科研中计算机的使用

Skills in Scientific Researches

作者：冷轩

时间：February 14, 2020

LaTeX 模板：*Ethan Deng & Liam Huang: ElegantBook*

关注时代需求，解决时代问题

献给那些帮助过我的人们

这页是背面

目录

前言

文档定位

1. 自己使用过程的积累，大部分转载自网上，我会尽量写明转载地址，难免有些忘了，还请见谅。
2. 对一个程序的应用不熟，看它的说明文档来的最快！

科研心得

前言一

Ubuntu 更新很快，每年都发布两个版本，4 月一个，10 月一个，很多技术技巧很快会成为历史[1]，注意技术技巧的时间是很有必要的。

我使用 Ubuntu 的过程可谓是曲曲折折、磕磕碰碰、一步一步走来。最开始接触到 Ubuntu 是 08 年在学校 BBS 上看到相关的帖子，被 Ubuntu 的 3D 特效和开放源代码所吸引。最开始是怎么装的 Ubuntu 记得不是很清楚，好像是下载光盘镜像刻录在光盘上，用光盘装的，装的是双系统。遇到的最大问题是，上外网的问题，学校上外网是用客户端的，按流量计费。但是在 Ubuntu 下搞定这个客户端一直没有很好解决。还有一个没有很好解决的是显卡驱动的问题。而最最重要的是 Windows XP 下的很多软件在 Ubuntu 下没有很好的对应版本，如 QQ 和银行网盾。学校玩 Ubuntu 的人也很少。于是虽说是双系统，但大部分还是在用 XP，Ubuntu 成为一个摆设。后来因为占用硬盘，干脆格掉了。这是第一个时期。过了一段时间，手又痒了，这次学会了用 U 盘装，还是双系统。虽然 Ubuntu 下有网页登录客户端，但始终没有摆脱 Windows，Ubuntu 再次成为摆设。没有摆脱 Windows，Ubuntu 的使用始终是很难发展的。第三个时期，也就是现在，可以总算对 Windows 说再见了。毕业了，没有了客户端，上网直接拨号，上网问题不存在了；电脑换了，Ubuntu 也发展了很多，显卡驱动问题不存在了；WebQQ 也完善了许多，QQ 问题不在了；Ubuntu 有了专门的软件商店，里面有很多免费的开放源代码的软件，再加上虚拟机 Virtualbox 功能强大，我在 Ubuntu 下设了个虚拟机装了 XP 来应对一些目前没法替代的软件，其实也就是网购、网银那些了，这样软件的问题可以说也不存在了。时至今日，我终于才迈过 Ubuntu 的门槛，而这花了我四年时间。这个时间真是有点长，有各种因素，最本质的还是自己真正花在这上面的时间太少。欣慰的是，我终究没有放弃这。

这个使用记是伴随 Ubuntu 的更新一起前进的，最开始是基于 11.10，现在基于 12.04 。有些技巧会过时，有些基本的却不会变，这得自己好好斟酌体会。

——20120505

[1]但是 Linux 的那些基本的东西可以说是不会变的

前言二

这是这份笔记结构的第一次重大调整，原名叫《Ubuntu 使用记》，主要是我使用 Ubuntu 过程中的记录。最开始写这份笔记的时候是 2012 年，现在已经是 2014 年，近两年了。为什么现在才考虑进行结构调整并改名为《计算机的使用》，或许是现在对计算机了解的更深入，使用的更多吧。本笔记的主题将是自己对计算机所有相关的学习记录，包括系统 Linux 和其中一个发行版 Ubuntu 的使用，也有脚本语言 Shell 的学习，也有 Fortran、C、C++ 等语言的学习。

——20140307

定位为纯粹的个人的计算机学习记录，记的都是自己想记下的东西。

——20140719

前言三

这是这份文档的第二次重大调整，原名叫《计算机的使用》，现改名叫《科研中计算机的使用》。缩小范围，针对科研服务的。

Ubuntu 使用大事件

2014.7.30——Ubuntu 和 Win7 双系统安装最终解决之道

这两天，真的是备受煎熬。由于历史原因，没想到自己对 Linux 系统的需求性越来越强，当时给笔记本安装 Ubuntu 时分配的硬盘空间太小了，不够用，要重装。说实话，自己是很不愿意重装的，毕竟用了很久了，很多东西要重新安装，配置，很麻烦的。一重装，问题就来了，这也确定了我接下来两天煎熬的日子。安装 Ubuntu 的时候，不能识别已经有的 Win7。怀疑系统版本问题，怀疑启动盘没制作好，怀疑电脑硬盘分区问题。最后已经存在的 Win7 也重装了，电脑系统算是彻底重装了。Win7 系统版本不好，装了不满意，网上下载，那天却莫名其妙的断网。重装了几次 Win7，硬盘分区也试了很多次，安装 Ubuntu 的时候总是检测不到已有的 Win7。很烦躁，不知道自己能不能搞定，甚至质疑自己当初重装系统的决定。不过，这个决定是正确的，这条路必须走，问题必须拿下。问题到了第二天有了转机，终于在网上搜到问题的解释。迷雾渐渐消散，我也似乎领会到了解决之道。

自己分区安装 Ubuntu，然后在 Win7 下用软件 EasyBCD 建立引导。Linux 文件系统清晰明了，很喜欢，我只分三个区就行了。1、/ boot 分区，包含了操作系统的内核和在启动系统过程中所要用到的文件，一般 200 MB 就行了；2、/swap 交换分区，物理内存大于 512MB 时分配与物理内存等容量，也是一种文件系统，它的作用是作为 Linux 的虚拟内存。在 Windows 下，虚拟内存是一个文件，pagefile.sys，而在 Linux 下，虚拟内存需要使用独立分区；3、/分区，剩下的容量都用于根分区了，觉得没必要细分，细分反而麻烦。至于如何在 Win7 下用软件 EasyBCD 建立引导，网上资料较多，搜索便是。

安装方法是正确的但也不是很顺利，安装过程不太正常，试了两次都是这样。怀疑制作安装盘的 U 盘出问题了。不巧，没其他 U 盘了，又开始借 U 盘，也不是很顺利，跟技术没关系就不讲了。果真是 U 盘的问题，期待许久的界面终于在第二天晚上快要回宿舍的时间见到了。这次折腾，不是没又收获，其实还挺大的。对 Ubuntu 的安装有了深入的了解，可以说学会了终极安装的方法。在这个过程中，也发现了新的东西。只是希望，以后折腾是在风险可控的范围内尝试。虽然这次数据备份了，但是我只有这台笔记本，安装不上，那真是麻烦了。

失败确实是成功之母。我真的是在屡屡失败之中成长起来的。心态也要锻炼起来，不要起伏波动太大，从容应对。

no zuo no die。就在写这个的过程中，我又折腾了。在试 nvidia 的显卡驱动，结果图形界面进不去了。好在还有命令行界面，多年的折腾经验告诉我，镇定，要镇定。果断网上搜索，一堆看法。后来想到，卸载 nivdia 驱动就行了，于是敲入 `sudo apt-get --purge remove nvidia-*` 命令。然后，图形界面又回来了。果断还是用系统默认的显卡驱动吧。

做事之前，先做风险评估。尝试固然很好，但要做风险评估。

2014.9.11——双显卡驱动最终解决之道

Ubuntu 的显卡驱动问题一直都是很头疼的问题。这次不是手贱了。一次更新后，开机在 5 个点亮完就没反应了。进入修复模式，偶然瞥到 “load fallback graphics devices [fail]”，不用说又是显卡驱动问题。其实没这个提示，出现这个问题也大概猜到是显卡驱动的问题。回想到上次的经历，这次问题的处理心里大概有个底。首先想办法进入命令行模式。以往版本的系统启动时都有这个选项的，14.04 版本隐藏了。本来想想办法调出这个模式的引导的。但在这之前抱着运气的态度再次尝试了修复模式。在修复模式下，选择进入低画质模式，虽然成功登陆，但是桌面空白，只有鼠标，可以理解，本来就是显卡驱动出问题了嘛。突然想起进入命令行模式的快捷键，Ctrl+Alt+F1，果然进入了。然后就是按照上次的经历，输入命令，移除所有关于 nvidia 的官方驱动，即：

```
apt-get --purge remove nvidia-*
```

成功了！图形界面进去了。可是问题没完，ubuntu 的开源显卡驱动，不太好。首先，显示面只占了方形区域，两边空白，鼠标还有闪烁，很不好用。安装 nivida 官方驱动又出问题，怎么办？我想先查看下我的电脑的显卡信息吧，`lspci |grep VGA`，赫然显示：

00:02.0 VGA compatible controller: Intel Corporation 2nd Generation Core Processor Family Integrated Graphics Controller (rev 09)

01:00.0 VGA compatible controller: NVIDIA Corporation GF108M [GeForce GT 630M] (rev ff)

我的电脑是双显卡，立即想起关于双显卡驱动的 Bumblebee Project。打开主页 http://www.bumblebee-project.org，按照说明，安装 Bumblebee，重启，无需再额外装 nvidia 驱动了，世界平静了。

2014.10.15——又进不了系统了

今天弹出软件更新，看着就更新了下。可是没想到，在下载 flash 插件的时候一直卡着下载不下来。我下意识的就知道问题要来了，但也没办法只能强关。重启后，果然出了问题，无线网卡打不开。我知道是之前更新的问题，于是想着重新更新下。可是还是卡在 flash 插件的更新那了。只能再次强关，这次就没那么幸运，系统果断进不了了。网上查询了下，貌似用 Ubuntu 启动优盘启动，然后在运行命令修护。虽然操作看着还会，只是挺麻烦的。后来想着开机时有修护模式，就试着进入了修护模式。但是第一次进入修护模式还是老问题。后来第二次进入修护模式，选了较低版本的内核，成功进入了修护模式。运行了软件包修护选项，经过一系列的修护后，终于又进入系统了。后来重启也正常了第一次进入修护模式没有成功，大概是最新的内核在更新的时候出了问题吧，所以选用久了内核成功了。Linux 系统真的好折腾人啊，不敢再更新了。

也别一朝被蛇咬，十年怕井绳。以后更新的时候，注意看看更新文件。那个可能引起麻烦的，Ubuntu 用了这么多年了，心里也有个数。不要因为困难而停止进步。

学习及参考资料

参考书

- 《鸟哥的 Linux 私房菜：基础学习篇》鸟哥，王世江；
- 《Linux 就是这个范儿》赵鑫磊，张洁；
- 《Linux 命令行与 Shell 脚本编程大全》Richard Blum，Christine Bresnahan，武海峰（译）；
- 《Unix&Linux 大学教程》Harley Hahn，张杰良（译）

不错的学习网站

- http://www.92csz.com/study/linux/
- http://vbird.dic.ksu.edu.tw/
- Python 学习：http://www.w3cschool.cc/python/python-tutorial.html

第 零 章 科研入门技能及心得

0.1 一些该注册的

gmail: 至少两个。一个注册用，一个通讯用。真名。

research gate

google scholar

ocrc

0.2 文献检索与管理

google scholar 检索

zotero 管理

0.3 Pdf 批量全文搜索

Recoll

http://www.linuxdown.net/install/faq/20160318_how_linux_5065.html

0.4 程序和文档版本管理——Git&Github

以我们的量子耗散动力学、光谱计算为例

1. 找个合适的模型作为标准模型；
2. 只要新增内容，就新复制一个，说明改进；
3. 所有新功能先在标准模型上实现，再应用；
4. 结合 git 版本控制。

参考：bilibili 搜索 git，如尚硅谷官方、庄七。

Git 基本概念

- 工作区：写代码
- 暂存区：临时存储
- 本地库：历史版本

分支管理

0.4.1 初始操作

(1) 本地库初始化（建立本地库）

- git init
  进入项目目录，在命令行输入 git init，会产生一个.git 的子目录，存放的是本地库相关的目录和文件，不要删除和胡乱修改。

(2) 设置签名
- git config user.name XX
- git config user.email XX

(3) 配置
- `git config --global core.editor "vim"`：使用 Vim 作为编辑器

0.4.2 提交操作

1. git status: 查看状态
2. `git add <file>`：添加追踪文件到缓存区 / `git rm --cached <file>`：移除缓存区追踪文件
3. `git commit <file>`：提交文件到库
4. `git commit -a`：直接将所有修改文件提交到库（也可以分成两步，先 git add 更新缓存区，再 git commit 提交到库）
5. `git commit -m ‘‘xxx’’ <file>`：跳过消息编辑器步骤，直接提交“xxx”更新记录

0.4.3 版本穿梭操作

1. `git log`：历史记录查询
   精炼形式：`git reflog`
2. `git reset --hard <index>`：历史版本选择
   注意，还有其他两个参数`--soft`、`--mixed`。基于这个功能，可以实现删除文件并找回，但前提是文件存在时的状态必须提交到了本地库。

0.4.4 比较文件差异

1. git diff <file>：将工作区中的文件和暂存区进行比较
2. git diff < 本地库中某个历史版本 > <file>：将工作区中的文件和本地库中历史文件进行比较

0.4.5 分支管理

1. `git branch -v`：查看分支
2. `git branch <name>`：创建分支
3. `git checkout <name>`：切换分支
4. 合并分支
   - 切换到要合并到的分支

- `git merge <branch name>`:

5. 解决合并冲突
   - 编辑文件，删除特殊符号
   - 把文件修改到满意的程度，保存退出
   - `git commit -m` ‘‘日志信息’’。注意此时不能带文件名。

0.4.6 远程仓库 Github

(1) 创建远程仓库：略！

(2) 推送到远程仓库
   - `git remote -v`：查看远程库
   - `git remote add <name> <https://address>` ：添加远程库
   - `git push <name> <branch>`：推送分支

(3) ssh 免密推送

第一部分

Linux

第一章 Linux 基本概念

1.1 发音

对 Linux 的发音大致有这么几种："里那克斯"、"里你克斯"与"里扭克斯"等。其实官方的标准发音为 `['li:nks]`，因为这个发音是创始人 Linus 的发音。如果你不认识这个音标，那么就读成"里那克斯"。当然你发音成什么，并没有人会说你，完全是一个人的习惯而已。

1.2 来源

Linux 是根据 UNIX 演变过来的。当年 linus 就是因为接触到了 UNIX 而后才自己想开发一个简易的系统内核的，他开发的简易系统内核其实就是 Linux。当时 linus 把开发的这个系统内核丢到网上提供大家下载，由于它的精致小巧，越来越多的爱好者去研究它。人们对这个内核添枝加叶，而后成为了一个系统。Linux 是免费的。其实这里的免费只是说 Linux 的内核免费。在 Linux 内核的基础上而产生了众多 Linux 的版本。

1.3 发行版

Linux 的发行版说简单点就是将 Linux 内核与应用软件做一个打包。较知名的发行版有：Ubuntu、RedHat、CentOS、Debain、Fedora、SuSE、OpenSUSE、TurboLinux、BluePoint、RedFlag、Xterm、SlackWare 等。

1.4 系统目录结构

在 Linux 中，所有设备都是用文件名来表示的，这与我们所熟悉的 Windows 系统有很大不同，Linux 中并没有所谓 C 盘、D 盘之类的说法，并且在 Linux 中只有文件和目录，所有文件和目录都是以倒树状结构挂载在"/"（读做根目录）目录下，也正是因此 Linux 系统的管理员被称为 root（树根）。

- **/bin——Shell 命令**

  bin 是 Binary(二进制) 的缩写。这个目录存放着最经常使用的命令，也就是 Shell 命令。提供给所有用户和脚本程序使用。所以，即使其他文件系统还没有被挂接时，他们也要能够工作，这是将这些内容放在根文件系统或根分区下的主要原因。/bin 目录不得包含任何子目录。

- **/sbin——必备的系统软件**

  必备的系统软件存放处。用于系统维护的软件和某些只限 root 用户使用的命令存储在/sbin、/usr/sbin 以及/usr/local/sbin 目录下。/sbin 目录存储的则是那些在系

统启动、恢复、还原以及修复过程中必备的系统工具，也是对/bin 下内容的补充。在/usr 被挂载才需要的程序一般放在/usr/sbin 目录下。用户自行安装的系统管理程序放在/usr/local/sbin 目录下。

- **/boot——启动过程所需内容**

  该目录包含了 Linux 启动过程所需要的所有内容。更为专业一点地说，/boot 目录下所保存的数据是用于内核开始执行用户级程序之前使用的东西。说白了就是跟内核有关的那点东西，引导加载器、内核镜像等。这里也可能包含备份的主引导扇区信息和扇区映像文件。

- **/cdrom——**
- **/dev——设备文件**

  dev 是 Device(设备) 的缩写。该目录下存放的是 Linux 的外部设备，在 Linux 中访问设备的方式和访问文件的方式是相同的。

- **/etc——配置文件**

  这个目录用来存放所有的系统管理所需要的配置文件和子目录。

- **/home——用户专有**

  /home 为用户专有目录，每个用户都会在/home 目录下有其对应的子目录，而且这个子目录一般与他的用户名相同。没有任何程序会依赖于这个目录，这个目录也不属于根文件系统。

  同一个程序，不同的用户有不同的配置方式，这些用户独有的配置文件就保存在它的 home 目录中，一般以“.”开头（隐藏文件）。如果某个程序需要创建多个配置文件，它就会在用户的 home 目录下创建一个以“.”开头的目录（隐藏目录），这些配置文件放入目录中。这种情况下，配置文件不再需要以“.”开头了。

  一个用户拥有对其 home 目录绝对的控制权，可以做任何事情。不仅仅是保存自己的图片、文档、音乐等，还能在这里安装软件，甚至在 home 目录下模拟整套 Linux 文件的系统布局都没问题。

- **/root——root 专有**

  这是 root 账户专有的 home 目录，除 root 之外，任何人不具备访问权限。而对于普通账户的 home 目录，root 是可以随意访问。是否为 root 账户提供专有 home 目录取决于开发者或者用户的使用习惯，不过还是建议提供这个/root 目录。

- **/lib——共享库和内核模块**

  这个目录存放着必备的共享库和内核模块。在系统启动阶段和运行根文件系统上的各种命令时，都要用到这些共享库。

- **/lost+found——**

  这个目录一般情况下是空的，当系统非法关机后，这里就存放了一些文件。

- **/media——可移动设备**

  Linux 系统会自动识别一些设备，例如 U 盘、光驱等等，当识别后，Linux 会把识别的设备挂载到这个目录下。曾经出现过很多其他名称的挂载点，如/cdrom、/mnt 或/mnt/cdrom 等，现在统一规范到/media 中。虽然很多人习惯将这些可移动设

备挂载在/mnt 子目录中，但这与传统不符，因为/mnt 实际上只是临时挂载点。如果系统中有不止一个同类的驱动器，则在其挂载类型目录后面带有一个 0 起始的数字，比如“cdrom0、cdrom1……”

- **/mnt——临时挂载**

  系统提供该目录是为了让用户临时挂载别的文件系统的，此目录的内容属于局部问题，不应该影响任何程序的运行。

- **/opt——额外安装软件**

  这里是系统额外安装软件的目录。每个软件会在/opt 下拥有一个自己的目录，那里面存放着它所有的可执行程序和静态数据。这个专有目录名应该与它本身的名字相一致，这样用户就很容易管理它们。也可以使用厂商名称，不过这个厂商名称必须是在 Linux 名称与号码分配管理局注册过的。厂商名下面还应该是软件名，毕竟同一个厂商可以有多个软件。

  /opt/bin、/opt/doc、/opt/include、/opt/info、/opt/lib 和/opt/man 是保留给本地系统管理员使用的。有些软件包可能会提供一个前端文件（链接或副本），系统管理员可以将这个文件放在这些保留目录中，但是在缺少这些保留目录时也要保证可以正常工作。

  用户调用的程序必须放在软件包目录下的 bin 子目录中。如果包含联机帮助，则要将它们放在对应软件包的 share/man 子目录下，而且必须拥有与/usr/share/man 目录中相同的组织结构。

  软件有可变文件，必须将这些可变文件安装在/var/opt 目录下。本机专有的配置文件必须安装在/etc/opt 目录下。没有哪些附加软件会将它们的文件放在/opt、/var/opt 和/etc/opt 之外，除非那个软件的某些文件必须放在特定位置，否在不能正常工作。比如，设备锁文件必须放在/var/lock 中，设备文件必须放在/dev 中。

- **/proc——**

  /proc 这个目录是一个虚拟的目录，它是系统内存的映射，我们可以通过直接访问这个目录来获取系统信息。这个目录的内容不在硬盘上而是在内存里，我们也可以直接修改里面的某些文件，比如可以通过下面的命令来屏蔽主机的 ping 命令，使别人无法 ping 你的机器：

- **/run——**

- **/srv——服务进程所需数据**

  某些服务进程启动之后，它们需要读取的数据会放在/srv 目录中。提供/srv 目录一方面为了方便用户找到特定服务的数据文件，另一方面也为某些服务放置它们的执行脚本（如 CGI 脚本）、只读或可写数据提供一个合适的地方。提供给特定用户的数据应该放在那个用户的 home 目录下。

  目前还没有规范的/srv 下子目录的命令方法。有一种方式是按照协议构造/srv 下的数据，如 ftp、rsync、www、cvs 等等。对于大型系统，可能利用功能管辖范围构造/srv 下的内容，如/srv/physics/www、/srv/compsci/cvs 等。它的具体内容在不同主机间会有很大的差异，因此，没有程序会依赖于/srv 下特定的子目录结构，也没

有哪个程序必须在/srv 下保存数据。不过，/srv 目录始终会作为服务进程数据存放地而一直存在着。

- **/sys——**

  /sys 这是 linux2.6 内核的一个很大的变化。该目录下安装了 2.6 内核中新出现的一个文件系统 sysfs ，sysfs 文件系统集成了下面 3 种文件系统的信息：针对进程信息的 proc 文件系统、针对设备的 devfs 文件系统以及针对伪终端的 devpts 文件系统。该文件系统是内核设备树的一个直观反映。当一个内核对象被创建的时候，对应的文件和目录也在内核对象子系统种被创建。

- **/tmp——临时文件**

  /tmp 必须能够被任何程序、任何用户访问，它存放临时文件。任何程序都不用对/tmp 目录下任何文件或目录负责，尽管如此，还是建议系统重启后，清楚/tmp 目录下的内容。

- **/usr 目录组织结构**

  /usr 是整个文件系统的第二个主要区段。用户的很多应用程序和文件都放在这个目录下，类似与 Windows 下的 program files 目录。

  - /usr/bin

    系统中绝大多数的可执行文件都放置在这里。脚本语言的解释器或扩展部分都存放在这里。这是因为 Shell 脚本解释器（Shell 脚本的第一行往往带有一个!#<path> 类型的注释）不能依赖于某个特别的目录，这就要求必须标准化它们的位置。Bourn Shell 和 C Shell 解释器已经被锁定在/bin 目录中，而 Perl、Python 和 TCL 却经常被放置在不同的地方，于是/usr/bin 下的内容往往是到其实际位置的符号链接。

  - /usr/sbin

    那些非必要的系统管理软件的可执行文件存放在这里。那些用于还原、更新、修复系统，以及挂载/usr 目录的系统管理程序都必须放在/sbin 目录下。

  - /usr/include

    系统内所有通用的 C 语言头文件都放置在这里。

  - /usr/lib

    /usr/lib 内的内容是库和目标文件，它们都是二进制文件，但不被用户和 Shell 脚本直接执行。应用程序可以在这里创建独立的子目录。如果应用程序在这里创建了独立的子目录，它所有的与体系结构有关的执行数据都必须放在这个子目录中。

  - /usr/local

    /usr/local 是整个文件系统的第三个主要区段，是供系统管理员安装局域性软件使用的。它保证了当系统更新时不会把它们覆盖掉。其中包含局域性 bin、etc、include、lib、man、sbin、share、src 等目录或符号链接。如果/usr/local/include 和/use/include 有相同的头文件时，前者优先被搜索。局部优先于全局，/usr/local/bin 的优先级也高于/usr/bin。

- /usr/share

  这里存放着所有与体系结构无关的数据文件，它的内容可以共享给运行在任何体系结构下的操作系统。例如，某个站点可能由 i386、Alpha 和 PPC 三种体系结构平台构成，但它们可能共同维护唯一的一个挂载自某处的/usr/share 目录。需要注意的是，虽然/usr/share 可以共享于不同体系结构之间，但必须保证它们运行的是不同系统结构平台的相同操作系统。任何程序或软件包包含或需要的不用修改的与体系结构无关的数据都应该保存在/usr/share（如果安装的是局域性软件，应该是/usr/local/share）目录中。

- /usr/share/man

  这个目录是专门为联机帮助而准备的，它包含了 “/” 和 “/usr” 下大部分程序的帮助信息。组织结构看似相当复杂，但还是有章可循的。各联机帮助页面被储存在 <mandir>/<locale>/man<section>/<arch> 这一路径中。/usr/share/man 目录下有如 man1、man2……man8 这样的子目录，它们的具体分工如下：

  - man1：存放公共可用的命令或软件的指南页面
  - man2：存放所有系统调用的说明
  - man3：存放库函数的说明信息
  - man4：存放特种文件的说明，如/dev 目录下的设备文件和提供网络协议支持的内核接口等
  - man5：存放大部分数据文件的格式说明，这包括各种包含文件、程序输出文件和系统文件
  - man6：存放游戏程序、演示程序和一些不太重要的程序的说明文档
  - man7：存放其他类型的在线帮助信息，如 troff 等文本处理宏的信息
  - man8：存放系统维护程序文档。

  上述目录也必须出现在/usr/share/man/<locale> 中，除非它们里面没有任何内容。这是为了提供多语言版本支持而提供的，其中 <locale> 代表了不同的语言。<locale> 可能的名称是 en、fr、ko、ja 等。同理，如果某些内容与体系结构有关的话，如设备驱动器或底层系统管理命令，则需要将这些内容放置在对应的 <arch> 目录下。/usr/local 下的软件在线帮助信息保存在/usr/local/man 中，组织结构与/usr/share/man 是相同的。

  数字 1 至 8 为前面讲述的 <section>。通常，每个帮助页都有一个独立文件，这个文件带有 “.<section>” 的后缀，如 ctrlatlde1.8。另外，某些大型软件系统的帮助文档可能不遵循这一规则，反倒会使用它们自己定义的后缀。如 X Window 就会使用.x 作为后缀。其他位置的帮助文档，提供多语言版本支持时，同样遵循/usr/share/man 的多语言版本支持规则。

- /usr/share/misc

  那些不需要在/usr/share 目录下拥有自己目录的应用程序，会将它们的体系结构无关的数据放置在这里。

- /var 目录组织结构

/var 目录包含着系统中绝大多数的随时变化的数据，比如系统日志。/var 的某些内容是不能共享给其他系统的，最典型的就是/var/logs、/var/lock 和/var/run。当然，某些内容是可以共享给其他系统的，特别是/var/mail、/var/cache/man 和/var/spool/news。/var 的创建是为了能够以只读的方式挂载/usr 目录。/usr 里面的东西一旦有被写入的要求，则必须转移到/var 下。

- /var/cache

  该目录的内容是来自应用程序的缓存数据，这些内容可以加快费事的运算或 I/O 处理任务。应用程序必须能够再生或恢复这些数据。与/var/spool 不同，删除缓存文件不会造成数据丢失错误。

- /var/lib

  该目录保存着与某个应用程序或系统有关的状态信息。所谓状态信息，就是程序运行过程中不断修改的数据，且仅与具体系统有关。对于控制程序的具体行为，用户不应通过修改/var/lib 下的文件来完成。在应用程序启动期间或同一个程序不同实例之间，状态信息通常是用于保存程序所处的状态的。一个应用程序（或一组相关联的应用程序）必须为其状态数据占有/var/lib 下的一个子目录。不过还有一个 misc 子目录，专门提供给那些不需要占用一个子目录的程序。其他子目录仅提供给那些包含在发行版中有这一需求的应用程序。

- /var/lock

  锁文件必须被保存在/var/lock 目录结构中。锁文件用于硬件设备和其他被多个应用程序共享的资源，例如串行设备锁文件，最初可能放在/usr/spool/locks 或/usr/spool/uucp 中，现在必须放在/var/lock 中。锁文件的命令方式必须是”LCK..“后接设备及本名。例如，/dev/ttyS0 的锁文件名为”LCK..ttyS0 “。锁文件的内容组织格式，必须是 HDB UUCP 锁文件格式。HDB 格式以 10 字节 ASCII 数值字符储存进程 ID，尾部带有换行符。例如，进程 1230，它的锁文件将包含 11 字符：空格、空格、空格、空格、空格、空格、1、2、3、0 和换行符。

- /var/log

  该目录包含了各种日志文件。大多数日志文件都保存在这里或相应的子目录中。

- /var/mail

  此处是用户邮件文件的存放处，所有邮箱文件必须是以 UNIX 邮箱格式存储。mail spool 必须能够访问/var/mail，而且 mail spool 文件取自具体用户。此目录移自/var/spool/mail，就是为了能够与几乎所有的 UNIX 实现相一致。这一改变，使得一个/var/mail 共享于多个主机以及多个不同的 Unix 实现成为可能。更为重要的是，这不需要转移 mail spool 的物理位置，不过，程序和头文件还需要转移到/var/mail 中。

- /var/opt

  /opt 下软件包的可变数据都应该安装到/var/opt 下对应的子目录中去，这

个子目录名称应该与/opt 下的子目录名称相一致，里面内容的组织结构没有特殊要求。

- /var/run

  从系统启动开始至当前的系统运行状态信息保存在这目录。该目录下的文件，在系统启动进程执行开始，必须被清空。程序可以在/var/run 下拥有一个子目录，鼓励程序使用不止一个运行时文件。进程标识文件（PID 文件），原来是放置在/etc 中的，现在必须放置在/var/run 中。PID 文件的命名规则是“<程序名 >.pid”，例如：crond 的 PID 文件名就是 crond.pid。PID 文件的内部格式依然保持不变，必须以 ASCII 码的数字字符描述进程标识符，同时末尾带有换行符。如，若 crond 的进程 ID 为 25，则/var/run/crond.pid 将包含三个字符：2、5 和换行符。记录当前谁正在使用系统的 utmp 文件也保存在这里。那些使用 Unix-domian 套接字的系统程序，必须将它们的套接字文件放置在该目录中。

- /var/spool

  /var/spool 中的数据就是供以后用的，一旦被处理完毕，就会被删除。

- /var/tmp

  /var/tmp 目录使得应用程序要求在系统重启期间保护临时文件或目录成为可能，因此，保存在/var/tmp 中的数据，比/tmp 中的数据得到更多的保护。/var/tmp 中的文件或目录，在系统启动时不得删除。

在 Linux 系统中，有几个目录是比较重要的，平时需要注意不要误删除或者随意更改内部文件。/etc：上边也提到了，这个是系统中的配置文件，如果你更改了该目录下的某个文件可能会导致系统不能启动。/bin、/sbin、/usr/bin、/usr/sbin：这是系统预设的执行文件的放置目录，比如 ls 就是在/bin/ls 目录下的。值得提出的是，/bin、/usr/bin 是给系统用户使用的指令（除 root 外的通用户），而/sbin、/usr/sbin 则是给 root 使用的指令。/var：这是一个非常重要的目录，系统上跑了很多程序，那么每个程序都会有相应的日志产生，而这些日志就被记录到这个目录下，具体在/var/log 目录下，另外 mail 的预设放置也是在这里。

1.5 文件属性

用 ls -l 查看当前目录下的文件时，共显示了 9 列内容（用空格划分列），都代表了什么含义呢?

