Mudlib--制作

作者:akuma 是北大侠客行 mud 老牌巫师柳残阳 经常在这里遇到有朋友就一些特定的 mudlib 提问，很多人，特别是初学者，经常忘了写是哪个 lib 里的问题。

一般来说，目前我们常见的就是 xkx 类，fy 类，xyj 类的 lib。 虽然这些 lib 最初都源自台湾安老大的 ES2 mudlib，但经过这么多年的发展或者不发展，各自都

有很多自己的特色（不是指游戏内容，而是目录结构，继承关系，乃至于函数命名风格等）有时候我自己不去对应的 lib 里翻找，也很难记清楚一个事情的来龙去脉。更不用说众多的改版站 自己做的东西了。

而且由于这么多年的变迁，各种 lib 都已经变得相当的复杂，很多基础的东西隐藏的比较深。包 括很多还在写东西的 wiz，自己对于一些基本的因果关系也不是那么熟悉和清楚了。

所以我就在想，是不是可以采用类似写编程书的方法，通过例子一步一步的把一些底层的东西 讲清楚说明白，对于大家来说，越是对底层了解的透彻（知其然并且知其所以然）在搭上层的 时候就越容易采取简单而且正确的方法；对我来说，也可以借这个机会整理我自己的思路，并 且把一些我自己也比较迷糊的概念整理明白。

所以才会考虑写这篇东西。 我希望可以通过从头整理一套简单清晰的 lib，把很多底层的概念配合者讲解一遍。 那么有四点需要说明： 这是比较业余的东西，我自己工作比较忙，可能很难保证更新的速度，但我一定尽我所能来

写，争取把这个系列完成掉。

由于演示的成分比较大，因此随着每一讲的深入，我们最终完成的 lib 可能会是个简陋的，无 法真正提供游戏内容的产品。但我希望他是一个简单的，思路清晰的东西，可以完成大部 分基本的游戏内容（比如可以提供简单的战斗，经验的获得等）

如果可能的话，我会在讲到每一点的时候，对照一些成熟的 lib 来讲。但是我的目录结构，乃 至于继承关系等可能跟他们有不同。

依然由于这是个人作品，我自己的思路也会比较固定，所以错误在所难免，如果大家发现描 述上的、概念上的错误，那纯熟正常，请大家尽可能提出来，我尽量 fix。

第一讲：让它跑起来

注：每一讲我都会上传一个相符的 lib，有些文件是旧的，有些是新的，我尽可能在 lib 里写清 楚注释。更详细的内容则在每讲的正文里写。

一个最简单的能跑的 lib 应该长成什么样子？每个基于 mudos 写 lpc 的人可能都会给出不同的答 案。我记得曾经有个朋友释出过一个不到 5k 的 lib。

我这个则还要小一点，tgz 之后是 1851 个 字 节 。嗯 • 还 好 。 我们对这个 lib 基本上不会有什么期待，但是他至少应该完成如下两个事情：

能跑起来，并且接受用户的连接（你用 zmud 也好，telnet 也好，总之是可以连到端口上）

连接后的用户可以输入 并且 lib 应该给予一定的反应(那么最简单的做法就是完成一个所谓 的 echo server 了—你输入什么，server 就 给你返回什么)

【配合本讲的 lib 版本为 0.1，文件名则是 newlib.0.1.tar.gz，见附件】 以下是目录结构：

```
    |--
adm
    |
`--  obj
    |
|-- master.c
    |                       `--
simul_efun.c
    |--
include
    |              `--
globals.h
    |--
log
    `--
obj
            `--
        user.c
```

首先说目录结构，一个好的清晰的目录结构用起来很舒服，我自己一般推崇单层目录结构，但 是考虑到习惯问题 ,我在/adm 下采用了和 xkx 类似的方法 ,即把 master 和 simul 放到了 /adm/obj/ 下。

大 致 讲 一下：

目前我们只创建了四个基本目录：/adm /include /log /obj。

adm 放的都是“独一无二”的东西，比如最关键的两个物件 master.c 和 simul_efun.c，还有以后会慢慢出现的各种 daemons。

include 是所有头文件所在的目录，目前只有一个 globals.h。 log 是用来存放各种 log 输出的，这没啥可讲的。 obj 是所有 lib 当中最终被用到的物件的实体文件，也就是会被最终 new()或者 load_object()

