且判断是否需要比武。其次对玩家 answer 的命令处理分析，然后决定该做什么。最后调用比武函数，函数中利用了战斗循环系统的 combat.c 来完成效果。

「**悬崖小路**」 - /d/lingjie/xuanya2

小路一直往北面延伸，似乎那里不是那么的陡峭。

靠近左右两边能看到下面的激流(river)，但这样很危险

这里明显的出口是 south 和 north。

> look river

你探头向急流望去。

水流相当的急，但是你却很好奇想靠近观望下(approach)。

> approach

你走近崖边探头向下望去。哎呀！一不小心掉了下去！

你的头撞在悬崖壁上，一下子昏了过去！

你的眼前一黑，接着什么也不知道了...

图 4-5 游戏场景展示

在玩家完成后续任务后，会最终来到悬崖处。此处故意设置了诱惑障碍，玩家一不小心可能就掉下悬崖，并且根据计算玩家的各项数据决定其是否昏迷。如图 4-4 便是其中的一种情况。

通过这样几重考验后，玩家面对的是终极怪物，设计中对该怪物定义了较高的武力，可以说该任务是针对高级玩家的。怪物的文件中，定义 void init()函数呼叫自动攻击函数 hurting()，处理各种战斗情况。如果这些都顺利完成的话，会在外物的尸体旁发现一本秘籍，秘籍的作用当然是教会玩家关于魔术师门派的终极魔法封印术。鉴于网络游戏，肯定存在对该书的倒卖现象，所以要从程序上加以限制。只允许该门派的弟子查阅，并且查阅后书便自行销毁 destruct，如果贩卖后，他派弟子翻阅并不会有任何效果，并且书从此消失。

刚才提到在悬崖处的游戏效果，玩家不慎掉入谷底其实是另一个游戏场景的出发点，在谷底的礁石附近会时不时的有涨潮和退潮，如果玩家没有在设定时间内离开，便会不断地被潮水击伤。这样一个效果在玩家刚进入时，void_init 函数就启动了，所以不会出现 bug（玩家不受到潮水影响）。

经过这样一个场景后，会来到设计中的一处断桥，此断桥的游戏效果是：一定时间内桥面由于风大而不能通行，一定时间内又畅通。当有新的玩家在进入该场景时，必须对场景调用 upset()函数。刷新该时间点的断桥合拢或断裂。保证每位玩家在游戏中看到的描述是一致的（防止一位玩家可以通行，而另一位看到的是无法通行）。

> **study fyshouji**
> 如果你是魔术师的职业，那么请你获取技能(getfy)。否则这本书对你没有一点意义。
> > i
> **你身上带着下列这些东西(负重 1%)：**
> 封印手记(Fyshouji)
> ☆粗布衣(Linen)
> > **look fyshouji**
> 一本薄薄的《封印手记》，扉页上写着：
>
> 封印术奥义
>
> > **getfy**
> 你已经学会了封印术了。

图 4-6 游戏效果展示

通过断桥是可以来到一处原始森林的，该森林专为某些魔法设计，在森林中随时都游荡着一些魔兽，并且这些魔兽本身就是可以被封印的。该设计考虑到魔术师门派没有专精打斗类魔法，所给其强大的召唤术，用来帮助战斗。同时在森林某些地方也与前面提到的礁石相连，关于密室等命令应该流传玩家之间，如果直接显示便会影响这个游戏的娱乐性。

Help 文件中添加了新的游戏地图，并且对门派下各种武功描述做了详细介绍。不过其中并未涉及类似密室等的命令。

讨论到这里，鉴于场景涉及的内容较为丰富，一一详细叙述会造成篇幅过多、重点不明确。所以采用如上的方式。课题设计中的部分成果和过程已做出概要说明，叙述程序的主要思路、主要实现效果，并且对场景做简要介绍。

第五章 课题总结

5.1 对毕业设计的小结

在毕业设计的初期，我查阅了大量的游戏程序网站，发现关于该课题的专业教材少之又少，造成了我一时间找不到入门的技巧。而后在玩该游戏的过程中，逐渐的了解了这是一个怎样的游戏，并且开始自学游戏中已经有的一些文件。通过程序和游戏中内容的对比，不断地实践，慢慢地掌握了游戏中常用的底层函数。在大量地编写任务场景时，学习游戏中已经创立的某些经典任务。掌握了游戏构架最底层的许多知识点，并且利用他们构架起了自己的魔法系统、魔术师门派等。达到了课题设计的目标。