第 1 列，包含的东西有该文件类型和所属主、所属组以及其他用户对该文件的权限。第一列共 10 位。其中第一位用来描述该文件的类型。上例中，我们看到的类型有“d”，“-”，其实除了这两种外还有“l”，“b”，“c”，” s” 等。

d 表示该文件为目录；

- 表示该文件为普通文件；

l 表示该文件为连接文件（linux file），上边提到的软连接即为该类型；

b 表示该文件为块设备文件，比如磁盘分区；

c 表示该文件为串行端口设备，例如键盘、鼠标。

s 表示该文件为套接字文件（socket），用于进程间通信。

后边的 9 位，每三个为一组。均为 rwx 三个参数的组合。其中 r 代表可读，w 代表可写，x 代表可执行。前三位为所属主（user）的权限，中间三位为所属组（group）的权限，最后三位为其他非本群组（others）的权限。下面拿一个具体的例子来述说一下。

一个文件的属性为-rwxr-xr– ，它代表的意思是，该文件为普通文件，文件拥有者可读可写可执行，文件所属组对其可读不可写可执行，其他用户对其只可读。

对于一个目录来讲，打开这个目录即为执行这个目录，所以任何一个目录必须要有 x 权限才能打开并查看该目录。例如一个目录的属性为 drwxr–r– 其所属主为 root，那么除了 root 外的其他用户是不能打开这个目录的。

第 2 列，表示为连接占用的节点（inode），若为目录时，通常与该目录地下还有多少目录有关系，关于连接（link）在以后章节详细介绍。

第 3 列，表示该文件的所属主。

第 4 列，表示该文件的所属组。

第 5 列，表示该文件的大小。

第 6 列、第 7 列和第 8 列为该文件的创建日期或者最近的修改日期，分别为月份日期以及时间。

第 9 列，文件名。如果前面有一个. 则表示该文件为隐藏文件。

1.6 文件类型

在前面的内容中简单介绍了普通文件 (-)，目录 (d) 等，在 linux 文件系统中，主要有以下几种类型的文件。

1）正规文件（regular file）：就是一般类型的文件，当用 ls –l 查看某个目录时，第一个属性为”-“的文件就是正规文件，或者叫普通文件。正规文件又可分成纯文字文件（ascii）和二进制文件（binary）。纯文本文件是可以通过 cat, more, less 等工具直接查看内容的，而二进制文件并不能。例如我们用的命令/bin/ls 这就是一个二进制文件。

2）目录（directory）：这个很容易理解，就是目录，跟 windows 下的文件夹一个意思，只不过在 linux 中我们不叫文件夹，而是叫做目录。ls –l 查看第一个属性为”d”。

3）连接档（link）：ls –l 查看第一个属性为“l”，类似 windows 下的快捷方式。这种文件在 linux 中很常见，而且笔者在日常的系统运维工作中用的很多，所以你要特意留意一下这种类型的文件。在后续章节笔者会介绍。

4）设备档（device）：与系统周边相关的一些档案，通常都集中在 /dev 这个目录之下！通常又分为两种：区块 (block) 设备档：就是一些储存数据，以提供系统存取的接口设备，简单的说就是硬盘啦！例如你的一号硬盘的代码是 /dev/hda1 等等的档案啦！第一个属性为“b “；字符 (character) 设备档：亦即是一些串行端口的接口设备，例如键盘、鼠标等等！第一个属性为“c “。

* linux 文件后缀名

对于后缀名这个概念，相信你不陌生吧。在 linux 系统中，文件的后缀名并没有具体意义，也就是说，你加或者不加，都无所谓。但是为了容易区分，linux 爱好者们都习惯给文件加一个后缀名，这样当用户看到这个文件名时就会很快想到它到底是一个什么文件。例如 1.sh, 2.tar.gz, my.cnf, test.zip 等等，如果你首次接触这些文件，你也许会感到很晕，没有关系，随着学习的深入，你就会逐渐的了解这些文件了。笔者所列举的几个文件名中 1.sh 代表它是一个 shell script ，2.tar.gz 代表它是一个压缩包，my.cnf 代表它是一个配置文件，test.zip 代表它是一个压缩文件。

另外需要你知道的是，早期 Unix 系统文件名最多允许 14 个字符，而新的 Unix 或者 linux 系统中，文件名最长可以到达 256 个字符！

1.7 库文件

1.7.1 什么是 Linux 库

在 Windows 平台和 Linux 平台下都大量存在着库。本质上来说库是一种可执行代码的二进制形式，可以被操作系统载入内存执行。由于 Windows 和 Linux 的本质不同，因此二者库的二进制是不兼容的。这里仅限于介绍 Linux 下的库。

1.7.2 库的种类

Linux 库有两种：静态库和共享库。二者的不同点在于代码被载入的时刻不同。静态库的代码在编译过程中已经被载入可执行程序，因此体积较大。共享库的代码是在可执行程序运行时才载入内存的，在编译过程中仅简单的引用，因此代码体积较小。

1.7.3 库存在的意义

库是别人写好的现有的，成熟的，可以复用的代码，你可以使用但要记得遵守许可协议。现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。共享库的好处是，不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例。

1.7.4 库文件是如何产生的

在 linux 下静态库的后缀是.a，它的产生分两步：

Step 1. 由源文件编译生成一堆.o，每个.o 里都包含这个编译单元的符号表

Step 2. ar 命令将很多.o 转换成.a，成文静态库。动态库的后缀是.so，它由 gcc 加特定参数编译产生。例如：`$ gcc -fPIC -c *.c $ gcc -shared -Wl,-soname, libfoo.so.1 -o libfoo.so.1.0 *`

1.7.5 Linux .o a .so .la .lo 的区别

- o: 编译的目标文件

- a: 静态库，其实就是把若干 o 文件打了个包
- so: 动态链接库（共享库）
- lo: 使用 libtool 编译出的目标文件，其实就是在 o 文件中添加了一些信息
- la: 使用 libtool 编译出的库文件，其实是个文本文件，记录同名动态库和静态库的相关信息

(1) libtool 的工作原理 libtool 是一个通用库支持脚本，将使用动态库的复杂性隐藏在统一、可移植的接口中；使用 libtool 的标准方法，可以在不同平台上创建并调用动态库。可以认为 libtool 是 gcc 的一个抽象，其包装了 gcc（或者其他的编译器），用户无需知道细节，只要告诉 libtool 需要编译哪些库即可，libtool 将处理库的依赖等细节。libtool 只与后缀名为 lo、la 为的 libtool 文件打交道。

libtool 主要的一个作用是在编译大型软件的过程中解决了库的依赖问题；将繁重的库依赖关系的维护工作承担下来，从而释放了程序员的人力资源。libtool 提供统一的接口，隐藏了不同平台间库的名称的差异等细节，生成一个抽象的后缀名为 la 高层库 libxx.la（其实是个文本文件），并将该库对其它库的依赖关系，都写在该 la 的文件中。该文件中的 dependency_libs 记录该库依赖的所有库（其中有些是以.la 文件的形式加入的）；libdir 则指出了库的安装位置；library_names 记录了共享库的名字；old_library 记录了静态库的名字。

当编译过程到 link 阶段的时候，如果有下面的命令：

$libtool –mode=link gcc -o myprog -rpath /usr/lib –L/usr/lib –la

libtool 会到/usr/lib 路径下去寻找 liba.la，然后从中读取实际的共享库的名字（library_names 中记录了该名字，比如 liba.so）和路径 (lib_dir 中记录了，比如 libdir=’/usr/lib’)，返回诸如/usr/lib/liba.so 的参数给激发出的 gcc 命令行。

如果 liba.so 依赖于库/usr/lib/libb.so，则在 liba.la 中将会有 dependency_libs=’-L/usr/lib -lb’或者 dependency_libs=’/usr/lib/libb.la’的行，如果是前者，其将直接把“-L/usr/lib –lb”当作参数传给 gcc 命令行；如果是后者，libtool 将从/usr/lib/libb.la 中读取实际的 libb.so 的库名称和路径，然后组合成参数“/usr/lib/libb.so”传递给 gcc 命令行。

当要生成的文件是诸如 libmylib.la 的时候，比如：

$libtool –mode=link gcc -o libmylib.la -rpath /usr/lib –L/usr/lib –la

其依赖的库的搜索基本类似，只是在这个时候会根据相应的规则生成相应的共享库和静态库。

注意：libtool 在链接的时候只会涉及到后缀名为 la 的 libtool 文件；实际的库文件名称和库安装路径以及依赖关系是从该文件中读取的。

(2) 为何使用 -Wl,–rpath-link -Wl,DIR？使用 libtool 解决编译问题看上去没什么问题：库的名称、路径、依赖都得到了很好的解决。但下结论不要那么着急，一个显而易见的问题就是：并不是所有的库都是用 libtool 编译的。

1.7.6 库文件命名规范

在 linux 下，库文件一般放在/usr/lib /lib 下，静态库的名字一般为 libxxxx.a，其中 xxxx 是该 lib 的名称。动态库的名字一般为 libxxxx.so.major.minor，xxxx 是该 lib 的名称，major 是主版本号，minor 是副版本号。

一般 libxxx.so 是一个链接文件，链接到 libxxx.so.0 等这样的文件。这样即使库版本更新了，文件名升级，调用方也不需要关心。只要重新做个软链接就行。

1.7.7 如何知道一个可执行程序依赖哪些库

ldd 命令可以查看一个可执行程序依赖的共享库，例如 # ldd /bin/lnlibc.so.6 => /lib/libc.so.6 (0Œ40021000)/lib/ld-linux.so.2 => /lib/ld- linux.so.2 (0Œ40000000) 可以看到 ln 命令依赖于 libc 库和 ld-linux 库

1.7.8 可执行程序在执行的时候如何定位共享库文件

当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径此时就需要系统动态载入器 (dynamic linker/loader) 对于 elf 格式的可执行程序，是由 ld-linux.so* 来完成的，它先后搜索 elf 文件的 `DT_RPATH` 段—环境变量 `LD_LIBRARY_PATH` —`/etc/ld.so.cache` 文件列表—/lib/,/usr/lib 目录找到库文件后将其载入内存

1.7.9 在新安装一个库之后如何让系统能够找到他

如果安装在/lib 或者/usr/lib 下，那么 ld 默认能够找到，无需其他操作。如果安装在其他目录，需要将其添加到/etc/ld.so.cache 文件中，步骤如下：

1. 编辑/etc/ld.so.conf 文件，加入库文件所在目录的路径
2. 运行 ldconfig，该命令会重建/etc/ld.so.cache 文件

1.8 Linux 编译使用链接库

以 GCC 命令参数为例。

为什么会出现 undefined reference to ’xxxxx’ 错误？首先这是链接错误，不是编译错误，也就是说如果只有这个错误，说明你的程序源码本身没有问题，是你用编译器编译时参数用得不对，没有指定链接程序要用到得库，比如你的程序里用到了一些数学函数，那么你就要在编译参数里指定程序要链接数学库，方法是在编译命令行里加入-lm。

1.8.1 -l 和-L 参数

-L 指定库的路径，-l 指定需连接的库名。

-l 参数就是用来指定程序要链接的库，-l 参数紧接着就是库名，那么库名跟真正的库文件名有什么关系呢？就拿数学库来说，他的库名是 m，他的库文件名是 libm.so，很容易看出，把库文件名的头 lib 和尾.so 去掉就是库名了。

好了现在我们知道怎么得到库名了，比如我们自已要用到一个第三方提供的库名字叫 libtest.so，那么我们只要把 libtest.so 拷贝到/usr/lib 里，编译时加上-ltest 参数，我们就能用上 libtest.so 库了（当然要用 libtest.so 库里的函数，我们还需要与 libtest.so 配套的头文件）。

放在/lib 和/usr/lib 和/usr/local/lib 里的库直接用-l 参数就能链接了，但如果库文件没放在这三个目录里，而是放在其他目录里，这时我们只用-l 参数的话，链接还是会出错，出错信息大概是："/usr/bin/ld: cannot find -lxxx"，也就是链接程序 ld 在那 3 个目录里找不到 libxxx.so。这时另外一个参数-L 就派上用场了，比如常用的 X11 的库，它放在/usr/X11R6/lib 目录下，我们编译时就要用-L/usr/X11R6/lib -lX11 参数，-L 参数跟着的是库文件所在的目录名。再比如我们把 libtest.so 放在/aaa/bbb/ccc 目录下，那链接参数就是-L/aaa/bbb/ccc -ltest 另外，大部分 libxxxx.so 只是一个链接，以 RH9 为例，比如 libm.so 它链接到/lib/libm.so.x，/lib/libm.so.6 又链接到/lib/libm-2.3.2.so，

如果没有这样的链接，还是会出错，因为 ld 只会找 libxxxx.so，所以如果你要用到 xxxx 库，而只有 libxxxx.so.x 或者 libxxxx-x.x.x.so，做一个链接就可以了 ln -s libxxxx-x.x.x.so libxxxx.so 手工来写链接参数总是很麻烦的，还好很多库开发包提供了生成链接参数的程序，名字一般叫 xxxx-config，一般放在/usr/bin 目录下，比如 gtk1.2 的链接参数生成程序是 gtk-config，执行 gtk-config –libs 就能得到以下输出"-L/usr/lib -L/usr/X11R6/lib -lgtk -lgdk -rdynamic -lgmodule -lglib -ldl -lXi -lXext -lX11 -lm"，这就是编译一个 gtk1.2 程序所需的 gtk 链接参数，xxx-config 除了–libs 参数外还有一个参数是–cflags 用来生成头文件包含目录的，也就是-I 参数，在下面我们将会讲到。你可以试试执行 gtk-config –libs –cflags，看看输出结果。现在的问题就是怎样用这些输出结果了，最笨的方法就是复制粘贴或者照抄，聪明的办法是在编译命令行里加入这个 ‘xxxx-config –libs –cflags‘，比如编译一个 gtk 程序：gcc gtktest.c ‘gtk-config –libs –cflags‘这样就差不多了。注意 ‘不是单引号，而是 1 键左边那个键。

除了 xxx-config 以外，现在新的开发包一般都用 pkg-config 来生成链接参数，使用方法跟 xxx-config 类似，但 xxx-config 是针对特定的开发包，但 pkg-config 包含很多开发包的链接参数的生成，用 pkg-config –list-all 命令可以列出所支持的所有开发包，pkg-config 的用法就是 pkg

-config pagName –libs –cflags，其中 pagName 是包名，是 pkg-config–list-all 里列出名单中的一个，比如 gtk1.2 的名字就是 gtk+，pkg-config gtk+ –libs –cflags 的作用跟 gtk-config –libs –cflags 是一样的。比如：gcc gtktest.c ‘pkg-config gtk+ –libs –cflags‘。

1.8.2 -include 和-I 参数

-include 用来包含头文件，但一般情况下包含头文件都在源码里用 #include xxxxxx 实现，-include 参数很少用。-I 参数是用来指定头文件目录，/usr/include 目录一般是不用指定的，gcc 知道去那里找，但是如果头文件不在/usr/include 里我们就要用-I 参数指定了，比如头文件放在/myinclude 目录里，那编译命令行就要加上-I/myinclude 参数了，如果不加你会得到一个"xxxx.h: No such file or directory" 的错误。-I 参数可以用相对路径，比如

头文件在当前目录，可以用-I. 来指定。上面我们提到的–cflags 参数就是用来生成-I 参数的。

1.8.3 -O 参数

这是一个程序优化参数，一般用-O2 就是，用来优化程序用的，比如 gcc test.c -O2，优化得到的程序比没优化的要小，执行速度可能也有所提高（我没有测试过）。

1.9 图形界面还是命令行界面

对于 linux 的应用，我想大多数都是用在服务器领域，对于服务器来讲真的没有必要跑一个图形界面。所以我们平时安装 linux 操作系统时往往是不安装图形界面的。说到这里也许你会有疑问，图形界面还能选择装或者不装？

是的，虽然 linux 和微软的 windows 一样同位操作系统，但是它们有一个很大的区别就是 windows 操作系统的图形界面是和内核一体的，俗称微内核，而 linux 操作系统图形界面就像一个软件一样，和内核并不是一体的。所以 linux 你可以选择不安装图形界面，这样不仅不影响服务器的正常使用还可以节省系统资源的开销，何乐而不为呢？

如果你对 linux 超级感兴趣，想使用 linux 就像使用 windows 一样，那你可以安装图形界面，可以像 windows 一样来体验 linux 也是蛮不错的。但是如果你想成为一个专业的 linux 系统工程师，那我建议你从第一天开始就不要去安装图形界面，从命令窗口开始熟悉它。

另外一点值得说的是，日常应用中，我们都是远程管理服务器的，不可能开着图形界面来让你去操作，虽然目前也有相应的工具支持远程图形连接服务器，可是那样太消耗网络带宽资源，所以从这方面来考虑还是建议你不要使用图形界面。

对于日常使用，还是图形界面方便。苹果电脑的操作系统就是 unix，界面多漂亮。当然真正发挥 linux 的作用还得用命令行界面，熟悉命令行界面才称得上是 linux 玩家吧。

1.10 路径的概念

- 路径：要到达一个目录所要经过的目录集合，如要到达`/usr/share/doc` 这个目录就要经过`/usr/share`。这很好理解，Linux 的文件系统是倒树型的文件结构，所有文件都是从根目录“/”开始的
- 绝对路径：总是从根目录开始的路径，如`/usr/share/doc`
- 相对路径：总是从当前路径开始算起的路径，如用户的当前目录为`/usr`，那么表示和`/usr/share/doc` 一样的目录的相对路径为`/share/doc`

绝对路径和相对路径的本质区别就是计算的起点不同，一个总是从根目录开始算起，而另一个则总是从当前目录算起。

1.10.1 命令查找顺序

。现在我们知道系统里面其实有不少的 ls 命令，或者是包括内建的 echo 命令，那么来想一想，如果一个命令 (例如 ls) 被下达时，到底是哪一个 ls 被拿来运行？很有趣吧！基本上，命令运行的顺序可以这样看：

以相对/绝对路径运行命令，例如『/bin/ls 』或『./ls 』；由 alias 找到该命令来运行；由 bash 内建的 (builtin) 命令来运行；透过 $PATH 这个变量的顺序搜寻到的第一个命令来运行。举例来说，你可以下达 /bin/ls 及单纯的 ls 看看，会发现使用 ls 有颜色但是 /bin/ls 则没有颜色。因为 /bin/ls 是直接取用该命令来下达，而 ls 会因为『alias ls='ls –color=tty' 』这个命令别名而先使用！如果想要了解命令搜寻的顺序，其实透过 type -a ls 也可以查询的到啦！上述的顺序最好先了解喔！

1.11 环境变量

Linux 环境变量能帮助提升 Linux Shell 体验。很多程序和脚本都是通过环境变量来获取系统信息、存储临时数据和配置信息。

1.11.1 什么是环境变量

Bash Shell 用一个称作环境变量（environment variables）的特性来存储有关 Shell 会话和工作环境的信息。这也是它们为什么被称作环境变量的原因。环境变量不仅对 Shell 会话可见，对所有 Shell 创建的子进程也可见。自定义变量则只对创建它们的 Shell 可见。这让环境变量对那些子进程中需要获得父进程信息的程序来说非常有用。举例来说，能不能在任何目录下运行某个命令，与 PATH 这个变量有很大的关系。例如运行 ls 这个命令时，系统就是透过 PATH 这个变量里面所记录的路径顺序来搜寻命令。如果在搜寻完 PATH 变量内的路径还找不到 ls 这个命令时，就会在屏幕上显示 “command not found ” 的错误信息。

如果说的专业一点，那么由于在 Linux System 下面，所有的线程都是需要一个运行码，而就如 “真正以 Shell 来跟 Linux 沟通，是在正确的登陆 Linux 之后！” 这句话说的，这个时候你就有一个 Bash 的运行程序，也才可以真正的经由 Bash 来跟系统沟通。而在进入 Shell 之前，由于系统需要一些变量来提供他数据的存取 (或者是一些环境的配置参数值，例如是否要显示彩色等等的)，所以就有一些所谓的 “环境变量” 需要来读入系统中了。这些环境变量例如 PATH、HOME、MAIL、SHELL 等等，都是很重要的。为了区别与自定义变量的不同，环境变量一律使用全大写字母来表示。

为什么环境变量的数据可以被子程序所引用呢？这是因为内存配置的关系！理论上是这样的：

当启动一个 shell，操作系统会分配一记忆区块给 shell 使用，此内存内之变量可让子程序取用若在父程序利用 export 功能，可以让自定义变量的内容写到上述的记忆区块当中 (环境变量)；当加载另一个 shell 时 (亦即启动子程序，而离开原本的父程序了)，子 shell 可以将父 shell 的环境变量所在的记忆区块导入自己的环境变量区块当中。透过这样

的关系，我们就可以让某些变量在相关的程序之间存在，以帮助自己更方便的操作环境喔！不过要提醒的是，这个『环境变量』与『bash 的操作环境』意思不太一样，举例来说，PS1 并不是环境变量，但是这个 PS1 会影响到 bash 的接口 (提示字符嘛)！相关性要厘清喔！

在不同的地方，或许说的是全局变量和局部变量。基本上可以认为“环境变量 = 全局变量”、“自定义变量 = 局部变量”。

1.11.2 查看环境变量——env

用 env 可以查看环境变量，env 就是 environment 的缩写。下面列出一些比较重要的环境变量：

- HOME
  代表用户的家目录。使用 `cd ~` 去到自己的家目录或者直接敲 cd 不接任何路径回到用户家目录，就是取用这个变量。有很多程序都可能会取用到这个变量的值。
- SHELL
  设置目前这个环境使用哪种 Shell，Linux 默认使用 /bin/bash 。
- HISTSIZE
  设置历史命令记录的条数。
- PATH
  设置运行文件搜寻的路径，目录与目录中间以冒号 (:) 分隔，由于文件的搜寻是依序由 PATH 的变量内的目录来查询，所以，目录的顺序也是重要。
- LANG
  设置语系数据。很多信息都会用到，举例来说，当我们在启动某些 perl 的程序语言文件时，它会主动的去分析语系数据文件，如果发现有它无法解析的编码语系，可能会产生错误。一般来说，我们中文编码通常是 zh_TW.Big5 或者是 zh_TW.UTF-8，这两个编码偏偏不容易被解译出来，所以，有的时候，可能需要修订一下语系数据。
- RANDOM
  这是随机数的变量。目前大多数的发行版都会有随机数生成器，那就是 /dev/random 这个文件。我们可以透过这个随机数文件相关的变量 ($RANDOM) 来随机取得随机数值。在 BASH 的环境下，这个 RANDOM 变量的内容，介于 0 ~ 32767 之间 ($2^{15} = 32768$)，所以，只要 echo $RANDOM 时，系统就会主动的随机取出一个介于 0 ~ 32767 的数值。如果想要使用 0 ~ 9 之间的数值，可以利用 declare 宣告数值类型，然后这样做就可以了：

  ```
  declare -i number=$RANDOM*10/32767 ; echo $number
  ```

env 命令还有一个常用的功能是给脚本指定解释程序，如：

```
#!/usr/bin/env python
```

由于在 Ubuntu 中 Shell 输入 python 默认调用的是 python2.*，所以这是指定解释器为 python2.* 类版本。又如

```
#!/usr/bin/env python3
```

由于 python 存在 python2.* 和 python3.* 两类版本而不兼容，所有这是指定解释器为 python3.* 类版本。

“#!” 是脚本中的约定符号，用来指定解释器的版本。

1.11.3 查看所有变量——set

Bash 不只有环境变量，还有一些与 Bash 操作接口有关的变量，以及用户自己定义的变量存在的。要查看这些变量可以使用 set 命令。set 除了环境变量之外，还会将其他在 Bash 内的变量通通显示出来。一般来说，不论是否为环境变量，只要跟我们目前这个 Shell 的操作接口有关的变量，通常都会用大写字母表示。也就是说，基本上，在 Linux 默认的情况中，使用大写的字母来表示的变量一般为系统内定需要的变量。下面说下这些变量中比较重要的几个：

- PS1：提示字符的配置

  这是 PS1 (数字的 1 不是英文字母)，这个东西就是命令提示字符。当我们每次按下 [Enter] 按键去运行某个命令后，最后要再次出现提示字符时，就会主动去读取这个变量值。上头 PS1 内显示的是一些特殊符号，这些特殊符号可以显示不同的信息，每个发行版的 Bash 默认的 PS1 变量内容可能有些许的差异，不要紧，” 习惯你自己的习惯 “就好了。可以用 man bash 去查询一下 PS1 的相关说明，以理解底下的一些符号意义。

  - `\d`： 可显示出 “星期月日” 的日期格式，如：“Mon Feb 2”
  - `\H`： 完整的主机名
  - `\h`： 仅取主机名在第一个小数点之前的名字
  - `\t`： 显示时间，为 24 小时格式的 “HH:MM:SS”
  - `\T`： 显示时间，为 12 小时格式的 “HH:MM:SS”
  - `\A`： 显示时间，为 24 小时格式的 “HH:MM”
  - `\@`： 显示时间，为 12 小时格式的 “am/pm” 样式
  - `\u` ： 目前使用者的账号名称，如 “root”
  - `\v`： BASH 的版本信息，如版本为 3.2.25(1)，仅取 “3.2” 显示
  - `\w`： 完整的工作目录名称，由根目录写起的目录名称。但家目录会以 ~ 取代
  - `\W`： 利用 basename 函数取得工作目录名称，所以仅会列出最后一个目录名
  - `\#`： 下达的第几个命令
  - `\$`： 提示字符，如果是 root 时，提示字符为 # ，否则就是 $

- $：关于本 shell 的 PID

  $ 本身也是个变量。它代表的是 “目前这个 Shell 的线程代号”，即是所谓的 PID (Process ID)。想要知道我们的 Shell 的 PID ，就可以用：`echo $$`，出现的数字就是你的 PID 号码。