出来的东西。暂时只有一个连线物件，也就是 user.c（你看，现在我们连 login.c 都省掉 了，因为我们的当前目标仅仅是能让 mud 接受连接而已，断线重连、帐号认证等工作要放到 以后完成）

这里我们先讲一点基础的东西：mudos 如何启动？ 我们不讲 mudos 自己，只说和 lib 有关的部分。 有人认为，mudos 最先载入的是 master.c（请注意，这个 master.c 不是那个可以用来拜师的

master）其实这个看法是有点问题的。

一般来说，mudos 最先载入的文件是 simul_efun.c。 我们知道，要想跑起一个 mud，除了 lib 以外，还需要 driver 和一个给 driver 用的配置文件

config.cfg 在 config.cfg 里 有三个文件是要被定义的 simul_efun.c，master.c 和 globals.h，

如果愿意的话，我们当然可以给这三个文件起个其他的名字，比如 master.c 改名叫 core.c，globals.h 可以叫做 dangzhongyang.h 之类的，随意。不过大家都习惯了，我就不改了。

重 点 1. 关 于 simul_efun.c

刚才我们说到，mudos 启动之后，会先尝试载入这个 simul_efun.c。 这里有一个概念就是所谓的 simul。众 所 周 知 mudos 为了游戏的目的 提供了大量的函数（efun）

来帮助我们完成工作，比如说像判断是否为玩家的 userp()，像是对字符串做操作的一系列函数

等

等。

然而 mudos 也不是万能的，他不可能为预先想到所有可能的需求，有时候我们必须自己动手封装 一些功能（比如说，log_file()这么方便而好用的功能，mudos 就没有提供）如果应用范围很 窄，我们当然可以直接把他写到代码里，但是像 log_file 这么大众化而且常用的功能，我们不 可能每次用到就 c/p 到对应的文件里去，一个是不方便，一个也容易出错不是？

于是，mudos 提供了一个方法来解决这个问题，这就是 simul efun，简单说就是“模拟的 efun”，这个功能可以帮助我们封装合心意的函数 并且在 lib 的任何地方像使用 efun 一样的使用他（虽 然说慢点吧，当然从效率上说，最快的办法是把 efun 移植到 mudos 里，真正做成一个 efun，然而这是另外一个话题，不是我们讨论的范畴）

完成这个工作所需要干的事儿不多，也很简单，就是把函数定义和体写到 config.cfg 指定的 simul_efun.c 当 中就好了（具体请参考我的 lib 当中 simul_efun.c 里的 log_file()函数）这 样 我们就能让他像个 efun 一样的工作了。

北大侠客行

题外话 1 很多 mudlib 经过了很长时间的发展之后，积累了大量的 simul，统统塞到 simul_efun.c 当中也很

困扰，不美观也不便于管理，因此他们会采用另外一个做法：继承。

简单说就是把 simul_efun.c 当作一个入口文件 把所有的 simul 函数分门别类的写到其他文件当 中去，在主文件里只用继承(inherit)的方法来把这些 “其他文件” 包 含进来 (用 include 貌似也可吧。我不是很确定)

题外话 2 关于 lpc 的 OOP 特性

lpc 这门语言出现的时间较早，虽然一般来说我们认为他是面向对象的，但是也并不完全符合面 向对象的所有观念。

比方说，他只支持对对象的函数引用，而不支持对对象的元素引用，当我们想获取或者改变一 个对象中的变量时，我们只能通过函数来实现；

再比方说，对于封装来讲，LPC 并不能完好的实现多态，并且只有限的支持变长参数表(varargs，lpc 只支持一个变长参数)。

题外话 2 纯熟废话，有兴趣的同学可以自己参考 OOP

重点 2.master.c 当中的 connect() 函数

我们本讲开篇就说了，当下最重要的是让 mudos 能够接受连接。所以这里我们有必要讲讲 mudos 接受用户连接的机制。

我们知道，LPC 最有趣的地方就在于他是 OOP 的(不完全无所谓，反正了解到 lib 里所有的东西都 是对象就 ok 了)那么给每个用户连线分配一个 object 是很正常也是很方便的管理方法。问题 来了，如何让 mudos 知道 “这是一个连线物件” 呢？