在短短的 3 个多月中，学习并且设计了游戏程序，对于学习通信工程的我也算是一个不小的挑战。我们学院涉及程序方面的课程有 C 程序设计和数据结构等几门课。但是一贯比较注重硬件的我们，软件这一块知识仍旧是一个软肋。这样的一个课题，有效地帮助我了解了整个网络游戏和其构架，并且掌握了 LPC 语言。我相信这样一次设计经历，对将来个人的发展也是大有益处。

5.2 课题未来的发展方向

MUD 是一种支持网络化学习环境和学习团体的计算机通信技术，是一种以客户机/服务器模式运行的计算机程序。MUD 起源于一种在网络上流行的多用户角色扮演类交互式游戏（Multi-userDungeons），而目前已经逐渐从多用户的交互式游戏，发展成为一种开放性的、广泛参与的合作学习与合作工作系统，多数非游戏 MUD，都是 MU 技术在学术领域和教育领域的应用。许多 MUD 将某些特定领域的人群联系成为虚拟的团体，进行合作的学习、研究与讨论，还有一些 MUD 的教育应用，不仅体现了 MUD 作为合作学习/工作环境的作用，也体现了 MUD 作为通信工具（CMC）的作用。如开展远程教学，目前有多个在因特网上开展远程教学的教育机构将 MUD/MOO 作为其重要的组成部分。

参考文献

[1] 谭浩强，C 程序设计，第二版，北京，清华大学出版社, 1999 年：38-65

[2] 谭浩强，C++程序设计，北京，清华大学出版社, 2004 年：19-85

[3] 沈被娜，计算机软件技术基础，第三版，清华大学出版社，2001

[4] 严蔚敏 数据结构，第二版， 清华大学出版社， 1994

[5] Ron Penton, MUD GAME PROGRAMMING, USA, a division of Course Technology, 2004

[6] Mudos 函数手册**,** www.mudbulider.com**,** 2000

致谢

关于这次毕业设计，首先要感谢沈文辉老师。沈老师不仅在知识学术上态度严谨认真，而且在教导我们的时候更显得耐心仔细。在过去的几个月里，沈老师在繁忙工作的同时，抽出大量时间为我们解答关于游戏设计中的相关问题。 在每次与老师讨论了设计课题之后，都对游戏设计的概念有了进一步的思考。学会了怎样利用工具、怎样学习专业程序员的编写思路、以及如何提高自己在游戏中的创新性。通过这一阶段学习后，从根本上了解了一个网络游戏程序员的工作，以及相关需要学习的知识。弥补了通信专业学生在游戏编程知识方面的不足。

最后，感谢所有在毕业设计中曾给予我鼓励、关心和帮助的老师、朋友。

附录 A

MUD GAME PROGRAMMING

Ron Penton
Department of computer science
The State University of New York at Buffalo

Logic Modules

Logic modules in the BetterMUD are simply Python scripts that can be attached to any entity type. Every entity has the capability to be given any number of logic modules, which allows you to mix and match behaviors.

For example, in my version of the BetterMUD, I have character logic modules named "combat" and "encumbrance." Chapter 18, "Making the Game," shows you how to implement these, so for now all you need to know is that when characters have these modules, they have the ability to attack other characters, and they have the ability to weigh the number of items a character is carrying. Now, if I take away the "combat" module from any character, that character cannot be attacked or attack anyone else, because the logic module is what gives the character those abilities. Likewise, if I take away "encumbrance," the game happily allows your characters to carry an infinite number of items without weighing them to see how much they can carry.

A cool thing about logic modules is that they use a flexible set of attributes that you can access from within C++. This means that from C++, you can ask a logic module to get a 32-bit signed value based on its name. I'll show you how this works a bit later on when explaining character quests.

