LPC 基礎教程

作者：**Larkin**
日期：**2004-11-01**

目 錄

MUD 是一種網路遊戲，是英文 Multiple User Dimension、Multiple User Dungeon 或 Multiple User Dialoguede 縮寫，可以翻譯爲多人世界，多人地下城或多人對話，我們俗稱爲“泥巴”。

MUD 的基本部分是運行於 UNIX 系統上,其實現編譯語言就是將要介紹給您的 LPC 語言。其實，如果您已經對 UNIX 的一些基本指令有所瞭解的話，exp:ls,mkdir,rm,mv,cp 等等，那麼您應該知道如何進入一個 mud 中的文字編譯環境，編一個小程式並存儲它了。這裏還需要說明一下，那就是 lpc 編譯出的語言在結構上非常類似於我們所學的 C 語言編譯程式,但是 lpc 與 C 語言還是有相當的差距的。總之,我想說明的意思就是關鍵是要掌握編程的核心,即編譯原理,知道了原理和方法,程式指令是死的,只要多下工夫,是能夠弄好弄精的。這裏,我想通過初級篇和進階篇的一些內容來向您說明 LPC 以及怎樣運用這門語言,不過程式是要多實踐的,光通過材料來學習還是不行的,多看看代碼,多編些東西,多動手實踐一下就會越來越熟悉它了。

本教程選取的是 Descartes of Borg 於 1993 年編寫的基礎 lpc 和中級 lpc 教程的內容.在這些教材的基礎上我做了一些改動，以供您更方便容易的學習 LPC 這門語言.

第一章: 編程環境基本介紹

1.1 UNIX 基本結構

Mudlib 編程語言 lpc 使用的是基本的 UNIX 命令及檔案結構,因此我們有必要想瞭解一下 UNIX 方面的知識和相關內容.與我們常用的 DOS 系統一樣, UNIX 也使用階層式的目錄結構. 所有的次目錄都附屬於根目錄之下. 而每個次目錄之下同樣可以有更多的次目錄. 一個目錄可以有兩種表示方法:

1) 用目錄的全名 (full name), 或稱作絕對名稱 (absolute name).

2) 使用相對名稱 (relative name).
絕對名稱就是從根目錄一路寫下來, 直到該目錄的名字爲止.exp:/daemon/skills/sword.c 就是根目錄下 daemon 目錄下的 skills 目錄下的 sword.c 程式.相對名稱使用的是相對於其他目錄的名字. 以上面的例子來說, 相對於/daemon,存在目錄 skills,不難得出,絕對目錄是從根目錄開始的,而相對目錄則靈活的多,隨便一個幾極的子目錄都可以成爲確定另一個目錄的相對根目錄.這裏還要確定一下,上述舉例中的/daemon/skills/我們都稱爲路徑,而 sword.c 就是我們所說的檔案名。怎麼樣,和 DOS 差不多吧,其實樹形目錄結構是很多系統共同的部分。

1.2 UNIX 基本命令

LPmud使用許多UNIX的指令,比較典型的指令有:ls,cd,rm,mv,edit等等.這些指令對於我們維護和使用 LPC 的編程環境是非常有用的.這裏先舉一些常用的指令供您參考：

pwd 顯示你目前所在的工作目錄.

cd 改變你目前的工作目錄.

ls 列出一個目錄裏面所有的檔案.(相當於 DOS 中的 dir)

rm 刪除指定的檔案.(相當於 DOS 中的 del)

mv 更改指定檔案的檔案名.(相當於 DOS 中的 rename)

cp 拷貝一個檔案到指定目錄的命令.(相當於 DOS 中的 copy)

mkdir 建立一個新的目錄.(相當於 DOS 中的 md)

rmdir 刪除一個目錄,不過該目錄必須是空目錄.(相當於 DOS 中的 rd)

more 如果一個程式太長了,使用這個指令可以分頁顯示該檔案代碼.(相當於 DOS 中的 type|more)

edit 進入編程模式的指令.

從上面的初次接觸中我們不難發現,無論是環境還是指令和我們常用的系統都是差不多的，原理是一樣的。

第二章: LPC 程式及其資料形態

2.1 程式特點

我們使用 LPC 編寫程式的所寫的內容可以統稱爲物件(objects).一般來說,我們運行一個程式的時候,是有開始和結束的.換句話說,就是所有的程式開始執行的時候,總有一個開頭的地方

和結束的地方,程式執行後就終止了.而 LPC 的程式不同，整個 mudlib 的 driver 系統運行的是我們用 LPC 編寫出來的一個個程式,這些程式在不同的時間和情況下被不斷的調用,雖然都是運行程式，但是 LPC 的程式在 mudlib 中是不存在絕對的觸發點和結束點的。這一點需要我們注意。

2.2 程式的即時性

本來整個 mud 遊戲可以全部用 C 語言來寫.這樣遊戲的執行速度將會快上很多, 然而這樣卻讓 mud 缺乏可塑性,使巫師在遊戲正在執行的時候無法即時加入新東西.DikuMUD 就是全部用 C 語言寫成的.而 LPMUD 的理論就是 driver 不該決定遊戲內容,而遊戲內容應該決定於遊戲中的個別事物,並能夠在遊戲執行時加上東西.這就是爲什麼 LPMUD 使用 LPC 程式語言.它能讓你用 LPC 定義遊戲內容,交給主運行系統根據需要讀取並執行.當我們用 LPC 寫了一個程式(假設是用正確的 LPC),當你上傳(send)成功後,一旦遊戲中的東西參考它，他就會即時發生作用.這就是使用 LPC 語言做到的即時性編程效果.

2.3 電腦是怎樣認識程式的

我們使用的任何一種編程語言,電腦都不能直接接受它，必須通過轉化才行.電腦語言是由 0 與 1 組成的一系列排列有序的代碼組成的.而我們所用的 BASIC、C、C++、Pascal 等等,這些電腦語言全都是過渡語言. 這些電腦語言讓你能把想法組織起來, 讓思考更易轉換成電腦的 0 與 1 語言.而轉換是通過我們常掛在嘴邊所說的編譯來實現的.不過對於不同類型的資料，電腦把他轉化過來存儲起來的時候將不是直接的 0 和 1 的形式,就是說,每一個 LPC 變數都有變數型態指導如何轉換資料.比方說我們事先說明 int x,就是說明 x 是一個整形的量值，通過對資料形態的說明，可以讓電腦明白每組 0,1 系列的資料的明確意思.

2.4 LPC 的資料型態

LPMud driver 具有以下的資料型態:
void,status,int,string,object,int *,string *,object *,mixed *有一些資料型態是我們經常使用的,具有非常重要的作用: float,mapping float *,mapping *.現在我們所用的LPmud的driver一般都是MUDOS，其中的資料形態和上面所例舉的差不多.

第三章: 函數 (functions)

3.1 什麼是函數?