这里就用到了一个很重要的 master_apply(这玩意儿我们下边讲到)，master_apply::connect() 他的原型是：

object connect()

这个函数的作用就在于，当 mud 接受到一个连接之后(不管你是用 zmud 还是 telnet，总之你连上 了 mudos 提供的端口服务) mudos 会调用 master 的 connect()函数，并且期望返回一个对象。

这个对象就相当于在 mudos 里挂了号了。mudos 会把它当作一个用户连线对象来对待，比如说会 认为他是 userp()和 interactive()之类的 (这几个函数有一些细微而且诡异的差别，有空我们 再说)。

北大侠客行

请看我的 lib 里的 connect()：

```
 object    connect()
{
        log_file("new_user_login",time()+"\n");
        return    new("/obj/user.c");
}
```

很简单，master.c 只要被动的等待 mudos 的呼叫，并且在合适的时候造一个 user_ob 就 ok 了。

这里多讲一句： 一个完整的登录过程，在 mudos 当中

包括两步： 调用 master apply 的 connect()获取连线物

件； 调用这个连线物件身上的另外一个 apply 函数：

logon()

其中第二步是为了给登录验证过程一个合适的调用接口用的。目前我们 0.1 版的 lib 还用不到这 么牛 b 的技术，所以他就悬空了。

未来适当的时候我们再补充上。

重点 3.master 的 apply

实际上master.c 当中除了构造函数 create()之外，基本都是apply。 什么是apply 呢？ 大概说一下，所谓 apply，就是那种由 mudos 隐形调用的函数接口，这些接口是为 mudos 提供服务的，我们通常在 lib 当中，只是被动的通过这个接口告诉 mudos 在某些情况下“可以”或者“不可以”，或者“应该是谁”这样子。

在普通对象身上的这种接口我们就把他叫做 apply，在 master 身上的就是所谓的 master apply

了。

通常大家见到的比较多的 apply 包括像 id(（被 present()隐含调用）像 reset(（被 reset 机

制调用)

master 里更多是跟权限有关的 master apply，比如 valid_xxx 一族~ ~ ~

ok，这里我们大概知道有这么个事情就 ok 了，具体以后碰到需要用的 apply，我们再像这次的

connect()一样讲解。

重点 4.globals.h

大家会不会遇到这样的问题，我定义了一个宏，却忘了他在哪个头文件里？或者是写一个.c 的 时候经常要包含无数的头文件，其实就为了他当中一两个宏而已？

感谢 mudos 的作者，他通过 globals.h 帮我们在一定程度上解决了这个问题： 当我们有些宏定义是非常全句化的(比如说对目录的定义，对一些重要 ob 的定义等等) 我们可

以把它们丢到 globals.h 里。

并且，更方便的在这里：我们不需要显性的在程序里 include 这个 globals.h，mudos 自动的帮我 们在每个.c 当中都包含了它。

所以，当你有一些宏很全局的时候，尽管塞给 globals.h 吧 ^_^

另外一点，请看现在这个非常简单的 globals.h

```
#ifndef __GLOBALS_H_____
#define __GLOBALS_H___
#define LOG_DIR "/log/"
#endif
```

发现有什么不一样没有？

我们使用了#ifdnef #define #endif 的方式。 这样做的好处是，避免出现由于.h 嵌套而导致的 redefine (比如说 a.c 包含了 a.h 和 b.h，不幸的

是 a.c 同时继承了 b.c，而 b.c 自己也包含了 b.h，这样就出现了实际上的嵌套)

重点 5.user.c

我们的 user.c 现在功能很简单。在正常的 lib 当中，为了方便登录认证和断线重连，这个文件被 分成了两个 即 login.c 和 user.c（也许有人直接是用 body.c？我不太确定 总之是这么个意思）。一个负责密码认证，一个负责真正的用户行为和数据，并且通过 exec()函数把 mudos 认为的用户 标签在两个之间切换。