Overall Physical Design

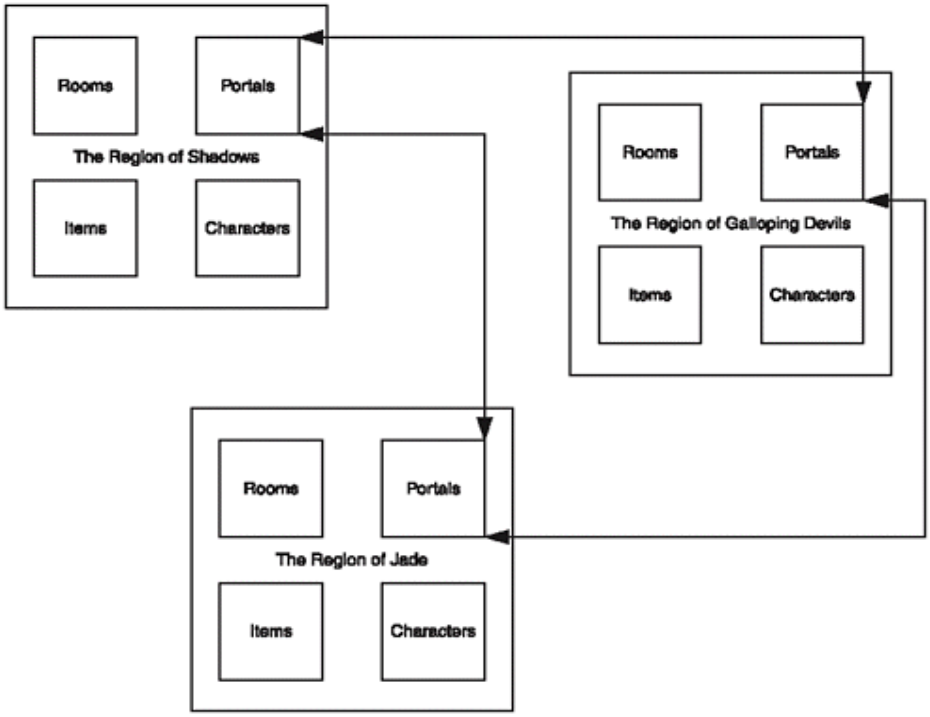
As well as having a completely flexible logical design, the BetterMUD has a flexible network layer, because it is more abstracted from the game than the SimpleMUD's network layer.

Like the SimpleMUD, the BetterMUD focuses on the idea of entities, so I won't need to spend much time going over entity concepts. In the BetterMUD, however, there are different kinds of entities.

Regions

The first major change from the SimpleMUD to the BetterMUD is the addition of regions. Most MUDs have regional systems, which allow you to organize your game more easily. Figure 11.3 shows a simple three-region layout. Regions also make it easy to group logic, and they ease the strain on your auto-saving system.

Figure 11.3. *Regions make it easy to group entities.*



Regions and Scripts

To understand regions, imagine a collection of rooms in the magical forest of the realm. Whenever evil monsters such as orcs and goblins enter the forest, they receive a curse that slightly lowers their stats. Without regions, there's really no easy way to do this. To make this work, you give a region the logical actions "character entered" and "character left," which are executed whenever characters enter or leave. So when an evil monster enters a magical forest, the magical forest's logic module curses the character, and when the character leaves the forest, the curse is removed. This is a nice, elegant system of implementing a large collection of scripts in just one area, instead of putting them in every room that is in the forest area.

Regions and Databases

Regions make things easier for the database, too. Imagine a large game, with thousands, or even tens of thousands of rooms. Whenever your game tries to do a complete database dump, it takes a long time, and the game is going to lag up for a second at least, or maybe much longer.
To prevent the lag, you may want your game to save one region at a time, splitting up the job over a long period of time, so that the game doesn't lag.

Regions and Data

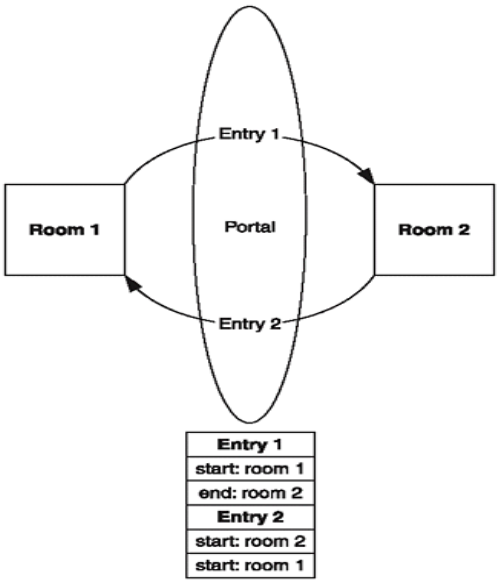
Regions are simple entities. They need to know only the basics: their name, ID, and description, as well as a logic module, and lists of all entities contained within the region.

Rooms

In the BetterMUD, rooms are similar to and simpler than SimpleMUD rooms. In the

BetterMUD, rooms are no longer involved with money on the ground, simply because money isn't a special case object. Neither do they deal with tracking who respawns in the room, because this functionality is provided in the logic modules you implement instead.
The physical aspect of a room in the BetterMUD deals only with a few things: room name, description, ID, associated region, exits from the room, characters in the room, and the items on the floor.