同任何函數一樣,LPC 函數獲得輸入值,然後返回輸出值.記得 Pascal 語言是把過程(procedure)和函數區分開來.LPC 並不這樣做,採用另外一種形式來區分.Pascal 稱爲過程的東西,在 LPC 就是無返回值(void)型態的函數.也就是說,程式或無返回值函數不傳回輸出值.Pascal 稱爲函數的東西,就是有傳回輸出值的.在 LPC 裏,最短的正確函式是:void do_nothing(){},這個函數不接受輸入,沒有任何指令,也不傳回任何值. 要寫出正確的 LPC 函數有三個部分:
1) 宣告 (declaration) 2) 定義 (definition) 3) 呼叫 (call) 就像變數一樣, 函數也要宣告. 這樣一來, 就可以讓我們的 driver 知道:
1) 函數輸出的資料是什麼型態
2) 有多少個輸入的資料以及它們的型態爲何.
比較普通的講法稱這些輸入值爲參數.所以,宣告一個函數的格式如下: 返回值型態 函數名

稱 (參數 1, 參數 2, ..., 參數 N);這樣是形式我們在很多編程語言中都遇見過，所以是不難理解和運用的.需要說明一點的是: 哪一個函數定義在前都沒有關係. 這是因爲函數並不是由前往後連續執行的. 函數只有被呼叫時才會執行. 唯一的要求是, 一個函數的宣告必須出現在函數的定義之前, 而且也必須在任何函數定義呼叫它之前.我們不妨舉出 write_vals() 和 add()兩個函數的例子，僅供參考:

```
/* 首先, 是函式宣告. 它們通常出現在物件碼的開頭. */
void write_vals();
int add(int x, int y);
/* 接著是定義 write_vals() 函式. 我們假設這函式將會在物件以外被呼叫. */
void write_vals()
{ int x;
/* 現在我們指定 x 爲呼叫 add() 的輸出值. */
x = add(2, 2);
write(x+"\n"); }
/* 最後, 定義 add() */
int add(int x, int y)
{ return (x + y); }
```

3.2 外部函數 (efuns)

也許你已經聽過有人提過外部函數.它們是外部定義的函數.外部函數是由 mud driver 所定義.如果您已經編寫 LPC 程式碼很久, 那麼實際上您已經接觸到了很多函數,exp: this_player(), write(), say(), this_object()...等等,這些看起來很像函數的式子其實就是外部函數.外部函數的價值在於它們比 LPC 函數要快得多, 因爲它們是事先就以電腦可以直接讀取和運行的二進位碼的格式存在著.這些外部函數是早就被定義和宣告好的內容，需要時您只需要直接呼叫調用它們就可以了.

創造外部函數是爲了處理普通的、每天都需要使用到的函數呼叫、處理 internet socket 的輸出與輸入、其他用 LPC 難以處理的事情的完成和實現.它們是在 driver 內以 C 寫成的, 並與 driver 一起編譯在 mud 開始之前, 這樣它們執行起來會快得多. 但是對您來說, 外部函數呼叫就像對您的函數呼叫一樣,它們還是需要知道兩件重要的事: 1) 它的返回值是什麼, 2) 它需要什麼參數.

外部函數的詳細資料及其形態,可以在你的 mud 中的 /doc/efun 目錄找到. 不過因爲每種 driver 的外部函數都不相同,所以無法給出一個詳細的類型表分類,不過,你可以通過「man」或「help」指令 (視 mudlib 而定) 找到詳細的資料. 例如指令「man write」會給你 write 外部函式的詳細資料.當然,「more /doc/efun/write」也可以.

3.3 如何定義你自己的函數

雖然在檔案中, 函數次序的先後是沒有什麼關係的, 但是定義一個函數的程式代碼的先後順序卻非常重要.當一個函數被呼叫時, 函數定義中的程式代碼按照出現的先後順序執行. 例如在 4.1 中的 write_vals() 中, 這個指令:

```
x = add(2, 2);
```

如果你想看到 write() 使用正確的 x 值, 就必須把它放在 write() 呼叫之前.

函數要返回一個值時, 由「return」指令之後跟著與函數相同資料型態的值來完成返回. 在先前的 add() 之中, 指令「return (x+y);」把 (x+y) 的值傳回給 write_vals()並指定給 x.也就是說「return」停止執行函數, 並返回程式代碼執行的結果給呼叫此函數的另一個函數.另外,它

將跟在它後面任何式子的值傳回呼叫的函數. 因此,要停止執行失去控制的無返回值函數, 使用 return 就可以了; 而後面不應加上任何東西. 這裏需要提醒您一點的是,使用「return」傳回任何函數的資料型態必須與函數式本身的資料型態相符合,這是一一對應的關係.

第四章: 基礎的繼承

4.1 從一個小程式開始

我們來看下麵這個小程式，它實現的是一個房間：

```
inherit ROOM;
void create()
{set("short","日月園");
set("long", "這是江南的工作室,其實就是一個盛開著各種花卉的小花園\n");
set("exits", ([ "down":"/d/xyj/kezhan", "north":"/u/power/doorroom", ]));
setup();
}
```

從這個簡單的程式中，我們不難得出以下幾點也存在幾個問題:

1) create() 是函數的定義.
2) 它呼叫 set()、set_exits(), 而這兩個函數在這段程式碼中並沒有宣告或定義.
3) 最上面有一行, 不是宣告變數或函式, 也不是函式定義 !

問題：

1) 爲什麼沒有宣告 create() ?
2) 爲什麼 set() 、set_exits() 已經宣告並定義過可以直接使用了?
3) 程式最上面一句話究竟是什麼意思，產生什麼作用呢?

這就是本章將要介紹的基礎的繼承,下麵我們開始.

4.2 物件導向程式設計(object oriented programming)

繼承(inheritance)是定義真正物件導向程式設計的特性之一. 它讓你創造通用的程式碼, 能以多種用途用於許多不同的程式中.一個 mudlib 所作的,就是創造這些通用的檔案(物件), 然後我們通過調用這些檔案來幫助我們製造特定物件. 如果你必須把定義前面工作室全部所需要的程式碼寫出來,大概必須要寫 1000 行程式碼才能得到一個房間所有的功能.顯然那是不切實際的. 再者,這種程式碼與玩家和其他房間的互動性很差, 因爲每一個創造者都寫出自己的函式以作出一個房間的功能. 所以, 你可能使用 query_long() 寫出房間的長敍述, 其他的巫師可能使用 long() . 這就是 mudlib 彼此不相容最主要的原因, 因爲它們使用不同的物件互動協定.

因此 OOP 的出現克服了這些問題.前面的工作室中,inherit ROOM 表示你已經繼承了 ROOM 的函數 ROOM 擁有普通房間所需要的全部函式定義其中. 當你要製造一個特定的房間, 你拿這個房間檔案中定義好的通用函式功能, 並加上你自己的函式 create()就可以很輕鬆地製造一個獨特的房間.

4.3 繼承的作用

例舉上面這個房間的例子,inherit ROOM 使你調用出了 ROOM 這個已經宣告好的函數，該函數包含了許多房間函數，如 set()和 set_exit()等等，正是基於這些函數都是實現宣告並定義好的，所以我們可以直接利用它們來進行我們的代碼編輯工作.在你的 creat() 函式裏, 你呼

叫那些函式來設定你房間一開始的值. 這些值讓你的房間不同於別的房間, 卻保留與記憶體中其他房間互動的能力.
由於 mudlib 之間還存在差異性，所以不同的 mudlib 就各有一套不同的標準函數供我們使用，例如 F_SSERVE 和 SSERVE 其實是同一個標準函數.因此在 coding 之前我們需要弄清楚我們所處的 mudlib 的環境以及提供的標準函數的一些具體情況再開始編程.