现在我们这里非常简单，不认证，所以直接就只有一个 user.c，未来再细化这里。 接下来，mudos 如何获取用户的指令呢？ 在正常的 mudlib 当中，我们通常通过一系列复杂的行为来实现之，例如通过一个“全时的”的

add_action(command_hook)钩子来获取用户输入 并且通过 commandd.c 之类名字的一个 daemon 来找到恰当的指令文件，并执行之。

未来我们也会完善这样的一套结构，不过现在么，既然我们只想完成一个 echo server。我们暂时用一个更简单的方式来处理他：string process_input(string arg)

这又是一个 apply，在“正常的”mudlib 里，我们通常用这个 apply 来实现 global alias 的解析，这里我们就直接通过

```
string process_input(string arg)
{
        write(arg+"\n");
}
```

这样的模式来完成“echo server”的工作了。

这边有个问题，大家有没有发现一个 string 型的函数我没有 return，这直接导致跑起来之后， 我们每个指令敲进去都会返回一个>what?

不要紧，反正这个版本我们只是大概的演示，下一次我们修好他。 好了，这次一共就四个文件，我们构建了一个可以跑起来并且有反应的 lib 了。大家可以跑一下

看看（附件里我提供了一个 bin 文件，包含了 driver 和 config.cfg，是 linux 版的，由于我在我

的服务器上跑的 driver 非常多，所以我给 driver 改名叫 mud 以避免不幸被 killall 掉 ）

如果您想在 linux 下跑，只要 chmod +x mud，并且重新配置一下 config.cfg 里的 mudlib 和 bin 的 目录就 ok 了。

如果您想在 win 下测试，那么很抱歉，我手头没有合适的 mudos.exe 版本（哪位好心的老大提供 一个）

小结和预告

第一次写这种东西（声明一下，我没写过语言类的书，所以不太会写，可能结构太乱，内容也 很分散，抱歉）大家有啥建议和意见欢迎跟帖，我尽量改进。

附件是两个文件：bin.tar.gz 是 linux 版的 mudos 和 config newlib.0.1.tar.gz 是和本讲座同步的 lib。可能有个问题，就是里边涉及到的中文是 utf8 格式

的，大家用起来可能是乱码，这个倒是可以用 uedit 转一下码（主要是我习惯了在 shell 下直接

敲代码，也就不改编码了，好在作为一个讲解用的 lib，不会有太多中文内容在里边）

下一讲，我想我会把登录验证和指令系统补全。这样我们就可以有空间做一些实验了（用指令 来写测试代码很方便的说）

另外，如果可能，也许会补一个不那么完善的权限系统，也许不会，我看进度把。 补充一句，bin.tar.gz 里用的就是标准的 v22pre11 的 mudos，我只做了两个小改动： 是在 mudos 跑起来的时候会生成一个以 config 文件名+”.drtmp”为名的临时文件，里边存放

了 mudos 自己的 pid

是稍微修改了 efun::sprintf() ，让它能够支持 functionp(，这是当时我为了简单化的实现背 包内物品存储时打的补丁，防止带有 functionp 的物品存储出错）

两个都没有什么实际用途（起码对这个 lib 来说是如此）所以大家如果用没有修改过的 pre11 也没问题的。

补充一下，上文说到的诡异的三个函数是：

userp()，interactive()和 living()

附件

跟 Akuma 一起从头打造 mudlib--【第一讲】 作者:Akuma

第二讲：指令系统

北大侠客行

我思考了一下，决定暂时跳过登录部分，先说指令。毕竟有了指令之后，就可以在线更新和重 启，不用不停的 kill driver 进程了。

在开始之前，我们先思考一个问题，指令是干什么用的，以及我们希望如何管理指令系统。 我们常讲一个词叫 I/O，也就是所谓的输入输出。 从最基本的意义上说 指令就是玩家敲的东西 他希望通过敲一些东西 告诉 mud “我要做什么”。

抛开网络层和 mudos 底层不讲 我们可以把指令看作是一个字符串事实上他的确是一个字符串，并且是以\n 或者\n\r 结尾的字符串，也就是说，很不幸，我们的一 telnet 为基础的 mud 不支持指 令 或 者 指 令 参 数 里 带 有 回 车 ^_^)

