如何编写自己的RPG游戏

Anjoy@USTC

July 15, 2012

目录

1 写在前面的话 1

2 工欲善其事, 必先利其器 2
2.1 一门合适的编程语言 2
2.2 一个高效的编辑器 2
2.2.1 VIM的安装和配置 3
2.2.2 EMACS的安装和配置 8
2.2.3 其他编辑器 8
2.3 选择一款合适的游戏引擎 8
2.3.1 一门合适的脚本语言 8
2.4 编译器 9
2.4.1 GCC 9
2.4.2 VS2010 9
2.5 选择合适的调试器 10
2.5.1 GDB 10
2.5.2 VS2010 10
2.6 Makefile使用 11
2.7 如何管理你的代码 11

3 选择合适的图形库 12
3.1 需要哪些功能 12
3.1.1 图象 13
3.1.2 声音 13
3.1.3 输入 13
3.2 ALLEGRO 14
3.3 SDL 14
3.4 HGE 14

4 资源压缩与打包 16

4.1 资源压缩算法 17
4.1.1 LASS算法 17
4.2 资源打包与提取 17
4.2.1 散列算法 17
4.2.2 FVN算法 18
4.3 D2P文件结构 19

5 游戏地图设计 22
5.1 地图要素 23

6 游戏角色设计 24
6.1 人物数据结构组织 25
6.1.1 25
6.1.2 25
6.2 Makefile使用 25

7 RPG游戏中的人工智能 26
7.1 AI流程图 27
7.2 常见人工智能算法 27
7.3 D2IM中用到的AI算法举例 27

8 游戏脚本系统 28
8.1 D2S系统的思考 30
8.2 脚本系统基本要求 32
8.3 D2S脚本解析系统 32
8.3.1 注释语句 32
8.3.2 变量及其赋值 32
8.4 数组 33
8.5 命令分割和组合 33
8.6 命令组合 34
8.7 单步和并行 34
8.8 脚本调用 36
8.8.1 IF语句 36
8.8.2 WHILE语句 37
8.9 逻辑运算 37
8.10多选择分支 38

9 关于D2IM 85

9.1 游戏背景 86
9.2 框架结构 86

Chapter 1

写在前面的话

角色扮演游戏(即ROLE-PLAYING GAME,缩写为RPG)是一种深受玩家喜爱的游戏模式.在游戏中,玩家扮演虚拟世界中的一个或者几个特定角色在特定场景下进行游戏.角色根据不同的游戏情节和统计数据(例如力量,灵敏度,智力,魔法等)具有不同的能力.而这些属性会根据游戏规则在游戏情节中改变.

作者一直想自己写一个有意思的RPG游戏,像暗黑破坏神和热血传奇那样的游戏.这种冲动的来源很简单,就是作者曾经痴迷于这两款游戏无法自拔,甚至不上课不考试,最后不得不复读了一年.上大学后,开始接触到一些牛人,原来他们都鄙视把时间浪费在玩游戏上.在他们的气场前作者感到很自卑很渺小,于是想着自己什么时候能玩出名堂.然后作者就接触了Linux,开始痴迷于它并各种尝试在上面写自己的代码.

后来作者的游戏瘾慢慢的消失不见了,随之而来的coding热情大增,因为作者没有其他的爱好.然后有了这份文档,希望你看到它后能少走作者走过的弯路.所有的源代码都是开放的,包括这份文档本身的latex代码.是的,作者崇尚开源精神,哪怕作者力量微薄.

准备好了么?那我们开始吧!

Chapter 2

工欲善其事，必先利其器

既然我们准备实现我们宏伟的目标,那么首先我们就需要为实现这个目标做好准备,所谓磨刀不误砍柴工.急急忙忙的开始并不是个好主意.那么我们先来精心挑选自己的工具吧.我们需要准备些什么呢?

2.1 一门合适的编程语言

可以使用的编程语言太多,作者也没有兴趣一一介绍.D2IM的开发使用了C语言.因为在作者眼中C语言是高效而优美的(很多人可能并不这样认为),你也可以使用C++或者JAVA,这是你的自由.

2.2 一个高效的编辑器

作者是VIM的脑残粉,所以作者毫不犹豫的推荐VIM.黑客世界中还有一款极其著名的编辑器叫EMACS,关于VIM和EMACS孰优孰劣,V党和E党常年论战但无定论.作者并不想引起争端,并且作者没有用过EMACS所以不便妄谈,总之最适合你的编辑器便是最好的编辑器.

作者将在下面给出VIM和EMACS的简单安装和配置方法.如果你使用的是其他编辑器,你可以跳过这两个小节,也可以在这两小节中选一种作为入门学习.学会任何一种都会终生受益的,请相信作者.

2.2.1 VIM的安装和配置

VIM是免费软件，官方网站为http://www.vim.org/，你可以在这上面找到它的下载连接。你只需根据自己的系统选择对应的下载即可。

Figure 2.1: VIM LOGO

如果你是windows用户，你可以点击这里下载VIM在windows下的安装文件，这个链接版本是V73.46，你也可以进入官网下载最新版本。下载完成后安装即可，注意在“Install Options”选项框出现时（如图2.2），至少选中下列选项：

- Vim executables and runtime files
- Vim console program (vim.exe)
- Create .bat files for command line use
- Add an Edit-with-Vim context menu entry
- Create a _vimrc if it doesn't exist
- Create plugin directories in VIM

如果你的WINDOWS是中文版，注意去掉Options中的“Native Language Support”,因为这个选项可能导致VIM的提示行显示乱码。

安装完成后会弹出一个VIM README文件，这时候关闭即可，如果你关心你也可以在...找到这个文件。

然后我们尝试安装VIM的中文帮助文档。VIM自带英文文档，如果我们需要我们也可以安装中文文档，这将对英语不好的同学提供很多便利。WINDOWS用户可以点击这里下载VIMDOC的汉化版。注意在选择VIM所在的安装文件夹时，安装程序尝试寻找vim.exe文件所在的目录，但是如果程序找不

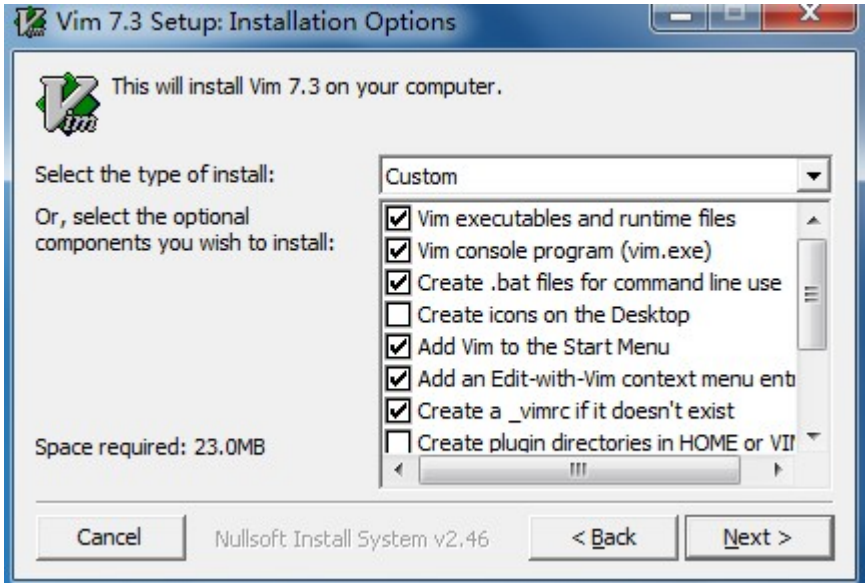


Figure 2.2: VIM安装

到，就需要你为安装程序指定（如图2.3）。指定VIM所在目录时一定要指定为vim.exe所在目录，否则安装完成后可能VIM找不到中文帮助文件。

如果你的系统是debian/ubuntu，你可以使用如下命令安装vim到你的系统上：

```
sudo apt-get install vim
```

你也可以选择使用源码编译的方式安装vim，这将给你更多为VIM自定义的便利。

```
./configure --with-feature=huge     \
            --prefix=/usr/bin       \
            CFLAGS="-O3 -????"
make
sudo make install
```

```
Align一个对齐的插件用来排版面对一堆乱七八糟的代码时用来对齐代码功能强大不过用到,,,,,的机会不多
```

Figure 2.3: VIMDOC安装

```
http://www.vim.org/scripts/script.php?script_id=521
Mru
http://www.vim.org/scripts/script.php?script_id=521给增加功能
vimMRU也就是保留最近打开的文件记录,,:打开MRU,退出q很方便有过一个支持,,菜单的
    类似的插件不过对于我这样的不用菜单的用户还是这个命令行的好用一点因为经常使
    用所以我映射

,,,到了
F2功能强大的代码注释工具用来注释或者取消注释支持很多语言可以对文本块操作写代码
,,,,
NERD_comments功能强大的代码注释工具用来注释或者取消注释支持很多语言可以对文本
    块操作写代码
,,,,离不了呵呵
,最常用到的快捷键是
"c

a.vim在
.c/.之间切换h写代码必备,

bufexplorer.vim列出当前打开的
buffer可以很容易的切换到和删除选定的,buffer必备插件之一,

c.vim
c/c++ support让你用编写,c/c程序时如虎添翼有很多贴心的功能每个功能都有快捷
    键++,,不过一部分和
,冲突NERD_comments如果经常编写一些单文件的程序
c但是不想写,makefile用这个他帮你完成,,,编译并链接F9,
```

```
ctrl-运行F9

calendar.vim日历插件有了它用
,,来写日记很方便vim

csExplorer.vim
color 浏览插件theme列出所有的,vim color 到一个列表中theme选中后按回车即可
    应用相,应的
color theme试验,color 时再也不用一次次输入theme:colo 了theme_name从上百
    个,
color 中选择自己喜欢的时有用themetheme

cscope_maps.vim的插件
cscopevim提供快捷键操纵,cscope好东东如果你在用,,的话cscope

favex.vim
FavEx : Favorite file and directory explorer 可以添加目录和文件到收藏
    夹，可以把,经常编辑的文件添加到收藏夹来，在文件打开以后，
"新增文件到收藏夹，ff"新增目录到fd收藏夹



lookupfile.vim五星级推荐的好插件我觉得它是
!上最伟大的插件之一vim提供多种方式查找文件让你在复,,杂的目树中也能轻松自如找到
    你要的文件
?

matchit.vim扩展了的
vim功能让可以匹配的不再仅仅是括号支持多种语言必备插件之一%,%,,.

parenquote.vim给选中的文字加上引号支持
,( { [ < ' " 选中后`,,"加上你想要添加的符号比如选中,abc后
,"得到(,(abc)

snippetEmu.vim扩展了的缩写功能
vimabbr支持占位符支持变量替换强烈推荐,,,

taglist.vim的代码浏览器
vim生成函数列表支持跳转可以根据光标λ置查询到当前的函数名使用,,,,的程序员必备
vim个人认为是最伟大的插件之一!

utl.vim给增加的识别功能
vimurl但是功能远不只是支持,url还有更多详情见,,的帮助utl

vcscommand.vim给整合了
vimcvs/功能subversion不用离开,环境也能执行常用的vimcvs/操作了subversion
```

```
viki.vim的
vimwiki没怎么用过据说很好用详情可以看滇狐的主页,,,
http://edyfox.codecarver.org/html/viki.html

vis.vim可以对选中的文本块执行操作
ex尤其是,visual 模式下block,自己是不支持的vim选中后.,
:B 加上命令ex

visincr.vim给增加生成递增或者递减数列的功能
vim支持十进制十六进制日期星期等功能强大灵活,,,,,

winmanager.vim给增加的功能
vimIDE提供目录浏览和,浏览功能buffer因为显示器太小感觉太占空间所以,,,单独使用
bufexplorer而且现在,的功能也够强大vim7netrw所以感觉比较鸡肋而且貌似很,,久没
    有更新所以基本不用
,

yankring.vim类似的
emacsking ring给,的也增加缓冲vimyank,本身只缓冲删除的字符串vim不缓
    冲,yank的内容
```

安装完成后，需要对VIM进行一定的配置。如果是WINDOWS，配置文件在VIM安装目录下（如图），名称为_vimrc，如果系统为Linux，则VIM配置文件存在于$HOME文件夹下，名称为.vimrc。打开对应配置文件，删除所有默认配置并加入以下内容：

```
set nocompatible
set number
set wrap
set autoindent
set noswapfile
set cindent
set nobackup
set expandtab
set smartindent
set nowritebackup
set noswapfile

set fileformat  =unix
set tabstop     =4
set shiftwidth  =4
set backspace   =2
set showtabline =0
```

```
set mouse       =
set selectmode  =key
set display     =lastline

syntax      on
color       murphy
filetype    plugin on
filetype    indent on

map j               gj
map k               gk
```

随着你对VIM的逐渐熟悉，你会继续加入一些自己的设置，上面过的设置只是最基本的。

2.2.2 EMACS的安装和配置

我不会用EMACS,等待有人来补齐这一章.

2.2.3 其他编辑器

我知道的有notepad++和everedit。

2.3 选择一款合适的游戏引擎

D2IM是2.5D游戏,所以我们需要一个合适的2D游戏引擎.作者接触过并使用过的游戏引擎有SDL,HGE和ALLEGRO.我将分别介绍它们.

2.3.1 一门合适的脚本语言

脚本语言是游戏可扩展的重要组成部分.使用最普遍的游戏脚本语言为LUA.

我们将引入自己的脚本语言,我们给它命名为D2S.在本书中,我们将详细的探讨如何构架并实现这个脚本解析引擎.

2.4 编译器

由于D2IM使用了C语言,故必须有一个C语言编译器.我使用的是GCC,你也可以使用微软的编译器.

2.4.1 GCC

我既不是Linux版主, 也不是Linux协会负责人, 我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样, 有一件Linux版衫. 4月5日stephenjy发帖求版衫 , 随即李喵喵投稿, 一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫, 同样的时间我在Linux协会邮件列表里也求, 一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上, 我直接跟版主说, 于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月, 仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫, 商量到钱的问题的时候向版主询问建议, 版主的回复是这种细节问题你们自己解.

2.4.2 VS2010

我既不是Linux版主, 也不是Linux协会负责人, 我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样, 有一件Linux版衫. 4月5日stephenjy发帖求版衫 , 随即李喵喵投稿, 一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫, 同样的时间我在Linux协会邮件列表里也求, 一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上, 我直接跟版主说, 于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月, 仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫, 商量到钱的问题的时候向版主询问建议, 版主的回复是这种细节问题你们自己解.

2.5 选择合适的调试器

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

2.5.1 GDB

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

2.5.2 VS2010

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

2.6 Makefile使用

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

2.7 如何管理你的代码

推荐使用SVN/GIT管理你的代码.共享你的源代码吧 我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

Chapter 3

选择合适的图形库

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.1 需要哪些功能

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.1.1 图象

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.1.2 声音

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.1.3 输入

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.2 ALLEGRO

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.3 SDL

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫，商量到钱的问题的时候向版主询问建议，版主的回复是这种细节问题你们自己解.

3.4 HGE

我既不是Linux版主，也不是Linux协会负责人，我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样，有一件Linux版衫. 4月5日stephenjy发帖求版衫 ，随即李喵喵投稿，一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫，同样的时间我在Linux协会邮件列表里也求，一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上，我直接跟版主说，于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月，仍然只有李喵喵同学一份投稿. 于是我和李

喵喵两个人决定着手开始做版衫, 商量到钱的问题的时候向版主询问建议, 版主的回复是这种细节问题你们自己解.

Chapter 4

资源压缩与打包

游戏中的脚本, 贴图, 声音, 配置文件等极多. 如果单靠建立文件夹难以为继, 一般的做法是压缩打包. 本章我们将简单LZSS压缩方法和打包文件制作, 并给出一个游戏资源管理器.