第五章: 變數 (variable) 處理

5.1 數值與物件

基本上, mud 裏頭的物件都不一樣的原因有兩個:
1) 有的物件擁有不同的函式
2) 所有的物件都有不同的數值
現在,所有的玩家物件都有同樣的函式.它們不一樣的地方在於它們自己所擁有的數值不同(例如 id),舉例來說,名字叫做 mzjl 的玩家跟 fyfbi,他們各自的 name 變數值不同,一個是"mzjl",另一個是"fyfbi".
所以, 遊戲中量值地改變伴隨著遊戲中物件值的改變.函式名稱就是用來處理變數的過程名稱. 例如說,create()函式就是特別用來初始化一個物件的過程. 函式之中, 有些特別的事稱爲指令. 指令就是負責處理變數的.

5.2 區域 (local) 和全域 (global) 變數

跟大多數程式設計語言的變數一樣, LPC 變數可以宣告爲一個特定函式的「區域」變數, 或是所有函式可以使用的「全域」變數. 區域變數宣告在使用它們的函式之內. 其他函式並不知道它們存在, 因爲這些值只有在那個函式執行時才儲存在記憶體中. 物件碼宣告全域變數之後, 則讓後面所有的函式都能使用它. 因爲只要物件存在, 全域變數就會佔據記憶體. 你只有在整個物件中都需要某個值的時候, 才要用全域變數. 看看下面兩段程式碼:

```
int x; int query_x() { return x; } void set_x(int y) { x = y; }
void set_x(int y) { int x; x = y; write("x 設定爲 "+x+" 並且會消失無蹤.\n");
```

第一個例子裏, x 宣告在所有的函式之外, 所以在 x 宣告之後的所有函式都能使用它. x 在此是全域變數.

第二個例子中, x 宣告在 set_x() 函式裏. 它只有在 set_x() 執行的時候存在. 之後, 它會消失. 在此, x 是區域變數.

5.3 處理變數的值

給 driver 的指令 (instruction) 用來處理變數值. 一個指令的範例是: x = 5;
上面的指令很清楚. 它把 5 這個數值指定給 x 變數. 不過, 這個指令牽涉到一些對普通指令來說很重要的觀念. 第一個觀念是運算式,一個運算式就是有值的一系列符號. 在上面的指令中, 運算式 5 的值指定給變數 x.常數是最簡單的運算式.一個常數就是不變的值, 像是整數 5 或是字串 "hello". 最後一個觀念就是運運算元.在上面的例子中,使用了 = 這個指定運算.

在 LPC 有更多其他的運運算元, 還有更複雜的運算式. 如果我們進入一個更複雜的層次, 例如:
y = 5; x = y +2;
第一個指令使用指定運運算元以指定常數運算式 5 的值給變數 y.第二個指令把(y+2)的值以加法運運算元把 y 和常數運算式 2 加起來,再用指定運運算元指定給 x.
換另一種方法來講, 使用多個運運算元可以組成複雜的運算式. 在前面的範例中, 一個指令 x = y + 2; 裏面含有兩個運算式: 1) 運算式 y+2 2) 運算式 x = y + 2

5.4 複雜的運算式

前面你大概已經注意到, 運算式 x = 5 「本身」也有個值是 5. 實際上, 因爲 LPC 運運算元如同運算式一樣也有自己的值, 它們能讓你寫出一些非常難解、看起來毫無意義的東西, 像是:
i = ((x=sizeof(tmp=users())) ? --x : sizeof(tmp=children("/std/monster"))-1)
基本上只是說:
把外部函式 users() 傳回的陣列指定給 tmp, 然後把此陣列元素的數目指定給 x. 如果指定給 x 的運算式值爲真 (不是 0) , 就指定 x 爲 1 並指定 i 的值爲 x-1 的值. 如果 x 爲僞, 則設定 tmp 爲外部函式 children() 傳回的陣列, 並指定 i 爲陣列 tmp 的元素數目再減 1. 你曾經用過以上的敍述嗎 ? 我很懷疑. 不過你可能看過或使用與它相似的運算式, 因爲一次合併這麼多的東西在一行裏面, 能提升你程式碼的執行速度. 比較常使用 LPC 運運算元這種特性的寫法大概像這樣:
x = sizeof(tmp = users());
while(i--) write((string)tmp[i]->query_name()+"\n");
取代這樣子的寫法:
tmp = users();
x = sizeof(tmp);
for(i=0;i<x;i++) write((string)tmp[i]->query_name()+"\n");
像是 for()、while() 、陣列.....等等東西稍後會解釋.
不過第一段程式碼比較簡潔, 執行起來也比較快.
下面提供一些 LPC 運運算元的說明.

5.5 LPC 運運算元

= 指定運運算元 (assignment operator):
範例 x = 5
說明:把「右邊」任何運算式的值指定給它「左邊」的變數.
注意, 你只能於左邊使用一個變數, 也不能指定給常數或複雜的運算式.

+ 加法運運算元 (addition operator):
範例 x + 7
左邊值加上右邊值的總和
說明:把右邊運算式的值加上左邊運算式的值.
對整數 (int) 型態值來說, 就表示數值總和.對字串(string)來說,表示右邊的值接在左邊的值後面("a"+"b"的值是 "ab"). 這個運運算元不改變任何原始值 (即變數 x 維持原來的值).

- 減法運運算元 (subtraction operator):

範例 x - 7
左邊運算式的值減去右邊的
說明:雖然它是減法, 但實際上與加法的特性是相同的. 字串 "ab" - "b" 的值是 "a".

* 乘法運運算元 (multiplication operator):
範例 x*7
說明:除了這個作數學乘法之外, 特性與加法、減法相同.

/ 除法運運算元 (division operator):
範例 x/7
值與說明 同上

+= 加法指定運運算元(additive assignment operator):
範例 x += 5
值與 x + 5 相同
說明:它把左邊的變數值和右邊的運算式值加起來, 把總和指定給左邊的變數. 例如 如果 x = 2... x += 5 指定 7 值給變數 x. 整個運算式的值是 7.

-= 減法指定運運算元 (subtraction assignment operator):
範例 x-=7
值:左邊的值減去右邊的值.
說明:雖然執行的是減法運算, 但特性與 += 相同.

*= 乘法指定運運算元 (multiplicative assignment operator):
範例 x *= 7
值:左邊的值乘上右邊的.
說明:除了乘法以外, 與 -= 和 += 相似.

/= 除法指定運運算元 (division assignment operator):
範例 x /= 7
值:左邊變數的值除以右邊的值.
說明:除了除法以外, 同上.

++ 後／前增加運運算元 (post/pre-increment operators):
範例 i++ 或 ++i
值:i++的值是 i,++i 的值是 i+1.
說明:++ 改變 i 的值, 將 i 加上 1. 但是, 運算式本身的值是多少, 要看你把 ++ 擺在哪里. ++i 是前增加運運算元. 這表示它的增加在給予值「之前」. i++ 是後增加運運算元. 它計算在 i 增加之前. 重點在哪 ? 好, 目前這對你來說無關緊要, 但是你應該記住它代表的意思.

-- 後／前減少運運算元 (post/pre-decrement operators):
範例 i-- 或 --i
值:i-- 的值是 i --i 的值是 i 減掉 1.

說明:運算特性同++.

== 相等運運算元 (equality operator):
範例 x == 5
值:真或僞 (非 0 或 0)
說明:它不更改任何值, 但是 如果兩個值相等就傳回真. 如果兩邊不相等則傳回僞.

!= 不等運運算元 (inequality operator):
範例 x != 5
值:真或僞
說明:如果左邊的運算式不等於右邊的運算式就傳回真. 如果它們相等則傳回僞.