然后，mudlib 根据一定的规则，找到一段适当的程序去解释玩家的指令，并且给予适当的操作 和反馈。

所以我们第一步可以把指令系统归结为如下的需求：

想办法获得玩家的输入

注意，这个规则一定要是规范的，这个规范我们可以随意制订，比如说用 100 到 999 的三 位数字表示特定含义的指令；当然了，理论上这虽然没问题，但是考虑到我们的 zmud 或者 telnet，玩家的指令是手敲的，你让他记这么多数字不太现实。

于是我还是决定继承传统 mud 的方法，用 “指令英文+空格+参数表” 的方式来设计这个规范。

我们定一个规则，让玩家输入的指令可以在 mudlib 里找到那段我们希望他来执行的代码去执行之。

如前所述，我们已经假定了指令规则是 “指令英文+空格+参数表”，那么指令的英文名字本身 就是一个很好的代号，我们只要通过一定的编码，赋予不同代号对应的程序段就好了。

本次讲解的内容分成两个部分，第一部分是简单实现指令，第二部分里我们再进一步细化它。

第一部分代码见附件 newlib.0.2.1.tar.gz

目录结构如下：

```
|-- adm
```

```
|   |-- daemons
| |`-- cmd_d.c
|   `-- obj
|   |--   master.c
|   `--   simul_efun.c
|-- cmds
|   `--   usr
|   `-- test.c
|--   include
|   `--   globals.h
|-- log
`-- obj
        `--
    user.c
```

重点 1.我们首先来解决“获得玩家输入”的部分

请看这个版本的/obj/user.c。大家还记得吗？上次当我们写第一个 echo server 的时候，留了 个尾巴。当时我们是通过 process_input()来“处理”指令。

这一回，我们希望可以把指令系统建立起来，因此我把 process_input()的 处理暂时注释掉了。 如何获得玩家的输入？

这里我继续使用一般 mudlib(es2 类)的做法，通过一个“全局的”add_action()来实现。请看代码：

```
 void   create()
{
        setup();
```

```
}
int     setup()
{
        log_file("user",sprintf("%O setup
    at   %s\n",this_object(),ctime(time())));
        enable_commands();
    add_action("cmd_hook","",1);
    return   1;
}
```

正常来说，我们不应该这么早进入这个部分，不过由于暂时还没有登录认证体系，就先这么凑合。这里只是用来表述流程：

一个 user_ob 被 master.c 的 connect()创建时（记得吗？这个时候他就是一个真正的连线物件了），apply 函数 create()被呼叫，于是他调用了 setup()。

我们可以看到，setup()只干了两件事：

```
enable_commands();
add_action("cmd_hook","",1);
```

add_action()大家在制作谜题的时候会经常遇到，就是给某个物件增加一个“临时的”指令，并且指定用于解释这个指令的函数名字。所有“接触到”这个物体的物体，都可以使用这个指令。我们这里的用法比较奇怪一些，他大概的意思就是“不管你输入什么，我都认为你是我这个 add_action 定义的指令”。

我一直觉得这个写法挺古怪的，不过为了兼容和尊重传统，我们继续这么搞吧。

enable_commands()是给物件设置一个标记，好让 add_action()有效。继续看 cmd_hook()这个执行函数。

```
int cmd_hook(string arg)
{
        string verb =  query_verb();
```

```
        return    CMD_D->do_cmd(this_object(),verb,arg);
}
```

query_verb()这个 efun 是用来获取玩家的“最近一个指令”，注意是指令，不包括参数表。参数

表是由 add_action 给定的解释函数（这里就是 cmd_hook 以参数形式传进来的）

然后，我们通过一个 daemons 来设法去执行这个指令（也就是 verb）

```
        return    CMD_D->do_cmd(this_object(),verb,arg);
```

CMD_D 一会再看。这里我们要注意的有一个要点，请看从 cmd_hook 被定义为 int 型函数开始，我 们一直在 return（CMD_D 里也是一路有 return 的）

为什么？这是 mudos 的一个机制 针对玩家的输入 我们一路处理下来 如果有为 true 的 return，那么系统会认为这个指令被正确的执行了。否则他就会想办法给出报错。