- ?：关于上个运行命令的回传值

  问号也是一个特殊的变量。在 Bash 里面这个变量很重要。这个变量是：“上一个运行的命令所回传的值”，上面这句话的重点是 “上一个命令” 与 “回传值” 两

个地方。当我们运行某些命令时，这些命令都会回传一个运行后的代码。一般来说，如果成功的运行该命令，则会回传一个 0 值，如果运行过程发生错误，就会回传“错误代码”，一般就是以非为 0 的数值来取代。

1.11.4 设置环境变量——export

当你登陆 Linux 并取得一个 bash 之后，你的 bash 就是一个独立的程序，被称为 PID 的就是。接下来你在这个 bash 底下所下达的任何命令都是由这个 bash 所衍生出来的，那些被下达的命令就被称为子程序了。因为子程序仅会继承父程序的环境变量，子程序不会继承父程序的自定义变量啦！所以你在原本 bash 的自定义变量在进入了子程序后就会消失不见，一直到你离开子程序并回到原本的父程序后，这个变量才会又出现！

换个角度来想，也就是说，如果我能将自定义变量变成环境变量的话，那不就可以让该变量值继续存在于子程序了？呵呵！没错！此时，那个 export 命令就很有用啦！如你想要让该变量内容继续的在子程序中使用，那么就请运行：

`export` 变量名称

如果仅下达 export 而没有接变量时，那么此时将会把所有的『环境变量』秀出来喔！

设置环境变量的方法是先创建一个自定义变量然后再把它导入到环境变量中。如下所示：

```
$ test=''testing a long string''
$ echo $test
testing a long string
$ exprot test
$ bash
$ echo $test
tesing a long string
$
```

导出自定义变量 test 后，启动了子 Shell 进程在子 Shell 中查看了 test 环境变量的值。这次，因为 export 命令让它变成了环境变量，环境变量保持了它的值。注意在导出自定义变量时，不能用美元符 $ 去引用变量名。

永久性的设置可以采用 Shell 的配置文件来设置。

1.11.5 设置 PATH 环境变量

PATH 环境变量是 Linux 系统上造成最多问题的变量。它定义了命令行输入命令的搜索路径。如果找不到命令，它会产生“command not found”的错误。

问题是通常应用会把可执行程序放到不在 PATH 环境变量中的目录。解决办法是保证 PATH 环境变量包含了所有存放应用的目录。可以添加新的搜索目录到现有的 PATH 环境变量，无需从头定义。PATH 中的目录之间是用冒号分隔的，所以只需引用原来的 PATH 值，然后再给字符添加新目录就可以了。通常将单点符加到 PATH 环境变量中，这个单点符代表当前目录，即：`PATH=$PATH`。

永久性的设置可以采用 Shell 的配置文件来设置。

1.11.6 取消环境变量

参见 declare 命令。

1.12 用户和组

1.12.1 用户和组的概念

Linux 是一个地道的多用户操作系统，可以添加 $2^{32}-1$ 个用户。对用户这个概念的理解还不能仅限于字面含义。在 Linux 系统中还有一些用户是用来完成特定任务的，比如 nobody、admin、ftp 等。需要注意的是，在 Linux 中不管用户名有多么 NB，只要不是 root，它就一定是普通用户，权利大小都是相同的。很多资料上说的所谓用户角色的那些概念，尤其是说角色不同权限不同的观点不是十分精准，很容易造成 Linux 用户有权限大小的印象。其实所谓的权限，则是不同的用户所能访问的文件不同产生的一种假象。而这种假象又不是单独一个用户的概念能够决定的，还要用到用户组的概念。

用户组是对某项功能具有相同权限的用户的集合。比较巧合的是用户名和用户组名相同，一个用户可以属于多个用户组，而且一个用户至少应该属于一个用户组。用户组能够在更高的层面来抽象用户所能够访问文件的数量，所以结合不同的用户组，就能够给每个用户构建出独一无二的文件访问列表，而且不同的用户之间还能做到交错纵横，有条不紊。在用户之间的协作上和隐私的保护上达到了一个良好的平衡。这才是用户角色存在的真实目的所在。

虽然用户角色不能跟权限靠上关系，但不同的角色有些时候还是有差别的，即是否拥有密码、home 目录以及 Shell[1]这些资源。有用户的主要任务就是运行某些服务程序以确保安全性，比如 nobody 用户就可以用于 Nginx[2]的工作进程。对于这类的用户一般是不分配密码和 Shell 的，甚至 home 目录也没有。

[1]严格说还是分配 Shell 了的，只是分配的是/sbin/nologin 这个特殊的 Shell，从其命名上看就是阻止你登录，实际上就是没由任何功能的 Shell（也可能显示几行声明文字）

[2]Nginx 是一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP/POP3/SMTP 代理服务器。它以极强的稳定性、丰富的功能集、极低的系统资源消耗、灵活多变的配置文件而闻名

1.12.2 /etc/passwd 文件

/etc/passwd 文件，根目录 “/” 下的 “etc” 文件夹下的 passwd 文件，从文件名上看，应该是跟密码有关。的确是这样，只是已经成为历史了。现在保存密码的文件是/etc/shadow，也就是传说中的影子文件。

/etc/passwd 文件是用来存放用户基本信息的。例如，我现在使用的系统中这个文件的内容如下：

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuuid:x:100:101::/var/lib/libuuid:
syslog:x:101:104::/home/syslog:/bin/false
messagebus:x:102:106::/var/run/dbus:/bin/false
... ...
hplip:x:114:7:HPLIP system user,,,:/var/run/hplip:/bin/false
pulse:x:115:122:PulseAudio daemon,,,:/var/run/pulse:/bin/false
phileas:x:1000:1000:phileas,,,:/home/phileas:/bin/bash
```

文件的每一个行都代表一个用户。换句话说，这个文件有几行，系统中就有个用户。每行由冒号 “:” 分割成 7 个字段，其结构如下：

用户名：密码：用户 ID：组 ID：用户全名：home 目录：Shell

“密码” 字段，虽然是这么叫的，但是没有密码放在这里，有的只是一个字母 “x”。这个文件比较重要的是用户 UID(User ID) 和 GID(Group ID)，无论是用户还是用户组，Linux 只靠它们来识别。为了区别不同的用户或用户组，需要保持它们自身在系统中的唯一性。

UID 和 GID 是可以相同的，因为它们代表的是不同的概念。都为 0 的 UID 和 GID 是 Linux 系统中比较特殊的，它们分配给 root 用户和 root 用户组。虽然系统只区别 0 和非零的 UID 或 GID，但是在使用习惯上还是进行了一些分段。0 自然不用说，就是给 root 的；1 ~ 499 是属于系统用户的，比如前面提到的那些待遇比较差的和一些甚至是占坑的就属于这类；500 ~ 4294967295 是分配给普通用户的。

为了看起来方便，完整的 passwd 文件整理为表格如下：

用户名	密码	用户 ID	组 ID	用户全名	home 目录	Shell
root	x	0	0	root	/root	/bin/bash
daemon	x	1	1	daemon	/usr/sbin	/usr/sbin/nologin
bin	x	2	2	bin	/bin	/usr/sbin/nologin
sys	x	3	3	sys	/dev	/usr/sbin/nologin
sync	x	4	65534	sync	/bin	/bin/sync
games	x	5	60	games	/usr/games	/usr/sbin/nologin
man	x	6	12	man	/var/cache/man	/usr/sbin/nologin
lp	x	7	7	lp	/var/spool/lpd	/usr/sbin/nologin
mail	x	8	8	mail	/var/mail	/usr/sbin/nologin
news	x	9	9	news	/var/spool/news	/usr/sbin/nologin
uucp	x	10	10	uucp	/var/spool/uucp	/usr/sbin/nologin
proxy	x	13	13	proxy	/bin	/usr/sbin/nologin
www-data	x	33	33	www-data	/var/www	/usr/sbin/nologin
backup	x	34	34	backup	/var/backups	/usr/sbin/nologin
list	x	38	38	Mailing List Manager	/var/list	/usr/sbin/nologin
irc	x	39	39	ircd	/var/run/ircd	/usr/sbin/nologin
gnats	x	41	41	Gnats Bug-Reporting System (admin)	/var/lib/gnats	/usr/sbin/nologin
nobody	x	65534	65534	nobody	/nonexistent	/usr/sbin/nologin
libuuid	x	100	101		/var/lib/libuuid	
syslog	x	101	104		/home/syslog	/bin/false
messagebus	x	102	106		/var/run/dbus	/bin/false
usbmux	x	103	46	usbmux daemon,,,	/home/usbmux	/bin/false
dnsmasq	x	104	65534	dnsmasq,,,	/var/lib/misc	/bin/false
avahi-autoipd	x	105	113	Avahi autoip daemon,,,	/var/lib/avahi-autoipd	/bin/false
kernoops	x	106	65534	Kernel Oops Tracking Daemon,,,	/	/bin/false
rtkit	x	107	114	RealtimeKit,,,	/proc	/bin/false
saned	x	108	115		/home/saned	/bin/false
whoopsie	x	109	116		/nonexistent	/bin/false
speech-dispatcher	x	110	29	Speech Dispatcher,,,	/var/run/speech-dispatcher	/bin/sh
avahi	x	111	117	Avahi mDNS daemon,,,	/var/run/avahi-daemon	/bin/false
lightdm	x	112	118	Light Display Manager	/var/lib/lightdm	/bin/false
colord	x	113	121	colord colour management daemon,,,	/var/lib/colord	/bin/false
hplip	x	114	7	HPLIP system user,,,	/var/run/hplip	/bin/false
pulse	x	115	122	PulseAudio daemon,,,	/var/run/pulse	/bin/false
phileas	x	1000	1000	phileas,,,	/home/phileas	/bin/bash

1.12.3 /etc/group 文件

/etc/group 文件存放的是用户组的基本信息，是/etc/passwd 中用户 ID 的来源。这个文件的部分内容如下：

```
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,xuanleng
tty:x:5:
disk:x:6:
lp:x:7:
... ...
```

```
utempter:x:112:
avahi-autoipd:x:113:
rtkit:x:114:
saned:x:115:
whoopsie:x:116:
avahi:x:117:
lightdm:x:118:
nopasswdlogin:x:119:
bluetooth:x:120:
colord:x:121:
pulse:x:122:
pulse-access:x:123:
xuanleng:x:1000:
sambashare:x:124:xuanleng
bumblebee:x:999:syslog,xuanleng
```

与/etc/passwd 文件类似，同样是每行代表一个用户组，只是这个文件比较简单，被冒号“:”分割成了四个字段。行结构如下所示：

用户组名：用户组密码：用户组 ID：用户组内的用户名

从/etc/group 文件的结构可以看出，不但用户有密码，用户组也是由密码的。用户组密码是给用户管理员用的，不过目前很少要配置用户组管理员。与/etc/passwd 一样，组密码也是储存在影子文件中，/etc/group 的影子文件是/etc/gshadow。

需要注意的是很多用户组并没有使用“用户组内的用户名”这个字段，而使用这个字段的，则会使用逗号“,”来分隔每个用户名。一个用户可以加入多个用户组，只要用户加入了这个用户组，则用户的名字就会出现在这个字段里。

为了看起来方便，完整的 group 文件整理为表格如下：

用户组名	用户组密码	用户组 ID	用户组内的用户名
root	x	0	
daemon	x	1	
bin	x	2	
sys	x	3	
adm	x	4	syslog,xuanleng
tty	x	5	
disk	x	6	
lp	x	7	
mail	x	8	
news	x	9	
uucp	x	10	
man	x	12	
proxy	x	13	
kmem	x	15	
dialout	x	20	
fax	x	21	
voice	x	22	
cdrom	x	24	xuanleng
floppy	x	25	
tape	x	26	
sudo	x	27	xuanleng
audio	x	29	pulse
dip	x	30	xuanleng
www-data	x	33	
backup	x	34	
operator	x	37	
list	x	38	
irc	x	39	
src	x	40	
gnats	x	41	
shadow	x	42	
utmp	x	43	
video	x	44	
sasl	x	45	
plugdev	x	46	xuanleng
staff	x	50	
games	x	60	
users	x	100	
nogroup	x	65534	
libuuid	x	101	
netdev	x	102	
crontab	x	103	
syslog	x	104	
fuse	x	105	
messagebus	x	106	
ssl-cert	x	107	
lpadmin	x	108	xuanleng
scanner	x	109	saned
mlocate	x	110	
ssh	x	111	
utempter	x	112	
avahi-autoipd	x	113	
rtkit	x	114	
saned	x	115	
whoopsie	x	116	
avahi	x	117	
lightdm	x	118	
nopasswdlogin	x	119	
bluetooth	x	120	
colord	x	121	
pulse	x	122	
pulse-access	x	123	
xuanleng	x	1000	
sambashare	x	124	xuanleng
bumblebee	x	999	syslog,xuanleng

1.13 一些注意

1. Linux 系统比 windows 精准，他区分大小写。
2. 注意命令中的空格
3. 在终端中输入密码时，密码是不回显的，就是说，密码不会被显示成“*”。输入完密码直接按回车就好了。
4. 直接在命令行中输入 cd 命令而不加任何参数，可以马上回到用户的主目录（home）

1.14 名词解释

1. GUI：Graphics User Interface 图形用户接口
2. CLI： command line interface 命令行接口

第二章 Linux 命令

2.1 命令行环境及命令格式

（1）命令行环境的含义

例如：`phileas@phileas-Aspire:~/Downloads$`

`phileas` 是用户名，`phileas-Aspire` 是主机名，`~/Downloads` 是当前目录。对普通用户来说，提示符显示为“`$`”，管理员 root 用户提示符为“`#`”。

（2）Linux 的命令格式

`$命令名␣[-选项]␣操作对象`

需要注意的是，命令名、选项或者操作对象之间都是至少要有一个空格，多个空格没有关系。选项和操作对象可以有多个，尤其是选项不是必需的，但也可以有多个。当有多个选项时，可以写在一起，前面通常要有一个“-”。操作对象也可以有 0 个、1 个或多个。

由于 Linux 基本是 C 语言编写的，所以无论命令名或选项，Linux 都是严格区分大小写。

2.1.1 通配符

在 bash 下，可以使用 * 来匹配零个或多个字符，而用？匹配一个字符。

2.1.2 管道符

管道符“|"，就是把前面的命令运行的结果丢给后面的命令。

2.1.3 作业控制

当运行一个进程时，你可以使它暂停（按 Ctrl+z），然后使用 fg 命令恢复它，利用 bg 命令使他到后台运行，你也可以使它终止（按 Ctrl+c）。

2.2 获得帮助的命令

2.2.1 help

Linux 命令可以分为内部命令和外部命令，所谓内部命令就是由 Linux 默认 Shell-bash 提供的命令，而非 bash 提供的命令就是外部命令。

对于内部命令，大家可以使用 help 命令来获得帮助，例如要获得 cd 命令的帮助可以使用如下命令来查询：

`$help␣cd`

其实，日常使用中大家碰到的绝大多数命令都是外部命令，所以不必刻意区分内部命令还是外部命令，只需要大家有这个概念就好。对于外部命令浏览其帮助文档，则用 man 命令。

2.2.2 man

man 命令[1]可以向用户提供快速的在线帮助，Linux 包括了一套完整的 man 和 man 帮助文档。这些 man 帮助文档涉及 Linux 系统的命令、程序、配置文件和程序库的功能等说明。一直以来，这些文档都是某些操作系统书籍的主要资料来源，自从有了互联网和图形用户界面后，直接使用这些文档的人变少了，但它仍是很重要的信息来源。操作系统安装好后，这些文档就可以马上使用。

man 的帮助文件有很多个，通常以章分类，存放在`/usr/share/man` 目录中，每章讨论一项专题，分为 man1、man2、man3 等子目录。章的划分符合 AT&T Unix 的文档结构，继承了其方便使用的特点，并且保持向后兼容（Unix 经常保持向后兼容）。大多数的用户命令可以在 man1 中找到，如果要使用其他手册，请在 man 和所查找命令之间加上具体手册的编号，如：

`$man␣password`

`$man␣5␣password`

第一个 man 命令查找的是 passwd 命令的 manpages，而第二个 man 命令查找的则是/ect/passwd 文件的 manpages，其中 5 表示配置文件手册类型，而非默认的可执行程序类型。

2.3 文件和目录管理命令

2.3.1 cd——改变当前路径的命令

`$cd␣`路径

特殊目录的表示方法及含义

- . 代表当前目录
- .. 代表上层目录
- ~ 代表当前登录用户的宿主目录
- ~用户名 代表进入~ 后用户的宿主目录
- - 代表前一目录，即进入当前目录之前操作的目录
- **直接在命令行中输入 cd 命令而不加任何参数，可以马上回到用户的主目录（home）**

2.3.2 ls——列出文件清单命令

ls 命令能够显示目录内容和查看文件详细信息，ls 意为 “list directory contents”。ls 命令的执行方式为：

[1]manual 的缩写，linux 中的命令很喜欢用缩写

`$ls␣-[选项]␣[文件名或者目录名]`

ls 命令选项：

- -l 列出所要查看的内容详细信息，不但包括文件名，还包括文件大小、访问权限和所有者等信息。具体说明详见 Linux 文件属性。
- -a 列出所有文件，包括隐藏文件也显示出来
- -R 列出当前目录下的所有内容，并且将子目录的内容也一起列出来
- -d 仅列出目录本身，而不显示当前目录下的内容

如果不加任何选项的话，ls 命令仅列出当前目录下的文件和目录名。

- 查看当前目录下的目录个数：`ls -l |grep "^d"|wc -l`

2.3.3 pwd——查看当前路径命令

pwd，Print Working Directory 的缩写。

如果有时在操作过程中忘记了当前的路径，则可以通过次命令来查看路径。

除了 pwd 命令，还有一个很有用的命令，那就是 tree 命令，它以更为直观的树状图的形式显示目录之间的关系。但是 Ubuntu 默认没有安装 tree，需要手动安装，具体安装方法如下：

`$sudo␣aptitude␣install␣tree`

2.3.4 mkdir——创建目录命令

`$mkdir␣目录名`

对于一次性建立多级目录的情况，可以使用-p 参数：

`$mkdir␣-p␣/home/dir1/dir2/dir3`

2.3.5 rmdir——删除空目录命令

`$␣rmdir␣目录名`

例如要删除目录/home/hxl/directory 这个空目录，则可以执行命令：

`$rmdir␣/home/hxl/directory`

注意：rmdir 可以删除的是非空目录，被删的目录下不能有文件或子目录。如果只是有子目录存在，还可以用-p 来删除；如果含有文件，那么-p 选项也无能为力了。

2.3.6 mv——移动或更名现有的文件或目录

linux 下重命名文件或文件夹的命令 mv 既可以重命名，又可以移动文件或文件夹.

例子：将目录 A 重命名为 B

mv A B

例子：将/a 目录移动到/b 下，并重命名为 c

mv /a /b/c

其实在文本模式中要重命名文件或目录的话也是很简单的，我们只需要使用 mv 命令就可以了，比如说我们要将一个名为 abc 的文件重命名为 1234 就可以这样来写：mv abc 1234，但是要注意的是，如果当前目录下也有个 1234 的文件的话，我们的这个文件是会将它覆盖的

2.3.7 rm——删除文件（目录）命令

rm 命令能够删除一个文件或者目录。

$rm␣[-选项]␣文件名或者目录名

对于 Ubuntu 来说，这个命令是比较危险的一个命令，因为一旦使用了这个命令删除的文件如果再进行磁盘写操作后将无法再恢复。在其他版本的一些 Linux 中，默认情况下给rm 命令加上一个-i 选项，可以在删除前对操作进行确认。

rm 命令选项：

- -f：强制删除（即使设置了-i 属性也不起作用）
- -r：删除目录
- -i：删除文件或目录前是否询问

2.3.8 cp——复制命令

cp 命令能够复制一个文件或者生成一个不同名但是内容相同的文件。

- 复制文件：$␣cp␣源文件名␣目标路径

  $␣cp␣/etc/passwd␣/home/hxl/：将目录/etc 下的passwd 文件复制到/home/hxl 目录中。
- 当前目录下生成不同名但内容相同的文件：$␣cp␣源文件名␣目标文件名

  $␣cp␣passwd␣passwd_new
- 复制目录时要加上选项-r，例如在当前目录下复制目录 a，并重命名为 b:

  $␣cp␣-r␣a␣b

2.4 查看命令

2.4.1 man——查看命令手册

man：manual 的缩写，用来查看命令的使用手册，如：man␣␣cp，了解命令用法的很好用的命令。

2.4.2 file——检测文件类型

file 是检测文件类型[2]的命令。file 命令的简单用法就是：file 文件名，例如：

file␣data.txt

data.txt: ASCII text

[2]文件类型就文件组织的方式，通常不同的文件类型执行不同的标准。例如我们熟知的：txt , doc , xls , pdf ...

就告诉我们，data.txt 是一个 text （即 txt) 类型的文件。txt 文件所采用的编码是 ascii 编码体系。所以 text 是文件类型；ASCII 是编码体系。

又如：

`file␣my.pdf`

my.pdf: PDF document, version 1.5

“PDF document” 告诉我们, my.pdf 是 pdf 类型的文件。版本执行的标准是：1.5 。

像 ascii , version 1.5 这些都是与文件类型密切相关的信息。如果需要更详细的信息，可以加参数：如：`file␣-i␣data.txt`。可以使用 man file 查看详细用法。

2.5 文本处理命令

2.5.1 awk

2.5.1.1 awk 常用内置变量

2.5.1.2 awk 统计命令

1、求和

```
cat data|awk '{sum+=$1} END {print "Sum = ", sum}'
```

例如：

```
xuanleng@xuanleng-Alienware-15:~/Desktop$ cat 2dspectra_xyz.ods|awk '{sum+=$1} END {print "Sum = ", sum}'
Sum =  4.79883e+06
```

2、求平均

```
cat data|awk '{sum+=$1} END {print "Average = ", sum/NR}'
```

3、求最大值

```
cat data|awk 'BEGIN {max = 0} {if ($1>max) max=$1 fi} END {print "Max=", max}'
```

4、求最小值（min 的初始值设置一个超大数即可）

```
cat data|awk 'BEGIN {min = 1999999} {if ($1<min) min=$1 fi} END {print "Min=", min}'
```

2.5.2 grep

grep，可以在多个文本文件中查找指定的字符串，如：在/home/phileas/Desktop 目录下搜索带字符串‘magic’的文件

`grep magic /home/phileas/Desktop/*`

其中 “*" 是通配符。

当然也可以搜索单个文件，如：`grep␣'BCL␣A'␣3BSD.pdb`

用于搜索的特殊符号：

- \< 和 \> 分别标注单词的开始与结尾。例如：
  `grep 'man'` ：会匹配‘Batman’、‘manic’、‘man’等’；
  `grep '\<man'` ：匹配‘manic’和‘man’，但不是‘Batman’；
  `grep '\<man\>'` ：只匹配‘man’，而不是‘Batman’或‘manic’等其他的字符串。
- '^'：指匹配的字符串在行首；
- '$'：指匹配的字符串在行尾。

`grep␣'BCL␣A'␣3BSD.pdb␣|␣grep␣"^HETATM"␣|␣grep␣NB␣|␣tr␣-s␣'␣'␣'␣'␣|␣cut␣-d'␣'␣-f7,8`

命令说明：

- ␣：表示空格；
- `grep␣'BCL␣A'␣3BSD.pdb`：意思是提取出文件 3BSD.pdb 中包含字符 BCL A 的内容；
- |：管道符，将之前命令的结果传给后面的命令；
- `grep␣"^HETATM"`：'^'，指匹配的字符串在行首；意思是提取出首行含有字符 HETATM 的内容；
- `grep␣NB`：提取出含有字符 NB 的内容；
- ␣`tr␣-s␣'␣'␣'␣'␣`cut -d' ' -f7,8,9|：删除字符和空格，留下所需数据；
- >NB.ods：>，重定向命令；将最终结果输入到 NB.ods 文件中。

2.5.3 tr

通过使用 tr，可以非常容易地实现 sed 的许多最基本功能。可以将 tr 看作为 sed 的（极其）简化的变体：它可以用一个字符来替换另一个字符，或者可以完全除去一些字符。也可以用它来除去重复字符。这就是所有 tr 所能够做的。

tr 用来从标准输入中通过替换或删除操作进行字符转换。tr 主要用于删除文件中控制字符或进行字符转换。使用 tr 时要转换两个字符串：字符串 1 用于查询，字符串 2 用于处理各种转换。tr 刚执行时，字符串 1 中的字符被映射到字符串 2 中的字符，然后转换操作开始。

（1）最常用选项的 tr 命令格式

`tr -c -d -s ["string1_to_translate_from"] ["string2_to_translate_to"] < input-file`

这里：

- -c 用字符串 1 中字符集的补集替换此字符集，要求字符集为 ASCII。

- -d 删除字符串 1 中所有输入字符。
- -s 删除所有重复出现字符序列，只保留第一个。即将重复出现字符串压缩为一个字符串。
- input-file 是转换文件名。虽然可以使用其他格式输入，但这种格式最常用。

（2）字符范围

指定字符串 1 或字符串 2 的内容时，只能使用单字符或字符串范围或列表。

a-z a-z 内的字符组成的字符串；

A-Z A-Z 内的字符组成的字符串；

0-9 数字串；

- `\octal` 一个三位的八进制数，对应有效的 ASCII 字符；

O*n 表示字符 O 重复出现指定次数 n，因此 [O*2] 匹配 OO 的字符串。

（3）tr 中特定控制字符的不同表达方式

速记符含义八进制方式

- `\a Ctrl-G` 铃声`\007`
- `\b Ctrl-H` 退格符`\010`
- `\f Ctrl-L` 走行换页`\014`
- `\n Ctrl-J` 新行`\012`
- `\r Ctrl-M` 回车`\015`
- `\t Ctrl-I` tab键`\011`
- `\v Ctrl-X` `\030`

（4）实例

1. 将文件 file 中出现的“abc”替换为“xyz”

   `cat file | tr "abc" "xyz" > new_file` **注意：**这里，凡是在 file 中出现的“a”字母，都替换成“x”字母，“b”字母替换为“y”字母，“c”字母替换为“z”字母。而不是将字符串“abc”替换为字符串“xyz”。
2. 使用 tr 命令“统一”字母大小写

   小写 –> 大写：`cat file | tr [a-z] [A-Z] > new_file`

   大写 –> 小写：`cat file | tr [A-Z] [a-z] > new_file`
3. 把文件中的数字 0-9 替换为 a-j

   `cat file | tr [0-9] [a-j] > new_file`
4. 删除文件 file 中出现的“Snail”字符

   `cat file | tr -d "Snail" > new_file` 注意：这里，凡是在 file 文件中出现的’S’,’n’,’a’,’i’,’l’字符都会被删除！而不是紧紧删除出现的"Snail”字符串。
5. 删除文件 file 中出现的换行“`\n`”、制表“`\t`”字符

   `cat file | tr -d "\n\t" > new_file` 不可见字符都得用转义字符来表示的，这个都是统一的。
6. 删除“连续着的”重复字母，只保留第一个

   `cat file | tr -s [a-zA-Z] > new_file`

7. 删除空行

   `cat file | tr -s "\n" > new_file`

8. 删除 Windows 文件“造成”的“^M”字符

   `cat file | tr -d "\r" > new_file` 或者 `cat file | tr -s "\r" "\n" > new_file`

   注意：这里-s 后面是两个参数“\r”和“\n”，用后者替换前者

9. 用空格符\040 替换制表符\011

   `cat file | tr -s "\011" "\040" > new_file`

10. 把路径变量中的冒号“:”，替换成换行符"\n"

    `echo $PATH | tr -s ":" "\n"`

2.5.4 cut

cut 是一个选取命令，就是将一段数据经过分析，取出我们想要的。一般来说，选取信息通常是针对“行”来进行分析的，并不是整篇信息分析的。

其语法格式为：cut [-bn] [file] 或 cut [-c] [file] 或 cut [-df] [file]

cut 命令从文件的每一行剪切字节、字符和字段并将这些字节、字符和字段写至标准输出。如果不指定 File 参数，cut 命令将读取标准输入。必须指定 -b、-c 或 -f 标志之一。

主要参数

- -b： 以字节（bytes）为单位进行分割。这些字节位置将忽略多字节字符边界，除非也指定了 -n 标志。
- -c： 以字符（characters）为单位进行分割。
- -d： 自定义分隔符，默认为制表符。
- -f： 与-d 一起使用，指定显示哪个区域（fields）。
- -n： 取消分割多字节字符。仅和 -b 标志一起使用。如果字符的最后一个字节落在由 -b 标志的 List 参数指示的
 范围之内，该字符将被写出；否则，该字符将被排除。

例如：