> 大於運運算元 (greater than operator):
範例 x > 5
值:真或僞
說明:只有在 x 大於 5 時爲真 如果相等或小於就爲僞

< 小於運運算元 (less than operator)
>= 大於或等於運運算元 (greater than or equal to operator)
<= 小於或等於運運算元 (less than or equal to operator):
範例 x < y x >= y x <= y
值:真或僞
說明:與 > 相似,除了<如果左邊小於右邊就爲真>=如果左邊大於「或等於」右邊則爲真<=如果左邊小於「或等於」右邊就爲真

&& 邏輯與運運算元 (logical and operator) || 邏輯或運運算元 (logical or operator):
範例 x && y x || y
值:真或僞
說明:如果右邊的值和左邊的值是非零值, && 爲真. 如果任何一邊是僞, 則 && 爲僞. 對 || 來說, 只要兩邊任何一個值是真, 則爲真. 只有兩邊都是僞值時, 才爲僞.

! 否定運運算元 (negation operator)
範例 !x
值:真或僞
說明:如果 x 爲真, 則 !x 爲僞 如果 x 爲僞, !x 就爲真.

-> 呼叫運運算元 (the call other operator)
範例 this_player()->query_name()
值:被呼叫函式的傳回值
說明:它呼叫右邊這個函式, 而這個函式位於運運算元左邊的物件之內. 左邊的運算式「必須」是一個物件, 而右邊的運算式「必須」是函式的名字. 如果物件之中沒有這個函式, 它會傳回 0 (更精確一點, 沒有定義 (undefined)).

? : 條件運運算元 (conditional operator)

範例 x ? y : z
值:上面的例子裏, 如果 x 爲真, 其值爲 y 如果 x 爲僞, 其值爲運算式 z
說明:如果最左邊的值爲真, 這整個運算式的值就是中間的運算式. 不然, 就把整個運算式的值定爲最右邊的運算式.

第六章: 流程控制 (flow control)

6.1 LPC 流程控制敍述

if(運算式) 指令;
if(運算式) 指令;
else 指令;

if(運算式) 指令;
else if(運算式) 指令;
else 指令

while(運算式) 指令;

do { 指令; } while(運算式);

switch(運算式)
{ case (運算式): 指令; break; default: 指令; }
我們討論這些東西之前,先談一下什麼是運算式和指令.運算式是任何有值的東西,像變數、比較式(像 x > 5, 如果 x 是 6 或 6 以上, 則其值爲 1, 不然其值爲 0)、指定式 (像 x += 2). 而指令是任何一行單獨的 LPC 碼, 像是函式呼叫、值指定式 (value assignment)、值修改式 (value modification) .....等等. 你也應該知道 && 、||、==、!=、! 這些運運算元. 它們是邏輯運運算元. 當條件爲真時, 它們傳回非零值, 爲僞時則傳回 0. 底下是運算式值的列表:

(1 && 1) 值: 1 (1 和 1)
(1 && 0) 值: 0 (1 和 0)
(1 || 0) 值: 1 (1 或 0)
(1 == 1) 值: 1 (1 等於 1)
(1 != 1) 值: 0 (1 不等於 1)
(!1) 值: 0 (非 1)
(!0) 值: 1 (非 0)
使用 && 的運算式中, 如果要比較的第一項測試值爲 0, 則第二項永遠不會測試之. 使用 || 時, 如果第一項爲真 (1), 就不會測試第二項.

6.2if()

我們介紹第一個改變流程控制的運算式是 if(). 仔細看看底下的例子:

```
1 void reset() {
2 int x;
3
4 ::reset();
5 x = random(100);
6 if(x > 50) set_search_func("floorboards", "search_floor");
7 }
```

每一行的編號僅供參考.

在第二行, 我們宣告一個稱爲 x 的整數型態變數. 第三行則優雅地留下一行空白, 以明示宣告結束和函式碼開始的界線. 變數 x 只能在 reset() 函式中使用.

第四行呼叫 room.c 中的 reset().

第五行使用 driver 外部函式的 random() 以傳回一個亂數字, 此數字介於 0 到參數減一. 所以在此我們想得到一個介於 0 到 99 的數字.

第六行中, 我們測試運算式 (x>50) 的值, 看它是真是僞. 如果爲真, 則呼叫 room.c 的函式 set_search_func(). 如果爲僞, 就不可能執行呼叫 set_search_func() .

第七行, 函式將 driver 的控制權交回呼叫此函式的函式 (在這個例子中, 呼叫 reset() 的是 driver 自己) , 也沒有傳回任何值.

如果你想執行一個以上的指令, 你必須按照以下的方法來做:

```
if(x>50) {
set_search_func("floorboards", "search_floor");
if(!present("beggar", this_object())) make_beggar();
}
```

注意運算式爲真時, 要執行的指令以 {} 包圍起來. 這個例子裏, 我們再次呼叫 room.c 中的 set_search_func() 來設定一個函式 (search_floor()) , 這個函式稍後被你設定爲: 玩家輸入 "search floorboards" 時, 呼叫 search_floor(). (注: 這種例子要看 mudlib 而定. Nightmare 有這個函式呼叫, 其他 mudlib 可能會有類似的東西, 也可能完全沒有這一類用途的函式) 接著, 另一個 if() 運算式檢查 (!present("beggar", this_object())) 運算式是否爲真. 測試運算式中的 ! 改變它後面運算式的真僞. 在此, 它改變外部函式 present() 的真僞值. 在此, 如果房間裏有個乞丐, present() 就傳回乞丐這個物件 (this_object()), 如果沒有乞丐, 則傳回 0. 所以, 如果房間裏面還有個活乞丐, (present("beggar", this_object())) 的值就會等於乞丐物件 (物件資料型態) , 不然它會傳回 0. ! 會把 0 變成 1 , 把任何非零值 (像是乞丐物件) 變成 0. 所以, 房間裏沒有乞丐時, 運算式 (!present("beggar", this_object())) 爲真, 反之, 有乞丐爲 0. 如果房間裏沒乞丐, 它呼叫你房間碼中定義的函式來製造一個新的乞丐, 並放進房間. (如果房間中已經有一個乞丐, 我們不想多加一個 :))

當然, if() 常常和一些條件一起出現 :). LPC 裏, if() 敍述的正式寫法爲:

```
f(運算式) { 一堆指令 }
else if(運算式) { 一堆指令 }
else { 一堆指令 }
```

這樣表示:

如果運算式爲真, 執行這些指令. 不然, 如果第二個運算式爲真, 執行第二堆指令. 如果以上皆僞, 執行最後一堆指令.