除了压缩与打包, 另一个重要的问题就是加密, 但是我们这里不涉及.

4.1 资源压缩算法

压缩资源的理由很简单,现在游戏传播一般使用网路分发方式,更小的文件大小会更方便.

压缩算法很多,我这里只详细的解释我们在D2IM中使用的LASS算法.

4.1.1 LASS算法

4.2 资源打包与提取

资源打包的最大理由是磁盘碎片.

然而打包后需要我们快速的在包文件中定位到对应的文件内容.D2IM中使用的通用打包文件后缀为.d2p,我们将在这里对其予以解释.

首先,我们使用文件名的方式对其进行索引.然而如何将多层级树形目录结构对应到片面列表结构,这里我们用到散列算法,d2p文件中使用的散列算法是FVN-1散列算法.这个算法对文件名类有很好的散列效果.

4.2.1 散列算法

散列算法又称又称HASH函数或者凑杂算法.HASH函数就是把任意长的输入消息串变化成固定长的输出串的一种函数.这个输出串称为该消息的杂凑值或者HASH值.一个杂凑函数应该至少满足以下几个条件:

- 输入长度是任意的
- 输出长度是固定的
- 对每一个给定的输入,计算出hash值是很容易的
- 给定杂凑函数的描述,找到两个不同的输入消息杂凑到同一个值是计算上不可行的

我们需要的HASH函数接口如下:

```
/*输入: filename 输入文件名路径名 /
 *       seed1/2 输入随机数种子   HASH输出
 *: crc1/2 输出校验
 *       return 值    HASH位宽由宏,决定D2P_CAPACITY_BIT_WIDTH
 */
uint32_t d2p_hash(
        const char *filename,
        uint32_t seed1, uint32_t seed2,
        uint32_t *crc1, uint32_t *crc2);
```

4.2.2 FVN算法

FNV哈希算法全名为Fowler-Noll-Vo算法,是以三位发明人Glenn Fowler,Landon Curt Noll,Phong Vo的名字来命名的.FNV能快速HASH大量数据并保持较小的冲突率,它的高度分散使它适用于HASH一些非常相近的字符串,比如URL,hostname,文件名,text,IP地址等.FNV算法有两个版本FNV-1和FNV-1a,下面给出两种算法的C描述:

```

#define FNV_32_PRIME     ((uint32_t) 0x01000193)
#define FNV_32_INIT      ((uint32_t) 0x811C9DC5)
static inline uint32_t fnv1_32(void *buf, size_t len)
{
    unsigned char   *bp   = (unsigned char *) buf;
    unsigned char   *be   = bp + len;
    uint32_t         hval = FNV_32_INIT;
    while (bp < be) {
        hval ^= (uint32_t) *bp++;
        hval *= FNV_32_PRIME;
    }
    return hval;
}
```

```
算法：高运算性能，低碰撞率，由
MurmurHashAustin 创建于年，现已应用到、Appleby2008Hadooplibstdc、++、等
    开源系统。年被雇佣，随后推出其变种的算
    法。nginxlibmemcached2011ApplebyGoogleGoogleCityHash官方网站：

https://sites.google.com/site/murmurhash/算法，自称超级快的算法，是的
```

```
MurmurHashhashFNV4倍。官方数据如下：—5

    OneAtATime - 354.163715 mb/sec
    FNV - 443.668038 mb/sec
    SuperFastHash - 985.335173 mb/sec
    lookup3 - 988.080652 mb/sec
    MurmurHash 1.0 - 1363.293480 mb/sec
    MurmurHash 2.0 - 2056.885653 mb/sec但也有文章声称，只有当的长度大于
        字节的时候，的运算速度才快于。“从计算速度上来看，只适用于已知长度的、
        长度比较长的字符”。

key10MurmurHashDJBMurmurHash研究一下算法

CityHash
```

4.3 D2P文件结构

先列出D2P的文件头

```
#define D2P_CAPACITY_BIT_WIDTH  ((uint32_t)20)
#define D2P_CAPACITY            ((uint32_t)(1<<(
    D2P_CAPACITY_BIT_WIDTH-1)))
typedef struct{
    uint8_t         signature1;
    uint8_t         signature2;
    uint8_t         signature3;
    uint8_t         version;

    uint32_t        seed1;
    uint32_t        seed2;

    uint32_t        capacity;
}d2phdr_t;
```

http://blog.csdn.net/fg2006/article/details/6838147 hash函数评定 Hash函数主要用于完整性校验和提高数字签名的有效性,目前已有很多方案。这些算法都是伪随机函数,任何杂凑值都是等可能的。输

出并不以可辨别的方式依赖于输入;在任何输入串中单个比特的变化,将会导致输出比特串中大约一半的比特发生变化。

首先，定义两个名词。 block ：打包数据的单位，可以是一个文件也可以是一个数据块。总之是资源包中的一块数据。 包，资源包：即包含多份数据的一个文件。

1 block size + block data 按数据块大小+数据块内容的方式将一批数据块逐个打包。这是比较简单实用的方法。往往还需要生成一个文本文件指明每个数据块是什么内容。如果按照默认顺序打包和读取就不需要了。这种方法主要用于包中的数据需要全部同时读出的情况。因为这种格式的包，从中间单独抽取某个数据块读出比较麻烦。只能跳过若干数据块。

2 offset table + block datas 这种方式下，首先在包头部写入一个offset table,即每个数据块在包中的起始位置。然后将所有数据块的内容逐个写入包中。这个offset table条目数一般比block数大1，这样 table[0]=0 table[1]=block 0 size table[2]=block 1 size + table[1] table[3]=block 2 size + table[2] ..... table[n]=block n-1 size + table[n-1] (block从0开始计数,共n个[0,n-1]) 读取的时候，根据block id,先在table中查到table[id]和table[id+1], 两者之差就是block size.table[id]是block地址。 当然也需要一个配套的文本文件记录每个block id对应的内容。 这种方式的好处是可以只读出需要的内容。且table所占的容量只比第一种方式稍大。是非常实用的方式。

3 trunk 方式 首先定义trunk：block id + block size + block data 只比第一种方式多了一个block id。但灵活性提高了。因为可以根据id判断当前的block是否是需要载入的。如果是就载入否则跳过这个block接着看下一个block,直到找到或没找到需要载入的block。block id不但可以指明数据块在同类数据中的id,而且可以指明是哪一种资源，这样就可以把不同种类的资源打包到一起。当然在前面的方式中可以在数据块中包含资源类型，但灵活性就差了，特别是必须在所有类型的数据块中统一写一些数据在相同位置来表示资源类型。而在trunk方式中，只要在block id中取一些位来定义资源类型，然后根据资源类型调用不同类型的载入函数去读取数据是相当灵活的。trunk方式的缺点是读取速度稍慢，因为必须一个个找过去。

4 file name 索引方式 不知道除了我有没有人用过这种方式，这是一种偷懒的方式。打包对象是文件。先在包头部写入一个文件名和offset 映

射表。文件名按字符串写入包中，所以这之前要写入文件名的字节数。 映射表的结构为： 文件名字节数+文件名字符串+文件offset 当然文件名必须用ascii编码。这儿还有个变通的方式。你可以用utf-8格式的编码，就不需要写文件名自己数了。 映射表之后就按顺序写入文件。 这种方式其实是offset table 方式的一种变体。不同的是直接将文件名写入包中。读取时可以指定文件名从包中读出，根从持久性设备上读单个文件的感觉一样。不必再定义额外的文件文件表明每个数据块的含义了。所以是一种比较懒的方式。缺点是浪费容量。

小结： 以上4种方式各有用处，用的最多的是offset table和trunk方式。此文需要持续补充更正完善 我既不是Linux版主, 也不是Linux协会负责人, 我只是一个普普通通的Linux用户. 我做版衫的目的只是想让今年和往年一样, 有一件Linux版衫. 4月5日stephenjy发帖求版衫 , 随即李喵喵投稿, 一片赞同之声之后一个月内无人理睬. 5月15日talentmonkey发帖求版衫, 同样的时间我在Linux协会邮件列表里也求, 一片赞同之声之后继续无人理睬. 之后紧接着boj的每周小聚那天晚上, 我直接跟版主说, 于是版主随手发了个帖子正式征集版衫. 在那之后又过了半个月, 仍然只有李喵喵同学一份投稿. 于是我和李喵喵两个人决定着手开始做版衫, 商量到钱的问题的时候向版主询问建议, 版主的回复是这种细节问题你们自己解.

Chapter 5

游戏地图设计

如果有人问我游戏程序设计中最难的环节是什么，那么我的回答一定是“游戏地图”。是的，游戏地图设计对于一个新手来说简直是一场噩梦！

常见的2.5D地图都采取了地砖贴图的方式，对于地面物体的显示、排序问题，从来都不是一个简单的问题。本章我尝试根据D2IM的地图设置，提出一个解决方案。

5.1 地图要素

Chapter 6

游戏角色设计

人物是游戏中最总要的元素，“我想”。

6.1 人物数据结构组织

6.1.1

6.1.2

6.2 Makefile使用

Chapter 7

RPG游戏中的人工智能

人工智能非为游戏而出现,这是一个庞大的CS领域分支.我们也没有必要去细细了解它,这里我们关心的仅仅就是它对游戏制作有关的这一部分.

在游戏中引入人工智能的作用无非是增加玩家的挑战性,创造更真实的虚拟世界和增加游戏的可玩性. 对玩家来说,对付复杂AI的敌人需要更高的技巧,挑战性有一定程度的增加.想象一款没有AI的游戏,NPC被卡在一块石头或树木前,傻傻的撞着树走,而不会饶个圈子走过去,这将使玩家感到索然无味.在现在游戏则很少见到这种情况,多种"寻路法"的计算已是游戏程序员们最基本的一种要求.

使NPC更加具有AI,更模拟人类的思维,行为,可以使玩家拥有更多的不可预料性,给玩家一种惊奇感,新鲜感.而实际上我们的做法是给NPC的一定的AI套路选择,例如:根据自身HP的多少,来进行"勇猛直前,边战边退,逃命要紧"等各类的AI算法的选择,将使玩家无法准确的推测NPC的行动,可玩性得到了发挥.

7.1 AI流程图

7.2 常见人工智能算法

7.3 D2IM中用到的AI算法举例

Chapter 8

游戏脚本系统

为什么游戏需要脚本系统？因为脚本是游戏剧情扩展的有效手段。脚本系统，地图编辑器，资源管理器，是一款良好的游戏作品中的必要环节。

D2S的语法和在游戏中的运行模型耗费了我很多心思。我开始想要使用单线程模型，每个游戏轮转执行一个命令。这会导致某个命令执行时间特别长导致游戏卡住的问题。我想支持start/end实现单步/并行混合脚本，这使得在并行时候调用子脚本出现困难。我想使用多线程模型，又存在一个线程间脚本命令冲突的问题。比如让一个Player走到NPC身边，然后和NPC对话。如果使用多线程模型，一个脚本控制Player走到NPC身边，等待这个任务结束后再和NPC对话，然而如果此时另外一个脚本使得Player自杀。这时候第一个脚本的等待状态就会一直持续下去，不得结束。这明显是错误的。

我想这是个很费脑筋的环节，我是不是该减弱脚本的功能？ 对，我决定减弱脚本系统的功能，任何脚本命令都是单步的，非单步命令将被集成为特殊命令。彻底的摒弃单步和非单步混合编程模式。

下面是一个关于这个问题的经典论述：

如今有一些不同类型的脚本系统可供程序员或者美术师使用，而且它用非常有条理和逻辑的思想恰当地做这些。第一种是简单的基于文本的，单线索的风格，就像我们程序员习惯的编码。在许多情况，它实际上基於 C，尽管以一种非常简单的形式。 大量这种类似“if this，then do that”的东西。大部分脚本倾向在范围内是相当线性的—意味着它通常由许多在次

序上彼此相接的命令组成。在世界中移动角色A指向B。当完成以后，让他讲话，完成以后，移动他指向C。相当简单的事情。

然后有复杂的东西--允许多重线索，和实际上允许可变情形。可变情形是当脚本开始时你实际上不能确知谁会出现在附近，但是你必须按这样的方式编写脚本以便任何人出现在附近它都将会工作。举例来说--一个正常的简单脚本会有三个家伙，全部被预先定义，全部有一组他们将会讨论的情形。一个可变的脚本将会有三个人,你不能保证是某一个特定的人，并必须按相同的方式工作。或者在一个极端的情形中，也许只有二个，或者甚至一个家伙将会在那里，使得三方交谈有一点困难。

Raven在Star Trek Voyager: Elite Force中面临的一个很大的问题是这样的情形，使用者可能会想要把一个角色从一条船的某个地方带到另外一个地方，但是从A点到B点的路径可能会随着每次游戏根本地改变。举例来说，他们需要让Munro(你所扮演的游戏主要角色)从发动机舱室到输送舱。 不幸的是由于游戏的非直线性，在事件到达这一点以前你可能已经破坏了涡轮升降机，或者也许 Jeffries 管被损害不能通过。假定当脚本开始的时候我们不知道世界的状态，我们不得不为几乎各种可能发生的事情编写脚本以便适用于这些‘如果。。。怎么办’的情形。而且它仅仅从那里变得更加糟糕。我们能建立的一些情形提供了如此多可能的组合情形，以致于为了一个满意的结论而准确测试每一个可能发生的事情几乎是不可能的。请和在SiN, Star Trek Voyager : Elite Force or Deus Ex中工作的任何人谈谈。QA部门传统地憎恨这些类型游戏，因为这已经使他们的工作比以前更加困难了 50 倍。

你能够想象为这些情形编写脚本是何等的困难。但那是今天的非线性游戏路径要求的事情，而且它为何博得了较多的开发支持从而能够努力实现它。

8.1 D2S系统的思考

任何一个指令都能被分割成多周期执行。

```
/*
**  args     : 输入参数数组
**  ret      : < 0 指令执行出错
**               = 0 指令执行完成
**               > 0 指令执行尚未完成
**
** 指令出错码    -1  : 找不到命令
**               -2  : 指令参数格式不对
*/
int do_cmd(char *args)
```

例如，在如下的脚本系统中，假设cmd1为即时命令，cmd2为过程命令。

```
while eva $(cmd1 -op) > $(calc $(cmd2 -op) - 2)
    cmd3 -op
    cmd4 -op
end
```

d2s引擎首先获得第一行文本，存入内部cmd_buf，扫描发现$(符号，于是取出$()中的“cmd1 -op”，并调用

```
    ret = do_cmd("cmd1 -op");
```

返回ret为0，表示已经执行完毕，于是继续扫描

规定$(cmd)可以嵌套和递归。 嵌套如：

```
    cmd -op $(cmd1 $(cmd2))
```

递归的情况如下，假设有一个脚本为cmd，内容如下：

```
# 脚本cmd
# 脚本在参数为小于时候返回命令本身10
```

```
# 并将返回值置为1
if eva $1 < 10
    echo '$(cmd' $(calc $1 + 1))
    exit 1
end
```

下面这句while命令调用它：

```
    while $(cmd 1)
    end
```

D2S解释器首先扫描第一行，发现$()，于是调用函数：

```
    ret = do_cmd("cmd 1");
```

首先返回ret为0，表示执行完毕，输出为"$(cmd 2)"，于是继续进行。

还有一种就是一般意义的递归：

```
# 脚本cmd
# 脚本在参数为小于时候返回命令本身10
# 并将返回值置为1
if eva $1 < 10
    echo $(cmd $(calc $1 + 1))
    exit 1
end
```