```
cut -d' ' -f7-9 a.txt
```

表示以空格为分隔符剪切出 a.txt 的第 7、8、9 列。

2.5.5 wc

语法：

`wc␣[-clw]␣[--help]␣[--version]␣[文件……]`

参数：

- `-c` 或`--bytes` 或`--chars` 只显示 Bytes 数。
- `-l` 或`--lines` 只显示列数。
- `-w` 或`--words` 只显示字数。

- --help 在线帮助
- --version 显示版本信息。

实例：

在默认的情况下，wc 将计算指定文件的行数、字数，以及字节数。使用的命令为：

wc testfile

运行结果例如：

```
3 92 598 testfile        # testfile文件的行数为3、单词数92、字节数598
```

也可以同时统计多个文件，如

```
$ wc testfile testfile_1 testfile_2   #统计三个文件的信息
3 92 598 testfile                     #第一个文件行数为3、单词数92、字节数598
9 18 78 testfile_1                    #第二个文件的行数为9、单词数18、字节数78
3 6 32 testfile_2                     #第三个文件的行数为3、单词数6、字节数32
15 116 708 总用量                     #三个文件总共的行数为15、单词数116、字节数708
```

2.5.6 sed

sed 是一个很好的文件处理工具，本身是一个管道命令，主要是以行为单位进行处理，可以将数据行进行替换、删除、新增、选取等特定工作，下面先了解一下 sed 的用法

sed 命令行格式为： sed [-nefri] ‘command’ 输入文本

常用选项：

- -n使用安静 (silent) 模式。在一般 sed 的用法中，所有来自 STDIN 的资料一般都会被列出到萤幕上。但如果加上 -n 参数后，则只有经过 sed 特殊处理的那一行 (或者动作) 才会被列出来。
- -e直接在指令列模式上进行 sed 的动作编辑；
- -f直接将 sed 的动作写在一个档案内，-f filename 则可以执行 filename 内的 sed 动作；
- -rsed 的动作支援的是延伸型正规表示法的语法。(预设是基础正规表示法语法)
- -i直接修改读取的档案内容，而不是由萤幕输出。

常用命令：

- a 新增，a 的后面可以接字串，而这些字串会在新的一行出现 (目前的下一行) ~
- c 取代，c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！
- d 删除，因为是删除啊，所以 d 后面通常不接任何咚咚；
- i 插入，i 的后面可以接字串，而这些字串会在新的一行出现 (目前的上一行)；
- p 列印，亦即将某个选择的资料印出。通常 p 会与参数 sed -n 一起运作 ~

- s 取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正规表示法！例如 1,20s/old/new/g 就是啦！

```
举例：（假设我们有一文件名为ab）
    删除某行
    [root@localhost ruby] # sed '1d' ab                    #删除第一行
    [root@localhost ruby] # sed '$d' ab                    #删除最后一行
    [root@localhost ruby] # sed '1,2d' ab                  #删除第一行到第二行
    [root@localhost ruby] # sed '2,$d' ab                  #删除第二行到最后一行

   显示某行
.   [root@localhost ruby] # sed -n '1p' ab                 #显示第一行
    [root@localhost ruby] # sed -n '$p' ab                 #显示最后一行
    [root@localhost ruby] # sed -n '1,2p' ab               #显示第一行到第二行
    [root@localhost ruby] # sed -n '2,$p' ab               #显示第二行到最后一行

   使用模式进行查询
    [root@localhost ruby] # sed -n '/ruby/p' ab            #查询包括关键字ruby所在所有行
    [root@localhost ruby] # sed -n '/\$/p' ab              #查询包括关键字$所在所有行，使用反斜线\屏蔽特殊含义

   增加一行或多行字符串
    [root@localhost ruby]# cat ab
    Hello!
    ruby is me,welcome to my blog.
    end
    [root@localhost ruby] # sed '1a drink tea' ab  #第一行后增加字符串"drink tea"
    Hello!
    drink tea
    ruby is me,welcome to my blog.
    end
    [root@localhost ruby] # sed '1,3a drink tea' ab #第一行到第三行后增加字符串"drink tea"
    Hello!
    drink tea
    ruby is me,welcome to my blog.
    drink tea
    end
    drink tea
    [root@localhost ruby] # sed '1a drink tea\nor coffee' ab   #第一行后增加多行，使用换行符\n
    Hello!
    drink tea
    or coffee
    ruby is me,welcome to my blog.
```

```
40      end
41
42     代替一行或多行
43      [root@localhost ruby] # sed '1c Hi' ab
                                #第一行代替为Hi
44      Hi
45      ruby is me,welcome to my blog.
46      end
47      [root@localhost ruby] # sed '1,2c Hi' ab
           #第一行到第二行代替为Hi
48      Hi
49      end
50
51     替换一行中的某部分
52     格式: sed 's/要替换的字符串/新的字符串/g'    (要替换的
       字符串可以用正则表达式)
53      [root@localhost ruby] # sed -n '/ruby/p' ab | sed 's/
           ruby/bird/g'     #替换ruby为bird
54     [root@localhost ruby] # sed -n '/ruby/p' ab | sed 's/
       ruby//g'          #删除ruby
55
56      插入
57      [root@localhost ruby] # sed -i '$a bye' ab          #
           在文件ab中最后一行直接输入"bye"
58      [root@localhost ruby]# cat ab
59      Hello!
60      ruby is me,welcome to my blog.
61      end
62      bye
63
64      删除匹配行
65
66       sed -i '/匹配字符串/d'  filename   (注: 若匹配字符串
            是变量, 则需要 “” , 而不是 ‘’ 。记得好像是)
67
68       替换匹配行中的某个字符串
69
70       sed -i '/匹配字符串/s/替换源字符串/替换目标字符串/g'
            filename
```

2.5.6.1 sed 删除行首行尾空格和 Tab

(1) 用 sed 实现删除行首空格: `sed 's/^[ \t]*//g'`

说明:

- 第一个/的左边是 s 表示替换，即将空格替换为空;
- 第一个/的右边是表示后面的以 xx 开头中括号表示“或”，空格或 Tab 中的任意一种。这是正则表达式的规范。中括号右边是 *，表示一个或多个;
- 第二个和第三个/中间没有东西，表示空 g 表示替换原来 buffer 中的，sed 在处理字符串的时候并不对源文件进行直接处理，先创建一个 buffer，但是加 g 表示对原 buffer 进行替换整体的意思是：用空字符去替换一个或多个用空格或 tab 开头的本体字符串。

（2）用 sed 实现删除行末空格：`sed 's/[ \t]*$//g'`

和上面稍微有些不同是前面删除了^ 符，在后面加上了美元符，这表示以 xx 结尾的字符串为对象。但是要注意在 KSH 中，Tab 并不是\t 而是直接打入一个 Tab 就可以了。

（3）删除行首空格、多余空格已经空行

```
cat 2dtemp|sed 's/^[ \t]*//g'|tr -s '\n'|tr -s ' ' > 2d.ods
```

2.6 输出命令

2.6.1 echo

echo 命令用于在屏幕上打印出指定的字符串。命令格式：

```
echo arg
```

可以使用 echo 实现更复杂的输出格式控制。

（1）显示转义字符

```
echo "\"It is a test\""
```

运行结果是：`"It is a test"`

其中双引号也可以省略。

（2）显示变量

```
name="OK"
echo "$name It is a test"
```

运行结果是：`OK It is a test`

同样双引号也可以省略。

如果变量与其它字符相连的话，需要使用大括号（{ }）：

```
mouth=8
echo "${mouth}-1-2009"
```

运行结果是：`8-1-2009`

（3）显示换行

```
echo "OK!\n"
echo "It is a test"
```

运行结果是：

```
OK!
It is a test
```

（4）显示不换行

```
echo "OK!\c"
echo "It is a test"
```

运行结果是：

```
OK!It si a test
```

（5）显示结果重定向至文件

```
echo "It is a test" > myfile
```

（6）原样输出字符串

若需要原样输出字符串（不进行转义），请使用单引号。例如：

```
echo '$name\"'
```

（7）显示命令执行结果

```
echo `date`
```

2.6.2 printf

printf 命令用于格式化输出，是 echo 命令的增强版。它是 C 语言 printf() 库函数的一个有限的变形，并且在语法上有些不同。printf 由 POSIX 标准所定义，移植性要比 echo 好。

如同 echo 命令，printf 命令也可以输出简单的字符串：

```
$printf "Hello, Shell\n"
Hello, Shell
$
```

printf 不像 echo 那样会自动换行，必须显式添加换行符 (\n)。

printf 命令的语法：

`printf format-string [arguments...]`

format-string 为格式控制字符串，arguments 为参数列表。

printf 命令与 C 语言中 printf() 功能和用法类似，不同的是：

- printf 命令不用加括号；
- format-string 可以没有引号，但最好加上，单引号双引号均可；
- 参数多于格式控制符 (%) 时，format-string 可以重用，可以将所有参数都转换；
- arguments 使用空格分隔，不用逗号。

请看下面的例子:

```
# format-string为双引号
$ printf "%d %s\n" 1 "abc"
1 abc
# 单引号与双引号效果一样
$ printf '%d %s\n' 1 "abc"
1 abc
# 没有引号也可以输出
$ printf %s abcdef
abcdef
# 格式只指定了一个参数, 但多出的参数仍然会按照该格式输出, format-string 被重用
$ printf %s abc def
abcdef
$ printf "%s\n" abc def
abc
def
$ printf "%s %s %s\n" a b c d e f g h i j
a b c
d e f
g h i
j
# 如果没有 arguments, 那么 %s 用NULL代替, %d 用 0 代替
$ printf "%s and %d \n"
and 0
# 如果以 %d 的格式来显示字符串, 那么会有警告, 提示无效的数字, 此时默认置为 0
$ printf "The first program always prints'%s,%d\n'" Hello Shell
-bash: printf: Shell: invalid number
The first program always prints 'Hello,0'
$
```

注意，根据 POSIX 标准，浮点格式%e、%E、%f、%g 与%G 是“不需要被支持”。这是因为 awk 支持浮点预算，且有它自己的 printf 语句。这样 Shell 程序中需要将浮点数值进行格式化的打印时，可使用小型的 awk 程序实现。然而，内建于 bash、ksh93 和 zsh 中的 printf 命令都支持浮点格式。

2.7 系统配置管理命令

2.7.1 date

语法:

```
date [-u] [-d datestr] [-s datestr] [--utc] [--universal] [--date=datestr] [--set=datestr] [--help] [--version] [+FORMAT] [MMDDhhmm[[CC]YY][.ss]]
```

参数说明:

- -d datestr：显示 datestr 中所设定的时间 (非系统时间)
- –help：显示辅助讯息
- -s datestr：将系统时间设为 datestr 中所设定的时间
- -u：显示目前的格林威治时间
- –version：显示版本编号

显示格式：

显示格式设定为一个加号后接数个标记，其中可用的标记列表如下

时间方面：

- %：印出%
- %n：下一行
- %t：跳格
- %H：小时 (00..23)
- %I：小时 (01..12)
- %k：小时 (0..23)
- %l：小时 (1..12)
- %M：分钟 (00..59)
- %p：显示本地 AM 或 PM
- %r：直接显示时间 (12 小时制，格式为 hh:mm:ss [AP]M)
- %s：从 1970 年 1 月 1 日 00:00:00 UTC 到目前为止的秒数
- %S：秒 (00..61)
- %T：直接显示时间 (24 小时制)
- %X：相当于%H:%M:%S
- %Z：显示时区

日期方面：

- %a：星期几 (Sun..Sat)
- %A：星期几 (Sunday..Saturday)
- %b：月份 (Jan..Dec)
- %B：月份 (January..December)
- %c：直接显示日期与时间
- %d：日 (01..31)
- %D：直接显示日期 (mm/dd/yy)
- %h：同%b
- %j：一年中的第几天 (001..366)
- %m：月份 (01..12)
- %U：一年中的第几周 (00..53) (以 Sunday 为一周的第一天的情形)
- %w：一周中的第几天 (0..6)
- %W：一年中的第几周 (00..53) (以 Monday 为一周的第一天的情形)
- %x：直接显示日期 (mm/dd/yy)

- %y：年份的最后两位数字 (00.99)
- %Y：完整年份 (0000..9999)

若是不以加号作为开头，则表示要设定时间，而时间格式为 MMDDhhmm[[CC]YY][.ss]，其中 MM 为月份，DD 为日，hh 为小时，mm 为分钟，CC 为年份前两位数字，YY 为年份后两位数字，ss 为秒数。使用权限：所有使用者。

当您不希望出现无意义的 0 时 (比如说 1999/03/07)，则可以在标记中插入 - 符号，比如说 date '+%-H:%-M:%-S' 会把时分秒中无意义的 0 给去掉，像是原本的 08:09:04 会变为 8:9:4。另外，只有取得权限者 (比如说 root) 才能设定系统时间。

当您以 root 身分更改了系统时间之后，请记得以 clock -w 来将系统时间写入 CMOS 中，这样下次重新开机时系统时间才会持续抱持最新的正确值。

实例：

```
# date
三 5月 12 14:08:12 CST 2010
# date '+%c'
2010年05月12日 星期三 14时09分02秒
# date '+%D' //显示完整的时间
05/12/10
# date '+%x' //显示数字日期，年份两位数表示
2010年05月12日
# date '+%T' //显示日期，年份用四位数表示
14:09:31
# date '+%X' //显示24小时的格式
14时09分39秒
# date '+usr_time: $1:%M %P -hey'
usr_time: $1:16 下午 -hey
```

2.7.2 source

2.7.3 export

2.7.4 ulimit

想象一个状况：我的 Linux 主机里面同时登陆了十个人，这十个人不知怎么搞的，同时开启了 100 个文件，每个文件的大小约 10MBytes，请问一下，我的 Linux 主机的内存要有多大才够？10*100*10 = 10000 MBytes = 10GBytes ... 老天爷，这样，系统不挂点才有鬼哩！为了要预防这个情况的发生，所以我们的 bash 是可以『限制用户的某些系统资源』的，包括可以开启的文件数量，可以使用的 CPU 时间，可以使用的内存总量等等。如何配置？用 ulimit 吧！

`ulimit [-SHacdfltu] [配额]`

选项与参数：-H：hard limit，严格的配置，必定不能超过这个配置的数值；-S：soft limit，警告的配置，可以超过这个配置值，但是若超过则有警告信息。在配置上，通常 soft 会比 hard 小，举例来说，soft 可配置为 80 而 hard 配置为 100，那么你可以使用到 90 (因为没有超过 100)，但介于 80 100 之间时，系统会有警告信息通知你！-a：后面不

接任何选项与参数，可列出所有的限制额度；-c： 当某些程序发生错误时，系统可能会将该程序在内存中的信息写成文件 (除错用)，这种文件就被称为核心文件 (core file)。此为限制每个核心文件的最大容量。-f： 此 shell 可以创建的最大文件容量 (一般可能配置为 2GB) 单位为 Kbytes -d： 程序可使用的最大断裂内存 (segment) 容量；-l： 可用于锁定 (lock) 的内存量 -t： 可使用的最大 CPU 时间 (单位为秒) -u： 单一用户可以使用的最大程序 (process) 数量。

2.7.5 history

2.7.6 basename

首先使用 –help 参数查看一下。basename 命令参数很少，很容易掌握。

```
Usage: basename NAME [SUFFIX]
  or:  basename OPTION... NAME...
Print NAME with any leading directory components removed.
If specified, also remove a trailing SUFFIX.

Mandatory arguments to long options are mandatory for short options too.
  -a, --multiple       support multiple arguments and treat each as a NAME
  -s, --suffix=SUFFIX  remove a trailing SUFFIX
  -z, --zero           separate output with NUL rather than newline
      --help     display this help and exit
      --version  output version information and exit

Examples:
  basename /usr/bin/sort          -> "sort"
  basename include/stdio.h .h     -> "stdio"
  basename -s .h include/stdio.h  -> "stdio"
  basename -a any/str1 any/str2   -> "str1" followed by "str2"
```

2.7.7 passwd——更改密码命令

passwd 是 “password” 的缩写[3]。在输入密码时，字符是不回显的，这可以防止他人偷看到新密码的长度，进一步降低被破解的可能性[4]。

2.7.8 iconv——文本编码转换命令

iconv -f encoding [-t encoding] [inputfile]... 转换给出文件的编码从一种到另外一种。

[3]类似的一个缩写命令 pwd 已经被 print working directory 用了，所以不要记错了

[4]可以体会到 Linux 系统的高度安全性吧

输入/输出格式规范:	
-f, –from-code=NAME	原始文本编码
-t, –to-code=NAME	输出编码
信息:	
-l, –list	列出所有已知编码字符集
输出控制:	
-c	忽略输出中的无效字符
-o, –output=FILE	输出文件
-s, –silent	suppress warnings
–verbose	打印进程信息
-?, –help	显示此帮助列表
–usag	提供简短的使用信息
-V, –version	显示程序版

对长选项必须或可选的参数同样适用于相应的短选项。

示例:

iconv -f=gbk -t=utf8 /home/phileas/ANSI-CJKmain.tex -o /home/phileas/utf8-cjkmain.tex

其中:-f=gbk -t=utf8 意思是将 gbk 编码转换成 utf8 编码;/home/phileas/ANSI-CJKmain.tex 是需要转换文件的全路径; -o 是输出文件; /home/phileas/utf8-cjkmain.tex 是输出文件的全路径。如果觉得写路径麻烦，那就直接用命令 cd 到该文件所处的文件夹下，就不用输入路径了。

2.7.9 ulimit 命令

ulimit 用于 shell 启动进程所占用的资源，是 shell 内建命令，语法格式:

ulimit [-acdfHlmnpsStvw] [size]

参数介绍:

- -a 显示当前所有的资源限制.
- -c size: 设置 core 文件的最大值. 单位:blocks
- -d size: 设置数据段的最大值. 单位:kbytes
- -f size: 设置创建文件的最大值. 单位:blocks
- -l size: 设置在内存中锁定进程的最大值. 单位:kbytes
- -m size: 设置可以使用的常驻内存的最大值. 单位:kbytes
- -n size: 设置内核可以同时打开的文件描述符的最大值. 单位:n
- -p size: 设置管道缓冲区的最大值. 单位:kbytes
- -s size: 设置堆栈的最大值. 单位:kbytes
- -t size: 设置 CPU 使用时间的最大上限. 单位:seconds
- -v size: 设置虚拟内存的最大值. 单位:kbytes
- -u number: 设置用户最大进程数（max user processes）

- -H 设置硬件资源限制.
- -S 设置软件资源限制.

在 Linux 下写程序的时候，如果程序比较大，经常会遇到“段错误”（segmentation fault）这样的问题，这主要就是由于 Linux 系统初始的堆栈大小（stack size）太小的缘故，一般为 10M。我一般把 stack size 设置成 256M，这样就没有段错误了！命令为：

ulimit -s 262140

如果要系统自动记住这个配置，就编辑/etc/profile 文件，在“`ulimit -S -c 0 > /dev/null 2>&1`”行下，添加“ulimit -s 262140”，保存重启系统就可以了。

调整 ulimit 的 stack size 参数调整为 unlimited 无限，使用 ulimit -s unlimited 时只能在当时的 shell 见效，重开一个 shell 就失效了。于是得在/etc/profile 的最后面添加 ulimit -s unlimited 就可以了，source /etc/profile 使修改文件生效。

2.8 Makefile

Makefile 的作用就是——自动化编译，一旦写好，只需要一个 make 命令来解析 Makefile，执行 Makefile 中描述的操作，整个工程就能自动编译，无论这个工程拥有多少个源代码文件。Makefile 定义了一系列的规则，来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至进行更复杂的功能操作，因为 Makefile 就像一个 shell 脚本一样，其中也可以执行操作系统的命令。

Makefile 是当前 Linux 下绝大多数软件的工程管理工具。而且它几乎已经成为了一个事实上的标准。即便不具备编写 Makefile 的技能，那也需要掌握 make 的使用方法。因为还有很大一部分需要的、或者感兴趣的软件都是采用源代码方式发行的。即便 rpm、deb 等软件打包方案在 Linux 世界已经大行其道，但是如果要使用这些软件的最新版本，基本上脱离不了使用 make 这个工具通过源代码来创建它。即便那些采用 Python、Perl 等解释语言开发的软件，绝大多数的情况也是利用 Makefile 作为安装脚本的。当然，更进一步说，有了一个很了不起的创意，并且准备分享给全世界，那么在 Linux 上就更需要掌握 make 乃至 Makefile 的编写技巧。

实际上 Makefile 是一类工程管理工具的工程描述文件的默认名称，让人头疼的是，这个文件名还不是唯一的。可以是 Makefile，可以是 makefile[5]，甚至是 GNUmakefile。Makefile 是 make 工程管理工具的工程描述文件，make 是类 UNIX 世界的一个传统工程管理工具，是它们文化的一部分，可以称作 Makefile 文化。传统 UNIX 的主要分支，比如 Solaris、HP-UNIX、AIX 等采用一套 Makefile 体系；BSD 系列如：FreeBSD、OpenBSD、NetBSD 等采用一套 Makefile 体系；Linux 采用另一套 Makefile 体系。而且即便属于同一体系，在某些方面都会显得大相径庭。

在 Linux 中采用的是 GNU make 这一体系，属于 GNU 计划中的一部分，可以说是所有体系中功能最为丰富、兼容性最好的一个 Makefile 文化的传承品。GNU make 已经是当前事实上的“标准”了，就好像它说的是标准普通话，其他 make 说的都是方言。

[5]注意 Linux 的文件名区分大小写

2.8.1 make 命令的基本用法

代码变成可执行文件，叫做编译（compile）；先编译这个，还是先编译那个（即编译的安排），叫做构建（build）。Make 是最常用的构建工具，诞生于 1977 年，主要用于 C 语言的项目。

make 是 Linux 系统下的一个工具，在任意目录中执行 make 都可以。make 会在执行路径中搜索 Makefile 文件。Makefile 并不是唯一的选择，还可以是 makefile 和 GNUmakefile。如果一个路径中同时存在它们三者，make 会选择一个。make 的选择顺序是 GNUmakefile、makefile 和 Makefile。只要选中了一个就不再理会其他的了。

选项：

make clean

清除上次的 make 命令所产生的 object 文件（后缀为“.o”的文件）及可执行文件。

make distclean

类似 make clean，但同时也将 configure 生成的文件全部删除掉，包括 Makefile。make clean 仅仅是清除之前编译的可执行文件及配置文件，而 make distclean 要清除所有生成的文件。

make install

将编译成功的可执行文件安装到系统目录中，一般为/usr/local/bin 目录。

make dist

产生发布软件包文件（即 distribution package）。这个命令将会将可执行文件及相关文件打包成一个 tar.gz 压缩的文件用来作为发布软件的软件包。

它会在当前目录下生成一个名字类似“PACKAGE-VERSION.tar.gz”的文件。PACKAGE 和 VERSION，是我们在 configure.in 中定义的`AM_INIT_AUTOMAKE(PACKAGE,␣VERSION)`。

make distcheck

生成发布软件包并对其进行测试检查，以确定发布包的正确性。这个操作将自动把压缩包文件解开，然后执行 configure 命令，并且执行 make，来确认编译不出现错误，最后提示你软件包已经准备好，可以发布了。

第 三 章 Shell & Shell Script

Shell 的学习与 Linux 的学习是密切相关的，要学 Linux 必须学 Shell。

3.1 什么是 Shell

Shell 是用户用来控制计算机的一个接口。与现代普通用户常见的计算机操作界面——图形化界面不同的是，这里指的是纯文字界面，称为命令行界面。只要能够操作应用程序的接口都能够称为壳程序。狭义的壳程序指的是命令行界面的软件，如常见的 bash 等。广义的壳程序则是图形接口的软件，因为图形接口其实也是操作各种应用程序来调用核心工作。命令行界面控制计算机高效、稳定、准确，适合程序员用户。

Shell 程序的出现是跟操作系统设计思想密切相关的。对于 Linux 系统，管理整个计算机硬件的是操作系统的核心 (Kernel)，这个核心是需要被保护。因为操作系统其实是一组软件，只不过这组软件控制着整个硬件与管理系统的活动监测。如果这组软件能被用户随意的操作，若使用者应用不当，将会使得整个系统崩溃。因为操作系统管理的就是整个硬件功能！所以不能够随便被一些没有管理能力的终端用户随意使用。

但是我们总是需要让用户操作系统的，所以就有了在操作系统上面发展的应用程序。用户可以透过应用程序来指挥核心，让核心达成我们所需要的硬件任务。应用程序是在最外层的，就如同鸡蛋的外壳一样，所以被称为壳程序 (Shell) 。通过 Shell 可以调用其他独立的应用程序，即通过 Shell 来操作其他应用程序来指挥核心完成工作，所以说 Shell 是用户用来控制计算机的一个接口。用户通过 Shell 将指令传递给 Kernel，Kernel 再来控制硬件完成工作。

3.2 常见的 Shell

简单点理解，Shell 就是系统跟计算机硬件交互时使用的中间介质，它只是系统的一个工具。实际上，在 Shell 和计算机硬件之间还有一层东西那就是系统内核了。打个比方，如果把计算机硬件比作一个人的躯体，而系统内核则是人的大脑，至于 Shell，把它比作人的五官似乎更加贴切些。回到计算机上来，用户直接面对的不是计算机硬件而是 Shell，用户把指令告诉 Shell，然后 Shell 再传输给系统内核，接着内核再去支配计算机硬件去执行各种操作。

Shell 是一种脚本语言，那么，就必须有解释器来执行这些脚本。Unix/Linux 上常见的 Shell 脚本解释器有 bash、sh、csh、ksh 等，习惯上把它们称作一种 Shell。我们常说有多少种 Shell，其实说的是 Shell 脚本解释器。

- bash

  bash 是 Linux 标准默认的 Shell，本教程也基于 bash 讲解。bash 由 Brian Fox 和 Chet Ramey 共同完成，是 BourneAgain Shell 的缩写，内部命令一共有 40 个。它基于

Bourne shell，吸收了 C shell 和 Korn shell 的一些特性。bash 完全兼容 sh，也就是说，用 sh 写的脚本可以不加修改的在 bash 中执行。

- sh

  sh 由 Steve Bourne 开发，是 Bourne Shell 的缩写，sh 是 Unix 标准默认的 Shell。

- ash

  ash shell 是由 Kenneth Almquist 编写的，Linux 中占用系统资源最少的一个小 Shell，它只包含 24 个内部命令，因而使用起来很不方便。

- csh

  csh 是 Linux 比较大的内核，它由以 William Joy 为代表的共计 47 位作者编成，共有 52 个内部命令。该 Shell 其实是指向/bin/tcsh 这样的一个 Shell，也就是说，csh 其实就是 tcsh。

- ksh

  ksh 是 Korn shell 的缩写，由 Eric Gisin 编写，共有 42 条内部命令。该 Shell 最大的优点是几乎和商业发行版的 ksh 完全兼容，这样就可以在不用花钱购买商业版本的情况下尝试商业版本的性能了。

3.3 Shell 的特性

- **记录命令历史**

  我们敲过的命令，Linux 是会有记录的，预设可以记录 1000 条历史命令。这些命令保存在用户的 home 目录中的.bash_history 文件中。有一点需要知道的是，只有当用户正常退出当前 Shell 时，在当前 Shell 中运行的命令才会保存至.bash_history 文件中。

  与命令历史有关的有一个有意思的字符那就是 “!” 了。常用的有这么几个应用：

  (1) !! （连续两 “!”），表示执行上一条指令；

  (2) !n（这里的 n 是数字），表示执行命令历史中第 n 条指令，例如 “!100” 表示执行命令历史中第 100 个命令；

  (3) ! 字符串（字符串大于等于 1），例如，!ta，表示执行命令历史中最近一次以 ta 为开头的指令。

- **命令和文件名补全**

  按 tab 键，它可以帮你补全一个指令，也可以帮你补全一个路径或者一个文件名。连续按两次 tab 键，系统则会把所有的指令或者文件名都列出来。

  (1) tab 接在一串命令的第一个字的后面为命令补全；

  (2) tab 接在一串命令的第二个字以后面则为文件或路径补全。

- **命令别名配置功能——alias**

  输入 alias 会看到目前系统预设的命令别名有哪些。自定义别名形如：