你可以只用 if() :
if(x>5) write("Foo,\n");
跟著一個 else if():
if(x > 5) write("X 大於 5.\n");
else if(x >2) write("X 小於 6, 大於 2.\n");
跟著 else:
if(x>5) write("X 大於 5.\n");
else write("X 小於 6.\n");
或是把上面列出來的東西全寫出來. 你有幾個 else if() 都沒關係, 但是你必須有一個 if() (也只能有一個), 也不能有一個以上的 else . 當然, 上面那個乞丐的例子中, 你可以在 if() 敍述中重複使用 if() 指令.
舉例來說, if(x>5) { if(x==7) write("幸運數字 !\n"); else write("再試一次.\n"); } else write("你輸了.\n");

6.3 while() 和 do {} while()

原型:
while(運算式) { 一堆指令 }
do { 一堆指令 } while(運算式);
這兩者讓你在運算式爲真時, 一直重複執行一套指令. 假設你想設定一個變數等於玩家的等級, 並持續減去隨機的金錢數量或可承受傷害值 (hp, hit points) 直到該等級變數爲 0 (這樣一來, 高等級的玩家失去的較多). 你可能會這樣做:

```
1 int x;
2
3 x = (int)this_player()->query_level();
4 while(x > 0) {
5 if(random(2)) this_player()->add_money("silver", -random(50));
6 else this_player()->add_hp(-(random(10));
7 x--;
8 }
```

由於 x 本身稍後會一直減 1 直到到 x = 0 , 所以 x = 0 時也是僞值 (爲 0).
第五行以 random(2) 隨機傳回 0 或 1. 如果它傳回 1 (爲真), (譯注: 補充完畢) 則呼叫玩家物件的 add_money() 將玩家身上的銀幣隨機減少 0 到 49 枚.
在第六行, 如果 random(2) 傳回 0, 我們呼叫玩家物件中的 add_hp() 函式來減少 0 到 9 點的可承受傷害.
第七行裏, 我們把 x 減 1.
第八行執行到 while() 指令的終點, 就回到第四行看 x 是否還大於 0 . 此回圈會一直持續執行到 x 小於 1 才結束. 但是, 你也許想在你執行一些指令「之後」再測試一個運算式. 比如用上面的例子, 如果你想讓每個人至少執行到一次指令, 甚至還不到測試的等級:

```
int x;
x = (int)this_player()->query_level();
do {
```

```
if(random(2)) this_player()->add_money("silver", -random(50));
else this_player()->add_hp(-random(10));
x--;
} while(x > 0);
```

這個例子真的很奇怪, 因爲沒幾個 mud 會有等級爲 0 的玩家. 而且, 你可以修改前面例子中的測試條件做到同樣的事. 不管如何, 這個例子只是要展現出 do {} while() 的如何工作. 如你所見, 此處在回圈開始的時候沒有初始條件 (在此不管 x 的值爲何, 立刻執行) , 回圈執行完之後才測試. 這樣能保證回圈中的指令至少會執行一次, 無論 x 爲何.

6.4 for() 回圈

原型:
for(初始值 ; 測試運算式 ; 指令) { 指令 }
初始值:
讓你設定一些變數開始的值, 用於回圈之內. 此處可有可無.
測試運算式: 與 if() 和 while() 的運算式相同. 當這一個 (或一些) 運算式爲真時, 執行回圈. 你一定要有測試運算式. 指令:
一個 (或一些) 運算式, 於每個回圈執行完畢之後執行一次. 此處可有可無.
注:
for(;運算式;) {} 與 while(expression) {} 「 完 全 相 同 」
範例:

```
1 int x;
2
3 for(x= (int)this_player()->query_level(); x>0; x--) {
4 if(random(2)) this_player()->add_money("silver", -random(50));
5 else this_player()->add_hp(-random(10));
6 }
```

這個 for() 回圈與前面 while() 的例子「完全相同」. 還有, 如果你想初始化兩個變數:

```
for(x=0,y=random(20);x<y;x++){ write(x+"\n"); }
```

在此, 我們初始化 x 和 y 兩個變數, 我們把它們用逗號分開來. 你可以在 for() 三個部分的運算式中如此使用.

6.5 敍述: switch()

```
原型: switch(運算式) {
case 常數: 一些指令
case 常數: 一些指令
.....
case 常數: 一些指令
default: 一些指令
}
```

這樣有點像 if() 運算式, 而且對 CPU 也好得多, 但是 switch() 很少有人使用它, 因爲它看起來實在很複雜. 但是它並非如此. 第一點, 運算式不是測試條件. case 才是測試. 用普

通的話來讀:

```
1 int x;
2
3 x = random(5);
4 switch(x) {
5 case 1: write("X is 1.\n");
6 case 2: x++;
7 default: x--;
8 }
9 write(x+"\n");
```

就是:

設定變數 x 爲一個 0 到 4 的亂數字. x = 1 的 case 中, 顯示 x 的值, 將 x 加上 1 之後再將 x 減 1. x = 2 的 case 中, 將 x 加上 1 之後再減 1. 其他情形下, x 減 1. 顯示 x 的值.

switch(x) 基本上告訴 driver, 變數 x 的值是我們想配合各個 case 的情形. 當 driver 找到一個能配合的 case 時, 這個 case 「以及所有在它之後」的 case 都會執行. 你可以使用 break 指令, 在執行一個 case 之後跳出 switch 敍述, 就像其他流程控制敍述一樣. 稍後會解釋這一點. 只要 switch() 流程還沒中斷, 任何 x 值都會執行 default 敍述. 你可以在 switch 敍述中使用任何資料型態:

```
string name;
name = (string)this_player()->query_name();
switch(name) {
case "descartes": write("You borg.\n");
case "flamme":
case "forlock":
case "shadowwolf": write("You are a Nightmare head arch.\n");
default: write("You exist.\n");
}
```

對我來說, 我會看到:

```
You borg.
You are a Nightmare head arch.
You exist.
```

Flamme、Forlock 、或 Shadowwolf 會看到:

```
You are a Nightmare head arch.
You exist.
```

其他人會看到: You exist.

6.6 改變函式的流程和流程控制敍述

以下的指令:

return continue break

能改變前面提過的那些東西,

它們原本的流程. 首先,

return
一個函式中, 不管它出現在哪里, 都會終止執行這個函式並將控制權交回呼叫這個函式的函式. 如果這個函式「不是」無傳回值 (void) 的型態, 就必須在 return 敍述之後跟著一個傳回值. 一個絕對值函式長得大概像這樣:
int absolute_value(int x) { if(x>-1) return x; else return -x; }
第二行裏, 函式終止執行, 並回到呼叫它的函式. 因爲在此, x 已經是正整數. continue 在 for() 和 while() 敍述中用得最多. 它停止目前執行的回圈, 把回圈送回開頭執行. 例如, 你想要避免除以 0 的情況:

x= 4; while(x > -5) { x-- if(!x) continue; write((100/x)+"\n"); } write("完畢.\n")
你會看到以下的輸出:
33
50
100
-100
-50
-33
-25
完畢.
爲了避免錯誤, 每一次回圈都檢查 x, 確定 x 不爲 0. 如果 x 是 0, 則回圈回到開頭處的測試運算式, 並不終止目前的回圈.

用 for() 運算式來說就是:
for(x=3; x>-5; x--) {
if(!x) continue;
write((100/x)+"\n");
}
write("完畢.\n");

這樣執行起來差不了多少. 注意, 這樣子跟前面輸出的結果一模一樣. 當 x = 1, 它測試 x 是否爲 0, 如果不是, 就顯示 100/x, 然後回到第一行, 將 x 減 1, 再檢查 x 是否是 0 , 如果爲 0, 回到第一行並把 x 再減 1.

break 它停止執行流程控制敍述. 不管它出現在敍述裏面的任何地方, 程式控制會結束回圈. 所以, 如果在上面的例子中, 我們把 continue 換成 break, 則輸出的結果會變成像這樣:
33
50
100
完畢.

continue 最常用於 for() 和 while() 敍述. 但是 break 常用於 switch().
switch(name)

```
{ case "descartes": write("You are borg.\n"); break;
case "flamme": write("You are flamme.\n"); break;
case "forlock": write("You are forlock.\n"); break;
case "shadowwolf": write("You are shadowwolf.\n"); break;
default: write("You will be assimilated.\n");
}
```

下面這個函式跟上面的一樣:

```
if(name == "descartes") write("You are borg.\n");
else if(name == "flamme") write("You are flamme.\n");
else if(name == "forlock") write("You are forlock.\n");
else if(name == "shadowwolf") write("You are shadowwolf.\n");
else write("You will be assimilated.\n");
```

但是 switch 敍述對 CPU 比較好. 如果這些指令放在多層巢狀 (nested) 的敍述中, 它們會改變最近的敍述.