从我们之前看到的”>what?”或者一般 mud 里定义的“>什么？”，一直到使用 notify_fail 来自 定义的错误返回等等。

重点 2.规范的管理和找到指令对应的处理函数

在前边我们已经获执行指令取到了所有必要的信息，包括指令名字，参数表（还有是“谁”的 指令）接下来我们就要开始想办法执行他了。

请看/adm/daemons/cmd_d.c（有些 lib 里可能叫做 commandd.c 之类

的） 目前我们只有这么一句

```
int do_cmd(object me,string    verb,string    arg)
{
        return    ("/cmds/usr/"+verb)->main(me,arg);
}
```

这也就是刚才在 cmd_hook()里调用的函数体了。

这里不用多解释，我们就是通过”/cmds/usr/”+verb 拼出了需要使用的指令的文件名字，然后 把“谁”和“什么”传到他的 main()函数里了。

其实在 lpc 里，main()并不是一个特定的函数，指令里用这个函数名作为入口，我猜是当年某个 习惯 c 的人随手定的规则。不过为了讲解的作用，我们还是继承下来。

我们特意创建了/cmds/usr/test.c 这样一个指令，大家可以自己看看。

ok，现在我们的指令流程已经初步创建起来了，如果你现在启动我们 0.2.1 版的 lib，连接上去，敲：

```
test    xxxxx
```

就会获得如下的反馈：

in cmd test: arg=xxxxx

这说明指令 test 已经正确的被执行了。 兴奋的我们准备再敲点别的。比如 asdfasdf asdfsdf 这样子。嗯……没有反应。

如果你这个时候去看/log，就会发现多了一个 error_handler 目录。很不幸，一般来说，看到这个就表示我们的 lib 有“runtime error”（运行时断错误，也就是说，程序可以被编译，没有语 法错误，但是在运行中出了错）

这说明我们的程序缺乏起码的容错性。

这也就是我们在 lib.0.2.2.tar.gz 当中需要解决的内容。

有点小忙……

稍后我把第二讲的下半部分补全。需要解决三个问题： 鲁棒性，无论如何我们不希望总出现 error_handler。也就是说，无论玩家怎么乱输入，我们都应该可以处理，而不是任由它报错。

便于管理，我们希望找到一个方法，让指令的处理文件有序的呆在一个地方，并且可以方便 且正确的被程序找到。

安全性，即权限，看本文的 wiz 居多，我们总不希望玩家可以随便执行巫师指令吧。那么我 们就要想个办法，让适合的人能执行适合的指令，而不是相反。

请等待第二讲－下～～谢谢。

附件

跟 Akuma 一起从头打造 mudlib--【第二讲】 作者:Akuma

第三讲：dbase 和登录过程

这几天过 10.1，反而比较忙，不常在家。 所以估计这八天里也就是今天这一讲了。 实际上 lib 的内容昨天就写完了。关键是怎么讲的问题。 第二讲里，本来说是准备接着把指令的权限部分完成，结果仔细想了一下，发现没法搞。原因有二：

第一个是，还没有登录过程，登进来的 user_ob 都不知道谁是谁，如何定权限？ 第二个是，一旦说到登录，那事情就变得复杂了，最起码的，"是谁" 的问题，也就是 id 怎么

存到 user_ob 身上呢？这起码应该有个 dbase。

所以想了半天，决定把这两块先补充上，然后再继续写权限的部分。 因此这一讲有两个重点，一个是 dbase 和相关的函数，另外一个就是做一个简单的登录过程。在

这之后后再按顺序把 save restore，security 这些搞定，我们就可以继续回到指令权限这个内 容上来了。

请看目录（对应的文件见附件 newlib.0.3.tar.gz）:

```
      ".
|-- adm
|   |-- daemons
|   |     |-- cmd_d.c
|   |`--   login_d.c
|   `-- obj
|   |--    master.c
|   `--    simul_efun.c
|-- cmds
|   `--    usr
|   `-- test.c
|--    include
|   `--    globals.h
|-- log
|-- obj
|   `--    user.c
`--    std
      `--
   body
         `-- user_dbase.c
```