```
alias lm='ls -al'
```

这样 lm 会等于 ls -al 这样的一个功能，即显示这个目录底下的所有文件 (包含隐藏档) 及所有的文件属性。这个就是 bash 所特有的功能之一了。我们可以通过 alias 把一个常用的并且很长的指令别名一个简洁易记的指令。如果不想用了，还可以用 unalias 解除别名功能。需要注意的是命令的别名在 Shell 注销后就会失效，如果要保留，就得写入 Shell 的配置文件。

- **通配符**

  除了完整的字符串之外，bash 还支持许多的通配符来帮助用户查询与命令下达。举例来说，想要知道 /usr/bin 底下有多少以 X 为开头的文件吗？使用：ls -l /usr/bin/X* 就能够知道。此外，还有其他可供利用的通配符，这些都能够加快使用者的操作。

3.4 Bash Shell 的内建命令

为了方便 Shell 的操作，Bash 内建了很多命令。对于如何知道命令是外部命令还是内建在 Bash 当中，可以利用 type 命令，其格式如下：

`type [-tpa] name`

选项与参数：

- 不加任何选项与参数时，type 会显示出 name 是外部命令还是 Bash 内建命令；
- -t：当加入 -t 参数时，type 会将 name 以底下这些字眼显示出它的意义：
  - file ： 表示为外部命令；
  - alias ： 表示该命令为命令别名所配置的名称；
  - builtin ： 表示该命令为 bash 内建的命令功能。
- -p ： 如果后面接的 name 为外部命令时，才会显示完整文件名；
- -a ： 会由 PATH 变量定义的路径中，将所有含 name 的命令都列出来，包含 alias。

3.5 Bash 的进站与欢迎信息

在终端机接口 (tty1∼tty6) 登陆的时候，会有几行提示的字符串，那就是进站画面。这些字符串存储在/etc/issue 文件里面。例如，

```
xuan@x:~$ cat /etc/issue
Ubuntu 14.04.2 LTS \n \l

xuan@x~$
```

issue 内的各代码意义

`\d` 本地端时间的日期；

`\l` 显示第几个终端机接口；

`\m` 显示硬件的等级 (i386/i486/i586/i686...)；

`\n` 显示主机的网络名称；

`\o` 显示 domain name；

`\r` 操作系统的版本 (相当于 uname -r)

`\t` 显示本地端时间的时间；

`\s` 操作系统的名称；

`\v` 操作系统的版本。

3.6 Shell 的配置文件

3.6.1 login 与 non-login Shell

login Shell 与 non-login Shell 的区别在于有没有 login：

- login shell：取得 bash 时需要完整的登陆流程的，就称为 login shell。举例来说，由 tty1～tty6 登陆，需要输入用户的账号与密码，此时取得的 bash 就称为“login shell”；
- non-login shell：取得 bash 接口的方法不需要重复登陆的举动，举例来说，

  （1）以 X window 登陆 Linux 后，再以 X 的图形化接口启动终端机，此时那个终端接口并没有需要再次的输入账号与密码，那个 bash 的环境就称为“non-login shell”。

  （2）在原本的 bash 环境下再次调用 bash 这个命令，同样的也没有输入账号密码，那第二个 bash (子程序) 也是“non-login shell”。

在这两种取得 bash 的情况中，bash 读取的配置文件不一样。一般说来，login Shell 读取的配置文件是：

- `/etc/profile`：这是系统整体的配置，最好不要修改这个文件；
- `~/.bash_profile` 或 `~/.bash_login` 或 `~/.profile`：属于用户个人配置文件，要改自己的数据，就写入这里。

3.6.2 /etc/profile

/etc/profile，login Shell 读取的配置文件。这个配置文件可以利用用户 ID (UID) 来决定很多重要的变量数据，这也是每个用户 login 取得 bash 时一定会读取的配置文件。所以如果想要帮所有用户配置整体环境，那就是修改这里。不过，没事还是不要随便改这个文件。这个文件配置的变量主要有：

- PATH：会依据 UID 决定 PATH 变量要不要含有 sbin 的系统命令目录；
- MAIL：依据账号配置好使用者的 mailbox 到 /var/spool/mail/账号名；
- USER：根据用户的账号配置此一变量内容；
- HOSTNAME：依据主机的 hostname 命令决定此一变量内容；
- HISTSIZE：历史命令记录笔数。

/etc/profile 还会调用外部的配置文件，如：/etc/profile.d/*.sh，即只要用户具有 r 的权限，那么在 /etc/profile.d/ 这个目录内且扩展名为.sh 的文件就会被 /etc/profile 调用。如果

需要帮所有用户配置一些共享的命令别名时，可以在这个目录底下自行创建扩展名为.sh 的文件，并将所需要的数据写入即可。这样的机制让我们的 bash 操作接口变的非常友善。

3.6.3 ~/.bash_profile、~/.bash_login 和 ~/.profile

对于 login Shell，bash 在读完了整体环境配置的 /etc/profile 并藉此调用其他配置文件后，接下来则是会读取用户的个人配置文件。在 login shell 的 bash 环境中，所读取的个人偏好配置文件主要有三个，依序分别是：

1. `~/.bash_profile`
2. `~/.bash_login`
3. `~/.profile`

其实 bash 的 login shell 配置只会读取上面三个文件的其中一个，而读取的顺序则是依照上面的顺序。也就是说，如果 `~/.bash_profile` 存在，那么其他两个文件不论有无存在，都不会被读取。如果 `~/.bash_profile` 不存在才会去读取 `~/.bash_login`，而前两者都不存在才会读取 `~/.profile` 的意思。会有这么多的文件，其实是因应其他 shell 转换过来的使用者的习惯而已。

3.6.4 ~/.bashrc

当取得 non-login shell 时，该 bash 配置文件仅会读取 `~/.bashrc` 而已！

3.6.5 其他相关配置文件

还有一些配置文件可能会影响到 bash 操作的：

- `~/.bash_history`

  默认情况下，我们的历史命令就记录在这里。而这个文件能够记录几笔数据，则与 HISTFILESIZE 这个变量有关啊。每次登陆 bash 后，bash 会先读取这个文件，将所有的历史命令读入内存，因此，当我们登陆 bash 后就可以查知上次使用过哪些命令。

- `~/.bash_logout`

  这个文件则记录了，当我注销 bash 后，系统再帮我做完什么动作后才离开，的意思。可以去读取一下这个文件的内容，默认的情况下，注销时，bash 只是帮我们清掉屏幕的信息而已。不过，我们也可以将一些备份或者是其他认为重要的工作写在这个文件中（例如清空缓存盘），那么当我们离开 Linux 的时候，就可以解决一些烦人的事情。

- `/etc/man.config`

  这个文件乍看之下好像跟 bash 没相关性，但是对于系统管理员来说，却也是很重要的一个文件。这个文件的内容规范了使用 man 的时候，man page 的路径到哪里去寻找。所以说的简单一点，这个文件规定使用 man 命令的时候，该去哪里查看数据的路径配置。

那么什么时候要来修改这个文件呢？如果是以 tarball 的方式来安装数据，那么 man page 可能会放置在 /usr/local/softpackage/man 里头，那个 softpackage 是软件名称，这个时候就得以手动的方式将该路径加到 /etc/man.config 里头，否则使用 man 的时候就会找不到相关的说明档。

事实上，这个文件内最重要的其实是 MANPATH 这个变量配置。我们搜寻 man page 时，会依据 MANPATH 的路径去分别搜寻。另外，要注意的是，这个文件在各大不同版本 Linux distributions 中，名称都不太相同，例如 CentOS 用的是 /etc/man.config ，而 SuSE 用的则是 /etc/manpath.config ，可以利用 [tab] 按键来进行文件名的补齐。

3.7 通配符与特殊符号

在 bash 的操作环境中还有一个非常有用的功能，那就是通配符 (wildcard) ，利用 bash 处理数据就更方便了。底下列出一些常用的通配符：

- `*`：代表“0 个到无穷多个”任意字符；
- `?`：代表“一定有一个”任意字符；
- `[ ]`：同样代表“一定有一个在括号内”的字符 (非任意字符)。例如 [abcd] 代表“一定有一个字符，可能是 a、b、c、d 这四个任何一个”；再说个例子`*[0]` 和`*[0]*`，前者是寻找结尾为 0 的任意文件，后者则是寻找包含 0 的任意文件，把各自符号的意义套入就能分析出来；
- `[ - ]`：若有减号在中括号内时，代表“在编码顺序内的所有字符”。例如 [0-9] 代表 0 到 9 之间的所有数字，因为数字的语系编码是连续的；
- `[^ ]`：若中括号内的第一个字符为指数符号“^”，那表示“反向选择”，例如 `[^abc]` 代表一定有一个字符，只要是非 a、b、c 的其他字符就接受的意思。

下面列出的则是 bash 环境中的特殊符号：

- `#`：批注符号，这个最常被使用在 script 当中，视为说明，在后的数据均不运行；
- `\`：跳脱符号，将“特殊字符或通配符”还原成一般字符；
- `|`：管线 (pipe)，分隔两个管线命令的界定；
- `;`：连续命令下达分隔符，连续性命令的界定 (注意！与管道符并不相同)；
- `~`：用户的家目录；
- `$`：取用变量前导符，亦即是变量之前需要加的变量取代值；
- `&`：工作控制 (job control)，将命令变成背景下工作；
- `!`：逻辑运算意义上的“非”（not）的意思；
- `/`：目录符号，路径分隔的符号；
- `>`，`>>`：数据流重导向，输出导向，分别是“取代”与“累加”；
- `<`，`<<`：数据流重导向，输入导向；
- `' '`：单引号，不具有变量置换的功能；
- `" "`：具有变量置换的功能；
- `` ` ` ``：两个“` ”中间为可以先运行的命令，亦可使用“`$( )`”；

- ( )：在中间为子 shell 的起始与结束；
- { }：在中间为命令区块的组合。

所以文档名尽量不要使用到上述的字符。这里只是简单介绍，更详细的介绍参见相关章节。

3.7.1 “ ~ ” 后缀的文件

当我们编辑文本文件时，保存后经常发现多了个后面带个“ ~ ”的同样名字的隐藏文件。这是前一个版本的系统自动备份的文件，万一发生错误和其他意外时，保留这些文件还是有一定作用的。如果觉得没必要，可以通过设置更改。

- 对于编辑器 gedit：编辑 => 首选项 => 编辑器，把保存文件时备份选项去掉就可以了；
- vim 的话可以在其配置文件.vimrc 中取消设置 backup 文件：`set nobackup`。

3.7.2 反斜杠

因为命令太长，于是可以就利用“ `\[Enter]` ”来将 [Enter] 这个按键“跳脱！”开来，让 [Enter] 按键不再具有“开始运行”的功能，好让命令可以继续在下一行输入。需要特别留意，[Enter] 按键是紧接着反斜杠 (\) 的，两者中间没有其他字符。因为 \ 仅跳脱“紧接着的下一个字符”而已！所以，万一写成：`\␣[Enter]`，亦即 [Enter] 与反斜杠中间有一个空格时，则 \ 跳脱的是“空格键”而不是 [Enter] 按键，这个地方请再仔细的看一遍！很重要！

如果顺利跳脱 [Enter] 后，下一行最前面就会主动出现 > 的符号，就可以继续输入命令。也就是说，那个 > 是系统自动出现的，不需要输入。

3.8 什么是 Shell 脚本

计算机语言从执行过程的角度可以分为编译型语言和解释型语言（当然还有其他语言，如 HTML、LaTeX 标记语言等），但无论哪种语言，所有的语言编写出来的程序执行时要解释成计算机代码。

- 编译型语言：编写完程序后直接通过编译器转换程计算机代码生成可执行文件，以后每次执行时就不用进行转换，直接执行可执行文件即可。

  很多传统的程序设计语言，例 Fortran、Ada、Pascal、C、C++ 和 Java，都是编译型语言。这类语言需要预先将我们写好的源代码 (source code) 转换成目标代码 (object code)，这个过程被称作“编译”。运行程序时，直接读取目标代码 (object code)。由于编译后的目标代码 (object code) 非常接近计算机底层，因此执行效率很高，这是编译型语言的优点。但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。例如，在 C++ 里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

- 解释型语言：编写完成后不用转换成计算机代码，而是在每次执行的时候再进行代码转换工作，脚本语言的好处是源文件能够直接进行修改，直接使用文本编辑器就能够打开。

  解释型语言也被称作“脚本语言”。执行这类程序时，解释器 (interpreter) 需要读取我们编写的源代码 (source code)，并将其转换成目标代码 (object code)，再由计算机运行。因为每次执行程序都多了编译的过程，因此效率有所下降。使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象；缺点是它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用 C 或 C++ 来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有 awk、Perl、Python、Ruby 与 Shell。

Shell 本身是一个用 C 语言编写的程序，它是用户使用 Unix/Linux 的桥梁，用户的大部分工作都是通过 Shell 完成的。Shell 既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

它虽然不是 Unix/Linux 系统内核的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，Shell 是最重要的实用程序，深入了解和熟练掌握 Shell 的特性极其使用方法，是用好 Unix/Linux 系统的关键。可以说，Shell 使用的熟练程度反映了用户对 Unix/Linux 使用的熟练程度。

Shell 有两种执行命令的方式：

- 交互式（Interactive）：解释执行用户的命令，用户输入一条命令，Shell 就解释执行一条。
- 批处理（Batch）：用户事先写一个 Shell 脚本 (Script)，其中有很多条命令，让 Shell 一次把这些命令执行完，而不必一条一条地敲命令。

Shell 脚本和编程语言很相似，也有变量和流程控制语句，但 Shell 脚本是解释执行的，不需要编译，Shell 程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到 Shell 提示符下执行。

说白了，Shell 脚本就是一些命令的集合。举个例子，我想实现这样的操作：

1）进入到/tmp/目录；

2）列出当前目录中所有的文件名；

3）把所有当前的文件拷贝到/home/xuan 目录下；

4）删除当前目录下所有的文件。

简单的 4 步在 Shell 窗口中需要你敲 4 次命令，按 4 次回车。这样是不是很麻烦？当然这 4 步操作非常简单，如果是更加复杂的命令设置需要几十次操作呢？那样的话一次一次敲键盘会很麻烦。所以不妨把所有的操作都记录到一个文档中，然后去调用文档中的命令，这样一步操作就可以完成。其实这个文档呢就是 Shell 脚本了，只是这个 Shell 脚本有它特殊的格式。

Shell 脚本能帮助我们很方便的去管理服务器，因为我们可以指定一个任务计划定时

去执行某一个 Shell 脚本实现我们想要需求。这对于 Linux 系统管理员来说是一件非常值得自豪的事情。现在的 139 邮箱很好用，发邮件的同时还可以发一条邮件通知的短信给用户，利用这点，我们就可以在我们的 Linux 服务器上部署监控的 Shell 脚本，比如网卡流量有异常了或者服务器 web 服务器停止了就可以发一封邮件给管理员，同时发送给管理员一个报警短信这样可以让我们及时的知道服务器出问题了。

有一个问题需要约定一下，凡是自定义的脚本建议放到/usr/local/sbin/目录下，这样做的目的是，一来可以更好的管理文档；二来以后接管你的管理员都知道自定义脚本放在哪里，方便维护。

Shell 脚本通常都是以.sh 为后缀名的，这个并不是说不带.sh 这个脚本就不能执行，只是大家的一个习惯而已。所以，以后你发现了.sh 为后缀的文件那么它一定会是一个 Shell 脚本了。

“#”表示注释，在前面讲过的。后面跟一些该脚本的相关注释内容以及作者和创建日期或者版本等等。当然这些注释并非必须的，如果你懒的很，可以省略掉，但是笔者不建议省略。因为随着你工作时间的增加，你写的 shell 脚本也会越来越多，如果有一天你回头查看你写的某个脚本时，很有可能忘记该脚本是用来干什么的以及什么时候写的。所以写上注释是有必要的。另外系统管理员并非你一个，如果是其他管理员查看你的脚本，他看不懂岂不是很郁闷。

该脚本再往下面则为要运行的命令了。

例如：test.sh

```
#!/bin/bash
## This is my first shell script.
## Wrote by Aming 2011-05-20

date
echo "Hello World."
```

3.8.1 Shell 脚本的第一行

“#!”是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种 Shell。

目前研发送测的 shell 脚本中主要有以下两种方式：

(1) `#!/bin/sh`

(2) `#!/bin/bash`

以上两种方式有什么区别？对于脚本的实际运行会产生什么不同的影响吗？

脚本 test.sh 内容：

```
#!/bin/sh
source pcy.sh #pcy.sh并不存在
echo hello
```

执行./test.sh，屏幕输出为：

```
./test.sh: line 2: pcy.sh: No such file or directory
```

由此可见，在#!/bin/sh 的情况下，source 不成功，不会运行 source 后面的代码。修改 test.sh 脚本的第一行，变为#!/bin/bash，再次执行./test.sh，屏幕输出为：

```
./test.sh: line 2: pcy.sh: No such file or directory

hello
```

由此可见，在#!/bin/bash 的情况下，虽然 source 不成功，但是还是运行了 source 后面的 echo 语句。为什么会有这样的区别呢？junru 同学作了解释：

1. sh 一般设成 bash 的软链

```
[work@zjm-testing-app46 cy]$ ll /bin/sh

lrwxrwxrwx 1 root root 4 Nov 13 2006 /bin/sh -> bash
```

2. 在一般的 linux 系统当中（如 redhat），使用 sh 调用执行脚本相当于打开了 bash 的 POSIX 标准模式

3. 也就是说 /bin/sh 相当于 /bin/bash –posix

所以，sh 跟 bash 的区别，实际上就是 bash 有没有开启 posix 模式的区别。so，可以预想的是，如果第一行写成 #!/bin/bash --posix，那么脚本执行效果跟#!/bin/sh 是一样的（遵循 posix 的特定规范，有可能就包括这样的规范："当某行代码出错时，不继续往下解释"）。

3.8.2 Shell 脚本的执行

Shell 脚本的执行很简单，有两种方法。

（1）作为解释器参数，即直接运行解释器，其参数就是 Shell 脚本的文件名，例如 sh test.sh 或更严格的 /bin/sh test.sh

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。另外使用 sh 命令去执行一个 shell 脚本的时候是可以加-x 选项来查看这个脚本执行过程的，这样有利于我们调试这个脚本哪里出了问题。例如，sh -x test.sh，则有

```
+date
Fri May 2o 11:24:53 CST 2011
+echo 'Hello World'
Hello World
```

（2）作为可执行程序。

（a）在脚本所在目录下，首先使脚本具有执行权限，有两种方法：

1. 命令操作：chmod +x ./test.sh；

2. 图形化操作：选中脚本 test.sh 右键勾选可以执行选项。

（b）然后执行程序：`./test.sh`

注意，一定要写成./test.sh，而不是 test.sh。这里的“./”实质指的是相对路径，即当前路径下。如果 test.sh 在/bin 目录下，当然也可以采用绝大路径来运行即“/bin/test.sh”，只不过相对路径更简便些。运行其它二进制的程序也一样，直接写 test.sh，Linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有/bin、/sbin、/usr/bin、/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用./test.sh 告诉系统说，就在当前目录找。

通过这种方式运行 bash 脚本，第一行一定要写对，好让系统查找到正确的解释器。

这里的" 系统"，其实就是 shell 这个应用程序，但我故意写成系统，是方便理解，既然这个系统就是指 shell，那么一个使用/bin/sh 作为解释器的脚本是不是可以省去第一行呢？是的。

3.8.3 Shell 脚本的注释

以“#”开头的行就是注释，会被解释器忽略。sh 里没有多行注释，只能每一行加一个 # 号。如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？每一行加个 # 符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

3.8.4 Shell 脚本的文件包含

像其他语言一样，Shell 也可以包含外部脚本，将外部脚本的内容合并到当前脚本。

Shell 中包含脚本可以使用：`. filename`

或

`source filename`

两种方式的效果相同，简单起见，一般使用点号 (.)，但是注意点号 (.) 和文件名中间有一空格。

例如，创建两个脚本，一个是被调用脚本 subscript.sh，内容如下：

`url="http://see.xidian.edu.cn/cpp/view/2738.html"`

一个是主文件 main.sh，内容如下：

```
#!/bin/bash
. ./subscript.sh
echo $url
```

执行脚本：

```
$chomd +x main.sh
./main.sh
```

```
http://see.xidian.edu.cn/cpp/view/2738.html
$
```

注意：被包含脚本不需要有执行权限。

3.9 输入输出重定向

Unix 命令默认从标准输入设备 (stdin) 获取输入，将结果输出到标准输出设备 (stdout) 显示。一般情况下，标准输入设备就是键盘，标准输出设备就是终端，即显示器。

3.9.1 输出重定向

命令的输出不仅可以是显示器，还可以很容易的转移向到文件，这被称为输出重定向。

命令输出重定向的语法为：

`$ command > file`

这样，输出到显示器的内容就可以被重定向到文件。

例如，下面的命令在显示器上不会看到任何输出：

```
1 $ who > users
```

打开 users 文件，可以看到下面的内容：

```
$ cat users
oko          tty01   Sep 12 07:30
ai           tty15   Sep 12 13:32
ruth         tty21   Sep 12 10:10
pat          tty24   Sep 12 13:07
steve        tty25   Sep 12 13:03
$
```

输出重定向会覆盖文件内容，请看下面的例子：

```
$ echo line 1 > users
$ cat users
line 1
$
```

如果不希望文件内容被覆盖，可以使用 » 追加到文件末尾，例如：

```
$ echo line 2 >> users
$ cat users
line 1
```

```
line 2
$
```

3.9.2 输入重定向

和输出重定向一样，Unix 命令也可以从文件获取输入，语法为：

```
$ command < file
```

这样，本来需要从键盘获取输入的命令会转移到文件读取内容。

注意：输出重定向是大于号 (>)，输入重定向是小于号 (<)。

例如，计算 users 文件中的行数，可以使用下面的命令：

```
$ wc -l users
2 users
$
```

也可以将输入重定向到 users 文件：

```
$ wc -l < users
2
$
```

注意：上面两个例子的结果不同：第一个例子，会输出文件名；第二个不会，因为它仅仅知道从标准输入读取内容。

3.9.3 重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

- 标准输入文件 (stdin)：stdin 的文件描述符为 0，Unix 程序默认从 stdin 读取数据；
- 标准输出文件 (stdout)：stdout 的文件描述符为 1，Unix 程序默认向 stdout 输出数据；
- 标准错误文件 (stderr)：stderr 的文件描述符为 2，Unix 程序会向 stderr 流中写入错误信息。

默认情况下，command > file 将 stdout 重定向到 file，command < file 将 stdin 重定向到 file。

如果希望 stderr 重定向到 file，可以这样写：

```
$command 2 > file
```

如果希望 stderr 追加到 file 文件末尾，可以这样写：

```
$command 2 >> file
```

其中数字 2 表示标准错误文件（stderr）。

如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：`$command > file 2>&1`

或

```
$command >> file 2>&1
```

如果希望对 stdin 和 stdout 都重定向，可以这样写：`$command < file1 >file2`

command 命令将 stdin 重定向到 file1，将 stdout 重定向到 file2。

全部可用的重定向命令列表

命令	说明
command > file	将输出重定向到 file
command < file	将输入重定向到 file
command >> file	将输出以追加的方式重定向到 file
n > file	将文件描述符为 n 的文件重定向到 file
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file
n >& m	将输出文件 m 和 n 合并
n <& m	将输入文件 m 和 n 合并
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入

3.9.4 Here Document

Here Document 目前没有统一的翻译，这里暂译为“嵌入文档”。Here Document 是 Shell 中的一种特殊的重定向方式，它的基本的形式如下：

```
command << delimiter
    document
delimiter
```

它的作用是将两个 delimiter（界定符）之间的内容 (document) 作为输入传递给 command。

注意：

- 结尾的 delimiter 一定要顶格写，前面不能有任何字符，后面也不能有任何字符，包括空格和 tab 缩进；
- 开始的 delimiter 前后的空格会被忽略掉。

下面的例子，通过 wc -l 命令计算 document 的行数：

```
$wc -l << EOF
    This is a simple lookup program
    for good (and bad) restaurants
    in Cape Town.
EOF
3
$
```

其中 EOF 是 end of file 的缩写。

也可以将 Here Document 用在脚本中，例如：

```
1 #!/bin/bash
2
3 cat << EOF
4 This is a simple lookup program
5 for good (and bad) restaurants
```

```
in Cape Town.
EOF
```

运行结果：

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

下面的脚本通过 vi 编辑器将 document 保存到 test.txt 文件：

```
#!/bin/sh

filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

运行脚本：

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

打开 test.txt，可以看到下面的内容：

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

3.9.5 /dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 /dev/null：

```
$ command > /dev/null
```

/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到”禁止输出“的效果。

如果希望屏蔽 stdout 和 stderr，可以这样写：

```
$ command > /dev/null 2>&1
```

3.10 变量

3.10.1 定义变量

定义变量时，变量名不加美元符号（$），如

`variableName="value"`

注意，**等号两边不能直接接空格符**，即`variableName␣=␣"value"` 是错误的，这可能和我们熟悉的所有编程语言都不一样。总的来说，变量名的命名须遵循如下规则：

- 首个字符必须为字母（a-z，A-Z）；
- 中间不能有空格，可以使用下划线（_）；
- 不能使用标点符号；
- 不能使用 bash 里的关键字（可用 help 命令查看保留关键字）；
- 通常大写字符为系统默认变量，自行配置变量可以使用小写字符。

变量定义举例：

```
myUrl="http://see.xidian.edu.cn/cpp/linux/"
myNum=100
```

3.10.2 使用变量

使用一个定义过的变量，只要在变量名前面加美元符号（$）即可，如：

```
your_name="mozhiyan"
echo $your_name
echo ${your_name}
```

变量名外面的花括号是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，比如下面这种情况：

```
for skill in Ada Coffe Action Java
do
    echo "I am good at ${skill}Script"
done
```

如果不给 skill 变量加花括号，写成`echo "I am good at $skillScript"`，解释器就会把`$skillScript` 当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。推荐给所有变量加上花括号，这是个好的编程习惯。需要注意的是，还有个类似的表达，`$()`，这是命令替换的一种表达。

3.10.3 重新定义变量

已定义的变量，可以被重新定义，如：

```
myUrl="http://see.xidian.edu.cn/cpp/linux/"
echo ${myUrl}

myUrl="http://see.xidian.edu.cn/cpp/shell/"
echo ${myUrl}
```

这样写是合法的,但注意,第二次赋值的时候不能写 $myUrl="http://see.xidian.edu.cn/cpp/shell/",使用变量的时候才加美元符（$）。

3.10.4 只读变量

使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash

myUrl="http://see.xidian.edu.cn/cpp/shell/"
readonly myUrl
myUrl="http://see.xidian.edu.cn/cpp/danpianji/"
```

运行脚本，结果如下：

```
/bin/sh: NAME: This variable is read only.
```

3.10.5 删除变量

使用 unset 命令可以删除变量。语法：

```
unset variable_name
```

变量被删除后不能再次使用；unset 命令不能删除只读变量。

举个例子：

```
#!/bin/sh

myUrl="http://see.xidian.edu.cn/cpp/u/xitong/"
unset myUrl
echo $myUrl
```

上面的脚本没有任何输出。

3.10.6 变量类型

运行 Shell 时，会同时存在三种变量：

1) 局部变量

局部变量在脚本或命令中定义，仅在当前 Shell 实例中有效，其他 Shell 启动的程序不能访问局部变量。

2) 环境变量

所有的程序，包括 Shell 启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候 Shell 脚本也可以定义环境变量。

3) Shell 变量

Shell 变量是由 Shell 程序设置的特殊变量。Shell 变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了 Shell 的正常运行。

详见环境变量相关章节。

3.10.7 特殊变量

（1）特殊变量列表

变量	含义
$0	当前脚本的文件名
$n	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是 $1，第二个参数是 $2
$#	传递给脚本或函数的参数个数。
$*	传递给脚本或函数的所有参数。
$@	传递给脚本或函数的所有参数。被双引号 (“ ") 包含时，与 $* 稍有不同，下面将会讲到。
$?	上个命令的退出状态，或函数的返回值。
$$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

例如，$ 表示当前 Shell 进程的 ID，即 pid，看下面的代码：

```
$echo $$
```

运行结果

3433

（2）命令行参数

运行脚本时传递给脚本的参数称为命令行参数。命令行参数用 $n 表示，例如，$1 表示第一个参数，$2 表示第二个参数，依次类推。

请看下面的脚本：