你現在應該完全瞭解 if()、for()、while()、do{} while()、switch(), 也該完全瞭解如何使用 return、continue、break 改變它們的流程. 使用 switch() 要比一大堆 if() else if() 來得有效率, 所以應該儘量使用 switch() . 我們也向你介紹過怎麼呼叫其他物件中的函式.第六章的內容應該說是整個基礎教程的關鍵，將這六章的內容結合起來，那麼現在你已經能夠做出簡單的房間、NPC 和其他簡單的物件了，下面的事就是多琢磨，多鑽研了.

第七章: 網路編程

在 MudOS 0.8.14 和 0.9.0 中的一個增強就是 Internet socket 功能被包含進來了。MudOS 與 TMI 研究者一直希望使網路上 MUD 的更緊密地通過通信集成在一起， Socket efun (或者 LPC socket) 依靠允許 LPC 開發者寫作基於 Internet socket 的應用程式提供了第一級的 MUD 集成性。 例如，已經存在用於 telnet、遠端 MUD finger、遠端 MUD tell、互聯 MUD 郵件遞送以及參與 MUDWHO 系統的 LPC 物件。
之所以要寫這麼一份文檔， 就是要把它作爲一個指導你如何使用 LPC socket 進行基於網路的互聯編程的指南。 它的定位將是那些有一定經驗的 LPC 程式員，他們已經理解 LPC 編程的大部分基礎內容，但希望寫一些基於網路的 LPC 服務。

7.1 Socket 模式

一共有五種不同的通信模式或者叫做套接字(socket)模式：
MUD、STREAM、DATAGRAM、STREAM_BINARY，和 DATAGRAM_BINARY。這五種模式的定義可以在 mudlib include 的 <socket.h> 中找到。

MUD 模式

MUD 模式是一個面向連接的通信模式， 這種模式中 LPC 資料類型可以由網路

從一個 MUD 傳送到另一個 MUD。例如，在 MUD 模式中你可以發送結構資料——如陣列和映射——穿過網路到另一個也使用 MUD 模式 socket 的 MUD 中。除 object 以外的所有的 LPC 資料類型都可以由 MUD 模式發送和接收。

jjgod 注：無法傳送物件造成了穿越 mud 的功能， 比如一直比較熱的分站漫遊功能受到一些阻礙，我們需要較爲麻煩的獲得一個物件身上所有的變數(比如一個 USER_OB， 需要查找 /clone/user/user.c 以及所有直接和間接繼承——可以用 deep_inherit_list 得到——的的檔中的變數)，然後一個一個的發送和接收。 這樣就造成相容性和可擴展性很成問題。

STREAM 模式

STREAM 模式也是一個面向連接的通信模式，和 MUD 模式不同的是，所有的數據都是以字串形式發送和接收的。所以，你可以通過 STREAM 模式從其他的網路端(比如你自己寫的一個 MUD 用戶端)發送到 MUD 中。由於無法直接發送和接收所有的 LPC 資料類型，STREAM 模式會顯得不那麼強大，但很多應用程序，像 telnet，用不著發送整數、陣列這樣的資料， 而都是以各個方向的字符流的形式流覽的。

MUD 模式 socket 其實就是使用特殊的代碼以通過 STREAM 模式來實現發送和接收 LPC 資料類型。因此，如果應用程式需要額外的資料提取，更適合用 MUD 模式。 但 MUD 模式由於其固有的限制，將比 STREAM 模式更慢，也將使用更多的記憶體。 注意，STREAM 模式是無法確保發送的字串能夠全部立即到達，實際上它們是一份一份的送到後， 再將之重新組合在一起(每一份將按順序到達)。

DATAGRAM 模式

和 MUD 與 STREAM 模式不同，DATAGRAM 模式是無連接的。你不需要確立一個連接就可以在 MUD 間傳輸資料了。因此，每份資料是採用一種叫“datagram”的資料包發送到目的地的，每個這種資料包都自帶定址資訊，能夠自覺地從網絡的一端行走到另一端。

因爲 DATAGRAM 模式沒有一個確定的連接，所以發出的 datagram 包有可能在網路中就這麼丟失了，沒有任何一個 MUD 能收到它。 例如 TMI 用 DATAGRAM 模式發送的一個到 Portals 的資料包如果在網路中丟失了， Portals 可能永遠收不到它，而且完全不知道曾經發送過這麼一個資料包。 而且 TMI 也無法得知這個資料包是否丟失了，因爲就算丟失了也不會收到任何錯誤資訊。

TCP 和 UDP

在 MUD 模式和 STREAM 模式中，MUD 間將建立一個 TCP 連接，TCP 是一種能夠保證資料被正確發送的協定，如果它發現資料包丟失了，就會嘗試重新發送它，直到正確收到爲止：它通過特定的演算法來發送資料，這種演算法使其能夠估算數據要花多長時間才能到達，如果過了那段時間還沒收到回答，它就重新發送一個資料包，直到收到確認資訊爲止。TCP 協定也確保了資料包能夠按順序到達，而且也不會收到兩個同樣的資料包。(這是 TCP 協議的一點很膚淺的描述，但也算簡要地說明瞭爲什麼這種協議使得資料傳輸變爲可信賴的。)

DATAGRAM socket 則不然。 UDP 是一種面向 datagram 的協議，它在 MUD 間

發送資料包時是完全和“連接”、“重新發送”等這些詞沾不上邊的。但是，既然 DATAGRAM 模式是不可靠的，爲什麼有人會用到它呢？當然，TCP 顯得更好：它通過重新發送確保了資料能夠正確到達、它考慮到了網路中所有可能出現的惡劣情況……而實際上，不少應用程式卻不在乎所有的資料是否正確到達了另一端。你可能要問了，既然不在乎還發送它幹什麼？好吧好吧，這是個不錯的問題，但現在就回答它太早了， 你只要記住有些情況是得用到 DATAGRAM 模式的，我們會在後邊詳細的解釋一下，別著急。

jjgod 注：例如 InterMud 這種鬆散集成的程式，就完全不用考慮資料包是否正確發到了， 你的 MUD 有的時候甚至僅僅是毫無目的的在網上尋找是否有能夠互聯的 MUD， 定時的 ping 也不期望別的 MUD 就一定能收到——收不到就收不到唄，最多不過是少了一個與之互聯的 MUD 而已，對自己沒什麼傷害。 相反，如果你與數百個 MUD 互聯，而和這些 MUD 之間傳送資訊都需要一次次的確認、 確認再確認，網路負擔就很可觀了。

7.2 創建 Socket

——————————

好的，讓我們從創建一個 MUD 模式 socket 開始， 我們可以寫這樣一個物件來實現：

```
#include <socket.h>
void create()
{
        int s;
        s = socket_create(MUD, "close_callback");
        if (s < 0)
        {
                write("socket_create: " + socket_error(s) + "\n");
                return;
        }
        write("Created socket descriptor " + s + "\n");
        socket_close(s);
}
void close_callback(int s)
{
        write("socket " + s + " has closed\n");
}
```

讓我們分解一下這個程式來看清楚一個 socket 是如何創建的。不過首先要說的是，到我們能夠用 socket 來發送資料還有一個漫長的路程要走，創建僅僅是第一步呢。所以給點耐心吧，弄明白每個例子以後再開始做。