当执行到echo这一行的时候，首先发现$(cmd，于是继续扫描参数，发现参数也是$(继续扫描。 恩，一般执行顺序为：

先得到一行，扫描发现第一个$(后，寻找匹配的)，如果在寻找过程中发现第二个$(就先找第二个的，再找第一个的。为深度优先搜索的解释顺序。实现的时候用树来解决。

默认一个任务有一个动作。

8.2 脚本系统基本要求

任何一个脚本系统都需要有注释语句，变量操作以及流程控制语句。D2S也不例外。

8.3 D2S脚本解析系统

D2S脚本系统是大小写敏感的。

8.3.1 注释语句

D2S使用井号（#）表示一行的注释，注意D2S只有这一种注释方式。如果在语句中需要使用#，应该使用
#。D2S的注释规则和Linux下Bash规则接近。

```
NAME@D2IM$ echo "The # here does not begin a comment."
NAME@D2IM$ echo 'The # here does not begin a comment.'
NAME@D2IM$ echo The \# here does not begin a comment.
```

8.3.2 变量及其赋值

D2S的变量命名的格式用正则表达式如下：

```
[_a-zA-Z]\+[_a-zA-Z0-9]*
```

变量的赋值需要使用到内嵌的let命令：

```
let var1 = 12                # 整数值
let var2 = 1.2               # 浮点数值
let var3 = hello world       # 字符串
let cmd  = "cmd1 or cmd2"    # 命令
let cmd  = cmd1  or cmd      # 两条命令
let var = ${eval ($var1+$var2)/23.1 - 12.4}
```

变量值的引用使用$(var)进行，注意()是可以省略的，省略括号的原则是不应引起混淆。建议始终使用$()形式。 变量的清除使用unlet关键字：

```
unlet var
```

8.4 数组

D2S中并不需要数组，因为我们总可以用如下方式引用值：

```
for n in ${seq 1 100}
    let arr$n = $n
end

let ptr = arr
let cnt = 1
while [ -ne $cnt 100 ]
    echo $ptr$cnt
    let cnt++
end
```

8.5 命令分割和组合

如果多个命令需要放在一行，可以使用&&操作符予以分割。举例：

```
cmd1 args && cmd2 args && cmd3 args
```

D2S解析时候会把&&前后视为两条不同的命令，如果不使用分隔符，第一个单词后面所有的单词都将视为第一个单词的参数。 &&操作符最主要的作用是和()操作符配合。小括号将多个命令合并为一个命令，括号内的所有命令将一起执行。

```
(cmd1 args && cmd2 args && cmd3 args)
```

8.6 命令组合

使用()将多个命令组合为一个命令，命令之间使用and，or，not，xor分割。单步执行时括号内内容将作为一个整体一次执行。

8.7 单步和并行

游戏脚本区别于普通脚本最重要的地方就是脚本解析需要在单步方式和并行方式之间不断切换。举个例子：如果有NPC1和NPC2，当他们发现有玩家进入视野范围，需要主动走到玩家跟前并开始对话。我们使用游戏内部支持的goto语句：

```
goto $(npc1) ${pos $player}
goto $(npc2) ${pos $player}
```

这样就会导致NPC1先走到player附近的位置，然后NPC2再走到player附近的位置。显然我们需要的是两个NPC“走”的动作同步进行。 再举个例子：在一个场景开始时候一般会有淡入，我们使用如下方式：

```
ldmp  entry.map
playm background.mp3
fadin 1000
```

这时候我们便需要单步运行，同步运行只会导致游戏脚本出现bug。为了解决这一问题，D2M使用了关键字start，start启动一个复合命令并返回对应的描述符。例如：

```

NAME@D2IM:~$ start sleep 1000
12
```

```
NAME@D2IM:~$ start (sleep 1000 && echo first)
15
first
NAME@D2IM:~$ start (sleep 1000 && echo first) && echo second
27
second
first
NAME@D2IM:~$
```

如果我们需要有规律并行命令，比如命令所有的monster开始进攻，我们可以这样：

```

unlet pids
for monster in ${monsterid}
    let pids = $(pids) ${start attack $(monster)}
end
wait $(pids)
```

如果我们需要大块无规律的并行命令，我们可以：

```
unlet pids
let pids = $(pids) ${start cmd1}
let pids = $(pids) ${start cmd2}
let pids = $(pids) ${start cmd3}
let pids = $(pids) ${start cmd4}
let pids = $(pids) ${start cmd5}
wait $(pids)
```

但这样较为繁琐，我们提供另一种更简洁的形式：

```
start
    cmd1
    cmd2
    cmd3
wait
```

注意start和wait命令必须成对出现，并且不支持嵌套。一个start出现后，必须在对应的wait出现后才能重新出现start。 如果在start和wait之间出现脚本调用，例如上面的例子中cmd2不是一个内置命令而是一个脚本，那么在cmd2脚本中允许出现start和wait。运行时如果出现脚本调用，调用时自动将环境设置为单步模式，直到脚本调用返回，再将环境设置为原来的模式。 始终

8.8 脚本调用

D2S不支持函数，对应的D2S使用脚本调用来完成类似功能。在D2S中脚本调用和内置命令同级，脚本解析时不予区分。如果脚本和内置命令重名，D2S将优先使用脚本。因此你可以如此重定义内置命令。当然这会带来效率的些许降低。 脚本搜索路径由全局变量$PATH决定。$PATH的初始值在/bin/init中设置如下：

```
let PATH = /bin:/sbin
export PATH
```

你也可以设置自己的PATH路径：

```
let PATH = your\_dir:$PATH
export PATH
```

8.8.1 IF语句

IF语句共有如下三种基本形式：

```
if condition
    commands
end

if condition
    commands
else
    commands
```

```
end

if condition
    commands
elif condition
    commands
elif condition
    commands
end
```

condition为一个判断状态，在脚本中可以用$?获得其取值。在脚本系统中我们常用exit命令设定$?取值。IF语句将获取$?寄存器的值并以此作为判断依据。

8.8.2 WHILE语句

```
while condition
    commands
end
```

8.9 逻辑运算

逻辑运算有四个，分别是and，or，not，xor。

```
if cond1 and cond2
    commands
end

if cond1 or cond2
    commands
end

command or command2

commands1 and command2
```

8.10 多选择分支

switch支持正则匹配，格式如下：

```
switch expr
    case pattern1:
        commands
    case pattern2:
        commands
    default:
        commands
end
```

注意switch和case之后的expr和pattern都从第一个非空白字符开始，如果expr或者pattern第一个字符为空格，应该使用
s转义，或者用""将其转化为字符串，转化规则和前面相同。

D2S的解析逻辑如下： 获取一行 扫描变量描述符，如果有，对对应变量进行求值（求值是递归的），然后替换代入。由于是宏的作用，故任何内容皆可以代入，包括关键字。

D2M 中 语 句 的 最 小 构 成 单 位 是 词，词是以空白字符分割而成的，空白字符包括制表符和空格。例如： goto (*player*)(x) (*y*)就是四个词。分别是*goto*、(player)、(*x*)和(y)，这个语句将使得player步行到(*x*，y)附近位置。

D2M暂时不支持输入输出重定向操作和管道，这是为了实现更加简单。如果我们需要将输出定向到文件或者用上一个命令的输出作为下一个命令的输入，我们可以使用如下形式：

```
fprint /tmp/test ${cmd}
```

我承认这不简洁，但是确实实现起来简单多了！

简单命令的第一个词是要执行的命令，其余的词都是这个命令的参数，例如： echo "hello world" echo 第一个echo 是命令，第二个词"hello world"是参数1，第三个词echo是参数2，而不再作为一个命令了。

D2S不支持管道，如果我们需要某个命令的输出，我们同样可以使用变量作中间媒介。

用类似伪代码的形式表示如下：

```
while(1) {
print_prompt();
get_input();
parse_input();
if “(” logout || “” exit)
break;
do_cmd();
}读取用户输入如何获取用户输入？一种方法是通过
```

getchar() 从标准输入每次读一个字符，如果读到的字符是 ‘ \’ ，说明用户键入了回车键，那么就把此前读到的字符串作为用户输入的命令。n代码如下：

```
int len = 0;
int ch;
char buf[300];
ch = getchar();
while(len < BUFSIZ && ch != '\n') {
buf[len++] = ch;
ch = getchar();
}
if(len == BUFSIZ) {
printf("command is too long\n");
break;
}
buf[len] = '\n';
len++;
buf[len] = 0;但是，我们注意到，在
```

bash 中，可以用 “” 和 “” 键在命令行中左右移动，可以用上下键调用以前使用的命令，可以用退格键来删除一个字符，还可以用<—> tab 键来进行命令行补全。我们的如果也要支持这些功能，那么就必须对这些键进行处理。这样仅仅对用户输入的读取就非常麻烦了。shell实际上，任何需要一个获取用户输入的程序，都会涉及到同样的问题，如何象

```
bash 那样处理键盘？GNU readline 库就是专门解决这个问题的，它把对键盘的操作完
```

全封装起来，对外只提供一个简单的调用接口。有了它，对键盘的处理就不再让人头疼了。关于

```
readline 库的详细信息，可以通过 man readline 来看它的帮助页面。在我们
```

的 shell 程序中，我是这样来使用的。 readline

```
char* line;
char prompt[200];
while(1) {
set_prompt(prompt);
if(!(line = readline(prompt)))
break;。。。。。。
```

}首先通过
set_prompt() 来设置要输出的提示符，然后以提示符作为参数调用 readline()，这个函数等待用户输入，并动态创建一块内存来保存用户输入的数据，可以通过返回的指针 line 得到这块内存。在每次处理完用户输入的命令之后，我们必须自己负责来释放这块内存。有了
readline 之后，我们就可以象 bash 那样使用键盘了。在通过
readline 获取用户输入之后，下一步就是对用户输入的命令进行分析。命令行分析对命令行的分析，实际上是一个词法分析过程。学过编译原理的朋友，都听说过

lex 和yacc 的大名，它们分别是词法分析和语法分析工具。Lex 和 yacc 都有的版本（GNUopen source 的思想实在是太伟大了，什么好东东都有免费的用），分别是 flex 和。 bison所谓“工欲善其事，必先利其器”，既然有这么好的工具，那我们就不必辛辛苦苦自己进行词法分析了。对，我们要用
lex 来完成枯燥的命令行词法分析工作。“去买本《与》（中国电力出版社）来看吧。第一次学当然稍微有点难度，不过一旦掌握了，以后再碰到类似问题，就可以多一个利器，可以节省劳动力了。
lexyacc在我们的这个
shell 程序中，用 flex 来完成词法分析工作。相对语法分析来说，词法分析要简单的多。由于我们只是做一个简单的，因此并没有用到语法分析，而实际上
在 shell bash 的实现代码中，就用到了语法分析和。 yacc关于
lex 的细节，在这里我就不能多说了。程序，通常分为三个部分，其中进行语法分析工作的就是它的第二部分：Lex “规则”。规则定义了在词法分析过程中，遇到什么样的情况，应该如何处理。词法分析的思路，就是根据前面定义的“语法规范”来把用户输入的命令行拆解成
shell首先，我们要把用户输入的命令，以空白字符（键或者空格）分隔成一个个的参数，并把这些参数保存到一个参数数组中。但是，这其中有几种特殊情况。
tab一、如果遇到的字符是“”、“”、“”或“”，由于这些符号是管道或者列表中所用到的分隔符，因此必须把它们当作一个单独的参数。
;><|二、以双引号（”）括起来的字符串要作为一个单独的参数，即使其中出现了空白字符、“”、“”、“”、“”。其实，在
;><|标准中，对引号的处理相当复杂，不仅包括双引号（”），还有单引号（’）、反引号（POSIX），在什么情况下，应该用什么样的引号以及对引号中的字符串应该如何解释，都有一大堆的条款。我们这里只是处理一种极简单的情况。 其次，如果我们遇到换行符（’
\’），那么就结束本次命令行分析。根据前面定义的n shell 语法规范，最上层的是列表命令，因此下一步是把所有的参数作为一个列表命令来处理。根据这个思路，我们来看对应的
lex 规则。

```
%%
"\"" {BEGIN QUOTE;}
[^\n"]+ {add_arg(yytext);}
"\"" {BEGIN 0;}
\n {BEGIN 0; do_list_cmd(); reset_args();}
";" {add_simple_arg(yytext);}
">" {add_simple_arg(yytext);}
"<" {add_simple_arg(yytext);}
"|" {add_simple_arg(yytext);}
[^ \t\n|<>;"]+ {add_arg(yytext);}
\n {do_list_cmd(); reset_args();}
. ;
%%
```

我们对这些规则逐条解释：一这条规则，目的是为了在命令行中支持引号，它们用到了

```
144 lex 规则的状态特性。、
1"\"" {BEGIN QUOTE;}、
2[^\n"]+ {add_arg(yytext);}、
3"\"" {BEGIN 0;}、
4\n {BEGIN 0; do_list_cmd(); reset_args();}、
```

1 如果扫描到引号（“），那么进入 QUOTE 状态。在这个状态下，即使扫描到空白字符或“”、“”、“”、“”，也要当作普通的字符。;><|、

2 如果处于状态，扫描到除引号和回车以外的字符，那么调用 QUOTE add_arg()函数，把整个字符串加入到参数数组中。、

3 如果处于状态，扫描到引号，那么表示匹配了前面的引号，于是恢复到默认状态。QUOTE、

4 如果处于状态，扫描到回车，那么结束了本次扫描，恢复到默认状态，并执行QUOTE do_list_cmd()，来执行对列表命令的处理。以下几条规则，是在处于默认状态的情况下的处理。、

```
5";" {add_simple_arg(yytext);}、
6">" {add_simple_arg(yytext);}、
7"<" {add_simple_arg(yytext);}、
8"|" {add_simple_arg(yytext);}、
9[^ \t\n|<>;"]+ {add_arg(yytext);}、
10\n {do_list_cmd(); reset_args();}、
```

5 如果遇到分号（），因为这是一个列表命令结束的操作符，所以作为一个单独的参数，执行; add_simple_arg()，将它加入参数数组。、

6 如果遇到，因为这是一个简单命令结束的操作符，所以作为一个单独的参数，执行 > add_simple_arg()，将它加入参数数组。、

7 如果遇到，因为这是一个简单命令结束的操作符，所以作为一个单独的参数，执行 < add_simple_arg()，将它加入参数数组。、

8 如果遇到管道符号（），因为这是一个管道命令结束的操作符，所以作为一个单独的参数，执行| add_simple_arg()，将它加入参数数组。、

9 对于不是制表符（）、换行符（’ tab\’）、n| 、、和分号（）以外的字符序列，作为一个普通的参数，加入参数数组。<>;、

10 如果遇到换行符，那么结束本次扫描，执行 do_list_cmd()，来执行对列表命令的处理。、

11 对于任意其它字符，忽略通过

lex 的“规则”把用户输入的命令行分解成一个个的参数之后，都要执行 do_list_cmd() 来执行对列表命令的处理。命令处理首先是对处于“语法规范”中最上层的列表命令的处理。

shell

1 列表命令的处理过程：依次检查参数数组中的每一个参数，如果是分号（），那么就认为分号前面的所有参数组成了一个管道命令，调用

; do_pipe_cmd() 来执行对管道命令的处理。如果扫描到最后，不再有分号出现，那么把剩下的所有参数作为一个管道命令处理。代码很简单：

```
static void do_list_cmd()
{
int i = 0;
int j = 0;
char* p;
while(argbuf[i]) {
```

```
if(strcmp(argbuf[i], ";") == 0) {// ;
p = argbuf[i];
argbuf[i] = 0;
do_pipe_cmd(i-j, argbuf+j);
argbuf[i] = p;
j = ++i;
} else
i++;
}
do_pipe_cmd(i-j, argbuf+j);
```

}接下来是对管道命令的处理。管道命令的处理管道是进程间通信（）的一种形式，关于管道的详细解释在《高级环境编程》第章：进程间通信以及《网络编程：第卷：进程间通信》第章：管道和中可以看到。

IPCunix14unix24FIFO我们还是来看一个管道的例子：

```
[root@stevens root]# echo “hello ”world|wc -c |wc -l
```