```
#!/bin/bash

echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

运行：`./test.sh Zara Ali`

运行结果为：

```
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

（3）$* 和 $@ 的区别

$* 和 $@ 都表示传递给函数或脚本的所有参数，不被双引号 (“ ") 包含时，形式是一样的，都以 “ $1" “$2" …“$n" 的形式输出所有参数。

但是当它们被双引号 (“ ") 包含时，“$*" 会将所有的参数作为一个整体，以 “$1 $2 … $n" 的形式输出所有参数；“$@" 会将各个参数分开，以 “$1" “$2" …“$n" 的形式输出所有参数。

（4）退出状态

$? 可以获取上一个命令的退出状态。所谓退出状态，就是上一个命令执行后的返回结果。

退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。

不过，也有一些命令返回其他值，表示不同类型的错误。

$? 也可以表示函数的返回值，后续将会讲解。

3.10.8 键盘读取变量

要读取来自键盘输入的变量，就要用到 read 命令了。其使用格式如下：

`read [-pt] variable` 选项与参数：

- -p： 后面可以接提示字符！
- -t： 后面可以接等待的 “秒数”。这个比较有趣，不会一直等待使用者。

read 之后不加任何参数，直接加上变量名称，那么底下就会主动出现一个空白行等待你的输入。如果加上 -t 后面接秒数，那么在设置的时间之内没有任何动作时，该命令就会自动略过了。如果是加上 -p ，在输入的光标前就会有比较多可以用的提示字符给我们参考，在命令的下达里面，比较美观。参见下面这个例子：

```
1 xuan@www:~$ read atest
2 This is an apple
3 xuan@www:~$ echo $atest
4 This is an apple
5 xuan@www:~$ read -p "Please keyin your name: " -t 30 named
6 Please keyin your name: xuan
7 xuan@www:~$ echo $named
8 xuan
9 xuan@www:~$
```

3.10.9 变量声明

declare 或 typeset 是一样的功能，就是在声明变量的类型。如果使用 declare 后面并没有接任何参数，那么 Bash 就会主动的将所有的变量名称与内容通通叫出来，就好像使用 set 一样。declare 的命令格式如下：

```
declare [-aixr] variable
```

选项与参数：

- -a： 将后面名为 variable 的变量定义成为数组 (array) 类型；
- -i： 将后面名为 variable 的变量定义成为整数数字 (integer) 类型；
- -x： 用法与 export 一样，就是将后面的自定义变量 variable 变成环境变量；
- +x： 撤销环境变量，将将后面的环境变量 variable 变成自定义变量；
- -r： 将变量配置成为 readonly 类型，该变量不可被更改内容，也不能 unset；
- -p： 列出后面变量的类型。

具体用法参考鸟哥的例子：

```
范例一：让变量 sum 进行 100+300+50 的加总结果
[root@www ~]# sum=100+300+50
[root@www ~]# echo $sum
100+300+50  <==咦！怎么没有帮我计算加总？因为这是文字型态的变量属性啊！
[root@www ~]# declare -i sum=100+300+50
[root@www ~]# echo $sum
450         <==F 乎？？

范例二：将 sum 变成环境变量
[root@www ~]# declare -x sum
[root@www ~]# export | grep sum
declare -ix sum="450"  <==果然出现了！包括有 i 与 x 的宣告！

范例三：让 sum 变成只读属性，不可更动！
[root@www ~]# declare -r sum
[root@www ~]# sum=tesgting
-bash: sum: readonly variable  <==老天爷～不能改这个变量了！

范例四：让 sum 变成自定义变量吧！
[root@www ~]# declare +x sum  <== 将 - 变成 + 可以进行『取消』动作
[root@www ~]# declare -p sum  <== -p 可以单独列出变量的类型
declare -ir sum="450" <== 看吧！只剩下 i，r 的类型，不具有 x F！
```

在默认的情况底下，bash 对于变量有几个基本的定义：

- 变量类型默认为『字符串』，所以若不指定变量类型，则 1+2 为一个『字符串』而不是『计算式』。所以上述第一个运行的结果才会出现那个情况的；
- bash 环境中的数值运算，默认最多仅能到达整数形态，所以 1/3 结果是 0；

3.11 替换

3.11.1 转义字符

如果表达式中包含特殊字符，Shell 将会进行替换。例如，在双引号中使用变量就是一种替换，转义字符也是一种替换。

举个例子：

```
#!/bin/bash

a=10
echo -e "Value of a is $a \n"
```

运行结果：

`Value of a is 10`

这里 -e 表示对转义字符进行替换。如果不使用 -e 选项，将会原样输出：

`Value of a is 10\n`

下面的转义字符都可以用在 echo 中：

转义字符	含义
\\	反斜杠
\a	警报，响铃
\b	退格（删除键）
\f	换页 (FF)，将当前位置移到下页开头
\n	换行
\r	回车
\t	水平制表符（tab 键）
\v	垂直制表符

可以使用 echo 命令的 -E 选项禁止转义，默认也是不转义的；使用 -n 选项可以禁止插入换行符。

3.11.2 命令替换

命令替换是指 Shell 可以先执行命令，将输出结果替换该命令。命令替换的一般作用是抽取一个命令的输出，然后使用 = 操作赋值到一个变量供以后使用。

命令替换有两种语法：

`‘command‘` 或 `$(command)`

注意是反引号，不是单引号，这个键位于 Esc 键下方。

命令由于简单，使用‘‘没问题。如果命令中有转义就麻烦了，因为\也是‘‘的转义符，不注意的情况下出现不是预想的结果，而且还很难查出 bug。`$()` 就不一样了，你在命令行上任何一条正确的命令都可以用`$()`。还有`$()` 支持多重嵌套，使用更加灵活，如 `fatherpath=$(dirname $(pwd))`。

下面的例子中，将命令执行结果保存在变量中：

```
#!/bin/bash

DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

运行结果：

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

3.11.3 变量替换

变量替换可以根据变量的状态（是否为空、是否定义等）来改变它的值。可以使用的变量替换形式：

- `${var}`：变量本来的值
- `${var:-word}`：如果变量 var 为空或已被删除 (unset)，那么返回 word，但不改变 var 的值。
- `${var:=word}`：如果变量 var 为空或已被删除 (unset)，那么返回 word，并将 var 的值设置为 word。
- `${var:?message}`：如果变量 var 为空或已被删除 (unset)，那么将消息 message 送到标准错误输出，可以用来检测变量 var 是否可以被正常赋值。若此替换出现在 Shell 脚本中，那么脚本将停止运行。
- `${var:+word}`：如果变量 var 被定义，那么返回 word，但不改变 var 的值。

3.12 数组

Bash 支持一维数组（不支持多维数组），并且没有限定数组的大小。类似与 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于 0。

3.12.1 定义数组

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

array_name=(value1 ... valuen)

例如:

```
array_name=(value0 value1 value2 value3)
```

或者

```
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量:

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

3.12.2 读取数组

读取数组元素值的一般格式是:

${array_name[index]}

例如:

```
valuen=${array_name[2]}
```

举个例子:

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

运行脚本，输出:

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

使用 @ 或 * 可以获取数组中的所有元素，例如：

```
${array_name[*]}
${array_name[@]}
```

举个例子：

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

运行脚本，输出：

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

3.12.3 获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

```
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

3.13 运算符

Bash 支持很多运算符，包括算术运算符、关系运算符、布尔运算符、字符串运算符和文件测试运算符。

3.13.1 算术运算

原生 Bash 不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。

expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

例如，两个数相加：

```
#!/bin/bash

val=`expr 2 + 2`
echo "Total value : $val"
```

运行脚本输出：

```
Total value : 4
```

注意：

- 表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样；
- 完整的表达式要被 ‘ ‘ 包含，注意这个字符不是常用的单引号，在 Esc 键下边。

再看一个使用算术运算符的例子：

```
#!/bin/sh

a=10
b=20
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

运行结果：

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

注意：

- 乘号 (*) 前边必须加反斜杠 (\) 才能实现乘法运算；
- if...then...fi 是条件语句，后续将会讲解。

算术运算符列表

运算符	说明	举例
+	加法	‘expr $a + $b‘ 结果为 30。
-	减法	‘expr $a - $b‘ 结果为 10。
*	乘法	‘expr $a * $b‘ 结果为 200。
/	除法	‘expr $b / $a‘ 结果为 2。
%	取余	‘expr $b % $a‘ 结果为 0。
=	赋值	a=$b 将把变量 b 的值赋给 a。
==	相等	用于比较两个数字，相同则返回 true。[$a == $b] 返回 false。
!=	不相等	用于比较两个数字，不相同则返回 true。[$a != $b] 返回 true。

注意：条件表达式要放在方括号之间，并且要有空格，例如 [$a==$b] 是错误的，必须写成 [$a == $b]。

3.13.2 关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

先来看一个关系运算符的例子：

```
#!/bin/sh

a=10
b=20
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
```

```
then
    echo "$a -lt $b: a is less than b"
else
    echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
    echo "$a -ge $b: a is greater or  equal to b"
else
    echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
    echo "$a -le $b: a is less or  equal to b"
else
    echo "$a -le $b: a is not less or equal to b"
fi
```

运行结果：

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

关系运算符列表

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true	[$a -eq $b] 返回 true
-ne	检测两个数是否相等，不相等返回 true	[$a -ne $b] 返回 true
-gt	检测左边的数是否大于右边的，如果是，则返回 true	[$a -gt $b] 返回 false
-lt	检测左边的数是否小于右边的，如果是，则返回 true	[$a -lt $b] 返回 true
-ge	检测左边的数是否大等于右边的，如果是，则返回 true	[$a -ge $b] 返回 false
-le	检测左边的数是否小于等于右边的，如果是，则返回 true	[$a -le $b] 返回 true

3.13.3 布尔运算符

先来看一个布尔运算符的例子：

```
#!/bin/sh

a=10
b=20

```

```
if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

运行结果：

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

布尔运算符列表

运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true	`[ ! false ]` 返回 true
-o	或运算，有一个表达式为 true 则返回 true	`[ $a -lt 20 -o $b -gt 100 ]` 返回 true
-a	与运算，两个表达式都为 true 才返回 true	`[ $a -lt 20 -a $b -gt 100 ]` 返回 false

3.13.4 字符串运算符

先来看一个例子：

```
#!/bin/sh

a="abc"
b="efg"

```

```
if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

运行结果：

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

字符串运算符列表

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true	[$a = $b] 返回 false
!=	检测两个字符串是否相等，不相等返回 true	[$a != $b] 返回 true
-z	检测字符串长度是否为 0，为 0 返回 true	[-z $a] 返回 false
-n	检测字符串长度是否为 0，不为 0 返回 true	[-z $a] 返回 true
str	检测字符串是否为空，不为空返回 true	[$a] 返回 true

3.13.5 文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。

例如，变量 file 表示文件“/var/www/tutorialspoint/unix/test.sh”，它的大小为 100 字节，具有 rwx 权限。下面的代码，将检测该文件的各种属性：

```
#!/bin/sh

file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]
then
   echo "File has read access"
else
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

if [ -f $file ]
then
   echo "File is an ordinary file"
else
   echo "This is sepcial file"
fi

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi

if [ -s $file ]
then
   echo "File size is zero"
else
   echo "File size is not zero"
fi

if [ -e $file ]
then
   echo "File exists"
```

```
else
    echo "File does not exist"
fi
```

运行结果：

```
File has read access
File has write permission
File has execute permission
File is an ordinary file
This is not a directory
File size is zero
File exists
```

文件测试运算符列表

运算符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true	[-b $file] 返回 false
-c file	检测文件是否是字符设备文件，如果是，则返回 true	[-b $file] 返回 false
-d file	检测文件是否是目录，如果是，则返回 true	[-d $file] 返回 false
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true	[-f $file] 返回 true
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true	[-g $file] 返回 false
-k file	检测文件是否设置了粘着位 (Sticky Bit)，如果是，则返回 true	[-k $file] 返回 false
-p file	检测文件是否是具名管道，如果是，则返回 true	[-p $file] 返回 false
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true	[-u $file] 返回 false
-r file	检测文件是否可读，如果是，则返回 true	[-r $file] 返回 true
-w file	检测文件是否可写，如果是，则返回 true	[-w $file] 返回 true
-x file	检测文件是否可执行，如果是，则返回 true	[-x $file] 返回 true
-s file	检测文件是否为空（文件大小是否大于 0），不为空返回 true	[-s $file] 返回 true
-e file	检测文件（包括目录）是否存在，如果是，则返回 true	[-e $file] 返回 true

3.14 字符串

字符串是 shell 编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。单双引号的区别跟 PHP 类似。

（1）单引号

```
str='this is a string'
```

单引号字符串的限制：

- 单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
- 单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

（2）双引号

```
your_name='qinjx'
str="Hello, I know your are \"$your_name\"! \n"
```

双引号的优点：

- 双引号里可以有变量；
- 双引号里可以出现转义字符

（3）拼接字符串

```
your_name="qinjx"
greeting="hello, "$your_name" !"
greeting_1="hello, ${your_name} !"

echo $greeting $greeting_1
```

（4）获取字符串长度

```
string="abcd"
echo ${#string} #输出 4
```

（5）提取子字符串

```
string="alibaba is a great company"
echo ${string:1:4} #输出liba
```

注意：计数从 0 开始。

（6）查找子字符串

```
string="alibaba is a great company"
echo `expr index "$string" is`
```

3.15 条件判断

3.15.1 if 语句

if 语句通过关系运算符判断表达式的真假来决定执行哪个分支。Shell 有三种 if ... else 语句：

- if ... then ... fi 语句；
- if ... then ... else ... fi 语句；
- if ... then ... elif ... then ... else ... fi 语句。

（1）if ... then ... fi 语句的语法：

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

如果 expression 返回 true，then 后边的语句将会被执行；如果返回 false，不会执行任何语句。最后必须以 fi 来结尾闭合 if，fi 就是 if 倒过来拼写。注意：expression 和方括号 ([ ]) 之间必须有空格，否则会有语法错误。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

运行结果：

```
a is not equal to b
```

（2）if ... then ... else ... fi 语句的语法

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

如果 expression 返回 true，那么 then 后边的语句将会被执行；否则，执行 else 后边的语句。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

运行结果：

```
a is not equal to b
```

（3）if ... then ... elif ... then ... else ... fi 语句可以对多个条件进行判断，语法为：

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

哪一个 expression 的值为 true，就执行哪个 expression 后面的语句；如果都为 false，那么不执行任何语句。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

运行结果：

```
a is not equal to b
```

（4）if 语句还可以和 test 命令结合使用，如

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
```

```
    echo 'The two numbers are equal!'
else
    echo 'The two numbers are not equal!'
fi
```

运行结果：

```
The two numbers are equal!
```

test 命令用于检查某个条件是否成立，与方括号 ([ ]) 类似。

（5）if 语句也可以写成一行，以命令的方式来运行，像这样：

```
if test $[2*3] -eq $[1+5]; then echo 'The two numbers are equal!'; fi;
```

3.15.2 if 语句中的 -a 到-z

```
[ -a FILE ] 如果 FILE 存在则为真。
[ -b FILE ] 如果 FILE 存在且是一个块特殊文件则为真。
[ -c FILE ] 如果 FILE 存在且是一个字特殊文件则为真。
[ -d FILE ] 如果 FILE 存在且是一个目录则为真。
[ -e FILE ] 如果 FILE 存在则为真。
[ -f FILE ] 如果 FILE 存在且是一个普通文件则为真。
[ -g FILE ] 如果 FILE 存在且已经设置了SGID则为真。
[ -h FILE ] 如果 FILE 存在且是一个符号连接则为真。
[ -k FILE ] 如果 FILE 存在且已经设置了粘制位则为真。
[ -p FILE ] 如果 FILE 存在且是一个名字管道(F如果O)则为真。
[ -r FILE ] 如果 FILE 存在且是可读的则为真。
[ -s FILE ] 如果 FILE 存在且大小不为o则为真。
[ -t FD ] 如果文件描述符 FD 打开且指向一个终端则为真。
[ -u FILE ] 如果 FILE 存在且设置了SUID (set user ID)则为真。
[ -w FILE ] 如果 FILE 如果 FILE 存在且是可写的则为真。
[ -x FILE ] 如果 FILE 存在且是可执行的则为真。
[ -O FILE ] 如果 FILE 存在且属有效用户ID则为真。
[ -G FILE ] 如果 FILE 存在且属有效用户组则为真。
[ -L FILE ] 如果 FILE 存在且是一个符号连接则为真。
[ -N FILE ] 如果 FILE 存在 and has been mod如果ied since it was last read则为真。
[ -S FILE ] 如果 FILE 存在且是一个套接字则为真。
[ FILE1 -nt FILE2 ] 如果 FILE1 has been changed more recently than FILE2, or 如果 FILE1 exists and FILE2 does not则为真。
[ FILE1 -ot FILE2 ] 如果 FILE1 比 FILE2 要老，或者 FILE2 存在且 FILE1 不存在则为真。
[ FILE1 -ef FILE2 ] 如果 FILE1 和 FILE2 指向相同的设备和节点号则为真。
[ -o OPTIONNAME ] 如果 shell选项 “OPTIONNAME” 开启则为真。
[ -z STRING ] “STRING” 的长度为零则为真。
[ -n STRING ] or [ STRING ] “STRING” 的长度为非零 non-zero则为真。
```

```
[ STRING1 == STRING2 ] 如果2个字符串相同。 "=" may be used instead of "==" for strict POSIX compliance则为真。
[ STRING1 != STRING2 ] 如果字符串不相等则为真。
[ STRING1 < STRING2 ] 如果 "STRING1" sorts before "STRING2" lexicographically in the current locale则为真。
[ STRING1 > STRING2 ] 如果 "STRING1" sorts after "STRING2" lexicographically in the current locale则为真。
[ ARG1 OP ARG2 ] "OP" is one of -eq, -ne, -lt, -le, -gt or -ge. These arithmetic binary operators return true if "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively. "ARG1" and "ARG2" are integers.
```

3.15.3 case 语句

case ... esac 与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构。case 语句匹配一个值或一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：

```
case 值 in
模式1)
    command1
    command2
    command3
    ;;
模式2)
    command1
    command2
    command3
    ;;
*)
    command1
    command2
    command3
    ;;
esac
```

case 工作方式如上所示。取值后面必须为关键字 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至;;。;; 与其他语言中的 break 类似，意思是跳到整个 case 语句的最后。取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。

下面的脚本提示输入 1 到 4，与每一种模式进行匹配：

```
echo 'Input a number between 1 to 4'
echo 'Your number is:\c'
read aNum
case $aNum in
    1)  echo 'You select 1'
    ;;
    2)  echo 'You select 2'
    ;;
    3)  echo 'You select 3'
    ;;
    4)  echo 'You select 4'
    ;;
    *)  echo 'You do not select a number between 1 to 4'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
Input a number between 1 to 4
Your number is:3
You select 3
```

再举一个例子：

```
#!/bin/bash

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d
         directory]"
      exit 1 # Command to come out of the program with
         status 1
      ;;
esac
```

运行结果：

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
```

```
Dir name is unix
$
```

3.16 循环

3.16.1 for 循环

for 循环一般格式为:

```
for 变量 in 列表
do
    command1
    command2
    ...
    commandN
done
```

列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量。in 列表是可选的，如果不用它，for 循环使用命令行的位置参数。

例如，顺序输出当前列表中的数字:

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

运行结果:

```
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
```

顺序输出字符串中的字符:

```
for str in 'This is a string'
do
    echo $str
done
```

运行结果:

```
This is a string
```

显示主目录下以.bash 开头的文件:

```
#!/bin/bash

for FILE in $HOME/.bash*
do
    echo $FILE
done
```

运行结果:

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

3.16.1.1 seq 命令

seq 命令用于产生从某个数到另外一个数之间的所有整数。

例一:

```
seq 1 10
```

结果是 1 2 3 4 5 6 7 8 9 10

例二:

```
for i in `seq 1 10`
do
echo $i
done
```

或者用

```
for i in $(seq 1 10)
```

也可以。

seq 的选项:

- `-f, --format=FORMAT`: use printf style floating-point FORMAT (default: %g)

  -f 选项指定格式

```
seq -f"%3g" 9 11
9
```

```
10
11
```

% 后面指定数字的位数默认是“%g”；

“%3g”那么数字位数不足部分是空格；

sed -f “%03g” 9 11 这样的话数字位数不足部分是 0；

% 前面制定字符串：

```
seq -f "str%03g" 9 11
str009
str010
str011
```

- `-s, --separator=STRING`: use STRING to separate numbers (default: /n)

  -s 指定分隔符默认是回车；

  `seq -s" " -f"str\%03g" 9 11`

  输出结果：str009 str010 str011

  要指定/t 做为分隔符号：

  `seq -s"‘echo -e "/t"‘" 9 11`

- `-w, --equal-width`: equalize width by padding with leading zeroes

  -w 指定输出数字同宽不能和-f 一起用

```
seq -w 98 101
098
099
100
101
```

输出是同宽的。

3.16.2 while 循环

while 循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为：

```
while command
do
   Statement(s) to be executed if command is true
done
```

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。

以下是一个基本的 while 循环，测试条件是：如果 COUNTER 小于 5，那么返回 true。COUNTER 从 0 开始，每次循环处理时，COUNTER 加 1。运行上述脚本，返回数字 1 到 5，然后终止。

```
COUNTER=0
while [ $COUNTER -lt 5 ]
do
    COUNTER='expr $COUNTER+1'
    echo $COUNTER
done
```

运行脚本，输出：

```
1
2
3
4
5
```

while 循环可用于读取键盘信息。下面的例子中，输入信息被设置为变量 FILM，按 <Ctrl-D> 结束循环。

```
echo 'type <CTRL-D> to terminate'
echo -n 'enter your most liked film: '
while read FILM
do
    echo "Yeah! great film the $FILM"
done
```

运行脚本，输出类似下面：

```
type <CTRL-D> to terminate
enter your most liked film: Sound of Music
Yeah! great film the Sound of Music
```

3.16.3 until 循环

until 循环执行一系列命令直至条件为 true 时停止。until 循环与 while 循环在处理方式上刚好相反。一般 while 循环优于 until 循环，但在某些时候，也只是极少数情况下，until 循环更加有用。

until 循环格式为：

```
until command
do
   Statement(s) to be executed until command is true
done
```

command 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。

例如，使用 until 命令输出 0 9 的数字：

```
#!/bin/bash

a=0

until [ ! $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

运行结果：

```
0
1
2
3
4
5
6
7
8
9
```

3.16.4 跳出循环

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，像大多数编程语言一样，Shell 也使用 break 和 continue 来跳出循环。

3.16.4.1 break 命令

break 命令允许跳出所有循环（终止执行后面的所有循环）。

下面的例子中，脚本进入死循环直至用户输入数字大于 5。要跳出这个循环，返回到 shell 提示符下，就要使用 break 命令。

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "You do not select a number between 1 to 5, game is over!"
            break
        ;;
    esac
```

```
done
```

在嵌套循环中，break 命令后面还可以跟一个整数，表示跳出第几层循环。例如：

```
break n
```

表示跳出第 n 层循环。

下面是一个嵌套循环的例子，如果 var1 等于 2，并且 var2 等于 0，就跳出循环：

```
#!/bin/bash

for var1 in 1 2 3
do
   for var2 in 0 5
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

如上，break 2 表示直接跳出外层循环。运行结果：

```
1 0
1 5
```

3.16.4.2 continue 命令

continue 命令与 break 命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环。

例如，对上面的例子进行修改：

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
        *) echo "You do not select a number between 1 to 5!"
            continue
            echo "Game is over!"
        ;;
    esac
done
```

运行代码发现，当输入大于 5 的数字时，该例中的循环不会结束，语句

```
echo "Game is over!"
```

永远不会被执行。

同样，continue 后面也可以跟一个数字，表示跳出第几层循环。

再看一个 continue 的例子：

```
#!/bin/bash

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
    Q=`expr $NUM % 2`
    if [ $Q -eq 0 ]
    then
        echo "Number is an even number!!"
        continue
    fi
    echo "Found odd number"
done
```

运行结果：

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

3.17 函数

3.17.1 函数的定义

函数可以让我们将一个复杂功能划分成若干模块，让程序结构更加清晰，代码重复利用率更高。像其他编程语言一样，Shell 也支持函数。Shell 函数必须先定义后使用。

Shell 函数的定义格式如下：

```
function_name () {
    list of commands
    [ return value ]
}
```

如果愿意，也可以在函数名前加上关键字 function：

```
function function_name () {
    list of commands
    [ return value ]
}
```

3.17.2 函数返回值

函数返回值，可以显式增加 return 语句；如果不加，会将最后一条命令运行结果作为返回值。

Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0 表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

如果一定要让函数返回字符串，那么可以先定义一个变量，用来接收函数的计算结果，脚本在需要的时候访问这个变量来获得函数返回值。

先来看一个例子：

```
#!/bin/bash

# Define your function here
Hello () {
    echo "Url is http://see.xidian.edu.cn/cpp/shell/"
}

# Invoke your function
Hello
```

运行结果：

```
$./test.sh
Hello World
$
```

3.17.3 函数参数

在 Shell 中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1 表示第一个参数，$2 表示第二个参数...

带参数的函数示例：

```
#!/bin/bash
funWithParam(){
    echo "The value of the first parameter is $1 !"
    echo "The value of the second parameter is $2 !"
    echo "The value of the tenth parameter is $10 !"
```

```
    echo "The value of the tenth parameter is ${10} !"
    echo "The value of the eleventh parameter is ${11} !"
    echo "The amount of the parameters is $# !"  # 参数个数
    echo "The string of the parameters is $* !"  # 传递给函数的所有参数
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

运行脚本：

```
The value of the first parameter is 1 !
The value of the second parameter is 2 !
The value of the tenth parameter is 10 !
The value of the tenth parameter is 34 !
The value of the eleventh parameter is 73 !
The amount of the parameters is 12 !
The string of the parameters is 1 2 3 4 5 6 7 8 9 34 73 !"
```

注意，$10 不能获取第十个参数，获取第十个参数需要 ${10}。当 n>=10 时，需要使用 ${n} 来获取参数。

另外，还有几个特殊变量用来处理参数，前面特殊变量章节中已经提到。

3.17.4 函数调用

调用函数只需要给出函数名，不需要加括号。

再来看一个带有 return 语句的函数：

```
#!/bin/bash
funWithReturn(){
    echo "The function is to get the sum of two numbers..."
    echo -n "Input first number: "
    read aNum
    echo -n "Input another number: "
    read anotherNum
    echo "The two numbers are $aNum and $anotherNum !"
    return $(($aNum+$anotherNum))
}
funWithReturn
# Capture value returnd by last command
ret=$?
echo "The sum of two numbers is $ret !"
```

运行结果：

```
The function is to get the sum of two numbers...
Input first number: 25
Input another number: 50
```

```
The two numbers are 25 and 50 !
The sum of two numbers is 75 !
```

函数返回值在调用该函数后通过 $? 来获得。

再来看一个函数嵌套的例子：

```
#!/bin/bash

# Calling one function from another
number_one () {
    echo "Url_1 is http://see.xidian.edu.cn/cpp/shell/"
    number_two
}

number_two () {
    echo "Url_2 is http://see.xidian.edu.cn/cpp/u/xitong/"
}

number_one
```

运行结果：

```
Url_1 is http://see.xidian.edu.cn/cpp/shell/
Url_2 is http://see.xidian.edu.cn/cpp/u/xitong/
```

3.17.5 终端调用函数

如果你希望直接从终端调用函数，可以将函数定义在主目录下的.profile 文件，这样每次登录后，在命令提示符后面输入函数名字就可以立即调用。

3.17.6 删除函数

像删除变量一样，删除函数也可以使用 unset 命令，不过要加上.f 选项，如下所示：

```
$unset .f function_name
```

第四章 Linux 的基本操作

4.1 软件安装

Linux 下的软件安装方法，基本分为以下几种：

- 源代码安装
- 软件管理工具 rpm/dpkg 安装
- 命令行网络在线安装
- 图形化网络在线安装

这几种方法，对新手而言，可以说由底层到高级，由复杂到简单。图形化网络在线安装是命令行网络在线安装的一种图形化表达，即有个图形化的软件来管理安装软件。一般说来，大部分常用软件都被收录到软件仓库中，使用在线安装就能搞定。需要采用源代码安装的一般是新出的软件或者个人作品。依据需要安装的软件，自行选择安装方法。不过，能通过在线安装的尽量用在线安装的方式。

4.1.1 源代码安装

从源代码安装软件是 Linux 系统中最原始也是最有效的一种方式。采用这种方式可以安装所有针对 Linux 系统开发的软件，以及绝大部分针对类 Unix 系统开发的软件。从源代码安装软件主要有以下几个步骤：

- **安装前的工作**
  - 解压源码，cd 到源码目录。
  - 检查源码中是否包含上次编译过的目标文件（*.O），用 make clean 或 make distclean 去除目标文件，保证新编译出来的可执行文件是在自己机器上编译完成的。
  - 阅读源码安装包目录下的 README 或 INSTALL 相关的文件，一般这些文件已经说清楚了怎么安装。
- **配置：./configure**