我們做的第一件事情就是 #include <socket.h>，所有的 socket 定義都在這個 socket.h 裏面。首先是定義，例如 MUD、STREAM 和 DATAGRAM，雖然每個名字都對應著一個數字，但寫得好的程式應該用巨集定義來代替數位：因爲某天你可能要修改這些定義(假如有一天 MUD 和 STREAM 代表的數字調換了，就不

必在代碼中一處一處的修改數字)， 也是因爲這樣可以讓人一看就明白你打算做什麼。
我們定義了一個整數變數 s， 很多 socket 應用程式中 s 都作爲 socket 的縮寫出現。然後我們以兩個參數調用 socket_create()，第一個參數是 socket 的模式(我們上面討論的那些)，注意我們使用巨集定義 MUD 來代表 MUD 模式。例子中第二個參數叫做關閉回叫函數， 當這個連接被關閉時就會由 MudOS 呼叫此函數。 回叫(callback)常常被用在 LPC socket efun 中，以通知物件重要網路事件的發生。注意，我們還可以以 STREAM 或 DATAGRAM 爲第一個參數來創建 STREAM 或 DATAGRAM socket。
所有的 socket efun 都會返回一個退出狀態或者返回一個特定的值。 這個值代表了此函數完整的狀態。我們規定負值代表錯誤或者警告，當返回一個錯誤代碼時，應用程式得決定如何回答它。 有時假如 MUD 管理員不修改本地的配置檔或者 MudOS 的話，根本沒有可能獲得成功的返回代碼， 那怎麼辦呢？
比如說上面的例子，如果返回了錯誤(s 小於 0)，我們就使用 socket_error() 這個 efun 來在螢幕上顯示出一段錯誤原因的資訊，這在調試時是很有用的，但在實際運行的時候，我們可能不希望所有的用戶都看到這些雜亂的資訊，所以那時還是換成 log_file() 來把錯誤記錄下來，以備以後改正。
如果 socket_create() 成功了，它將返回一個大於等於 0 的整數，這個整數代表了一個 socket、一個 socket 描述符(descriptor)， 或者一個檔描述符，這三個名字都源於 UNIX 術語。 如果 socket_create() 返回負值，就說明有錯誤發生，沒有創建出什麼 socket。 但這並不是獲得一個錯誤的理由，實際上， 最常見的錯誤往往是因爲指定錯了 socket 模式造成的(如果你採用 socket.h 中的定義來指定，通常不那麼容易出錯，因爲每個巨集定義都很明顯)，或者是 socket 溢出了。MUD 管理員可以通過配置 MudOS 能使用的 socket 數目來避免這個問題，默認這個數目是 16， 但你最好配置一下以適應自己 MUD 的需求，增加這個數字就可以使你能用到更多的 LPC socket 了。注意，每個到達的 LPC socket 都將佔用一個用於玩家登錄和打開檔的 socket。 若你需要同時保證大量的玩家和大量打開的 socket， 就得考慮增加這個最大值以保證此進程能夠打開足夠多的檔描述符。
所有的 socket 物件都得小心不要丟失了 socket，socket 物件不像其他的 LPC 物件，它們的數目是有限的。 所以，如果我們在上面的例子中成功的創建了一個 socket，後面一定要記住關閉它。 我們知道，丟失資源的軌跡稱爲“洩漏”，socket 洩漏發生在當一個物件創建了 socket，使用了一會兒卻在不再用到它的時候忘了關閉它，結果這個物件就一直佔用著一個 socket， 別的物件只能看著乾著急。不過如果一個物件被析構了，它那裏的所有 LPC socket 都會被自動關閉的。另一個值得注意的是，每個 socket 都有它獨一無二的 socket 描述符(或者 socket 編號)，因此如果某物件創建了一個 socket， 另一個物件創建了第二個 socket， 這兩個物件不可能收到相同的 socket 描述符。我們可以利用這一點來實現一些功能，例如以 socket 描述符來作爲一個 mapping 的索引(index)以記下每個打開的 socket 的資訊。 但記住，如果一個 socket 被關閉了，它就可以被 socket_create() 重新用到來作爲新 socket 的描述符。
jjgod 注：現在的 ES2 mudlib 中這些頭檔似乎都移到了 /include/net 中。
如 socket 模式定義在 <net/socket.h> 中，但 socket 錯誤的信

息定義在〈socket_err.h〉中。

7.3 用戶端/伺服器模型

在繼續介紹剩下的 socket efun 之前， 現在我們最好停下來復習一些基本的網路概念。面向連接的通信常常是以用戶端/伺服器端模型構成的， 在這個模型中，任何一個連接要麼是用戶端，要麼就是伺服器端。由用戶端發起連接、(向伺服器端)請求一些服務。伺服器端則是在等待用戶端的連接請求，如果等到了的話就提供相應的服務。例如 FTP 就是這麼工作的。 用戶通過連接到服務器發出一個請求，伺服器就會滿足用戶端的請求。伺服器端和用戶端不同之處在於，它可能同時在爲多個客戶提供服務。
MUD 和 STREAM 就用到了用戶端/伺服器端模型， 用戶端和伺服器端採用些微有點不同的方式來確立連接。後面我們還會討論到使用 DATAGRAM 模式的點對點(peer-to-peer)模型， 這種模型和用戶端/伺服器端模型也略微有點不同，例如在確立連接時。
你也可以同時啓用多個服務，每個服務都以一個“眾所周知”或者說是“約定俗成”的埠來區分。埠是從 1 到 65535 之間的一個整數，儘管如此，大多數的 MUD 都採用 1025 到 65535 這個範圍內的埠號，因爲前 1023 個端口都是爲 telnet、ftp 這些已經成爲標準的應用保留的。 要讓用戶端和服務器端之間能夠進行通訊， 伺服器端得先創建一個 socket，將之幫定到一個約定的埠上，然後監聽連接的請求。另一方面，用戶端也必須創建一個 socket，然後連接到那個約定的埠上。也就是說用戶端將連接到伺服器端所綁定和監聽著的那個埠。這就是爲什麼稱之爲“眾所周知”埠的原因了，——用戶端得先知道有這麼個埠，才能去連接。約定的埠號往往定義在 Internet RFC 文檔中，以一個大家認可的標準形式存在。
我們將在下面討論如何通過執行 socket efun 來建立一個用戶端與伺服器端的連接。不過在用戶端能夠發起一個連接請求之前，伺服器端還需要做一些准備工作，所以，先讓我們啓動伺服器。

7.4 綁定到一個埠

在伺服器端用 socket_create() 創建了一個 socket 之後(而且當然返回值大於等於 0)，下一個合理的步驟是綁定到一個埠，這個工作是由 socket_bind() 完成的。讓我們在上面的例子裏添加一點代碼試試。 首先，聲明一個新的整數來記錄呼叫錯誤，也就是那個 socket_bind() 的返回值啦。

```
        int error;
// 現在讓我們添加一個到 socket_bind() 的呼叫:
        error = socket_bind(s, 12345);
        if (error != EESUCCESS)
        {
                write("socket_bind: " + socket_error(error) + "\n");
                socket_close(s);
```

```
                return;
        }
```