在这个例子中，有三个简单命令和两个管道。第一个命令是

```
echo “hello ”
```

，它在屏幕上输出world hello 。由于它后面是一个管道，因此，它并不在屏幕上输出结果，而是把它的输出重定向到管道的写入端。world第二个命令是

```
wc -
```

，它本来需要指定输入源，由于它前面是一个管道，因此它就从这个管道的读出端读数据。也就是说读到的是c hello ，worldwc -c 是统计读到的字符数，结果应该是。由于它后面又出现一个管道，因此这个结果不能输出到屏幕上，而是重定向到第二个管道的写入端。12第三个命令是

```
wc -
```

。它同样从第二个管道的读出端读数据，读到的是，然后它统计读到了几行数据，结果是行，于是在屏幕上输出的最终结果是。11211在这个例子中，第一个命令只有一个“后”管道，第三个命令只有一个“前”管道，而第二个命令既有“前”管道，又有“后”管道。在我们处理管道命令的

do_pipe_cmd()函数中，它的处理过程是：首先定义两个管道

```
prefd
```

和，它们分别用来保存“前”管道和“后”管道。此外，还有一个变量 postfd prepipe 来指示“前”管道是否有效。然后依次检查参数数组中每一个参数，如果是管道符号（），那么就认为管道符号前面所有的参数组成了一个简单命令，并创建一个“后”管道。如果没有“前”管道（管道中第一个简单命令是没有“前”管道的），那么只传递“后”管道来调用

```
|do_simple_cmd()
```

，否则，同时传递“前”管道和“后”管道来调用 do_simple_cmd()。执行完以后，用“前”管道来保存当前的“后”管道，并设置“前”管道有效标识，继续往后扫描。如果扫描到最后，不再有管道符号出现，那么只传递“前”管道来调用

prepipedo_simple_cmd()。代码如下：

```
int i = 0, j = 0, prepipe = 0;
int prefd[2], postfd[2];
char* p;
while(argv[i]) {
if(strcmp(argv[i], "|") == 0) { // pipe
p = argv[i];
argv[i] = 0;
```

```
pipe(postfd); //create the post pipe
if(prepipe)
do_simple_cmd(i-j, argv+j, prefd, postfd);
else
do_simple_cmd(i-j, argv+j, 0, postfd);
argv[i] = p;
prepipe = 1;
prefd[0] = postfd[0];
prefd[1] = postfd[1];
j = ++i;
} else
i++;
}
if(prepipe)
do_simple_cmd(i-j, argv+j, prefd, 0);
else
do_simple_cmd(i-j, argv+j, 0, 0);
```

最后，我们分析简单命令的处理过程。简单命令处理过程我们已经看到，对列表命令和管道命令的处理，实际只是一个分解过程，最终命令的执行还是要由简单命令来完成。在简单命令的处理过程中，必须考虑以下情况：、区分内部命令和外部命令

1根据简单命令的定义，它的第一个参数是要执行的命令，后面的参数作为该命令的参数。要执行的命令有两种情况：一种是外部命令，也就是对应着磁盘上的某个程序，例如、等等。对这种外部命令，我们首先要到指定的路径下找到它，然后再执行它。

wcls二是内部命令，内部命令并不对应磁盘上的程序，例如、等等，它需要自己来决定该如何执行。例如对

cdechoshell cd 命令，就应该根据它后面的参数改变当前路径。shell对于外部命令，需要创建一个子进程来执行它，而对于内部命令，则没有这个必要。外部命令的执行，是通过

exec 函数来完成的。有六种不同形式的 exec 函数，它们可以统称为 exec 函数。我们使用的是 execv()。关于 的细节，请看《环境高级编程》第章：进程控制。execunix8对于内部命令，我们目前支持五种，分别是：：退出解释器

exitshell：改变目录

cd：回显

echo：导入或显示环境变量

export：显示命令历史信息

history这几个内部命令分别由

do_exit()、do_cd()、do_echo()、do_export()、do_history()来实现。、处理重定向

2在简单命令的定义中，包括了对重定向的支持。重定向有多种情况，最简单的是输入重定向和输出重定向，分别对应着“”和“”。

<>输入重定向，就是把“”后面指定的文件作为标准输入，例如：

<

wc < xxx表示把
xxx 这个文件的内容作为 wc 命令的输入。输出重定向，就是把“”后面指定的文件作为标准输出，例如：
>
echo “hello ”world > xxx表示把
echo “hello ”world 的结果输入到 xxx 文件中，而不是屏幕上。为了支持重定向，我们首先对简单命令的参数进行扫描，如果遇到“”或者“”那么就认为遇到了重定向，并把“”或者“”符号后面的参数作为重定向的文件名称。
<><>对于输入重定向，首先是以只读方式打开“”后面的文件，并获得文件描述符，然后将该文件描述符复制给标准输入。
<对于输出重定向，首先是以写方式打开“”后面的文件，并获得文件描述符，然后将该文件描述符复制给标准输出。
>具体实现在
predo_for_redirect() 函数中：、管道的实现
3管道的实现实际上也是一种重定向的处理。对于“前”管道，类似于输入重定向，不同的是，它是把一个指定的描述符（“前”管道的输出端）复制给标准输入。对于“后”管道，类似于输出重定向，不同的是，它把一个指定的描述符（“后”管道的输入端）复制给标准输出。在对管道的处理上，还必须要注意管道和输入或输出重定向同时出现的情况，如果是一个“前”管道和一个输入重定向同时出现，那么优先处理输入重定向，不再从“前”管道中读取数据了。同样，如果一个“后”管道和一个输出重定向同时出现，那么优先处理输出重定向，不再把数据输出到“后”管道中。至此，我们已经描述了实现一个简单的

shell 解释器的全部过程，相应的代码和 makefile 在我们的网站上可以下载。希望大家能够结合代码和这篇文章，亲自动手做一次，以加深对shell 解释器的理解本文来自博客，转载请标明出处：
CSDNfile:///C:/Documents%20and%20Settings/asdf/Desktop/Web/的编写shell特殊符号()东之博客%20-%20%20-%20博客CSDN.htm摘要：本期的目的是向大家介绍的概念和基本原理，并且在此基础上动手做一个简单解释器。同时，还将就用到的一些

shellshell 环境编程的知识做一定讲解。linux本文适合的读者对象对环境上的语言开发有一定经验；
linuxc 对环境编程（比如进程、管道）有一点了解。linux概述本章的目的是带大家了解的基本原理，并且自己动手做一个解释器。为此，
shellshell首先，我们解释什么是解释器。
shell其次，我们要大致了解解释器具有哪些功能；
shell最后，我们具体讲解如何实现一个简单的
shell 解释器，并对需要用到一些环境编程的知识做一定讲解，并提醒你如果想深入掌握，应该去看哪些资料。 linux解释器是什么？

Shell 解释器是一个程序。对，是一个程序，而且，它就在我们的身边。在系统中，当我们输入用户名和密码登陆之后，我们就开始执行一个解释器程序，通常是Shelllinuxshell /bin/，当然也可以是别的，比如bash/bin/。（详细概念请看第一期中的有关部分）shshell提示：在
/etc/passwd 文件中，每个用户对应的最后一项，就指定了该用户登陆之后，要执行的解释器程序。shell在
linux 字符界面下，输入
man bash调出
bash 的帮助页面帮助的最开始就对下了一个定义：
bash
bash 是一个兼容于 sh 的命令语言解释器，它从标准输入或者文件中读取命令并执行。它的意图是实现 IEEE 标准中对POSIX 和工具所规范的内容。shell解释器的作用

Shell 在登陆 linux 系统之后，屏幕上就会出现一行提示符，在我的机器上，是这样的： [root@stevens root]# 这行提示符就是由解释器打印出来的，这说明，现在已经处于bash bash 的控制之下了，也同时提示用户，可以输入命令。用户输入命令，并回车确认后，分析用户的命令，如果用户的命令格式正确，那么就按照用户的意思去做一些事情。bashbash比如，用户输入：

[root@stevens root]# echo “hello, ”world那么，就负责在屏幕上打印一行“bashhello ”。world 如果，用户输入： [root@stevens root]# cd /tmp 那么，就把用户的当前目录改变为bash /。tmp所以，解释器的作用就是对用户输入的命令进行“解释”，有了它，用户才可以在

shell linux 系统中任意挥洒。没有它的帮助，你纵然十八般本领在身，也施展不出。每次在“解释”完用户命令之后，又打印出一行提示符，然后继续等待用户的下一个命令。这种循环式的设计，使得用户可以始终处于

bash bash 的控制之下。除非你输入、明确表示要退出 exitlogout 。bash语法梗概

Shell 我们不停的命令 bash 做这做那，一般情况下它都很听话，按你的吩咐去做。可有时候，它会对你说：“嗨，老兄，你的命令我理解不了，无法执行”。例如，你输入这样的命令： [root@stevesn root]# aaaaaa 会告诉你：bash bash: aaaaaa: command not found 是的，你必须说的让它能听懂，否则它就给你这么一句抱怨，当然也还会有其它的牢骚。那么，什么样格式的命令，它才能正确理解执行了？这就要引出

shell 的语言规范了。作为一个命令语言解释器，有一套自己的语言规范，凡是符合这个规范的命令，它就可以正确执行，否则就会报错。这个语言规范是在

Shell IEEE 的第二部分：“和规范”中定义的。关于这份规范，可以在这里看到。POSIXshelltools官方的东西，总是冗长而且晦涩，因为它要做到面面俱到且不能有破绽。如果读者有兴趣，可以仔细研究这份规范。而我们的目的只是理解的实现思想，然后去实现一个简单的

shell shell 解释器，所以没必要陷入枯燥的概念之中。现在请继续在

linux 字符界面下输入 man ，调出bash bash 的帮助页面，然后找到“语法”那一部分，我们就是以这里的描述作为实现的依据。 shell在帮助的“

bashshell 语法”一节，是这样来定义shell 语法的：

1 简单命令简单命令是可选的

()一系列变量赋值, 紧接着是空白字符分隔的词和重定向符号, 最后以一个控制操作符结束. 第一个词指明了要执行的命令, 它被作为第 0 个参数. 其余词被作为这个命令的参数.这个定义可以这样来理解：、

1 可以有变量赋值，例如

a=10 b=20 export a b、

2 “词”是以空白字符分隔开的，空白字符包括制表符（）和空格，例如：tab

ls /tmp就是两个词，一个，一个

ls /tmp、可以出现重定向符号，重定向符号是“”和“”，例如：

3><

echo “hello ”world > /tmp/log、

4 简单命令结束于控制操作符，控制操作符包括：

|| & && ; ;; ( ) |例如，用户输入：

ls /tmp用户最后敲的回车键就是控制操作符，表示要结束这个简单命令。

newline如果用户输入：

echo “”100 ; echo “”200那么这是两个简单命令，第一个结束于“”，第二个结束于

;。newline、
5 简单命令的第一个词是要执行的命令，其余的词都是这个命令的参数，例如：
echo “hello ”world echo第一个
echo 是命令，第二个词“hello ”是参数，第三个词world1 echo 是参数，而不再作为一个命令了。2简单命令是
shell 语法中最小的命令，通过简单命令的组合，又可以得到管道命令和列表命令。
1 管道（命令）管道是一个或多个简单命令的序列，两个简单命令之间通过管道符号（“”）来分隔
|例如

echo “hello ”world | wc -l就是一个管道，它由两个简单命令组成，两个简单命令之间用管道符号分隔开。我们可以看到，管道符号“”也是属于上面提到的控制操作符。

|根据这个定义，一个简单命令也同时是一个管道。管道的作用是把它前面的那个简单命令的输出作为后面那个简单命令的输入，就上面这个例子来说：

echo “hello ”world 本来是要在标准输出（屏幕）上打印“ hello ”world 的，但是管道现在不让结果输出到屏幕上，而是“流”到 wc -l 这个简单命令，wc -l 就把“流”过来的数据作为它的标准输入进行计算，从而统计出结果是 1 行。关于管道更详细的内容，我们在后面具体实现管道的时候再说明。

1 列表（命令）：列表是一个或多个管道组成的序列，两个管道之间用操作符
;, &, &&, 或 || 分隔。我们看到，这几个操作符都属于控制操作符。例如

echo “hello ”world | wc -l ; echo “nice to meet ”you就是一个列表，它由两个管道组成，管道之间用分号（）隔开
;分号这种控制操作符仅仅表示一种执行上的先后顺序。

1 复合命令这个定义比较复杂，实现起来也有相当难度，在咱们这个示例程序中，就不实现了。以上是

shell 语法规范的定义，我们的 shell 程序就是要以此规范为依据，实现对简单命令、管道和列表的解释。对于列表中的控制操作符，我们只支持分号（），其它的留给读者自己来实现。;接下来，我们具体介绍如何实现一个简单的解释器。
shell实现实例
shell 程序主框架主程序很简单，它在做一些必要的初始化工作之后，进入这样一个循环：
u 打印提示符并等待用户输入
u 获取用户输入
u 分析用户输入
u 解释执行；如果用户输入或者
logout exit 之后，才退出这个循环。用类似伪代码的形式表示如下：

```
while(1) {
print_prompt();
get_input();
parse_input();
if “(” logout || “” exit)
```

```
break;
do_cmd();
}
```

读取用户输入如何获取用户输入？一种方法是通过

getchar() 从标准输入每次读一个字符，如果读到的字符是‘ \’，说明用户键入了回车键，那么就把此前读到的字符串作为用户输入的命令。n代码如下：

```
 int len = 0;
int ch;
char buf[300];
ch = getchar();
while(len < BUFSIZ && ch != '\n') {
buf[len++] = ch;
ch = getchar();
}
if(len == BUFSIZ) {
printf("command is too long\n");
break;
}
buf[len] = '\n';
len++;
buf[len] = 0;
```

但是，我们注意到，在 bash 中，可以用“”和“”键在命令行中左右移动，可以用上下键调用以前使用的命令，可以用退格键来删除一个字符，还可以用<—> tab 键来进行命令行补全。我们的如果也要支持这些功能，那么就必须对这些键进行处理。这样仅仅对用户输入的读取就非常麻烦了。shell实际上，任何需要一个获取用户输入的程序，都会涉及到同样的问题，如何象

```
bash
```

那样处理键盘？GNU readline 库就是专门解决这个问题的，它把对键盘的操作完全封装起来，对外只提供一个简单的调用接口。有了它，对键盘的处理就不再让人头疼了。关于

```
 readline
```

库的详细信息，可以通过 man readline 来看它的帮助页面。在我们的 shell 程序中，我是这样来使用的。 readline

```
char* line;
char prompt[200];
while(1) {
set_prompt(prompt);
if(!(line = readline(prompt)))
break;。。。。。。

}
```

首先通过

```
 set_prompt()
```

来设置要输出的提示符，然后以提示符作为参数调用 readline()，这个函数等待用户输入，并动态创建一块内存来保存用户输入的数据，可以通过返回的指针 line 得到这块内存。在每次处理完用户输入的命令之后，我们必须自己负责来释放这块内存。有了

```
 readline
```