  设置参数，它被用来检查安装源码的 Linux 系统的相关软件属性，创建 Makefile 文件。这一步很关键，随后需要的安装信息都是从这一步中获取的。
- **编译：make**

  根据 Makefile 文件的指示展开编译工作，利用 gcc 将源码编译成目标文件。这些目标文件通过函数库连接产生一个完整的可执行文件。此时这个可运行文件还没有被安装到预定的目录，仍然在当前编译目录下。也就是说，在当前编译目录下虽然已经有了实际可用的软件程序，但是从这个目录下运行程序不太方便，还需安装到 Linux 系统上的常见位置上。
- **安装：sudo make install**

make 根据 Makefile 文件里面关于 install 的项目，将上一个步骤所编译完成的文件安装到预定的目录中。一般有 etc、lib、bin 和 man 等目录分别代表配置文件、函数库、执行文件、线上说明文件。由于是安装到系统目录中，所以要临时开启超级用户权限，即在 make 命令前加上 sudo。

利用源码安装软件有一定的灵活性，但是在大多情况下，利用源码安装是费力不讨好的事。这种方式安装的软件因为没有做软件相关检查会导致它依赖的其他软件不存在或这版本不正确，从而有可能无法正常运行。现在的 Linux 发行版大多有自己优秀的软件管理工具，先用软件管理工具安装，实在不行再源码安装。

4.1.2 软件管理工具 rpm/dpkg 安装

君子善假于物，由于从源码安装存在软件依赖性的问题，Linux 引入了软件管理机制进行软件包的安装、更新和卸载。软件发行商需要在固定的操作系统平台上将需要安装或升级的软件及其相关软件打包成一个特殊格式的文件。这个软件包中包含了软件包描述文件，其提供了软件包的各种信息，例如软件包的名字、版本、类别、说明摘要、创建时要执行什么指令、安装时要执行什么操作，以及软件包所要包含的文件列表等等。用户得到了软件包在系统中执行特定的命令后，软件包会根据内部的脚本检测需要的软件是否存在。若安装环境 OK 的话，安装工具就有条不紊地开展安装工作。在安装完成后安装工具还会将信息写入软件管理机制提供今后升级、删除、查询与验证。

下面用表格的形式简单说明一下 Linux 系统上最通用的两个软件管理工具 rpm 和 dpkg。

	RPM	DPKG
全称	Red Hat Package Manager(Red Hat 包管理器)	Debian Package(Debian 软件包管理器)
文件名	RPM 软件包的命令遵循名称-版本-修正版-类型这种格式。例如 Magiclinux-3.0.2-1.i386.rpm	DEB 软件包的命令遵循名称-版本-修正版-类型这种格式。例如 Magiclinux-3.0.2-1.i386.deb
软件包内容	软件包描述文件 (SPEC)、预安装脚本、多个二进制文件、安装后脚本	源码包 xxx_x.x-x.tar.gz、源代码信息总结包 xxx_x.x-x.dsc、用 dpkg 管理的完整二进制包 xxx_x.x-x-xxxx.deb、供 dput 使用的包 xxx_x.x-x-xxxx.change。 dsc 包提供了 control 文件、copyright 文件、changelog 文件、Rules 文件、Compat 文件和 dirs 文件。control 文件声明了重要的变量，dpkg 通过这些变量来管理软件包。
安装	rpm -ivh <rpm 软件包名 >，如 rpm-ivh Mysoftware-1.2-1.i386.rpm	dpkg -i <deb 软件包名 >
查询	当知道软件名称，在卸载前需要确定这个软件的完整名称时，可以使用 rpm -qa xxx* 查找 RPM 包软件。xxx 指软件名称开头的几个字母。 查找次软件向系统写入的文件列表，使用： rpm -ql <rpm 包名 >	统配符模式进行模糊查询 dpkg -l xxx* 查询系统中属于 xxx 软件的文件： dpkg –listfiles xxx
卸载	rpm -e <rpm 软件名 >，如 rpm -e Mysoftware (注意此处是软件名而不是软件包名)	dpkg -e <deb 软件名 >

这里只是简单介绍下 RPM 和 DPKG，更详细的介绍参见后续章节。需要说明的是 RPM 在被安装之前，为了避免文件被错误安装会先检查系统的硬派容量、操作系统版本等，所以 RPM 文件必须要在相同的 Linux 环境下才能够安装。为了避免我们的环境与作者打包的 Linux 环境不同的问题，很多软件作者除了提供 RPM 包外，还提供了以 ***.src.rpm 这种格式来命令的 SRPM 包。SRPM 提供了参数配置档，即 configure 与 makefile。这样虽然 SRPM 内容是源码，但仍含由所需要的相依赖软件说明、以及所有 RPM 文件所提供的数据。在安装该软件时，先将该软件以 RPM 管理的方式编译 RPM 文件，然后将编译完成的 RPM 文件安装到 Linux 系统当中。

4.1.3 命令行网络在线安装

虽然使用软件工具 rpm/dpkg 使 Linux 下软件安装简单不少，但是 dpkg/rpm 这些机制并没有完全解决软件依赖性的问题。例如在安装 PHP 软件包时，系统就可能会提示一些错误信息，指出需要其他一些软件包的支持。再去系统安装盘或网上找到这些软件安装是很麻烦的。如果要在原先已经部署好的 Linux 服务器中追加一些应用服务时，更是很难避免不出现软件包的依赖问题。于是出现了更为方便的网络在线安装策略。

很多 Linux 开发商都提供网络在线升级策略。这种策略的实施过程是系统先制作这些相依赖的软件列表，在安装某个软件包的时候，先到这个列表上去找，同时与系统已

安装的软件相比较。这样没安装的相依赖软件就能一次性全部安装起来。不同 Linux 开发商采用的工具不同，dpkg 管理机制出现了 APT 在线安装升级机制，而 RPM 则根据开发商的不同所采用工具也不一样，其中有 Red Hat 系统的 yum，SUSE 系统的 Yast Online Update(YOU)，Mandriva 的 urpm 软件等。现在来介绍下 Linux 系统上最流行的两个在线安装升级工具：apt 和 yum。

yum：Yellowdog Updater Modified 的缩写，yellowdog 是一个 Linux 的发行版。Red Hat 将这种升级技术利用到自己的发行版中形成了现在的 yum。

基本操作：

- 安装：yum install <package_name>
- 升级：yum update <package_name>
- 卸载：yum remove <package_name>
- 查询
  - 查找软件包：yum search <keyword>
  - 列出所有已安装的软件包：yum list installed
  - 获取软件包信息：yum info <package_name>
  - 列出软件包提供哪些文件：yum provides <package_name>
- 清除缓存：下载的软件包和 header 储存在 cache 中不会自动删除。使用 yum clean 完成清楚硬盘空间的工作
  - 清除 header：yum clean headers
  - 清除下载的 rpm 包：yum clean packages
  - 清楚缓存的软件包及旧 headers：yum clean all

apt：Advance Packaging Tool 的缩写。它包含以”apt-’’ 开头的多个工具：apt-get、apt-cache 和 apt-cdrom 等，在 Debian/Ubuntu 发行版中使用。

基本操作：

- 安装：apt-get install <package_name>
- 升级
  - 刷新软件源，建立更新软件包列表：apt-get update
  - 将系统中的所有软件包一次性升级到最新版本：apt-get upgrade
- 卸载
  - 卸载软件：apt-get remove <package_name>
  - 卸载软件的同时清除配置：apt-get purge remove <package_name>
- 查询
  - 查找软件包：apt-cache search <keyword> or <regular expression>
  - 获取指定软件包的详细信息：apt-cache show <package_name>
  - 获取软件包版本和软件包的依赖关系：apt-cache showpkg <package_name>
- 清除缓存
  - 清除整个软件包缓存区：apt-get clean
  - 按照依赖关系清理缓存区中多余的软件包：apt-get autoclean

yum 和 apt 的原理类似，但 apt 的执行效率要高于使用 Python 写成的 yum。既然 yum 是利用 Pthyon 脚本语言实现的，所以在 Python 升级问题上要慎重，否在会导致 yum 工作不正常，系统无法实现在线安装升级。这里只是简单介绍，更详细的介绍参见相关章节。

4.1.4 图形化网络在线安装

图形化网络在线安装是命令行网络在线安装的一种图形化表达，即使用图形化的安装升级软件。在 Ubuntu 上有专门的软件中心来管理安装软件类似苹果系统的软件商店。类似的还有 Synaptic，Fedora 也用这个，在 SUSE 和 openSUSE 系统上使用 YaST2。

4.2 实用快捷键

- Tab：这个键是最有用的键了，也应该是敲击概率最高的一个键。因为当你打一个命令打一半时，它会帮你补全的。不光是命令，当你打一个目录时，同样可以补全。
- Ctrl + L：清屏，使光标移动到第一行。
- Ctrl + D：退出当前终端，同样你也可以输入 exit。
- Ctrl + C：这个是用来终止当前命令的快捷键，当然你也可以输入一大串字符，不想让它运行直接 Ctrl + C，光标就会跳入下一行。
- Ctrl + Z：暂停当前进程，比如你正运行一个命令，突然觉得有点问题想暂停一下，就可以使用这个快捷键。暂停后，可以使用 fg 恢复它。

第二部分

Ubuntu

第五章 基本概念

5.1 软件包及软件仓库

5.1.1 Ubuntu 软件包

Ubuntu 软件包的文件格式不是常见的 RPM 格式，是 DEB 格式。DEB 格式最早是由 Debian 使用，由于 Ubuntu 是从 Debian 分支发展而来，所以也继承了这种软件包格式。DEB 软件包可以分为两类：扩展名为 DEB 的二进制软件包和扩展名为 DSC 的源码软件包。通常安装的软件只会用到二进制软件包，即 DEB 格式，而源码包只在查看、编译软件等特殊的时候才会用到。

- DEB 软件包间的依赖关系

  DEB 软件包间常见的依赖关系有：Depends、Recommends 和 Conflicts。看这三个单词的字母意思就大致明白依赖关系是怎么回事了。假设有两个 DEB 软件包 A 和 B：

  - Depends：A 依赖 B，意味着安装软件包 A 时系统必须已安装软件包 B；
  - Recommends：意味着开发这推荐用户在安装软件包 A 的同时也安装软件包 B；
  - Conflicts：意味着软件包 A 和 B 不能共存，若软件包 B 已经安装，那么安装 A 的时候可能要自动卸载掉 B。

- DEB 软件包的命名规则

  对于常见的 deb 文件，命名规则如：`gfce_ver-rev_arch.deb`

  其中 gfce 是软件包的名字；ver 代表版本；rev 代表修订版本；arch 是该软件包对应的硬件平台，如最常见的 i386。

5.1.2 软件仓库（软件源）

软件仓库（软件源）是由 Ubuntu 软件包的维护者维护并公开发布的 DEB 软件包的集合，它可位于网络，如软件包服务器、HTTP、FTP 服务器、光盘和硬盘等各种介质上。通常用户安装软件时不会自己下载 DEB 文件安装，而是由软件包管理工具根据用户的动作和配置计算依赖关系后，从软件仓库下载相应的软件包并安装。

显然软件仓库的一方面能保证大家用到安全的软件，降低了中病毒的风险；另一方面是方便大家安装使用软件，提高效率。软件仓库和苹果的软件商店等其他系统商店，本质是一样的，但后者是商业的。

需要注意的是：Ubuntu 和 Debian 都使用 DEB 软件包格式，但其软件仓库并不通用，不能把 Debian 的源设置为 Ubuntu 的源。

5.2 Ubuntu 实用工具

5.2.1 dpkg——底层软件包管理工具

dpkg 是 Ubuntu 软件包管理工具的基础，使用 dpkg 工具可以实现软件包的安装、卸载、查询、编译，以及打包应用程序等功能。但是由于当时受到互联网条件的制约，以及设计时忽视了 DEB 软件包之间复杂的依赖关系，所以 dpkg 工具无法自动解决 DEB 软件包之间的依赖关系。

dpkg 命令格式如下：

`dpkg [ -i` -r | -P | -l | -L | -s | -S] packagefilename|

这里只列出常用的，详细说明见手册（`man dpkg`），选项说明如下：

- -i 安装软件包
- -r 卸载软件包，但不删除软件包的配置文件
- -P 完全卸载软件包，包括所有相关配置文件
- -l 查看当前系统中已安装软件包的信息
- -L 查看当前系统中指定软件包的所安装的相关文件
- -s 查询已安装的指定软件包的详细信息
- -S 查询系统中某个文件所属的软件包

具体应用实例：

- 安装软件包　`$sudo␣dpkg␣-i␣packagename.deb`
- 卸载已安装的软件包　`$sudo␣dpkg␣-r␣packagename`
- 查看软件包所包含的内容　`$sudo␣dpkg␣-c␣packagename.deb`

5.2.2 alien——软件包转换工具

alien 软件包格式转换工具是一种转换软件包格式的专业工具，支持多种 linux 软件包格式之间的转换。

使用 alien 工具将 RPM 包转换为 DEB 软件包。转换后的软件包可以使用 dpkg 进行安装，具体操作如下：

```
sudo alien mplayer-1.0.6-i386.rpm
```

5.2.3 apt——高级软件包管理工具

apt（Advanced Package Tool）工具可以完成所有的软件包的管理工作，包括维护系统中的软件包数据库、自动检查软件包依赖关系、安装和升级软件包、从软件源镜像站点主动获取相关软件包等。apt 确切地说是一组命令的组合，这些命令组合又有若干参数，这样来完成各种工作。

- 安装
  - `apt-get install gfceu`，安装名为 gfceu 的软件包。在输入这条命令后，apt 系统会自动进行以下工作：

- 扫描软件包仓库列表，寻找并检查各种依赖关系，得出将要安装、升级或删除的软件包列表并从软件包镜像下载这些软件包文件；
- 先安装其依赖的软件包；
- 安装并配置软件包 gfceu。

自动解决依赖关系、自动下载减轻了传统管理 RPM 包系统用户的负担，是系统更容易使用。用户也可以在参数中附加多个软件包的名字，效果和多次重复改命令基本一致。

- `apt-get install gfceu=ver`，安装指定版本的软件包，即安装 gfceu 软件包的 ver 版本
- `apt-get install gfceu --reinstall`，重新安装 gfceu 软件包
- `apt-get install gfceu -f`，修复安装
- `apt-get source package`，下载该包的源代码
- `apt-get build-dep package`，安装相关的编译环境
- `apt-get check`，检查是否有损坏的依赖

- 升级
  - `apt-get update`，更新源
  - `apt-get upgrade`，更新已安装的包
  - `apt-get dist-upgrade`，升级系统
- 卸载
  - `apt-get remove gfceu`，删除 gfceu 软件包
  - `apt-get remove gfceu --purge`，删除包 gfceu，包括配置文件等
  - `apt-get clean`
  - `apt-get autoclean`，清理无用的包
- 查找
  - `apt-cache search keyword1 keyword2 ...`，这会搜索并列出所有描述中同时含有 keyword1 和 keyword2 等关键字的软件包
  - `apt-cache show package_name`，这是通过软件包名字来搜索
  - `apt-cache showsrc sl`，查询源码包 sl 的信息，命令返回的结果包括：版本（version）、编译依赖关系（Build-Depends）、适用计算机架构（Architecture）、镜像站点中存放位置（Directory），以及源码包中包含的文件（file）等。
  - `apt-cache depends package`，了解使用该包依赖那些包
  - `apt-cache rdepends package`，查看该包被哪些包依赖

第六章 安装及软件应用

6.1 Ubuntu Desktop

6.1.1 下载

到官网http://www.ubuntu.org.cn下载桌面版镜像，镜像有 CD 版、DVD 版，CD 版约 700 兆，正如官网描述的，DVD 版比 CD 版多是各种语言包。所以 CD 版就够了，需要的语言包等安装好下载就是了。

6.1.2 安装

官网已经说的很清楚了。我用的是方法是在 windows 下制作 U 盘安装盘。具体见官网。由于我是只装 ubuntu 系统。所以是一键完成，没出现要分区的操作，很简单。需要注意的是，安装时有问你，要不要带网络安装，带网络安装会更新一些文件和获取语言包，除非你对自己的网速很自信，那就不要带网络安装。我是带网络安装的，那个更新下载过程真是耗时呀，两个多小时，搞的退又不能退，进又很慢，只能慢慢等了。所以不要带网络安装，等装好了，再统一更新。

6.2 Fcitx

Fcitx 是一款不错的中文输入法。

- `sudo apt-get install fcitx-table-wbpy`
- System Settings $\Longrightarrow$ Language Support $\Longrightarrow$ Keyboard input method system 由默认 IBus 改成 fcitx
- reboot
- 从 Ubuntu 右上角顶栏的小键盘图标中打开，Configure。在 Input Method 选项下添加输入法。搜索 Pinyin 添加进去，就可以了。快捷键的设置，都在小键盘图标中。

6.3 uGet 下载工具

https://blog.csdn.net/u010445843/article/details/70184121

Windows 下的下载工具--迅雷，之所以下载速度快，乃是它能搜索资源、为己所用，而不是仅仅从原始地址这单一资源处下载。Ubuntu 下也有类似的工具，那就是 aira2。aira2 是一个命令行下载工具，可以配合其他图形界面的下载软件使用。我用的是 uget+aria2。uget 本身是一个小巧实用的多线程下载工具，加上 aria2 作为插件，下载速度有明显提高。

(1)uGet 主页：http://ugetdm.com。Ubuntu 安装：

```
sudo add-apt-repository ppa:plushuang-tw/uget-stable
```

```
sudo apt update
sudo apt install uget
```

(2) 安装 aria2 插件：`sudo apt install aria2`

(3) 打开 uGet 在设置中找到插件，选择 aria2。注意，重启生效。

6.4 Virtualbox

6.4.1 安装 virtualbox

我用的虚拟机软件是 virtualbox，是 SUN 公司开发的，现在 SUN 公司被收购了。还是从软件商店装，方便，安全，简单。输入名字搜索就是了。

6.4.2 创建虚拟机

创建虚拟机的过程很简单，引导说的很清楚，跟著引导一步一步走就行了。即使设置的不好，等创建完虚拟机还是可以改的。创键的出来的虚拟机相当于一台裸机。到是在这台裸机上装 xp 花了我一番功夫。关键问题在于裸机硬盘没有分区，没有格式化。我走了个弯路。我在这个虚拟机中进入 winPE 然后用瞬间分四个区。再将 xp 安装在第一个分区中。装好后，又于不需要这么多区，又将其他三个区合并了。真是个大弯，当时有点急，winPE 中肯定有分区工具，没去用。这也是安装 xp 的经验不足造成的。

6.4.3 为虚拟机启用 USB 设备

[1] 虽然虚拟机勾选了 USB 设备，但 XP 还是用不了。还要做的一件事就是将 vboxusers 添加到用户组中。说是说在 System > Administration > Users and Groups 中，可是我没有找到。其实用命令更简单：sudo gedit /etc/group 找到 vboxusers:x:125:，在其后添加自己的用户名就 ok。

6.4.4 添加增强功能

本来启动虚拟机中的 xp 后，在虚拟机的 device 中就有安装增强功能的选项。不知道为什么，我点击后报错，下载不了。没办法，自己下载，手动完成吧。在http://dlc.sun.com.edgesuite.net/virtualbox/中找到相应的版本下载下来。在这个过程中，我也郁闷惨了，增强功能的镜像有四十多兆，我下载的只有一十六兆，有问题。但是我没注意。结果老出问题，搞了很久。这个镜像要拷贝到文件系统下，/usr/share/virtualbox 文件夹中，还有一个文件夹也可以，但是这个文件夹路径较短。需要注意的是不管是哪个版本，名字一律改成，VBoxGuestAdditions.iso，安装时才回识别。由于是要拷贝到文件系统下，需要超级用户权限，方法有两个：一、用 root 用户的身份进入，方法是，没有设置 root 用户密码的，先设置 root 用户密码，在终端中，输入 sudo passwd root，按照提示给 root 指定一个密码。最后用 logout 登出当前账户，再用 root 和刚刚设置的密码就可以

[1]学习自http://www.webupd8.org/2011/02/get-your-usb-drives-to-work-with.html

了。这里的 root 就是用户名。root 用户下，就可以在文件系统中用鼠标右键的复制粘贴了。root 用户不安全，尽量避免用 root 用户。所以用，第二种方法好些。二、就是用 sudo 命令临时开启超级用户权限，用 cp 命令来复制、粘贴，命令如下：

sudo cp /home/phileas/VBoxGuestAdditions.iso /usr/share/virtualbox

其中/home/phileas/VBoxGuestAdditions.iso 是镜像的路径，

/usr/share/virtualbox 是要拷贝到的文件夹的路径，两路径间有空格。拷贝完成了，然后还是启动虚拟机中的 xp，在虚拟机的 device 中点击安装增强功能的选项，就开始安装。安装完后，会提示重启虚拟机中的 xp 才会生效。这个镜像其实是装在虚拟机 xp 中的，而不是主机中的。这也就意味着还有第三种方法（简单方法）。那就是，直接将镜像拷贝到虚拟机的 xp 中，在 xp 中解压安装。增强功能安装好后，就可以在虚拟的文件共享中，将主机 ubuntu 的文件夹共享给虚拟机中的 xp。两个系统，就通过这个共享文件夹，实现文件共享。这整个过程感觉挺有意思的，一个世界中包含另一个世界，两个世界通过一个通道相连。

6.4.5 遇到的问题

1. [2]播放视频全是发白的，但是声音没问题。
   开启了 2D 加速吧，把那个选项关了，只留 3D 加速或者 3D 加速也关了（虚拟 XP3D 加速没啥用），重启 XP 就好了。

6.5 Texlive

（1）下载镜像直接下载 DVD 光盘镜像的。不建议网络安装。官网主页:http://www.tug.org/texlive/

（2）安装

(a) 文字界面安装 文字界面也简单，在终端中运行 Perl 脚本 install-tl。这就是在终端中运行一个文件。运行一个文件，当然需要告诉电脑文件在哪里了，于是在终端输入该脚本的全路径就行了，如：

```
perl␣/home/phileas/texlive/install-tl
```

或先转到相应的目录下，再执行，如：

```
cd␣/home/phileas/texlive
perl␣install-tl
```

运行后，会出现各种选项和提示。可以看到，texlive 是装在文件系统中的，你可以更改安装的路径，或开启终端中的超级用户的权限，再来运行这个脚本。开启超级用户权限的方法是：在终端中输入命令：sudo -i。这样就可以安装了。

[2]参见：http://forum.ubuntu.com.cn/viewtopic.php?f=65&t=362895

(b) 图形界面安装 建议使用图形安装界面，因为宏包用图形界面管理很方便，过程如下：

1. `sudo␣-i` 开启超级用户权限，这是因为默认安装到文件系统中，而改动文件系统是需要超级用户权限的，也可以不开启，到时选个可以安装的路径就行了。
2. `apt-get␣install␣perl-tk` 安装 perl-tk 包才能用图形安装界面
3. 挂载镜像
   先将光盘镜像挂载到一个文件夹中，如：/home/phileas/texlive，意思是：根目录/下的 home 文件夹 phileas 中的 texlive 文件夹。先找出光盘镜像的全路径，很简单，选中你的光盘镜像右键属性中有写，这个路径或许当你拷贝进这个文件时你就已经知道了，我放的位置是，
   /home/phileas/desktop/texlive2011-20110705.iso，最后一项是该文件名。挂载方法是在终端中，敲命令：
   sudo mount /home/phileas/Desktop/texlive2011-20110705.iso/ /home/phileas/texlive ，注意两个路径之间有空格。这样就挂载好了。(注意路径区分大小写)
4. cd 挂载的镜像文件夹下，终端输入 perl install-tl - -gui
5. 有些选项，自己看着办。注意勾选建立符号链接 (symbol link)，很简单，勾选上就有了，不建立这个，之后在终端开启图形管理宏包界面的命令 tlmgr - - gui 就会报错说没有这个命令，其实还是要再加链接。所以这里记得勾选。(安装时没勾选也没关系，这部分内容在 texlive 相应版本的说明文档中说的很清楚。)
6. 点击安装
7. 如果安装时忘了勾选符号链接，则需添加环境变量。在`~/.bashrc` 文件中添加
   ```
   export PATH=$PATH:/usr/local/texlive/2018/bin/x86_64-linux
   export MANPATH=$MANPATH:/usr/local/texlive/2018/texmf-dist/doc/man
   export INFOPATH=$INFOPATH:/usr/local/texlive/2018/texmf-dist/doc/info
   ```
   然后在终端中使环境变量生效：
   ```
   source ~/.bashrc
   ```
8. 调出图形管理宏包界面，在终端输入 tlmgr - -gui，这时要改宏包更新源，在 option 选项改成网络为默认更新源。不然还是更新不了。

6.6 Texworks

6.6.1 安装

官网：https://www.tug.org/texworks/

```
sudo add-apt-repository ppa:texworks/stable
sudo apt-get update
sudo apt-get install texworks
```

6.6.2 简述

[转] http://blog.sina.com.cn/s/blog_630306a50101fjwy.html

Texworks 是目前我用的最多的 LaTeX 编辑软件，不是跟 ddswhu 交流，我还一直不知道，里面还蕴含着很多方便快捷的缩写。这里就是主要谈谈这些快捷缩写。这些缩写都在一个叫 completion 的文件中，到 Texworks 安装目录下找找。

首先，对 TeXworks 的自动补全功能解释一下：

1、在 TeXworks 的编辑窗里面键入 xa，按下 tab，出现了 `\alpha`，这就是最简单的补全，对简单命令的补全。

2、在 TeXworks 键入 usep，按下 tab 建，得到了 `\usepackage{}`，这就是最普通的补全，给出命令后的必须参数括号，并且光标停留在括号内。

3、在 TeXworks 键入 usepo，按下 tab，得到了 `\usepackage[]{}`，这是对含有可选参数的命令的补全，光标停在可选参数的中括号内，当我们把可选参数补完之后，按下 ctrl+tab 组合键，光标进入后面的必需参数括号内（后面的位置称为 placeholder）。其中 ctrl+tab 是移向往下最接近的一个 placeholder，shift+tab 是移向往上最近的一个 placeholder。

在刚才的例子中，我们只按了一次 tab，假如我们键入的引导词是若干个命令的引导词的前部分，则继续按下 tab 键会在这几个命令中切换，得到你想要的命令。

好了，为了使用自动补全，我们需要记住引导词。在 TeXworks 中，已经定义了很多的引导词，而且也允许用户自己定义新的引导词。

下面对常用的引导词归类。

6.6.3 环境类

对于环境的补全，引导词第一个字母均为 b，后面字母个数不定，但是，对绝大多数的环境，只需要使用环境名的前三个字母就行，即为"b+xyz+[tab]"。

比如 itemize 环境，根据规则，我们需要键入"bite"，然后按下 tab 键，即得到了

```
\begin{itemize}
\item

\end{itemize}
```

符合此规则的环境有 document，abstract，align，tabular，appendix，bmatrix，pmatrix，cases，description，center，equation，enumerate，eqnarray，figure，flalign，gather，item，letter，list，minipage，multiline，picture，split，subequations，theorem，titlepage，trivlist，varwidth，verbatim，等。

注意事项：如果环境名开头带有 the，则 xyz 为除去 the 之后的环境名的前三个字母。比如 bind=theindex 环境、bbib=thebibliography 环境。

另外需要注意的是：星号环境在原来引导词后加 s，即为"b+xyz+s+[tab]"，如果环境有可选项，需要使用可选项，则需要在末尾加上 o(option 的意思)，即为"b+xyz+o+[tab]"。

几个特殊的环境:

align :b+ali(s) alignat :b+ali+at(s) aligned :b+ali+ed alignedat :b+ali+edat(o)

verbatim :bver verse :bvers tabular :b+tab tabularx :b+tabx tabbing :b+tabb table :b+tabl、b+tbl (s,o,so)

居左、居右环境、居中 flushleft+flushright： b+fl+l/r \centering : cen

6.6.4 字体

（1）普通字体命令

（1.1）\textbf, \texttt, \textsf, \textsc, \textsl, \textit, \textup

方法一、由字体属性的两个关键字构成，比如 sc+[tab 键]，textit 有问题，em 表示\emph{}

方法二、\text(b/t/s/i/w...)+[tab键]

注意：\textwidth 也是 \textw

（1.2）属性的第二种表示方式、" 属性关键字 +d"

bfd: \bfseries ttd: \ttfamily sfd: \sffamily scd: \scshape sld: \slshape itd: \itshape upd: \upshape emd: \em