Figure 11.4. *Rooms point to portals that control the entry of players.*



Portals

Portals are a new concept in the BetterMUD. In the SimpleMUD, each room simply had four exit IDsthose are the IDs of the rooms attached to that room. In the BetterMUD, it's not that simpleevery room has a list of portals, which are basically structures that describe a path from one point to another. Every portal has one or more entries into it, as shown in Figure 11.4. Rooms are never explicitly linked; instead, they point to a portal, and the portal manages the entry of a player to different rooms depending on which room they entered the portal from.

Characters

In SimpleMUD, there were players and enemies; therefore. it was awkward for players to fight each other without copying large amounts of code, which is always a sign of a weak design. In the BetterMUD, I've unified the concepts of players and enemies into one entity type: characters.
Basically, a character is any living being in the game. Characters can hold stuff, move around, see things, and die.

Attributes

Characters have attributes, much like the attributes in the SimpleMUD, but instead of hard-coding these attributes, you're going to be able to access them via strings. This is because characters in the BetterMUD are somewhat flexible. You load the attributes from a text file on disk whenever the MUD starts, and the scripts and everything else have access to those variables.

Absolutely none of the attributes are hard-coded in the BetterMUD. This was designed to give you complete flexibility over what you want to do with the engine. I'm not going to say "players must have health and hitpoints!", because quite frankly, you may decide not to have those kinds of things in your MUD. When I get to Chapter 18, you'll see how this whole thing works out; believe me, it's cool.

Containers

Characters need containers for various things, such as for a list of items that the character is currently carrying.

Characters also have a collection of logic modules (in fact, all entites have these), and a special collection of logic modules named command modules. Command modules allow characters to interface with the game. Every character can have a personalized collection of commands, so you can do things like giving blacksmiths a "repair" command to repair broken itemsan ability other characters won't have.

Conditions

Characters will have all sorts of logic modules, and you can use them for varying purposes. You could make logic modules represent conditionsstuff like "on fire" or "poisoned." With this kind of system, you can say, "If a character has the logic module on fire, that character is actually on fire!"

Conditions are usually time based, which means that they last for a certain duration, or that they have an event that repeats. (For example, the "on fire" condition logic would take off X hitpoints every second until it burned out.)

Items

Items are the final entity type in the game, and they are basically anything inanimate. Items are always owned by a room or a character, and the items know who owns them.

Items have a listing of attributes, which corresponds to the attributes the players have. As with characters, there are no default item attributes; you customize everything. I show you some recommended values in Chapter 18.

There are two main types of items in the game; single items and collections of items. Single items are always single items; they typically represent large items such as weapons and armor, or even smaller stuff such as potions and scrolls.

Collections of items, on the other hand, are single items that have a count value, and thus act like many of the same item. Only certain types of items can be combined into collec-tionsmostly things such as coins, jewels, diamonds, and maybe even weapon-type stuff such as stars that can be thrown, or arrows for a bow.

Collection objects can be split, but only by dropping them, giving them to other characters, or picking them up. Whenever a collection object enters a new domain (either a room or an inventory of a character), the object automatically merges with other collection objects of the same type. Strictly speaking, splitting doesn't have to work this way, but it's easier on the interfaces if it does.

MUD 游戏编程

Ron Penton 著

布法罗纽约州立大学 计算机科学学院

逻辑模块

在 BetterMud 中的逻辑模块只是一些 Python 脚本，它们能够连接到任何实体类型。每个实体都能够被赋予任何数量的逻辑模块，允许游戏中的行为进行混合匹配。

例如，在我的 BetterMUD 形式中，有关于“战斗”和“负载”的人物特性逻辑模块。第 18 章将展示这些工具的使用，现在只需知道：什么时候角色有该模块，从而使他们有能力攻击其他人物并且拿起相应重量的物品。现在，如果我拿走某些人物的战斗模块，那么他们将不能够被攻击或者攻击他人。原因是战斗模块就是给予人物那些战斗能力。同样的道理，如果我拿走了负载模块，那么游戏将允许你的人物拿起任何重量的物品，而无需对其重量做出限制。

关于逻辑模块一件比较酷的事情是，编程中使用了一种灵活的属性设置，以至于你能够用 C++访问。意味着从 C++就可以要求逻辑模块根据其名称获取 32 为符号值。我将在解释人物角色特性时展示其工作原理。