之后，我们就可以象 bash 那样使用键盘了。在通过

```
 readline
```

获取用户输入之后，下一步就是对用户输入的命令进行分析。命令行分析对命令行的分析，实际上是一个词法分析过程。学过编译原理的朋友，都听说过

lex 和yacc 的大名，它们分别是词法分析和语法分析工具。Lex 和 yacc 都有的版本（GNUopen source 的思想实在是太伟大了，什么好东东都有免费的用），分别是 flex 和。 bison所谓“工欲善其事，必先利其器”，既然有这么好的工具，那我们就不必辛辛苦苦自己进行词法分析了。对，我们要用
lex 来完成枯燥的命令行词法分析工作。“去买本《与》（中国电力出版社）来看吧。第一次学当然稍微有点难度，不过一旦掌握了，以后再碰到类似问题，就可以多一个利器，可以节省劳动力了。
lexyacc在我们的这个
shell 程序中，用 flex 来完成词法分析工作。相对语法分析来说，词法分析要简单的多。由于我们只是做一个简单的，因此并没有用到语法分析，而实际上
在 shell bash 的实现代码中，就用到了语法分析和。 yacc关于
lex 的细节，在这里我就不能多说了。程序，通常分为三个部分，其中进行语法分析工作的就是它的第二部分：Lex “规则”。规则定义了在词法分析过程中，遇到什么样的情况，应该如何处理。词法分析的思路，就是根据前面定义的“语法规范”来把用户输入的命令行拆解成
shell首先，我们要把用户输入的命令，以空白字符（键或者空格）分隔成一个个的参数，并把这些参数保存到一个参数数组中。但是，这其中有几种特殊情况。
tab一、如果遇到的字符是“”、“”、“”或“”，由于这些符号是管道或者列表中所用到的分隔符，因此必须把它们当作一个单独的参数。
;><|二、以双引号（”）括起来的字符串要作为一个单独的参数，即使其中出现了空白字符、“”、“”、“”、“”。其实，在
;><|标准中，对引号的处理相当复杂，不仅包括双引号（”），还有单引号（’）、反引号（POSIX），在什么情况下，应该用什么样的引号以及对引号中的字符串应该如何解释，都有一大堆的条款。我们这里只是处理一种极简单的情况。 其次，如果我们遇到换行符（’

\’），那么就结束本次命令行分析。根据前面定义的n shell 语法规范，最上层的是列表命令，因此下一步是把所有的参数作为一个列表命令来处理。根据这个思路，我们来看对应的

lex 规则。

```
%%


"\"" {BEGIN QUOTE;}
[^\n"]+ {add_arg(yytext);}
"\"" {BEGIN 0;}
\n {BEGIN 0; do_list_cmd(); reset_args();}
";" {add_simple_arg(yytext);}
">" {add_simple_arg(yytext);}
"<" {add_simple_arg(yytext);}
"|" {add_simple_arg(yytext);}
[^ \t\n|<>;"]+ {add_arg(yytext);}
\n {do_list_cmd(); reset_args();}
. ;
```

%%我们对这些规则逐条解释：一这条规则，目的是为了在命令行中支持引号，它们用到了

144 lex 规则的状态特性。、
1"\"" {BEGIN QUOTE;}、

```
2[^\n"]+ {add_arg(yytext);}、
3"\"" {BEGIN 0;}、
4\n {BEGIN 0; do_list_cmd(); reset_args();}、
```

1 如果扫描到引号（“），那么进入 QUOTE 状态。在这个状态下，即使扫描到空白字符或“”、“”、“”、“”，也要当作普通的字符。;><|、

2 如果处于状态，扫描到除引号和回车以外的字符，那么调用 QUOTE add_arg()函数，把整个字符串加入到参数数组中。、

3 如果处于状态，扫描到引号，那么表示匹配了前面的引号，于是恢复到默认状态。QUOTE、

4 如果处于状态，扫描到回车，那么结束了本次扫描，恢复到默认状态，并执行QUOTE do_list_cmd()，来执行对列表命令的处理。以下几条规则，是在处于默认状态的情况下的处理。、

```
5";" {add_simple_arg(yytext);}、
6">" {add_simple_arg(yytext);}、
7"<" {add_simple_arg(yytext);}、
8"|" {add_simple_arg(yytext);}、
9[^ \t\n|<>;"]+ {add_arg(yytext);}、
10\n {do_list_cmd(); reset_args();}、
```

5 如果遇到分号（），因为这是一个列表命令结束的操作符，所以作为一个单独的参数，执行; add_simple_arg()，将它加入参数数组。、

6 如果遇到，因为这是一个简单命令结束的操作符，所以作为一个单独的参数，执行 > add_simple_arg()，将它加入参数数组。、

7 如果遇到，因为这是一个简单命令结束的操作符，所以作为一个单独的参数，执行 < add_simple_arg()，将它加入参数数组。、

8 如果遇到管道符号（），因为这是一个管道命令结束的操作符，所以作为一个单独的参数，执行| add_simple_arg()，将它加入参数数组。、

9 对于不是制表符（）、换行符（’ tab\’）、n| 、、和分号（）以外的字符序列，作为一个普通的参数，加入参数数组。<>;、

10 如果遇到换行符，那么结束本次扫描，执行 do_list_cmd()，来执行对列表命令的处理。、

11 对于任意其它字符，忽略通过

lex 的“规则”把用户输入的命令行分解成一个个的参数之后，都要执行 do_list_cmd() 来执行对列表命令的处理。命令处理首先是对处于“语法规范”中最上层的列表命令的处理。

```
 shell
```

1 列表命令的处理过程：依次检查参数数组中的每一个参数，如果是分号（），那么就认为分号前面的所有参数组成了一个管道命令，调用

; do_pipe_cmd() 来执行对管道命令的处理。如果扫描到最后，不再有分号出现，那么把剩下的所有参数作为一个管道命令处理。代码很简单：

```
static void do_list_cmd()
{
int i = 0;
```

```
int j = 0;
char* p;
while(argbuf[i]) {
if(strcmp(argbuf[i], ";") == 0) {// ;
p = argbuf[i];
argbuf[i] = 0;
do_pipe_cmd(i-j, argbuf+j);
argbuf[i] = p;
j = ++i;
} else
i++;
}
do_pipe_cmd(i-j, argbuf+j);
}
```

接下来是对管道命令的处理。管道命令的处理管道是进程间通信（）的一种形式，关于管道的详细解释在《高级环境编程》第章：进程间通信以及《网络编程：第卷：进程间通信》第章：管道和中可以看到。

IPCunix14unix24FIFO我们还是来看一个管道的例子：

```
[root@stevens root]# echo “hello ”world|wc -c |wc -l
```

在这个例子中，有三个简单命令和两个管道。第一个命令是

echo “hello ”，它在屏幕上输出world hello 。由于它后面是一个管道，因此，它并不在屏幕上输出结果，而是把它的输出重定向到管道的写入端。world第二个命令是

wc -，它本来需要指定输入源，由于它前面是一个管道，因此它就从这个管道的读出端读数据。也就是说读到的是c hello ，worldwc -c 是统计读到的字符数，结果应该是。由于它后面又出现一个管道，因此这个结果不能输出到屏幕上，而是重定向到第二个管道的写入端。12第三个命令是

wc -。它同样从第二个管道的读出端读数据，读到的是，然后它统计读到了几行数据，结果是行，于是在屏幕上输出的最终结果是。l1211在这个例子中，第一个命令只有一个“后”管道，第三个命令只有一个“前”管道，而第二个命令既有“前”管道，又有“后”管道。在我们处理管道命令的

do_pipe_cmd()函数中，它的处理过程是：首先定义两个管道

prefd 和，它们分别用来保存“前”管道和“后”管道。此外，还有一个变量 postfd prepipe 来指示“前”管道是否有效。然后依次检查参数数组中每一个参数，如果是管道符号（），那么就认为管道符号前面所有的参数组成了一个简单命令，并创建一个“后”管道。如果没有“前”管道（管道中第一个简单命令是没有“前”管道的），那么只传递“后”管道来调用

|do_simple_cmd()，否则，同时传递“前”管道和“后”管道来调用 do_simple_cmd()。执行完以后，用“前”管道来保存当前的“后”管道，并设置“前”管道有效标识，继续往后扫描。如果扫描到最后，不再有管道符号出现，那么只传递“前”管道来调用

prepipedo_simple_cmd()。代码如下：

```
int i = 0, j = 0, prepipe = 0;
int prefd[2], postfd[2];
char* p;
while(argv[i]) {
if(strcmp(argv[i], "|") == 0) { // pipe
```

```
p = argv[i];
argv[i] = 0;
pipe(postfd); //create the post pipe
if(prepipe)
do_simple_cmd(i-j, argv+j, prefd, postfd);
else
do_simple_cmd(i-j, argv+j, 0, postfd);
argv[i] = p;
prepipe = 1;
prefd[0] = postfd[0];
prefd[1] = postfd[1];
j = ++i;
} else
i++;
}
if(prepipe)
do_simple_cmd(i-j, argv+j, prefd, 0);
else
do_simple_cmd(i-j, argv+j, 0, 0);
```

最后，我们分析简单命令的处理过程。简单命令处理过程我们已经看到，对列表命令和管道命令的处理，实际只是一个分解过程，最终命令的执行还是要由简单命令来完成。在简单命令的处理过程中，必须考虑以下情况：、区分内部命令和外部命令

1根据简单命令的定义，它的第一个参数是要执行的命令，后面的参数作为该命令的参数。要执行的命令有两种情况：一种是外部命令，也就是对应着磁盘上的某个程序，例如、等等。对这种外部命令，我们首先要到指定的路径下找到它，然后再执行它。

wcls二是内部命令，内部命令并不对应磁盘上的程序，例如、等等，它需要自己来决定该如何执行。例如对

cdechoshell cd 命令，就应该根据它后面的参数改变当前路径。shell对于外部命令，需要创建一个子进程来执行它，而对于内部命令，则没有这个必要。外部命令的执行，是通过

exec 函数来完成的。有六种不同形式的 exec 函数，它们可以统称为 exec 函数。我们使用的是 execv()。关于 的细节，请看《环境高级编程》第章：进程控制。execunix8对于内部命令，我们目前支持五种，分别是：：退出解释器

exitshell：改变目录
cd：回显
echo：导入或显示环境变量
export：显示命令历史信息
history这几个内部命令分别由
do_exit()、do_cd()、do_echo()、do_export()、do_history()来实现。、处理重定向

2在简单命令的定义中，包括了对重定向的支持。重定向有多种情况，最简单的是输入重定向和输出重定向，分别对应着“”和“”。

<>输入重定向，就是把“”后面指定的文件作为标准输入，例如：

<

wc < xxx表示把

xxx 这个文件的内容作为 wc 命令的输入。输出重定向，就是把“”后面指定的文件作为标准输出，例如：

>

echo “hello ”world > xxx表示把

echo “hello ”world 的结果输入到 xxx 文件中，而不是屏幕上。为了支持重定向，我们首先对简单命令的参数进行扫描，如果遇到“”或者“”那么就认为遇到了重定向，并把“”或者“”符号后面的参数作为重定向的文件名称。

<><>对于输入重定向，首先是以只读方式打开“”后面的文件，并获得文件描述符，然后将该文件描述符复制给标准输入。

<对于输出重定向，首先是以写方式打开“”后面的文件，并获得文件描述符，然后将该文件描述符复制给标准输出。

>具体实现在

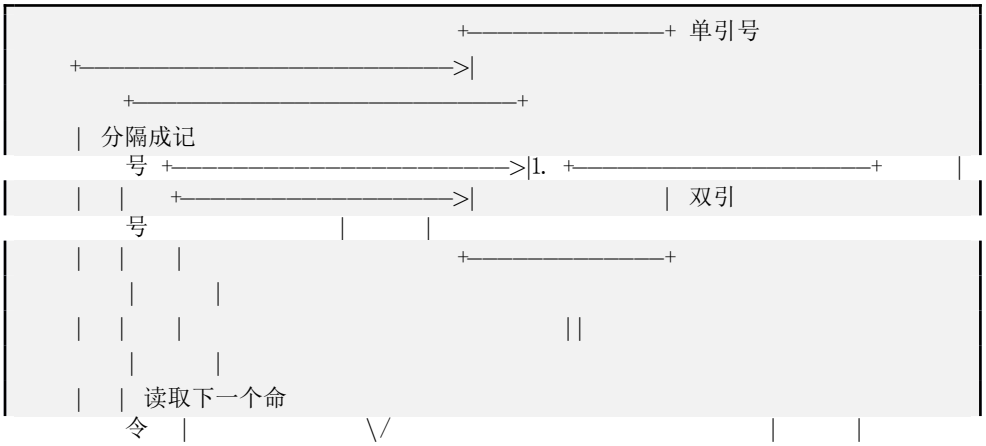
predo_for_redirect() 函数中：、管道的实现

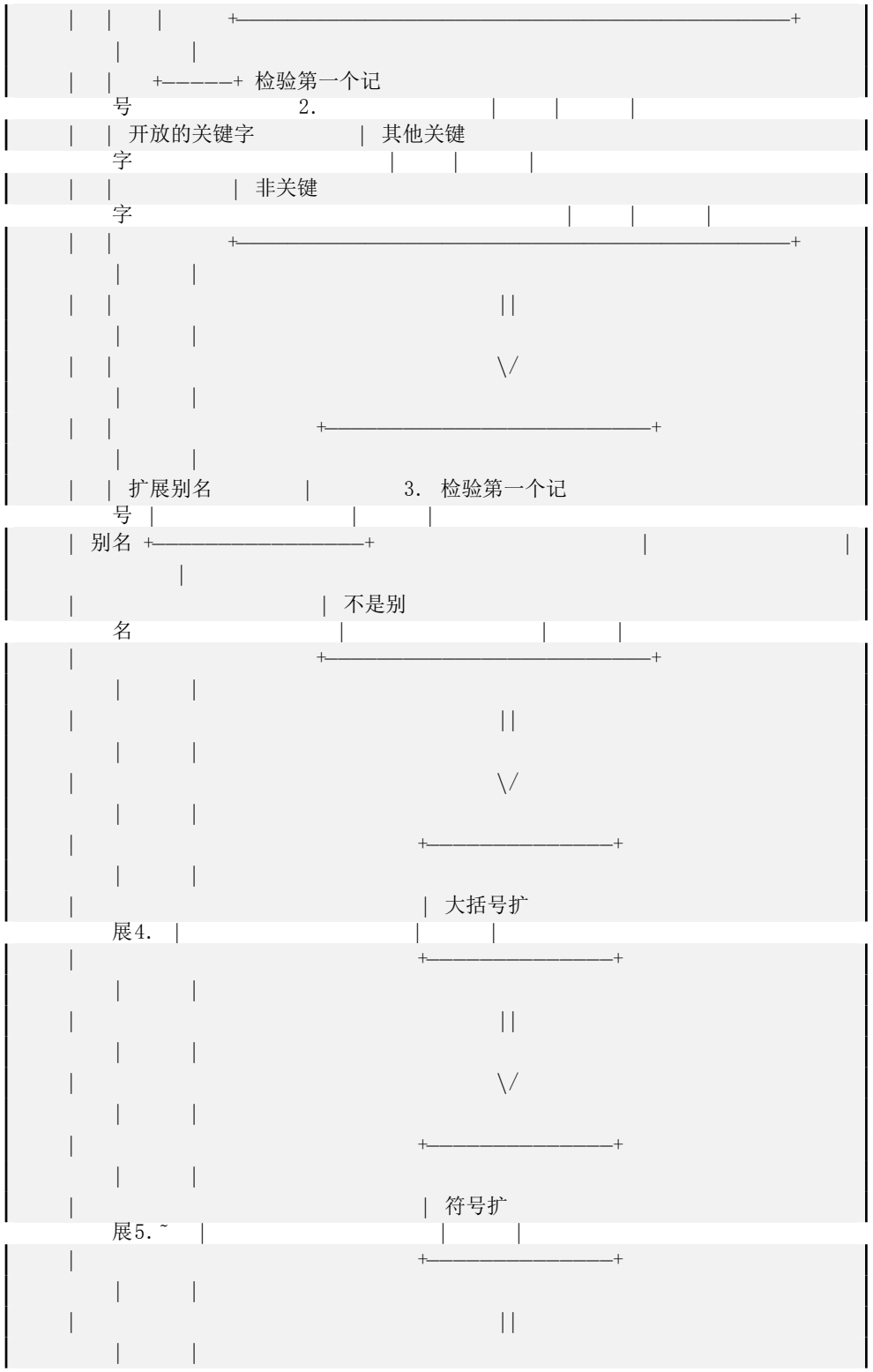
3管道的实现实际上也是一种重定向的处理。对于“前”管道，类似于输入重定向，不同的是，它是把一个指定的描述符（“前”管道的输出端）复制给标准输入。对于“后”管道，类似于输出重定向，不同的是，它把一个指定的描述符（“后”管道的输入端）复制给标准输出。在对管道的处理上，还必须要注意管道和输入或输出重定向同时出现的情况，如果是一个“前”管道和一个输入重定向同时出现，那么优先处理输入重定向，不再从“前”管道中读取数据了。同样，如果一个“后”管道和一个输出重定向同时出现，那么优先处理输出重定向，不再把数据输出到“后”管道中。至此，我们已经描述了实现一个简单的