重点 1.dbase

我们依然秉承这样的原则：所有新添加的内容都是暂时够用就好，打底子的东西我尽量细致的 讲。这块熟悉了，以后我们根据需要添加细节就容易了，那个时候反而可以一笔带过。

请看/std/body/user_dbase.c。一般来说，es2 继承过来的 lib 都是爱用/feature/dbase.c 这样 的结构。我自己受到 kok lib 的影响比较大，所以倾向于用/std/body 这样的结构，这个不是很 关键的东西。大家只要知道，如果要对照，就去看 /feature/dbase.c 就好了（一般来说 dbase 又是从 treemap 继承过来的，不过我个人比较喜欢不超过两层的继承关系）

dbase 是什么？ 如我们之前所讲过的，lpc 并不是完全意义上的面向对象，他不支持对数据域的直接访问，因此

这个 dbase 就是对 ob 身上的数据以及数据处理接口函数的一个集合。 这里我们这个 lib 第一次用

到了继承（inherit）具体如果想了解，大家可以搜一下面向对象有关的说明，一般性的，只 要知道被继承的文件里的数据和函数可以直接当作继承后的对象身上的数据和函数就好。

以我们现在使用的 user_ob 为例，我们总是需要知道、或者设置他"是谁"，"有多少 hp"这样的 问题，归纳起来，最常用的可能就是"哪个属性"是"多少"这样的话题了。对比 lpc 的数据类 型，这正是 mapping（映射）的拿手好戏。

所谓 mapping，其实在更通用的数据结构书里，也就是 hash 表（哈希表）这东西的内部机制不 讲了，大家只要了解，他分为 key 和 value 两部分通常可以写成(["a":1,"b":2])这样的形式 ， 并且可以用 map["a"]这样的形式来引用和赋值。

所以我们第一步就是给 user_ob 设计了一个叫做 dbase 的 mapping，用来存储大部分的数据（事实 上，包括 ROOM，NPC，item，都是有类似的结构，一般的 lib 也都是继承了同样的 dbase.c，但是 我们这里暂时只给 user_ob 使用，因此叫做 user_dbase.c，至于以后其他的对象是否也继承它、 还是单独写各自的 xxx_dbase，要根据需求再定）

具体请看

```
mapping dbase = ([]);
mixed set(mixed prop,mixed
data)
{
        return dbase[prop] = data;
}
mixed query(mixed prop)
{
```

```
        return    dbase[prop];
}
void delete(mixed prop)
{
        map_delete(dbase,prop);
}
```

现在这个版本相当粗略，我们只定义了一个叫做 dbase 的“全局” mapping，和三个函数

set(),query(),delete()。

事实上，真正使用的 lib 里，往往要比这个复杂的多，至少有三处： 多一个 temp_dbase 的 static mapping（在不同的 driver 下，static 也可能被写作 nosave，表示不会被 save_object()所整理存盘）用来存储那些只在对象存在过程中被使用，但是不需要存盘的数据；

至少还应该有一个 add()函数（实际上就是 set(xxx,query(xxx)+data)） 这个 dbase 应该支持多层结构，也就是他的任何一个 data 都可以又是个 mapping，这就是我们的 prop 用 mixed 来定义的原因

关于这个，我们多讲一句，set(“aaa/bbb”,xxx)这种结构大家可能都见过，这就是表示 aaa 这个 key 的内容是([“bbb”:xxx])。这会使得我们的这几个函数变得负责不好读，而且暂时也 用不到，因此我们把这个需求留待以后再处理。现在我们只要能支持读写单纯的 key = value 形式就好了。

有了 dbase，我们就可以读/写用户物件上的数据了。

重点 2.登录过程

很遗憾，现在这个版本的 lib，我们依然没法把它写完整。为什么呢？因为我们还没有 save/restore 机制。

请看 user.c

```
 int setup();
void create()
{
//    setup();
}
/*
string    process_input(string arg)
{
        write(arg+"\n");
}
*/
//    apply of    user_ob :: logon()
//    after master->connect(),mudos calls user_ob->logon()
//    we    use this to    put user_ob into    LOGIN_D
//    and start Login
authentication void logon()
{
        call_out("login_timeout",60);
        this_object()->set("login_temp"
    ,1); LOGIN_D->logon(this_object());
}
void login_timeout()
{
        if(this_object()->query("login_temp"))
                destruct(this_object());
}
```