物理特性的总设计

除了拥有一个完全可变通的逻辑设计，BetterMUD 还有相对较为灵活的网络层，因为 BetterMUD 比起 SimpleMUD 更为抽象。

如同 SimpleMUD，BetterMUD 关注与实体这一想法。因此我将不用花很多的时间解释实体的概念。但是在 BetterMUD 中有不同类型的实体。

区域范围

从 SimpleMUD 到 BetterMUD 一个最重大的变化就是增加了区域的概念。

多数 MUD 有区域系统，允许你更容易地管理你的游戏。如图 11.3 展示了一个简单的三区域布局设计。区域概念让逻辑分组变得简单，同时这样一个概念也减轻了自动存储系统的压力。

Figure 11.3. *Regions make it easy to group entities.*

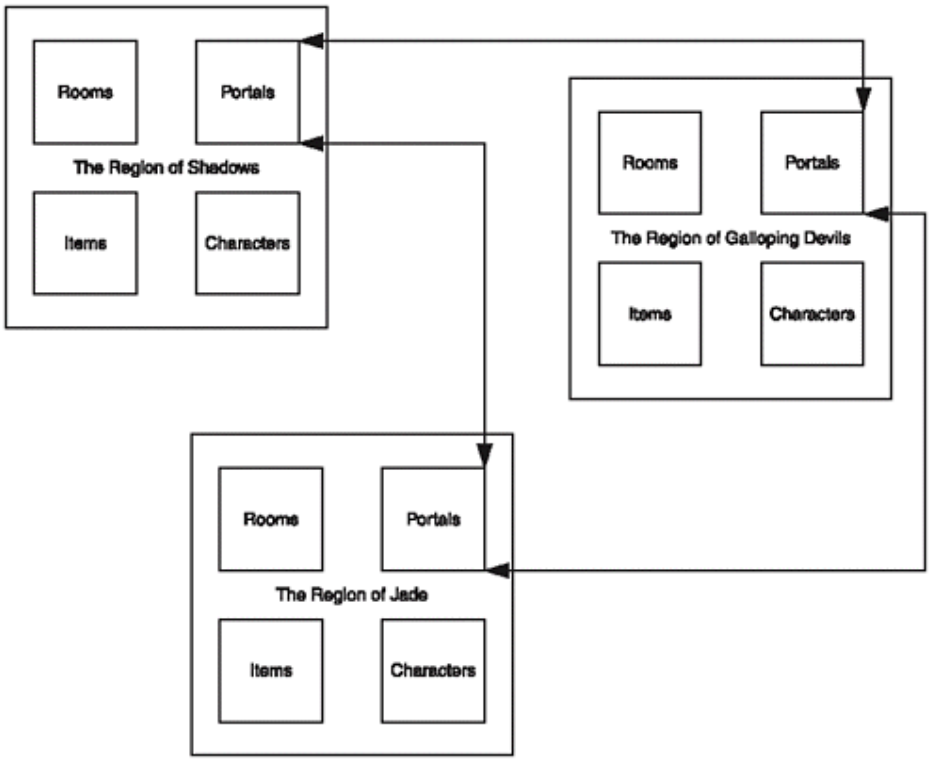


图 11.3 区域轻松地对实体的分组

区域和脚本

为了理解区域，想象一个位于魔法树林的房间集合。每当邪恶的怪物进入树林时，他们就会遭到诅咒并且轻微减轻他们的状态数据。如果没有区域的话，实际上不容易做到这点。要完成这一任务，为了一个区域给角色进入和角色离开的逻辑动作，他们分别在进入和离开时执行。所以，当邪恶怪物进入魔法树林时，树林的逻辑模块就会诅咒该角色，当此角色离开树林市，则停止诅咒。这一系统非常适合，它只在一个区域中实现一个大脚本集，而不是将他们至于树林区域中的每个房间。

区域和数据库

对于数据库，区域也让处理变得简单起来。设想一个具有成千上万房间的大型游戏。每当游戏试图存储数据库是，都需要花费大量的时间，那么游戏至少会滞后 1 秒，也许会滞后更长时间。

为了阻止这种滞后，可能要游戏一次只保存一个区域，将任务拆在比较长的时间里来完成，这样游戏就不会滞后。

区域和数据

区域是简单的实体，他们只需要知道基本情况：他们的名称、ID 和描述，以及逻辑模块和包含在区域中的所有实体列表。

房间

在 BetterMUD 中，房间相似于或者说更简单于 SimpleMUD。并且房间也不再包括地上的金钱，因为金钱不再是一个特殊的物件。同样不再处理跟踪房间里的大量繁衍，因为这些功能有逻辑模块来完成。