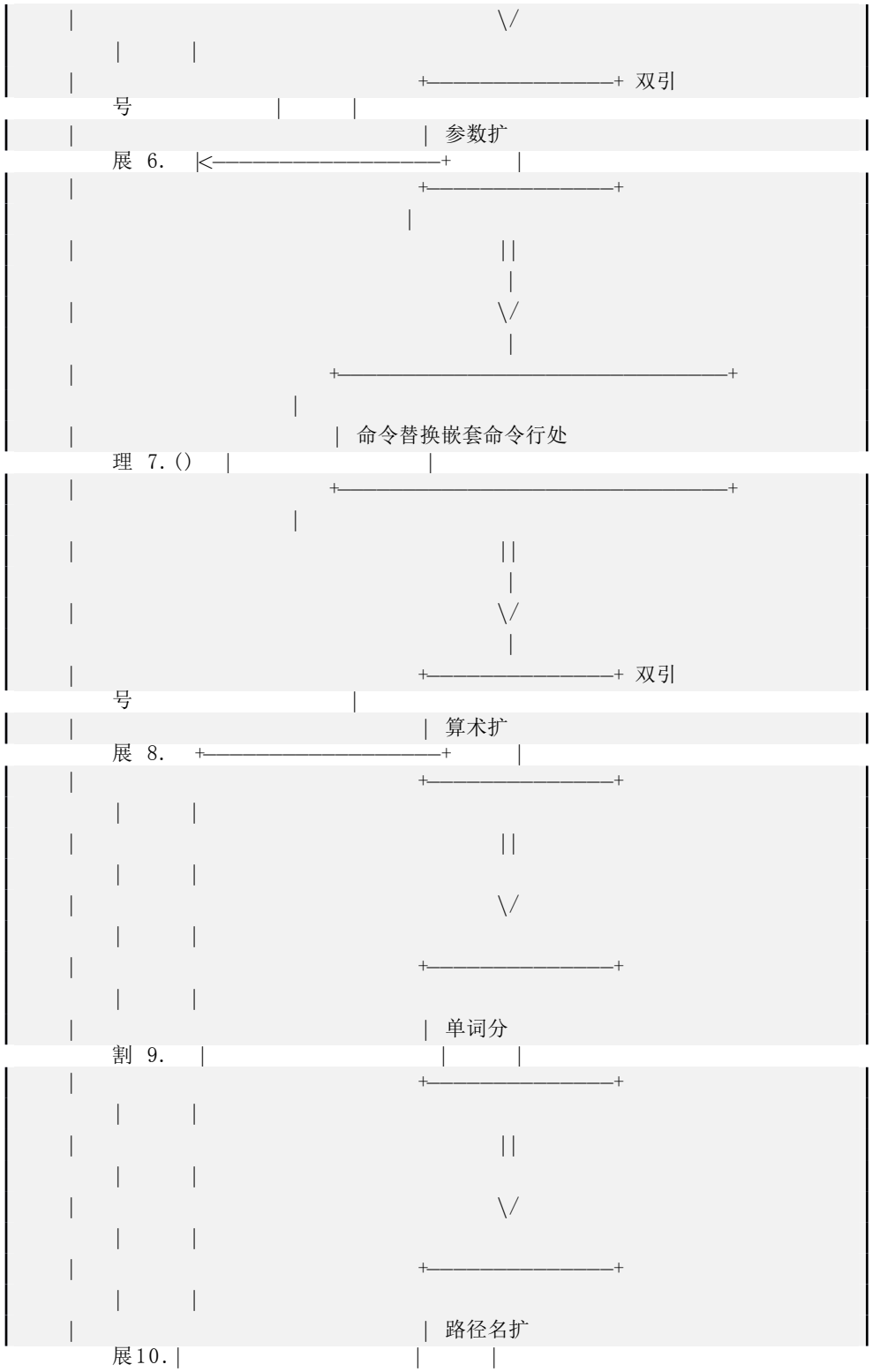
shell 解释器的全部过程，相应的代码和 makefile 在我们的网站上可以下载。希望大家能够结合代码和这篇文章，亲自动手做一次，以加深对shell 解释器的理解恩。。看一下。。我记得一个行

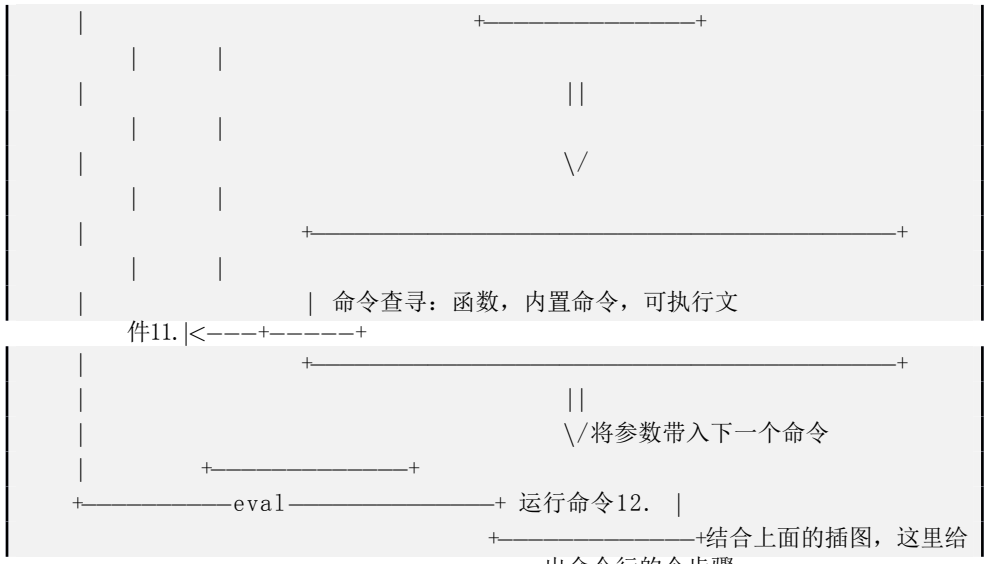
shell 要被扫描次9 才会送入解释我建议看一下那个教程后再写这一章

bash不过我忘了这个文档了




检验第一个记号
2.
开放的关键字
其他关键字
非关键字
扩展别名
3. 检验第一个记号
别名
不是别名
大括号扩展4.
符号扩展5.~





12、将命令行分成由固定元字符集分隔的记号；

1

SPACE, TAB, NEWLINE, ; , (,), <, >, |, &记号类型包括单词，关键字，

I/重定向符和分号。0、检测每个命令的第一个记号，查看是否为不带引号或反斜线的关键字。

2如果是一个开放的关键字，如和其他控制结构起始字符串，，

iffunction或，则命令实际上为一复合命令。{(在内部对复合命令进行处理，读取下一个命令，并重复这一过程。如果关键字不是复合命令起始字符串shell如(等一个控制结构中间出现的关键字then)，则给出语法错误信号。、依据别名列表检查每个命令的第一个关键字；

3如果找到相应匹配，则替换其别名定义，并退回第一步否则进入第步。该策略允许递归别名，还允许定义关键字别名。如

;4alias procedure=function、执行大括号扩展，例如

4a{b,c变成}ab ac、如果位于单词开头，用

5~替换$HOME。~使用的主目录替换

usr~。user、对任何以符号

6开头的表达式执行参数$变量()替换;、对形式

7$(string)的表达式进行命令替换；这里是嵌套的命令行处理。、计算形式为

8$((string))的算术表达式；、把行的参数，命令和算术替换部分再次分成单词，这次它使用

9中的字符做分割符而不是步骤的元字符集；$IFS1、对出现

10*，?，[/ 对执行路径名扩展，也称为通配符扩展；]、按命令优先级表跳过别名

11()，进行命令查寻；、设置完

12I/重定向和其他操作后执行该命令。0二、关于引用、单引号跳过了前个步骤，不能在单引号里放单引号

110、双引号跳过了步骤，步骤，也就是说，只处理个步骤。
21~59~106~8也就是说，双引号忽略了管道字符，别名，替换，通配符扩展，和通过分隔符分裂成单词。

~双引号里的单引号没有作用，但双引号允许参数替换，命令替换和算术表达式求值。可以在双引号里包含双引号，方式是加上转义符
″，还必须转义\″$，`，。\三、的作用；

eval的作用是再次执行命令行处理，也就是说，对一个命令行，执行两次命令行处理。这个命令要用好，就要费一定的功夫。我举两个例子，抛砖引玉。

eval、例子：用

11技巧实现的控制结构evalshellfor用技巧实现的控制结构。

evalshellfor

```
[root@home root]# cat myscript1
#!/bin/sh
```

```
evalit(){
        if [ $cnt = 1 ];then
                eval $@
                return
        else
                let cnt=cnt-1
                evalit $@
        fi
        eval $@
}
cnt=$1
echo $cnt | egrep "^[1-9][0-9]*$" >/dev/null
if [ $? -eq 0 ]; then
        shift
        evalit $@
else
        echo 'ERROR!!! Check your input!'
fi
[root@home root]# ./myscript1 3 hostname
home
home
home
[root@home root]# ./myscript1 5 id |cut -f1 -d' '
uid=0(root)
uid=0(root)
uid=0(root)
uid=0(root)
uid=0(root)注意：里有两个很特殊的变量，它们保存了参数列表。

bash



$，保存了以*指定的分割符所分割的字符串组。$IFS，原样保存了参数列表，也就是
$@"$1""$2"...这里我使用了函数递归以及实现了结构。

evalfor当执行
eval 时，它经历了步骤如下：$@第步，分割成
1eval $@第步，扩展
6为$@hostname第步，找到内置命令
11eval重复一次命令行处理，第步，找到
11命令，执行。hostname注意：也许有人想当然地认为，何必用呢？直接来执行命令就可
    以了嘛。

eval$@例子：一个典型错误的例子
```

```
2错误！这里给个典型的例子大家看看。




[root@home root]# a="id | cut -f1 -d' '"
[root@home root]# $a：无效选项
id -- f请尝试执行‘
id --’来获取更多信息。help
[root@home root]# eval $a
uid=0(root)如果命令行复杂的话包括管道或者其他字符

()，直接执行字符串的内容就会出错。分析如下。$a的处理位于第步——参数扩展，也就
    是说，跳过了管道分析，于是
$a6"|", "cut", "-f1", "-d"都变成了命令的参数，当然就出错啦。id但使用了，它
    把第一遍命令行处理所得的
eval"id", "|", "cut", "-f1", "-d"这些字符串再次进行命令行处理，这次就能正
    确分析其中的管道了。总而言之：要保证你的命令或脚本设计能正确通过命令行处
    理，跳过任意一步，都可能造成意料外的错误！
```