這段內容要加在 socket_close(s) 之前。 那麼，這有什麼用呢？好吧，先看第一個參數， 嗯，沒什麼奇怪的，就是我們從 socket_create() 那裏得到的 socket 描述符。每個關於 socket 的呼叫都要用到這個描述符，以使得 MudOS 明白我們想操作的是哪個 socket。 你還記得伺服器端是經常同時運作超過一個客戶 socket 的吧？所以必須保證它們不被混淆。第二個參數很簡單，就是埠號，埠號必須爲從 1024 到 65535 的整數，你看對不對？ (其實 0 也是合法的，我們會在稍後談到這個。)

呼叫 socket_bind() 之後， 我們再檢查一下返回值，看看是否發生了錯誤，目前的情況下也就是和 EESUCCESS 比較一下，在絕大多數場合(socket_create() 除外) EESUCCESS 都代表這個 socket efun 成功執行了。和 socket_create() 類似，如果發生了錯誤，我們也使用 socket_error() 來將它以字元串顯示出來。那麼輸出錯誤以後我們就返回了，對不對？錯了！上面我們提到了洩漏這個詞，如果我們就這麼返回了，這個物件就會一直佔用著這個不再被用到的 socket 從而導致別人沒法利用它。因此當我們確定以後不再用到這個 socket 時，千萬記住關閉它。從 socket_create() 的呼叫開始，直到 socket_close() 被呼叫，這個 socket 都是打開的。因而，要做個良好的 socket 使用者，記住在你的工作完成時關閉它。

socket_bind() 有個挺有點“名氣”的返回值——EEADDRINUSE。 爲什麼這麼說呢？如果有人綁定了一個 socket 到 12345 埠， 然後另一個 socket 也嘗試向綁定到同一個埠上，第二個 socket 的綁定將會失敗。這很好理解，socket 綁定到一個埠之後，該埠就屬於那個 socket 了， 其他試圖摻和進來的就會收到 EEADDRINUSE 這個錯誤資訊。這是個很常見的錯誤。 正確的解決方案是：1) 檢查一下是否同一個服務被啓動了兩次。 若是這樣，關掉一個，因爲一個就夠了。 2) 是否有幾個的開發者爲不同的服務選擇了同一個端口。這可不行，要避免這種情況可以讓一個“埠管理員”來分配埠，其他人不得插手。遺憾的是，網路不是孤立的，埠的分配需要所有你希望與之通信的 MUD 都同意才行。

jjgod 注：在提到 EEADDRINUSE 錯誤描述符時， 作者用了“notorious(聲名狼藉的)”這個詞，但在下文中並沒有貶義， 所以就按照作者的本意將之譯爲“名氣”便罷。

7.5 安全

在繼續下一步之前，我們需要回答幾個關於非法的 socket 描述符的問題。當我們傳入的值是一個錯誤的 socket 描述符時，會怎樣呢？別擔心，MudOS 會馬上捕獲你的行爲，並告訴你這是行不通的。舉個例子，當你傳入一個小於 0、或大於等於系統可用的 socket 最大值的描述符，MudOS 會發現你犯了個錯誤並返回 EEFDRANGE。若你打算做得更狡猾一點的話，可以嘗試給出一個合法但卻不是目前在用的數值，MudOS 還是會發覺並返回 EBADF。好吧，讓我們再狡猾一點，傳入一個別的物件正在用的 socket 描述符。會怎麼樣呢？這就到了解釋爲 LPC socket 建立的兩層安全系統的時候了。

第一層的安全使用主控(master)物件來驗證哪些物件可以、哪些物件不能使用 socket。 這可能在一些 MUD 中比較有用，例如我們打算讓一些開發者有許可權使用 socket，而另一些沒有， 或者出現個別的開發者濫用網路使用權而被禁止使用 socket 的情況(後面的例子中那樣的開發者最好讓他離開算了)。爲了實現上述的意圖，MudOS 調用了 valid_socket() 這個函數，valid_socket() 返回 0 或者 1 以標示請求的 socket 操作是否被允許。 如果沒有 valid_socket() 這麼個函數存在，MudOS 就假定返回值是 0，這樣所有的 LPC socket 操作將被全部禁止。所以，大多數的 MUD 的 master.c 都會包括下面這個 valid_socket()：

```
int valid_socket(object eff_user, string fun, mixed *info)
{
        return 1;
}
```

第二層的安全更爲嚴格些，它用於防止一個物件通過某些手段妨害其他的物件。我們已經知道， 當一個 socket 創建時，呼叫 socket_create() 的那個物件就等於擁有了這個 socket。每當一個 socket efun 被呼叫時，它都會比較一下呼叫者是否那個擁有者。如果不同，這次呼叫 socket efun 就被中斷了。下面有段很是可惡的代碼：

```
int s;
for (s = 0; s < 100; s++)
        socket_close(s);
```

它並不會成功的關閉 MUD 中所有的 socket，而只能關閉所有這個呼叫者自己擁有的那些 socket， 因爲所有其他的 socket 都由第二層安全措施保護著呢。如果被第一和第二層安全措施中的任一個拒絕，將返回 EESECURITY 來告訴那位呼叫者：socket efun 因爲安全原因拒絕了他的請求。如果你在寫代碼的過程中遇到了這樣的錯誤，多數是因爲你傳入了一個錯誤的 socket 描述符，畢竟這偶爾是會出現的。

7.6 監聽連接

當 socket 一創建出來，埠一被綁定，伺服器就必須開始監聽連接了。這是由 socket_listen() 來完成的。和 socket_bind() 一樣，一個參數是要監聽的 socket，第二個參數則是那個“監聽回叫(listen callback)”函數。回憶一下 socket_create() 中那個“關閉回叫”函數？ socket_listen() 也指定了一個當接到連接就會呼叫的函數，在這個函數中，伺服器可以決定是接受這個連接呢，還是關閉它。在大多數的伺服器中，確實沒有什麼就這樣關掉一個連接的理由，因爲這樣通常過於武斷，應該避免。若用戶端和伺服器端達成了某些鑒定的協定(例如密碼檢查)時，伺服器端好歹應該返回一段消息，告訴客戶端爲何連接被關閉了。當然，這可能更像風格的問題，但你得知道，要對一個連上了馬上就被莫名其妙地斷開了的用戶端或伺服器端進行調試，是非常困難的。如果你非得這麼做的話，就要確保你在日誌檔中記錄下了關閉連接的理由，這樣管理員或者開發者才方便診斷原因，解決問題。

下麵的代碼，讓一個已經創建了並綁定到埠上的 socket 開始進行連接監聽：

```
        error = socket_listen(s, "listen_callback");
        if (error != EESUCCESS)
        {
                write("socket_listen: " + socket_error(error) + "\n");
                socket_close(s);
                return;
        }
```

這和上邊的代碼簡直就是一樣的麼： 我們呼叫 socket_listen() 函數，檢查返回值看看是否成功了，若發生了錯誤則輸出包括了指定錯誤描述的錯誤的信息，然後，因爲我們已經完成了工作，就關閉連接並返回。事實上，很多必需的 socket 應用程式碼中都是按照這個模式寫成的。

很明顯，下一步就是討論這個 listen_callback 函數，是嗎？ 按道理說沒錯，但我們還是不這麼做，而是換換空氣，改爲看點用戶端的代碼吧。理由很簡單，在用戶端可以發起一個連接以前，討論伺服器端的問題太不現實了，因爲這之前伺服器端並不會運行比上邊更多的代碼。因此，稍微離題一下，去討論客戶端的問題是有必要的。

編後語

這個是總http://www.ltyz.gx.cn下載的，加入了自己的一些修改整理而成的一個LPC的學習文檔，文檔中難免有出錯的地方，請指教。

———Larkin 小新