BetterMUD 中，关于房间物理方面的特性，只处理如下几件事：房间名称、描述、ID、相关区域、房间出口、房间中的角色以及地板上的物品。

Figure 11.4. *Rooms point to portals that control the entry of players.*

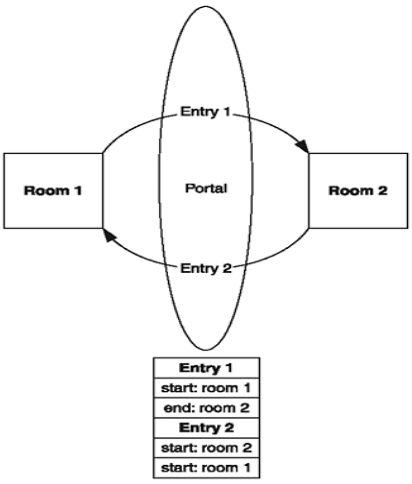


图 11.4 控制玩家条目的入口

入口

入口在 BetterMUD 中是一个新的概念，SimpleMUD 中，每个房间只有 4 个出口 ID，他们是与此房间相连的房间 ID。而在 BetterMUDZHONG ，则没有这么简单，每个房间都有一个入口列表，入口实质上是一种数据结构。他描述从一个点到另外一个点的路径。每个入口中都有一个或者多个条目，如图 11.4 中，房间之间从不会明确的连接在一起，相反，他们指向一个入口，此入口管理玩家到不同房间的条目，这种管理取决于他们从哪个房间进入入口。

人物角色

在 SimpleMUD 中有玩家和敌人，如果没有复制大量的代码，那么处理玩家与其他人物战斗是非常不灵活的。这样的情况通常算一个比较糟糕的设计。BetterMUD 中，将玩家和敌人统一成一种实体类型：角色。

从本质上来讲，角色是一个能够互动的生物。角色能够持有东西、走来走去、看见东西以及死亡。

属性

角色有属性，与 SimpleMUD 中相似，但不是对这些属性进行硬编码，而能够借助于字符串来访问他们。这是因为 BetterMUD 中的角色比较灵活。当 MUD 启动时，则从光盘的一个文件加载属性，脚本和其他所有东西都有权访问这些变量。

在 BetterMUD 中绝对没有硬编码属性。这种设计使我们能够完全灵活地使用引擎完成需要的事情。不会发生“玩家必须健康和有生命点数”。因为坦诚的讲，有可能决定在 MUD 中没有这类事情。在第 18 章中会看到这一切是如何解决的。

容器

各种人物的事件都需要有容器。（当前角色正在运送物品的列表）

角色有一个逻辑模块集，特殊的逻辑模块集——命令模块，它是角色和游戏的接口。每个人物角色都可以有一个个性化的命令集，因此可以完成类似这样的事情：给定铁匠一个“repair”命令来修理损坏的物品。这一能力是其他角色所不具备的。

条件

角色有各种各样的逻辑模块，可以将他们用于各种不同的目的。可以使逻辑模块表示条件——如“on fire”或者“poisoned”这样的条件。有了这种系统，可以判断：如果一个角色拥有了“on fire”模块，那么他实际上着火了。

一般情况下条件是基于时间的，这就意味着它们持续一定的时间，或者他们有一个重复发生的事件（例如，“on fire”模块每秒取走 X 点生命值，直到全部取完为止）。

物品

物品是游戏中最后的实体类型，它们是没有任何生命的东西。物品总是被房间或角色所拥有，并且物品知道谁拥有它们。

物品有属性列表，与玩家所具有的属性相对应。与角色一样，没有默认的物品属性。你可以定制任何一件。第 18 章中会展示推荐值。

游戏中有两种物件类型：单个物件和物品集。单个物件总是单个的物件，一般情况下，它们表示大型物品如武器盔甲等，或者小东西比如药物和卷轴等。

另外，物品集是一个拥有计数值的简单物件，于是它的作用如同许多相同的物品。只有某些物品类型能够组合成集合——主要是如金币、宝石、钻石等，也可能是诸如能够抛

出去的星状物，或者是弓箭等武器类。

集合的物件是可以被分离，但只局限于丢下它们；把它们交给其他角色；或者是拾起它们。当一个集合对象进入新的区域时，对象自动将其他相同类型的集合对象整合在一起。严格的讲，分离不一定要采用这样的方法，但是如果这样做，便利于地处理不同程序语言之间的接口。