表达式求值 D2S自身不支持表达式求值，D2S脚本中表达式求值借助命令calc命令完成。calc命令的格式如下：

```
calc expr
```

例如：

```
let var1 = 12
let var2 = 73.2
let res = ${calc ($var1+$(var)1/2+$var2)/3.2}
echo $res
```

```
表达式求值，可直接复制到中运行。

VC

#include<stdio.h>
#include<malloc.h>
#include<string.h>
#include<stdlib.h>
```

```
static int gv_a;

typedef struct list    //创建队列元素型
{
    char str[10];
    struct list*pNext;
}LIST;
typedef struct queue    //创建队列表型
{
    LIST *front;
    LIST *rear;
}QUEUE;

typedef struct oprstack     //创建运算符栈型
{
    char opr[100];
    int a;
}OPRSTACK;

typedef struct valstack     //创建运算栈型
{
    float val[30];
    int a;
}VALSTACK;

typedef struct element       //创建表达式栈元素型
{
    char c;
    struct element *pNext;

}ENT;

typedef struct stack            //创建表达式栈 型
{
    ENT *pTop;
    ENT *bottom;
}STACK;



bool calc(char *str,float *result);
float evaluationofexpression(char *str);

int main(void)
```

```
{
    char str[50];
    float result;
    printf("请输入要计算的表达式：");
    scanf("%s",str);
    result = evaluationofexpression(str);
    if (gv_a)
        return 0;
    printf("结果：%f",result);
    printf("\n");


    return 0;
}
/*=========================================================*/
/*不带括号表达式求值=====================================*/
/*=========================================================*/
void insert_char(char *p,char ent,int site)          //向字符串中插入一个元素
{
    int i=0,j;

    while (p[i]!='\0')
        ++i;
    if(site>i)
    {
        p[i] = ent;
        p[i+1] = '\0';
        return;
    }
    for (j=i;j>=site;--j)
    {
        p[j+1] = p[j];
    }
    p[site] = ent;

}

void del_char(char *p,int site)                  //删除字符串中的一个元素
{
    int i=0,j=site;

    while (p[i] != '\0')
        ++i;
    if (site>i)
```

```
        return;
    while (p[j] != '\0')
    {
        p[j] = p[j+1];
        ++j;
    }


}
int find_str(char *p1,char *p2)          //寻找字符串在另一个字符串中第
    一次出现的位置，失败返回-1.
{
    int i=0,j=0,a=1;

    while (p1[i]!='\0')
    {
        j = 0;
        a = 1;
        while (p2[j]!='\0')
        {
            if (p1[i+j] != p2[j])
            {
                a = 0;
                break;
            }

            ++j;
        }
        if (a == 1)
            return i;

        ++i;

    }

    return -1;
}

void form_expression(char *p)            //格式化表达式
{
    char *tmp1 = "--";
    char *tmp2 = "-";
    char *tmp3 = "!";
    int i=0,s1,s2,s3,j;
```

```
    //printf("AAAA");
    s1 = find_str(p,tmp1);          //将字符串中的“——”替换成“”+
    //printf("AAAA");
    while (s1 != -1)
    {
        del_char(p,s1);
        del_char(p,s1);
        insert_char(p,'+',s1);
        s1 = find_str(p,tmp1);

    }
    //printf("%d",s1);
    //printf("AAAA");
    if (p[0] == '-')            //如果第一个字符是‘’则在前面加-0
        insert_char(p,'0',0);
    //printf("AAAA");

    s2 = find_str(p,tmp2);          //将表达式中的‘’替换成‘!’-
    for (j=0;p[j]!='\0';++j)
    {
        if (p[j] == '-' && p[j-1]>=48 && p[j-1]<=57)
            insert_char(p,'+',j);



    }


    //printf("AAAA\n");
    s3 = find_str(p,tmp3);

    while (s3 != -1)            //将表达式中的‘!’替换成“”+-
    {
        del_char(p,s3);
        insert_char(p,'-',s3);
        insert_char(p,'+',s3);
        s3 = find_str(p,tmp3);
    }
    //printf("AAAA\n");


}

void initlist(QUEUE*p)  //初始化链表
{
```

```
    p->front = (LIST*)malloc(sizeof(LIST));
    p->rear = p->front ;
    p->front ->pNext = NULL;

}
void addlist(QUEUE*p,char *c)                //入队
{
    LIST *temp = (LIST*)malloc(sizeof(LIST));
    temp->pNext = NULL;
    strcpy(p->rear ->str ,c);
    p->rear ->pNext = temp;
    p->rear = temp;
}
void dellist(QUEUE *p,char *c)           // 出队
{
    if (p->front == p->rear )
    {
        strcpy(c,"!");
        return;
    }
    LIST *temp;
    temp = p->front;
    p->front = p->front ->pNext ;
    strcpy(c,temp->str );
    free(temp);

}

void oprstack(OPRSTACK*p)        //初始化运算符栈
{
    p->a = 0;

}

char delstack(OPRSTACK*p)      //运算符栈 出栈
{
    char temp;
    if (p->a  == 0 )
        return '!';
    temp = p->opr [p->a ];
    p->a  = p->a -1;
    return temp;

}
void addstack(OPRSTACK*p,char c)         //运算符栈 入栈
```

```
{
    p->a = p->a +1;
    p->opr[p->a ] = c;
}

void initvalstack(VALSTACK *p)          //初始化运算栈
{
    p->a = 0;
}

float delvalstack(VALSTACK *p)          //运算栈 出栈
{
    float temp;
    if (p->a == 0)
        exit(0);
    temp = p->val [p->a ];
    p->a = p->a -1;
    return temp;
}

void addvalstack(VALSTACK *p,float val)     //运算栈 入栈
{
    p->a = p->a +1;
    p->val [p->a ] = val;
}

int opr_assign(char opr)            //运算符优先级赋值
{
    if ((opr == '+')||(opr =='-'))
        return 1;
    if ((opr == '*')||(opr == '/'))
        return 2;
    else
        return 0;


}

int opr_compare(char a,char b) //运算符优先级比较：a>b 返回，1a<返回b，
    相等返回-10
{
    int opr1,opr2;
    opr1 = opr_assign(a);
    opr2 = opr_assign(b);
```

```
    if (opr1>opr2)
        return 1;
    else if (opr1<opr2)
        return -1;
    else
        return 0;
}

bool error_correct(char *str)        //改正表达式中的非法输入
{
    int i;
    for (i = 0;str[i]!='\0';++i)
    {
        if ((str[i]!=42)&&(str[i]!=43)&&(str[i]!=45)&&(str[i]!=46)
            &&(str[i]!=47)&&(str[i]<48 || str[i]>57))
            return false;
    }
    return true;
}

float resultval(float a,float b,char c)          //计算两个双精度的运算
    结果
{
    float result;
    if (c == '+')
        result = a+b;
    else if (c == '-')
        result = a-b;
    else if (c == '*')
        result = a*b;
    else
        result = a/b;
    return result;

}


void freequeue(QUEUE *p)             //释放队列所占用的内存空间
{
    char temp[50];
    while (p->front != p->rear )
        dellist(p,temp);
    free(p->rear );
}
bool ec_expression(char *p)     //纠错
```

```
{
    int i=0;
    int site;
    while (p[i]!='\0')
        ++i;
    //printf("%s\n",p);
    if (p[i-1]>57 || p[i-1]<48)
        return false;

    site = find_str(p,"++") ;
    if (site!=-1)
        return false;

    site = find_str(p,"+*") ;
    if (site!=-1)
        return false;

    site = find_str(p,"+/");
    if (site!=-1)
        return false;

    site = find_str(p,"+.") ;
    if (site!=-1)
        return false;

    site = find_str(p,"-+") ;
    if (site!=-1)
        return false;

    site = find_str(p,"-*");
    if (site!=-1)
        return false;
    site = find_str(p,"-/");
    if (site!=-1)
        return false;

    site = find_str(p,"-.") ;
    if (site!=-1)
        return false;

    site = find_str(p,"*+") ;
    if (site!=-1)
        return false;

    site = find_str(p,"**") ;
```

```
    if (site!=-1)
        return false;

    site = find_str(p,"*/") ;
    if (site!=-1)
        return false;

    site = find_str(p,"*.") ;
    if (site!=-1)
        return false;

    site = find_str(p,"/+") ;
    if (site!=-1)
        return false;
    site = find_str(p,"/*") ;
    if (site!=-1)
        return false;
    site = find_str(p,"//") ;
    if (site!=-1)
        return false;
    site = find_str(p,"/.") ;
    if (site!=-1)
        return false;
    return true;

}
/*不带括号表达式的值=========================================*/

bool calc(char *str,float *result)        //计算字符串表达式结果
{
    QUEUE queue;
    OPRSTACK oprstk;
    VALSTACK valstk;
    OPRSTACK temp1,temp2;
    int i,j;
    char t,tstr[10],t1[2];
    int cmp;
    char que[10];
    float temp3;
    float tmp1,tmp2,res;



    initlist(&queue);            // 完成各个线性表的初始化
    initvalstack(&valstk);
```

```
    oprstack(&oprstk);
    oprstack(&temp1);
    oprstack(&temp2);

    addstack(&oprstk,'#');



    if (error_correct(str) == false )
    {
        gv_a = 1;
        printf("表达式中存在非法字符!");
        return false;
    }

    if (ec_expression(str) == false)
    {
        gv_a = 1;
        printf("表达式格式不正确!");
        return false;
    }

    form_expression(str);
    //printf("%s\n",str);

    for (i = 0;str[i]!='\0';++i)
    {
        if ((str[i]>=48 && str[i]<=57)||(str[i] == 46)||str[i] ==
            45)     //当字符是数字0-9 或“” . 或“”时进行压
            栈 -temp1
        {
             addstack(&temp1,str[i]);   //调试 : 3

        }
        else
        {
            while (temp1.a != 0)            //T
            {
                t = delstack(&temp1);
                addstack(&temp2,t);

            }
            for (j=0;temp2.a !=0;++j)               //T
            {
                tstr[j] = delstack(&temp2);
```

```
            }
            tstr[j] = '\0';
            addlist(&queue,tstr);
            //printf("%s\n",tstr);

            cmp = opr_compare(str[i],oprstk.opr [oprstk.a ]);
            //printf("%d\n",cmp);

            if (cmp<=0)                         //T
            {
                while (oprstk.opr [oprstk.a ]!='#')        //T
                {
                    t1[0] =  delstack(&oprstk);
                //  printf("%c",t1[0]);
                    t1[1] = '\0';
                    addlist(&queue,t1);

                }
                addstack(&oprstk,str[i]);

            }
            else        //T
            {
                addstack(&oprstk,str[i]);

            }

        }             //T

    }
        while (temp1.a != 0)                //T
            {
                t = delstack(&temp1);
                 addstack(&temp2,t);

            }
        for (j=0;temp2.a !=0;++j)                //T
            {
                tstr[j] = delstack(&temp2);
            }
        tstr[j] = '\0';
        addlist(&queue,tstr);
        while (oprstk.opr [oprstk.a ]!='#')
        {
            t1[0] =  delstack(&oprstk);
```

```
            t1[1] = '\0';
            addlist(&queue,t1);
        }

         //tralist(&queue); 调试                    // ：队列错误



        while (queue.front != queue.rear )              //T
        {
             dellist(&queue,que);
             //printf("%c\n",que[0]);              调试：//正常que
             if ((que[0]>=48) && (que[0]<=57)||que[0] == 45)
                             //T
             {

                 temp3 = (float)atof(que);
                 //printf("%f\n",temp3);          调试：//正常temp3
                 addvalstack(&valstk,temp3);

             }
             else                                       //T
             {
                // printf("%c\n",que[0]); 调试：          //que正常[0]
                 tmp1 = delvalstack(&valstk);
                 tmp2 = delvalstack(&valstk);
                 res =  resultval(tmp2,tmp1,que[0]);
                 //printf("%f",res);
                 addvalstack(&valstk,res);

             }
        }

        *result = delvalstack(&valstk);
        freequeue(&queue);                      //  释放队列所占用的空
            间queue

        return true;

}


/*=====================================================*/
/*带括号表达式求值=====================================*/
/*=====================================================*/
```

```
void initstk(STACK *p)                    //初始化栈
{
    p->bottom = (ENT*)malloc(sizeof(ENT));
    p->pTop = p->bottom  ;
    p->bottom ->pNext = NULL;
}

void addstk(STACK *p,char ent)            //压栈
{
    ENT *temp = (ENT*)malloc(sizeof(ENT));
    temp->c = ent;
    temp->pNext = p->pTop ;
    p->pTop = temp;
}

char delstk(STACK *p)                 //出栈
{

    ENT *temp;
    char t;
    if (p->pTop == p->bottom )
        return '!';
    temp = p->pTop ;
    p->pTop = p->pTop ->pNext ;
    t = temp->c ;
    free(temp);
    return t;

}

void freestack(STACK *p)
{
    while (p->pTop != p->bottom )
        delstk(p);
    free(p->bottom );
}
bool check_brackets(char *p)              //检查括号
{
    OPRSTACK temp;
    int i;
    temp.a = 0;

    for (i=0;p[i]!='\0';++i)
    {
```

```
        if (p[i] == 40)
            addstack(&temp,p[i]);
        if (p[i] == 41)
            if (temp.a = 0)
                return false;
            else
                delstack(&temp);


    }

    if (temp.a == 0)
        return true;
    else
        return false;

}
bool check_obs(char *p)
{
    int i;
    for (i=0;p[i]!='\0';++i)
    {
        if (p[i] == '(')
            if (i == 0)
            {   if ((p[i+1]<48 || p[i+1]>57 )&& p[i+1]!=45 && p[i
                +1]!=40)
                    return false;
            }
            else
            {
                if ((p[i-1]!='+') && (p[i-1]!='-') &&(p[i-1]!='*')
                    && (p[i-1]!='/') && p[i-1]!=40)
                    return false;
                if ((p[i+1]<48 || p[i+1]>57 ) && p[i+1]!=45 && p[i
                    +1]!=40)
                    return false;
            }

        //  printf("AAAAA\n");
        if (p[i] == ')')
            if (p[i+1] == '\0')
            {   if ((p[i-1]<48 || p[i-1]>57) && p[i-1]!=41)
                    return false;
            }
            else
```

```
            {
                if (p[i+1]!='+' && p[i+1]!='-' && p[i+1]!='*'&& p[
                    i+1]!='/'&&p[i+1]!=41)
                    return false;
                if ((p[i-1]<48 || p[i-1]>57) && p[i-1]!=41)
                    return false;

            }

    }
    return true;
}

/*表达式求值==================================*/
float evaluationofexpression(char *str)            //表达式求值
{
    STACK stk;
    STACK stk1,stk2;
    char tmp1,tstr[50],tmp3,tstr1[50];
    int i,j,k,l,m,n,o;
    float ftmp;
    char *string,fstring[50];
    int decpt,  sign;
    float result;
    gv_a = 0;


    initstk(&stk);
    initstk(&stk1);
    initstk(&stk2);

    if (check_brackets(str) == false)
    {
        printf("括号不匹配!");
        gv_a = 1;
    }
    if (check_obs(str) == false)
    {
        printf("格式错误!");
        gv_a = 1;
    }
```

```
    //printf("%s",str); 调试：参数正常        //
    for (i=0;str[i]!='\0';++i)
    {
        if (str[i]!=41)                    //如果不是右括号则进行压栈 str
            addstk(&stk,str[i]);
        else
        {
            while (stk.pTop ->c !=40)    //如果是右括号则出栈 str 将出
                栈元素进行压栈知道出现左括号 str1
            {
                tmp1 =  delstk(&stk);
                addstk(&stk1,tmp1);

            }

            delstk(&stk);                  //删除栈中的左括号str
            j = 0;
            while (stk1.pTop != stk1.bottom )    //将中的元素全部转入数
                组中str1tsrt 以纠正顺序
            {
                tstr[j] = delstk(&stk1);
                    ++j;
            }
            tstr[j] = '\0';

            calc(tstr,&ftmp);
            string = fcvt(ftmp,5,&decpt,&sign);
        //  printf("%s\n",string);
        //  printf("%d\n",decpt);
        //  printf("%d\n",sign);
            fstring[49] = '\0';
            o = 0;
            while (fstring[o]!='\0')
            {
                fstring[o] = 65;
                ++o;
            }


            if (sign == 0)
                if (decpt <= 0)
                {
                    fstring[0] = '0';
                    fstring[1] = '.';

                    for (m = 0;m<-decpt;++m)
```

```
                {
                    fstring[2+m] = '0';
                }


            }
            else
                fstring[decpt] = 46;

        else
        {
            if (decpt <= 0)
            {
                fstring[0] = '-';
                fstring[1] = '0';
                fstring[2] = '.';
                for (m = 0;m<-decpt;++m)
                {
                    fstring[3+m] = '0';
                }
            }
            else
            {
                fstring[0] = '-';
                fstring[decpt+1] = '.';
            }
        }
        m = 0;
        n = 0;
        while (string[n]!='\0')
        {
            if (fstring[m] == 65)
            {
                fstring[m] = string[n];
                ++n;
            }
            ++m;

        }
        fstring[m] = '\0';
    /*  for (n=0;string[n]!='\0';++m)
        {
            if (m == decpt)
                fstring[m] = '.';
            else
```

```
                {
                    fstring[m] = string[n];
                    ++n;
                }

            }
            fstring[m] = '\0';
            printf("%s\n",fstring);
            //printf("%d\n",n);
        //  printf("%d\n",m);
        //  printf("%s\n",fstring);
            */

            for (k=0;fstring[k]!='\0';++k)
            {
                addstk(&stk,fstring[k]);
            }



        }




    }
    //travstk(&stk);
                  //调试：正常

    while (stk.pTop != stk.bottom )
    {
        //printf("AAAA\n");
        tmp3 =  delstk(&stk);
        addstk(&stk2,tmp3);

        //printf("%c\n",tmp3);

    }
    //printf("AAAA\n");                调试：错误！！！//
     //travstk(&stk2);


    for (l=0;stk2.pTop != stk2.bottom  ;++l)
        tstr1[l] = delstk(&stk2);
    tstr1[l] = '\0';
```

```
    //printf("%s\n",tstr1);
    calc(tstr1,&result);
    //printf("AAAA");
    freestack(&stk);                    //  释放stk stk1 stk2 所占用的内
        存空间
    freestack(&stk1);
    freestack(&stk2);

    return result;

}
```

我建议自己把Lua的程序看完了后再研究这一章的实现。

```
在中常用的特殊符号罗列如下：

shell

# ; ;; . , / \\ 'string'| ! $ ${} $? $$ $*

\"string\"* ** ? : ^ $# $@ `command`{} [] [[]] () (())

|| && {xx,yy,zz,...}~ ~+ ~- & \\< ...\\> + - %= == !=

# 井号 (comments)这几乎是个满场都有的符号，除了先前已经提过的第一行
\"\"
#!/bin/bash井号也常出现在一行的开头，或者位于完整指令之后，这类情况表示符号后
    面的是注解文字，不会被执行。

# This line is comments.
echo \"a = $a\" # a = 0由于这个特性，当临时不想执行某行指令时，只需在该行开
    头加上
 # 就行了。这常用在撰写过程中。
#echo \"a = $a\" # a = 0如果被用在指令中，或者引号双引号括住的话，或者在倒
    斜线的后面，那他就变成一般符号，不具上述的特殊功能。



~ 帐户的 home 目录算是个常见的符号，代表使用者的
 home 目录：cd ；也可以直接在符号后加上某帐户的名称：~cd ~或者当成是路径的一
     部份：user~/bin
~+ 当前的工作目录，这个符号代表当前的工作目录，她和内建指令的作用是相同
    的。 pwd
# echo ~+/var/log
~- 上次的工作目录，这个符号代表上次的工作目录。
# echo ~-/etc/httpd/logs
```

; 分号 (Command separator)在
shell 中，担任连续指令功能的符号就是分号。譬如以下的例子：\"\"\"\"cd ~/backup ; mkdir startup ;cp ~/.* startup/.