这里我们调整了一下，把原来在 create()里的 setup()过程干掉了，增加了一个叫做 logon()的
函数。

注意，这是一个 apply，它在 master apply::connect()之后会被调
用。 我们用这个函数把已经获得了连线物件权限的 user_ob 发给了
LOGIN_D。

```
 void    logon()
{
            call_out("login_timeout",60);
            this_object()->set("login_temp"
       ,1); LOGIN_D->logon(this_object());
}
```

注意这里有一个 call_out，用来防止有用户连进来什么都不干，所以有个 60 秒的超时处理。然

而为了防止正常登录用户也在 60 秒之后被踢。所以用一个 login_temp 的标记来做判断，这个标 记会在 LOGIN_D 成功登录之后被清理掉。从而保证登录进来的用户不会被踢（这里本来应该是 个 temp mark，不过咱们没有，就先凑合用 dbase 存）

未来/adm/daemons/login_d.c 会被用来处理整个的登录握手或新用户创建过程。只是目前还是 一个很粗糙的结构。

```
 void    logon(object ob)
{
            write("Welcome to my New mudlib\n");
            input_to("get_id", ob);
}
```

这是被 user_ob 的 logon()所呼叫的处理函数，你看多好。连 WELCOME 都有了 。

我们直接用 input_to 获取用户输入的 id。

正常过程，这里应该判断是否有该用户，用户 id 是否合法，然后是获取密码，对用户 data 做

restore 等流程。这里就简化了。

总之最后是进入到 enter_world()过程 应该把用户丢到合适的 r oom(可惜我们还没有 room，，，)，给用户适当的权限(我们也没有)打开用户的指令输入权利(这个有了，就是 user_ob→ setup()，大家还记得吧，之前被我们从 create()里干掉的 setup()终于又被我们找了个机会用 回 来 了)。

这里打一个补丁，setup()里我们增加了 set_living_name()这个函数，是为了让 find_player() 可以找到这个“玩家”。

如此，一个完整的登录过程，虽然粗糙，但是结构有了。 下面我们就可以从容的完成权限分配，save/restore 等 内 容 了 。 请 期 待 下 一 讲 ~ ps.随着 lib 逐渐搭起来，我也很犹豫，到底是把一些我们后来发展起来的比较新的模式放进去，

还是尽可能贴近大家比较熟悉的老 lib。

不过考虑到这个 lib 的讲解的性质，还是决定尽可能贴近老 lib，毕竟大家用的熟了，更多还是 在这些体系下写东西。

附件

跟 Akuma 一起从头打造 mudlib--【第三讲】 作者:Akuma

第四讲

存盘和读取的底层机制 简单的房间系统和简

单的 look 指令，物件的 move 指 令 权 限 还 是 没

有 ……

登录过程里依然没有密码验证，但是可以区分新老用户了，并且可以存取数据。 这是本章的基本内容。具体请见附件。

详细的文字说明，我明天补上。

（抱歉我老婆发烧，这两天都在照顾她，晚上才赶出来的 code，说明就实在来不及了） 另外注一下，所有文件当中的中文都是 utf8 格式，大家用的时候可以批量转一下码（utf8>gb）现在已经可以这样了：

欢迎来到测试 MUD

请输入您的 ID:akuma 读取存盘
数据成功，开始进入游戏。 世界的中
心 -

这里就是世界的中心。人来人往，车水马龙。到处都是人，摩肩接踵的，非常热闹。

这里有：
akuma
> test 连线进入这个世界。 目录结构如下：

```
|-- adm
|   |-- daemons
|   |     |-- cmd_d.c
|   |     `--   login_d.c
|   `-- obj
|   |--   master.c
|   `--   simul_efun.c
|-- cmds
```

```
|   `--   usr
|   |--   look.c
|   `-- test.c
|-- d
|   `-- wiz
|   `--   center.c
|-- data
|   `-- user
|   |--a
|   |   `-- akuma.o
|   `-- t
|       |-- test.o
|       `--   test2.o
|--   include
|   `--   globals.h
|-- log
|-- obj
|   `--   user.c
`--   std
    |-- body
```

```
|   |-- move.c
|   |-- room_dbase.c
|   |-- user_dbase.c
|   `--   user_save.c
`-- room.c
```