;; 连续分号 (Terminator)专用在
case 的选项，担任 Terminator 的角色。
case \"$fop\" inhelp) echo \"Usage: Command -help -version filename\";;version) echo \"version 0.1\" ;;esac

. 逗号 (dot就是“点”,)在
shell 中，使用者应该都清楚，一个 dot 代表当前目录，两个 dot 代表上层目录。
CDPATH=.:~:/home:/home/web:/var:/usr/local在上行
CDPATH 的设定中，等号后的 dot 代表的就是当前目录的意思。如果档案名称以
dot 开头，该档案就属特殊档案，用 ls 指令必须加上 -a 选项才会显示。除此之外，在 regularexpression 中，一个 dot 代表匹配一个字元。

'string' 单引号 (single quote)被单引号用括住的内容，将被视为单一字串。在引号内的代表变数的符号，没有作用，也就是说，他被视为一般符号处理，防止任何变量替换。
$

\"string\" 双引号 (double quote)被双引号用括住的内容，将被视为单一字串。它防止通配符扩展，但允许变量扩展。这点与单引数的处理方式不同。

heyyou=homeecho \"$heyyou\" # We get home

`command` 倒引号 (backticks)在前面的单双引号，括住的是字串，但如果该字串是一列命令列，会怎样？答案是不会执行。要处理这种情况，我们得用倒单引号来做。

fdv=`date +%F`echo \"Today $fdv\"在倒引号内的
date +%F 会被视为指令，执行的结果会带入 fdv 变数中。

, 逗点 (，标点中的逗号comma)这个符号常运用在运算当中当做区隔用途。如下例
\"\"
#!/bin/bashlet \"t1 = ((a = 5 + 3, b = 7 - 1, c = 15 / 3))\"echo \"t1= $t1, a = $a, b = $b\"

/ 斜线 (forward slash)在路径表示时，代表目录。

cd /etc/rc.dcd ../..cd /通常单一的
/ 代表 root 根目录的意思；在四则运算中，代表除法的符号。
let \"num1 = ((a = 10 / 2, b = 25 / 5))\"

\\ 倒斜线在交互模式下的
escape 字元，有几个作用；放在指令前，有取消的作用；放在特殊符号前，则该特殊符号的作用消失；放在指令的最末端，表示指令连接下一行。 aliases

type rmrm is aliased to `rm -i'# \\rm ./*.log上例，我在
rm 指令前加上 escape 字元，作用是暂时取消别名的功能，将 rm 指令还原。
bkdir=/home# echo \"Backup dir, \\$bkdir = $bkdir\"Backup dir, $bkdir = /home上例
echo 内的 \\，$bkdirescape 将 $ 变数的功能取消了，因此，会输出，而第二个 $bkdir 则会输出变数的内容$bkdir /。home

| 管道 (pipeline)
pipeline 是 UNIX 系统，基础且重要的观念。连结上个指令的标准输出，做为下个指令的标准输入。
who | wc -l善用这个观念，对精简
script 有相当的帮助。

! 惊叹号(negate or reverse)通常它代表反逻辑的作用，譬如条件侦测中，用
!= 来代表不等于\"\"
if [\"$?\" != 0]thenecho \"Executes error\"exit 1fi在规则表达式中她担任反逻辑
\"\" 的角色
ls a[!0-9]上例，代表显示除了
a0, a1 .... a9 这几个文件的其他文件。

: 冒号在
bash 中，这是一个内建指令：什么事都不干，但返回状态值\"\" 。0
:
echo $? # 回应为 0
: > f.$$上面这一行，相当于
cat /dev/null >f.。不仅写法简短了，而且执行效率也好上许多。$$有时，也会出现以下这类的用法

: ${HOSTNAME?} ${USER?} ${MAIL?}这行的作用是，检查这些环境变数是否已设置，没有设置的将会以标准错误显示错误讯息。像这种检查如果使用类似
test 或这类的做法，基本上也可以处理，但都比不上上例的简洁与效率。 if

? 问号 (wild card)在文件名扩展
(Filename expansion)上扮演的角色是匹配一个任意的字元，但不包含 null 字元。
ls a?a1善用她的特点，可以做比较精确的档名匹配。

* 星号 (wild card)相当常用的符号。在文件名扩展
(Filename expansion)上，她用来代表任何字元，包含 null 字元。
ls a*a a1 access_log在运算时，它则代表乘法。
\"\"
let \"fmult=2*3\"除了内建指令，还有一个关于运算的指令，星号在这里也担任
letexpr乘法的角色。不过在使用上得小心，他的前面必须加上\"\"escape 字元。

** 次方运算两个星号在运算时代表次方

\"\" 的意思。
let \"sus=2**3\"echo \"sus = $sus\" # sus = 8

$ 钱号(dollar sign)变量替换
(Variable Substitution)的代表符号。
vrs=123echo \"vrs = $vrs\" # vrs = 123另外，在
Regular Expressions 里被定义为行 \"\" 的最末端 (end-of-line)。这个常用
在、、grepsedawk 以及 vim(vi) 当中。

${} 变量的正规表达式
bash 对 ${} 定义了不少用法。以下是取自线上说明的表列
${parameter:-word} ${parameter:=word} ${parameter:?word} ${
parameter:+word} ${parameterffset} ${parameterffset:length} $
{!prefix*} ${#parameter} ${parameter#word} ${parameter##word}
${parameter%word} ${parameter%%word} ${parameter/pattern/
string} ${parameter//pattern/string}

$*
$* 引用的执行引用变量，引用参数的算法与一般指令相同，指令本身为，其后为，然后依
此类推。引用变量的代表方式如下：script01
$0, $1, $2, $3, $4, $5, $6, $7, $8, $9, ${10}, ${11}.....个位数的，
可直接使用数字，但两位数以上，则必须使用
{} 符号来括住。
$* 则是代表所有引用变量的符号。使用时，得视情况加上双引号。
echo \"$*\"还有一个与
$* 具有相同作用的符号，但效用与处理方式略为不同的符号。

$@
$@ 与 $* 具有相同作用的符号，不过她们两者有一个不同点。符号
$* 将所有的引用变量视为一个整体。但符号 $@ 则仍旧保留每个引用变量的区段观念。

$#这也是与引用变量相关的符号，她的作用是告诉你，引用变量的总数量是多少。

echo \"$#\"

$? 状态值 (status variable)一般来说，
UNIX(linux) 系统的进程以执行系统调用exit()来结束的。这个回传值就是值。回传给
父进程，用来检查子进程的执行状态。status一般指令程序倘若执行成功，其回传值
为；失败为
0 。1
tar cvfz dfbackup.tar.gz /home/user > /dev/nullecho\"$?\"$$由于进程
的是唯一的，所以在同一个时间，不可能有重复性的
ID 。有时，会需要产生临时文件，用来存放必要的资料。而此亦有可能在同一时间被使用
者们使用。在这种情况下，固定文件名在写法上就显的不可靠。唯有产生动态文件
名，才能符合需要。符号或许可以符合这种需求。它代表当
前PIDscriptscript$$shell 的。 PID
echo \"$HOSTNAME, $USER, $MAIL\" > ftmp.$$使用它来作为文件名的一部份，
可以避免在同一时间，产生相同文件名的覆盖现象。

ps: 基本上，系统会回收执行完毕的，然后再次依需要分配使用。所以 PID script 即使临时文件是使用动态档名的写法，如果script 执行完毕后仍不加以清除，会产生其他问题。

( ) 指令群组 (command group)用括号将一串连续指令括起来，这种用法对 shell 来说，称为指令群组。如下面的例子：(cd ~ ; vcgh=`pwd` ;echo $vcgh)，指令群组有一个特性，会以产生shell 来执行这组指令。因此，在其中所定义的变数，仅作用于指令群组本身。我们来看个例子subshell

```
# cat ftmp-01#!/bin/basha=fsh(a=incg ; echo -e \"\\n $a \\n\")echo $a#./ftmp-01incgfsh
```

除了上述的指令群组，括号也用在 array 变数的定义上；另外也应用在其他可能需要加上字元才能使用的场合，如运算式。escape

(( ))这组符号的作用与 let 指令相似，用在算数运算上，是 bash 的内建功能。所以，在执行效率上会比使用指令要好许多。 let

```
#!/bin/bash(( a = 10 ))echo -e \"inital value, a = $a\\n\"(( a++))
    echo \"after a++, a = $a\"
```

{ } 大括号 (Block of code)有时候 script 当中会出现，大括号中会夹着一段或几段以分号做结尾的指令或变数设定。\"\"

```
# cat ftmp-02#!/bin/basha=fsh{a=inbc ; echo -e \"\\n $a \\n\"}echo $a#./ftmp-02inbcinbc
```

这种用法与上面介绍的指令群组非常相似，但有个不同点，它在当前的 shell 执行，不会产生。 subshell大括号也被运用在函数 \"\" 的功能上。广义地说，单纯只使用大括号时，作用就像是个没有指定名称的函数一般。因此，这样写也是相当好的一件事。尤其对输出输入的重导向上，这个做法可精简 script script 的复杂度。此外，大括号还有另一种用法，如下

{xx,yy,zz,...}这种大括号的组合，常用在字串的组合上，来看个例子

mkdir {userA,userB,userC}-{home,bin,data}我们得到 userA-home, userA-bin, userA-data, userB-home, userB-bin,userB-data, userC-home, userC-bin,userC-，这几个目录。这组符号在适用性上相当广泛。能加以善用的话，回报是精简与效率。像下面的例子data

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

如果不是因为支援这种用法，我们得写几行重复几次呀！

[ ] 中括号常出现在流程控制中，扮演括住判断式的作用。

```
if [ \"$?\" != 0 ]thenecho \"Executes error\"exit1fi
```

这个符号在正则表达式中担任类似范围 \"\" 或集合 \"\" 的角色

```
rm -r 200[1234]
```

上例，代表删除 2001, 2002, 2003, 2004 等目录的意思。

[[ ]]这组符号与先前的
[] 符号，基本上作用相同，但她允许在其中直接使用 || 与&& 逻辑等符号。
#!/bin/bashread akif [[$ak > 5 || $ak< 9]]thenecho $akfi

|| 逻辑符号这个会时常看到，代表
or 逻辑的符号。

&& 逻辑符号这个也会常看到，代表
and 逻辑的符号。

& 后台工作单一个
& 符号，且放在完整指令列的最后端，即表示将该指令列放入后台中工作。
tar cvfz data.tar.gz data > /dev/null&

\\< ...\\> 单字边界这组符号在规则表达式中，被定义为边界的意思。譬如，当我们想找寻
\"\" the 这个单字时，如果我们用
grep the FileA你将会发现，像
there 这类的单字，也会被当成是匹配的单字。因为 the 正巧是的一部份。如果我们要必免这种情况，就得加上 there 边界\"\" 的符号
grep '\\' FileA

+ 加号 (plus)在运算式中，她用来表示加法。
\"\"
expr 1 + 2 + 3此外在规则表达式中，用来表示很多个的前面字元的意思。
\"\"
grep '10\\+9' fileB1091009100009100009310010009这个符号在使用时，前面必须加上#escape 字元。

– 减号 (dash)在运算式中，她用来表示减法。
\"\"
expr 10 – 2此外也是系统指令的选项符号。

ls –expr 10 – 2在
GNU 指令中，如果单独使用 – 符号，不加任何该加的文件名称时，代表标准输入的意思。这是\"\" 指令的共通选项。譬如下例GNU
tar xpvf –这里的
– 符号，既代表从标准输入读取资料。不过，在
cd 指令中则比较特别
cd –这代表变更工作目录到上一次工作目录。
\"\"

% 除法 (Modulo)在运算式中，用来表示除法。
\"\"
expr 10 % 2此外，也被运用在关于变量的规则表达式当中的下列

${parameter%word}${parameter%%word}一个
% 表示最短的 word 匹配，两个表示最长的 word 匹配。

= 等号 (Equals)常在设定变数时看到的符号。

vara=123echo \" vara = $vara\"或者像是
PATH 的设定，甚至应用在运算或判断式等此类用途上。

== 等号 (Equals)常在条件判断式中看到，代表等于
\"\" 的意思。
if [$vara == $varb]下略
...

!= 不等于常在条件判断式中看到，代表不等于
\"\" 的意思。
if [$vara != $varb]下略
...

^这个符号在规则表达式中，代表行的开头
\"\" 位置，在中也与叹号[]\"!\"()一样表示“非”输出输入重导向

/
> >> < << :> &> 2&> 2<>>& >&2文件描述符

(File Descriptor)，用一个数字（通常为）来表示一个文件。0-9常用的文件描述符如下：文件描述符名称常用缩写默认值

0 标准输入 stdin 键盘
1 标准输出 stdout 屏幕
2 标准错误输出 stderr 屏幕我们在简单地用
< 或时，相当于使用> 0< 或（下面会详细介绍）。 1>
* cmd > file把命令的输出重定向到文件中。如果已经存在，则清空原有文件，使用的选项可以防止复盖原有文件。
cmdfilefilebashnoclobber
* cmd >> file把命令的输出重定向到文件中，如果已经存在，则把信息加在原有文件後面。
cmdfilefile
* cmd < file使命令从读入
cmdfile
* cmd << text从命令行读取输入，直到一个与相同的行结束。除非使用引号把输入括起来，此模式将对输入内容进行变量替换。如果使用
textshell<<- ，则会忽略接下来输入行首的，结束行也可以是一堆再加上一个与相同的内容，可以参考後面的例子。tabtabtext
* cmd <<< word把（而不是文件）和後面的换行作为输入提供给。
wordwordcmd

* cmd <> file以读写模式把文件重定向到输入，文件不会被破坏。仅当应用程序利用了这一特性时，它才是有意义的。

filefile

* cmd >| file功能同，但即便在设置了>时也会复盖文件，注意用的是noclobberfile而非一些书中说的，目前仅在|!中仍沿用csh实现这一功能。>!

: > filename 把文件\"filename截断为长度\"0.# 如果文件不存在, 那么就创建一个长度的文件与0('touch的效果相同').

cmd >&n 把输出送到文件描述符n

cmd m>&n 把输出到文件符的信息重定向到文件描述符 mn

cmd >&- 关闭标准输出

cmd < &n 输入来自文件描述符n

cmd m<&n 来自文件描述各个mn

cmd <&- 关闭标准输入

cmd <&n- 移动输入文件描述符而非复制它。（需要解释）n

cmd >&n- 移动输出文件描述符而非复制它。（需要解释） n注意：实际上复制了文件描述符，这使得

>&cmd > file 与2>&1cmd 2>&1 >的效果不一样。file

Chapter 9

关于D2IM

这一章我将试图涉及一些在前面没有涉及到但是我认为很重要的东西。

9.1 游戏背景

9.2 框架结构

某一时刻至多只有一个脚本在运行。但是可以有零到多个任务在执行。

```
while( 游戏尚未结束 ){
    process_input();
    process_script();
    update();
}
```

可见由三部分组成。首先看输入处理：

```
process_input(){
    while( 获取每一个输入事件 ){
        if( 禁用输入 ){丢弃该事件

        }else{
            if( 为键盘事件 ){当前焦点窗口处理之
                GUI
                if( 未被处理 ){丢弃该事件

                }
            }else( 为鼠标事件 ){递归让所有尝试截获
                GUI
                if( 未被截获 ){游戏逻辑对输入进行截获

                    if( 未被截获 ){丢弃该事件

                    }
                }
            }else{丢弃，不支持的输入事件

            }
        }
    }
}
```

其次是脚本和任务处理问题：

```
process_script(){
    if( 当前执行队列为空 ){
        if( 当前有脚本在运行 ){
            process_scriptline()
            if( 命令为start ){
                while( 成功取到一条命令 ){
                    switch 命令
                        case: start报错，不能嵌套
                            start
                        case: wait
                            break
                        default:单行命令处理


    if( 当前有脚本在运行 ){
        if( 任务计数器为0 ){取一条命令

            if( 命令为start ){
                while取一条命令，不为(wait){创建并启动一个任务任务计数器自增

                    1
                }
            }
        }解锁任务计数器

    }
}
```