（2）数学字体命令：

\mathbf, \mathrm, \mathcal, \mathsf, \mathtt, \mattit

引导词为"m+ 字体属性关键字"。比如：mbf\mcal

6.6.5 希腊字母类

方法：” x+[c(大写符号)]+ 符号首字母”

适用的字母有:

\alpha, \beta , \chi, \delta, \gamma, \Gamma, \iota, \mu, \lambada, \Lambda, \mu, \nu, \omega, \Omega, \pi, \sigma, \zeta, \rho, \tau, \upsilon, \xi, \Xi

注意以下相同首字母的写法 (特殊)：

\epsilon: x+e \varepsilon: x+v+e \eta: x+et \phi :x+p \varphi :x+v+p \phi :x+ph \Phi :x+c+ph \varphi :x+v+ph \psi :x+ps \Psi :x+c+ps \tau :x+t \theta :x+th

6.6.6 章节命令

cha =\chapter{}

sec(o) =\section{}

ssec(o) =\subsection{}

sssec(o) =\subsubsection{}

6.6.7 参考文献

bbib =`\begin{thebibliography}` bibitem =`\bibitem` bibitemo =`\bibitem[]` bibstyle =`\bibliographystyle{}` biblio =`\bibliography{}`

6.6.8 杂项与普通命令

1、括号

dd : `\( \)`

d+希腊字母表达式=`\(希腊字母\)`

例如：dxa = `\(\alpha\)`

2、普通命令

usep =`\usepackage{}`

foot =`footnote foot`

frac =`\frac`

fbox =`\fbox`

fboxo =`\framebox`

href =`\href`

incg =`\includegraphics{}`

incgo =`\includegraphics[]{}`

ncol(newcolumn) = `&`

newc =`\newcommand{}{}`

newe =`\newenvironment{}{}{}`

newpg =`\newpage`

pgref =`\pageref{}`

pgs =`\pagestyle{}`

sqrt =`\sqrt{}`

toc =`\tableofcontents`

listf =`\listoffigures`

list =`\listoftables` multic =`\multicolumn{}{}{}`

6.7 Gnuplot

6.7.1 安装

(一)

注意不要直接安装，即`sudo␣apt-get␣install␣gnuplot`，通常缺少一个外界的库，容易造成 terminal set to unknown 错误，进而不能显示绘制图片。采用以下命令安装：

```
sudo␣apt-get␣install␣gnuplot-x11
```

或者分两步安装

sudo␣apt-get␣install␣libx11-dev

sudo␣apt-get␣install␣gnuplot

（二）源代码编译安装

- 先执行sudo␣apt-get␣install␣libx11-dev 安装 x11，否则不能显示图形！若不安装，则会出现，在终端下输入 gnuplot 后显示 terminal set to unknown。
- 去官网下载 gnuplot 版本并解压
- su␣root
- 切换到 gnuplot 所在目录
- 配置./configure
- 编译 make
- make install
- 运行：在终端输入 gnuplot 就行了
- 例子：plot sin(x)

6.7.2 文件设置

6.7.2.1 设置数据分隔符

```
set datafile separator ","
#设置gnuplot读取逗号分隔的数据文件
```

有时，我们的数据文件中各个数据之间是用逗号作为分隔符的，比如标准的以“CSV”（Comma-Separated Values）为后缀的数据文件。如果在逗号之后没有空格分隔，默认情况下 gnuplot 是无法直接读取的。这时可以有两种方案，第一种是提前处理一下数据文件，比如将逗号替换为空格，随便一个文本处理软件都能很轻松的做这种替换。但是有时我们有很多这样的数据文件，每个都这样处理一下也挺麻烦的。第二种方法就是在 gnuplot 中给出文件分隔符的信息，让 gnuplot 能够读懂我们的文件。

例如：

Plot␣'sample.csv'␣using␣1:2

gnuplot 会抱怨说：

```
gnuplot> plot 'sample.csv' using 1:2
                                    ^
         warning: Skipping data file with no valid points
                                     ^
         x range is invalid
```

正确的方法是这样的：

plot␣'sample.csv'␣using␣1:2␣"%lf,%lf"

格式字符串的格式与 C 语言中 scanf 的格式字符串是类似的，实际上 gnuplot 就是用的 scanf 函数来读取数据。%lf 表示按照 double 型浮点数类型来读取。需要注意的是

gnuplot 的格式化字符串不支持`%lf`。gnuplot 的格式化字符串还有很多的用法，这里就不多介绍了，有兴趣的可以参考帮助文档相关章节。

更一般的参考说明文档

```
Set datafile separator
The command set datafile separator tells gnuplot that data fields in subsequent input files
by a specific character rather than by whitespace. The most common use is to read in csv (co
value) files written by spreadsheet or database programs. By default data fields are separat
Syntax:
set datafile separator {whitespace | tab | comma | "<chars>"}
Examples:
# Input file contains tab-separated fields
set datafile separator "\t"
# Input file contains comma-separated values fields
set datafile separator comma
# Input file contains fields separated by either * or |
set datafile separator "*|"
```

6.7.2.2 重命名数据文件

```
datafile = "FMO-sink-initial1.ods"
#将数据文件命名为“datafile”，方便调用
```

6.7.3 图片设置

6.7.3.1 设置输出图片格式和名称

```
set term post eps color enh solid
#设置输出图片格式
set output "FMO-sink-initial1.eps"
#设置输出图片名称
```

其他图片格式的，有

- PS 格式：彩色（推荐使用）：

  ```
  set term postscript enhanced color
  set output "*.ps"
  ```

  另：（未测）

  ```
  set term postscript landscape
  set output "*.ps"
  ```

- EPS 格式：黑白（曲线类型控制未知）：

```
set term postscript eps enhanced
set output "*.eps"
```

忘了是啥：

```
set term post eps color solid enh
set output "*.eps"
```

- PNG 格式：

```
set terminal png size 1500,1000 enhanced font "Helvetica,20"
set output '*.png'
```

- PDF 格式需要装有 ps2pdf，然后：

```
set term postscript enhanced color
set output "| ps2pdf - *.pdf"
```

注：关于 pdf 图片，输出后周围的大片空白可以在终端中使用如下命令去除：pdfcrop *.pdf 会生成一个新文件：*-crop.pdf 使用此 pdf 文件在 Latex 等处都将十分整洁方便，妈妈再也不用担心我的图片了。

6.7.3.2 设置图片/字体大小

```
set size 1.0,1.0
#设置图片大小
```

字体大小设置比较复杂，此命令是按比例缩放图片大小，字体大小不变，相当于字体相对变大。

6.7.3.3 设置图片标题

设置/取消命名（位于图上方）

```
set title "figurename""
unset title
```

注：在多图设置（见 8 小节）中，命名命令对其后所有图生效，要在每个图中重新命名。之后不需要命名时要记得取消。

6.7.3.4 设置图注

1. 图注在图中的位置：`set␣key␣left/right␣top/bottom/center`
2. 图注增加/取消外框：`set/unset␣key␣box`
3. 图注间距：`set␣key␣spacing␣1.5`
4. 图注位于图外：`set␣key␣lmargin/rmargin/tmargin/bmargin(below)`

5. 取消图注：unset␣key

   图注设置参数一览：

```
set key {on|off} {default}
             {{inside | outside} | {lmargin | rmargin | tmargin | bmargin}
               | {at <position>}}
             {left | right | center} {top | bottom | center}
             {vertical | horizontal} {Left | Right}
             {{no}reverse} {{no}invert}
             {samplen <sample_length>} {spacing <vertical_spacing>}
             {width <width_increment>}
             {height <height_increment>}
             {{no}autotitle {columnheader}}
             {title "<text>"} {{no}enhanced}
             {{no}box { {linestyle | ls <line_style>}
                        | {linetype | lt <line_type>}
                          {linewidth | lw <line_width>}}}
     unset key
     show key
```

6.7.4 坐标轴设置

6.7.4.1 设置/取消坐标名称

```
set xlabel/ylabel "name"
unset xlabel/ylabel
```

6.7.4.2 设置坐标轴范围

- 设置给定坐标轴范围

  `set xrange/yrange [1:100]`
- 设置负向坐标 `set xrange/yrange [100:1]`
- 只设置一端坐标（另一端将自动调整）

  `set xrange/yrange [:100]`
- 恢复自动坐标轴范围（此时人工给定设置依然保留）

  `set auto x/y`
- 恢复给定坐标轴范围

  `set noauto x/y`

注：坐标轴范围，可以不设置，自动判断

6.7.4.3 设置坐标轴标识

- 设置坐标轴最大间隔（显示数值）

  `set xtics/ytics 0.5`
- 设置坐标轴小间隔

  `set mxtics/mytics 2`

  说明：将每个大间隔平分为 2 份
- 设置坐标轴刻度的精度

  `set xtics format "%.2f"`

  `set ytics format "%.0f"`

  说明："%.2f" 指的是浮点型、精确到小数点后两位
- 设置标识文本显示倾斜度

  `set xtics rotate by -45`

  说明：标识右倾 45 度
- 设置指数坐标轴

  `set log x/y`

  `set xtics 1.5`

  说明：设置指数坐标轴的间隔时，数字代表 10 的指数（待明）。注意此时坐标的值域必须大于 0。
- 移动坐标标记

  `set x/ytics [out] offset 1,1`

  "1,1" 为对应的移动向量的 x,y 值。

6.7.5 代码示例

6.7.5.1 一维绘图图

```
#!/bin/sh

# plot linear absorption spectrum
# Usage:
# ./plot_1d.sh diagonal.ods

for i in $*

do

echo "Processing $i ..."

bname=`basename ${i} .ods`

# call gnuplot to plot the 1D map
gnuplot << EOF


```

```
set term wxt enhanced
#加强的文本模式，可以输出稍复杂一点的文字标签


set datafile separator ","
#设置数据文件中数据的分隔形式，设置为逗号“，”隔开，对应常用的CSV（Comma-Separated Values）文件
datafile = "${i}"
#将数据文件命名为“datafile”，方便调用


#set term png
#set output "${bname}.png"
set term post eps color enh solid
set output "${bname}.eps"
#设置图片输出格式和名称


set title 'Linear Absorption'
set xlabel '{/Symbol w} (cm^{-1})'
set ylabel 'Absorption (a.u.)'
#设置图片的标题和x,y坐标轴的名称


#set xrange [0:500]
#set yrange [300:800]
#设置数据范围，可以不设置，自动判断


set size 1,1
#设置图片大小


set key right top
#设置图例的位置


#set xtics format "%.2f"
#set ytics format "%.0f"
#设置坐标轴刻度的精度，"%.2f"指的是浮点型、精确到小数点后两位


plot datafile using 1:2 with lines  linewidth 2 title "B820 absorption"

EOF

done
```

效果如下：

6.7.5.2 二维三维绘图

给数据加入空白行 Shell 脚本：

```
# awk -f addblanks.awk 2dspectra_random.ods > a.dat
BEGIN { FS = "," }
$1 != prev {printf "\n"; prev=$1} # print blank line
{print} # print the line
```

二维等高图代码:

```
#!/bin/sh

# plot linear absorption spectrum
# Usage:
# ./plot_1d.sh diagonal.ods

for i in $*

do

echo "Processing $i ..."

bname=`basename ${i} .ods`

awk -f addblanks.awk $i > "${bname}_blanks.dat"

# call gnuplot to plot the 1D map
gnuplot << EOF


set term wxt enhanced
#加强的文本模式，可以输出稍复杂一点的文字标签


set datafile separator ","
#设置数据文件中数据的分隔形式，设置为逗号“,”隔开，对应常用的CSV（Comma-Separated Values）文件
datafile = "${bname}_blanks.dat"
#将数据文件命名为“datafile”，方便调用

```

```

#set term png
#set output "${bname}.png"
set term post eps color enh solid
set output "${bname}.eps"
#设置图片输出格式和名称


set title 'T = 0 fs'
set xlabel '{/Symbol w_{t}} (cm^{-1})'
set ylabel '{/Symbol w}_t (cm^{-1})'
#设置图片的标题和x,y坐标轴的名称


#set xrange [0:500]
#set yrange [300:800]
#设置数据范围，可以不设置，自动判断


set size 1,1
#设置图片大小


set key right top
#设置图例的位置
#unset key


#set xtics format "%.2f"
#set ytics format "%.0f"
#设置坐标轴刻度的精度，"%.2f"指的是浮点型、精确到小数点后两位


#===================普通绘图===================#
set isosamples 50
#设置网格大小
set view 60, 30
#设置视角，默认视角是60,30,分别表示绕x轴和z轴的旋转角度。另外，一些terminal，例如wxt，支持直接用鼠标拖动图像改变视角。
set xyplane 0.2
#设置底部平面与曲面距离的相对距离
#或set xyplane at <z坐标> 设置底部平面与曲面距离的绝对距离
set hidden3d
#隐藏曲面重叠部分
#splot datafile with lines
#用线绘制3D曲面


#=================等高线绘图=================#
unset key
set contour base
set cntrparam levels incremental -1000,300,2000
splot datafile with lines title " "

unset surface
```

```
set view map
set size square
set cntrparam levels incremental -1000,300,2000
set key at screen 0.85,0.8
replot

set pm3d at b
set key at screen 1.2,1.2
set cntrparam levels incremental -1000,300,2000
set palette rgbformulae 22,13,-31
replot

EOF

#rm "${bname}_blanks.dat"

done
```

三维绘图代码:

```
#==================Pm3d绘图====================#
unset key
#pm3d绘图时, NxM的数据只能画出(N-1)x(M-1)的图像
set pm3d
#用色彩来表示不同的z值, palette-mapped 3d.
set palette rgbformulae 22,13,-31
#设置色板
replot

set pm3d at bs
#色彩图除了画在曲面上,还可以画在底部或顶部。b,s,t 三个字母分别代表底部、曲面和顶部,at 之后可以是任一个字母,也可是三个字母的任意组合。
replot
#splot datafile with lines

set pm3d map
#从上往下看数据
set size square
replot


#=================Image绘图===================#
set size square
plot datafile with image
#Image绘图与Pm3d绘图这两种方式无法简单判断优劣,只能根据实际需要选择。当像素比较多而数据又比较平滑的时候,其实两者差别不大。
```

效果如下:

T = 0 fs
2000
1500
1000
500
0
-500
-1000
0
100
200
300
400
500
600
ωτ (cm-1)
ωt (cm-1)

T = 0 fs
1.7e+03
1.4e+03
1.1e+03
800
500
200
-100
-400
ωt (cm-1)
ωτ (cm-1)

T = 0 fs
ωt (cm-1)
ωτ (cm-1)
2000
1500
1000
500
0
-500
-1000

T = 0 fs
2000
1500
1000
500
0
-500
-1000
ωτ (cm-1)
ωt (cm-1)

6.8 Intel Fortran

6.8.1 编译器的获取

对于学生，可以下载 Intel 非商业版的 fortran 和 c++ 等编译器。

下载地址：http://software.intel.com/en-us/non-commercial-software-development

新地址：https://software.intel.com/en-us/qualify-for-free-software

按要求填写邮件和地址，之后，Intel 会给该邮件发一封邮件，内有下载链接及激活的密钥。需要注意的是邮箱必需是学校的邮箱。

比如下载的 ifort 编译器为：`l_fcompxe_ia32_2013.1.117.tgz`

6.8.2 编译器的安装

- 预装 Java 运行环境；
- 预装 g++，`sudo␣apt-get␣install␣build-essential`；
- 解压安装文件，进入安装文件目录；
- 有两个安装脚本，install.sh 和 install_GUI.sh，前者是文本安装界面，后者（带 GUI）是图形安装界面，建议采用图形安装界面。安装时注意开启超级用户权限，即`sudo ./install_GUI.sh`
- 安装过程会要求输入密钥，照实输入；
- 路径设置：在`~/.bashrc` 中加入，`sudo ~/.bashrc`
  `source /opt/intel/composer_xe_2011_sp1.10.319/bin/compilervars.sh intel64`
  （不同的版本，路径或者名称不一样）
- 注销生效
- 查看是否安装成功：`which ifort`，如果成功会显示编译器路径。

6.9 Git & Github

Git 官网：http://git-scm.com

Github 官网：https://github.com

比较好的学习网站：

http://www.liaoxuefeng.com 对 Git 的作用，历史，诞生都讲的比较清楚，入门很有用！

6.10 远程登录服务器——ssh

SSH 为 Secure Shell 的缩写，由 IETF 的网络工作小组（Network Working Group）所制定；SSH 为建立在应用层和传输层基础上的安全协议。SSH 是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。利用 SSH 协议可以有效防止远程管理过程中的信息泄露问题。SSH 最初是 UNIX 系统上的一个程序，后来又迅速扩展到其他操作

平台。SSH 在正确使用时可弥补网络中的漏洞。SSH 客户端适用于多种平台。几乎所有 UNIX 平台—包括 HP-UX、Linux、AIX、Solaris、Digital UNIX、Irix，以及其他平台，都可运行 SSH。

SSH 分为客户端和服务端。服务端是一个守护进程，一般是 sshd 进程，在后台运行并响应来自客户端的请求。提供了对远程请求的处理，一般包括公共密钥认证、密钥交换、对称密钥加密和非安全连接。客户端一般是 ssh 进程，另外还包含 scp、slogin、sftp 等其他进程。客户端不是必须的。

工作机制：

1. 客户端发送一个连接请求到远程服务端；
2. 服务端检查申请的包和 IP 地址，再发生密钥给 SSH 客户端；
3. 客户端再将密钥发回服务端，自此建立连接。

6.10.1 服务端

1、安装服务器

```
sudo apt-get install openssh-server
```

2、查看服务器

```
ps -e|grep ssh
```

如果看到 sshd 则表示 sshserver 已经启动，如果只有 ssh-agent，则表示没有启动。

3、启动服务器

- `sudo service ssh start`
- `sudo /etc/init.d/ssh stop` ，# 停止
- `sudo /etc/init.d/ssh start`，# 启动
- `sudo /etc/init.d/ssh restart` ，# 重启

4、配置

修改配置文件：`sudo gedit /etc/ssh/sshd_config`，并重启服务。

- 把配置文件中的 “PermitRootLogin without-password" 加一个” #" 号，把它注释掉；再增加一句 “PermitRootLogin yes"；保存。
- ssh 默认端口是 22，Port 20，需要的话，自行修改。
- ssh 默认配置是允许 root 登录的，可以修改配置表禁止其登录：PermitRootLogin no

5、登录

`ssh` 服务器用户名@服务器地址，例如：`ssh sm06@10.125.6.30`

6.10.2 用户端

遇到更换服务器的情形，使用如下命令删除原密钥。

```
ssh-keygen -f "/home/xuanleng/.ssh/known_hosts" -R 159.226.204.95
```

冒号内为 known_hosts 文件全路径，“159.226.204.95” 为要删除的服务端地址。

6.10.3 远程文件传输

6.10.3.1 scp

在 Linux 下一般通过 scp 命令在本地机和服务器（超算）间传输文件。所以使用时一定要明确是在本地终端使用还是在超算终端使用。

1、从服务器上下载文件

（1）从本地终端下载

`scp␣username@servername:/path/filename␣.`

其意思按字面理解，其中“.”表示当前目录。例如，

```
phileas@phileas-computer:~$␣scp␣lengxuan@10.22.20.156:/home/lengxuan/1/sum_als.ods␣.
```

其中 lengxuan 为超算用户名，10.22.20.156 为超算地址。

（2）从超算终端下载

命令反过来就是了，例如：

`[lengxuan@hpc␣1]$␣scp␣sum_als.ods␣phileas@10.22.21.70:/home/phileas`

其中`[lengxuan@hpc␣1]`表示在超算用户 lengxuan 主目录下的 1 目录中。使用大致是这样，这样不方便用，因为对自己的 IP 不是很清楚。

2、上传本地文件到服务器

原理同上

3、从服务器上下载整个目录

也是分从本地终端下载和从服务器上下载两种，原理同上，和上传（下载）文件不同的是，上传（下载）目录多了一个参数，如

`scp␣-r␣root@192.168.0.101:/var/www/test␣␣/var/www/␣␣`

4、从本地上传目录到服务器

原理同上

6.10.3.2 sftp

sftp 是 Secure File Transfer Protocol 的缩写，安全文件传送协议。可以为传输文件提供一种安全的加密方法。sftp 与 ftp 有着几乎一样的语法和功能。SFTP 为 SSH 的一部分，是一种传输档案至 Blogger 伺服器的安全方式。其实在 SSH 软件包中，已经包含了一个叫作 SFTP(Secure File Transfer Protocol) 的安全文件传输子系统，SFTP 本身没有单独的守护进程，它必须使用 sshd 守护进程（端口号默认是 22）来完成相应的连接操作，所以从某种意义上来说，SFTP 并不像一个服务器程序，而更像是一个客户端程序。SFTP 同样是使用加密传输认证信息和传输的数据，所以，使用 SFTP 是非常安全的。但是，由于这种传输方式使用了加密/解密技术，所以传输效率比普通的 FTP 要低得多，如果您对网络安全性要求更高时，可以使用 SFTP 代替 FTP。

1、终端登录

`sftp` 服务器用户名@服务器地址，例如：`sftp sm06@10.125.6.30`，登入成功后终端呈现出:sftp>....

2、操作

在 sftp 的环境下的操作就和一般 ftp 的操作类似了，`ls`、`rm`、`mkdir`、`dir`、`pwd`，等指令都是对远端进行操作。如果要对本地操作，只需在上述的指令上加“`l`”变为：`lls`、`lcd`、`lpwd` 等。

3、上传与下载

- 上传：put /path/filename(本地主机) /path/filename(远端主机)；
- 下载：get /path/filename(远端主机) /path/filename(本地主机)。

6.10.4 解决 ssh 的 “Write failed: Broken pipe" 问题

问题场景

服务器环境：阿里云 Linux CentOS 主机

客户端：Mac OSX Terminal

问题现象

用 ssh 命令连接服务器之后，如果一段时间不操作，再次进入 Terminal 时会有一段时间没有响应，然后就出现错误提示：

```
Write failed: Broken pipe
```

只能重新用 ssh 命令进行连接。

解决方法

方法一：如果您有多台服务器，不想在每台服务器上设置，只需在客户端的 `~/.ssh/` 文件夹中添加 config 文件，并添加下面的配置：

```
ServerAliveInterval␣60
```

方法二：如果您有多个人管理服务器，不想在每个客户端进行设置，只需在服务器的 `/etc/ssh/sshd_config` 中添加如下的配置：

```
ClientAliveInterval␣60
```

方法三：如果您只想让当前的 ssh 保持连接，可以使用以下的命令：

```
$␣ssh␣-o␣ServerAliveInterval=60␣user@sshserver
```

在 linux 下终端用超算也会遇到同样的问题，按上述方法解决。

6.11 Linux 作业管理系统——PBS

PBS（Portable Batch System）最初由 NASA 的 Ames 研究中心开发，主要为了提供一个能满足异构计算网络需要的软件包，用于灵活的批处理，特别是满足高性能计算的需要，如集群系统、超级计算机和大规模并行系统。PBS 的主要特点有：代码开放，免费获取；支持批处理、交互式作业和串行、多种并行作业，如 MPI、PVM、HPF、MPL。PBS 是功能最为齐全，历史最悠久，支持最广泛的本地集群调度器之一。PBS 的目前包括 openPBS、PBS Pro 和 Torque 三个主要分支。其中 OpenPBS 是最早的 PBS 系统，目前已

经没有太多后续开发；PBS pro 是 PBS 的商业版本，功能最为丰富；Torque 是 Clustering 公司接过了 OpenPBS，并给与后续支持的一个开源版本。

6.11.1 Ubuntu14.04 安装配置 Torque

[转]http://blog.sina.com.cn/s/blog_4a0a8b5d0102v2p1.html

其实 Ubuntu 有编译好的 Torque，只需要安装 torque-server torque-mom torque-scheduler，再加些配置即可。但是发现它弄得 scp 秘钥有问题，配置了半天 rsa 秘钥也没有解决日志移位的问题。所以还是自己编译吧！

1、下载 Torque 源代码

http://www.adaptivecomputing.com/support/download-center/torque-download/

用最新版本就可以了，这里以 torque-4.1.7.tar.gz 为例。

2、解压并编译

```
tar -xzvf torque-4.1.7.tar.gz

cd torque-4.1.7

./configure --prefix=/usr/local/torque
```

提示缺少 openssl-dev 和 libxml2-dev 之类的，补上它们……

configure: error: TORQUE needs lib openssl-devel in order to build

输入 sudo apt-get install libssl-dev

configure: error: TORQUE needs lib libxml2-devel in order to build

输入 sudo apt-get install libxml2-dev

……

1) 出现问题

```
configure: error: TORQUE needs libxml2-devel in order to build
```

解决：

```
./configure LDFLAGS="-L/home/xuanleng/anaconda3/lib" --prefix=/usr/local/torque
```

2) 出现问题

```
libtool: Version mismatch error. This is libtool 2.2.6 Debian-2.2.6a-4, but the
libtool: definition of this LT_INIT comes from libtool 2.2.6b.
libtool: You should recreate aclocal.m4 with macros from libtool 2.2.6 Debian-2.2.6a-4
libtool: and run autoconf again.
make[2]: *** [wktools4] Error 63
make[2]: Target 'all' not remade because of errors.
make[1]: *** [all-recursive] Error 1
make: *** [all] Error 2
*** Exited with status: 2 ***
```

解决：

```
autoreconf -fi  configure.ac
```

直至./configure –prefix=/usr/local/torque 没有提示缺少库，ready to make 为止。

像这样提示：

```
Building components: server=yes mom=yes clients=yes
                     gui=no drmaa=no pam=no
PBS Machine type    : linux
Remote copy         : /usr/bin/scp -rpB
PBS home            : /var/spool/torque
Default server      : rccm
Unix Domain sockets :
Linux cpusets       : no
Tcl                 : disabled
Tk                  : disabled
```

```
make

sudo make install
```

3. 设置环境变量并刷新

刷新环境变量需要注意时效性，如果 root 或 sudoer 退出终端，在没有重启机器的前提下，那么还是要刷新下的，不然可能会提示没有 trqauthd 之类的错误。

```
sudo vi /etc/profile
```

添加

```
#Torque
export PATH=/usr/local/torque/bin:/usr/local/torque/sbin:
    $PATH
```

刷新环境变量

```
sudo -s
source /etc/profile
```

4.、安装

需管理员权限，仍然在 torque-4.1.7 文件夹下。

```
sudo ./torque.setup root
```

如果出现：

```
mxio@Node1:~/Downloads/torque-4.1.7$ sudo ./torque.setup root
./torque.setup: 1: ./torque.setup: trqauthd: not found
trqauthd failed to start!!! exiting setup
```

错误，那么请检查第三步并刷新 source /etc/profile。

出现类似下面的成功：

```
pbs_server port is: 15001
trqauthd daemonized - port 15005
trqauthd successfully started
initializing TORQUE (admin: root@Node1)
You have selected to start pbs_server in create mode.
If the server database exists it will be overwritten.
do you wish to continue y/(n)?y
root       495     1  1 17:40 ?        00:00:00 pbs_server -t create
Max open servers: 9
Max open servers: 9
```

5、配置

需管理员权限

```
sudo -s
```

查看计算机名

```
hostname
```

输出 Node1 （服务器计算机名）

```
vi /etc/hosts
```

将计算机名添加进 hosts，我选择注释掉 127.0.1.1 Node1，改成：

```
127.0.0.1        Node1 localhost
#127.0.1.1      Node1
```

进入 torque 主目录进行环境变量设置：

```
cd /var/spool/torque
vi server_priv/nodes
```

添加：

```
Node1 np=32
```

即计算机名和 CPU 数目，这个视具体情况而定。

```
vi mom_priv/config
```

添加：

```
$pbs_server = 127.0.0.1
```

```
vi server_name
```

添加：

```
Node1
```

6、启动 client daemon

```
pbs_mom
```

7、重启 pbs server daemon

```
qterm
pbs_server
```

8、启动 scheduler daemon

```
pbs_sched
```

9、检查服务是否正确启动

```
ps -aux | grep pbs #check all is running
qstat -q #check the presence of the queue
qmgr -c 'p s' #check server & queue settings
pbsnodes -a  #check if the nodes are listed and u
```

10、配置列队

```
qmgr -c "set queue batch resources_default.walltime = 360:00:00"
qmgr -c "set server query_other_jobs = True"
qmgr -c "set queue batch resources_max.ncpus=32"
```

11. 测试列队

首先退出 root

```
exit
```

```
source /etc/profile
echo "sleep 30" | qsub
qstat
```

12、配置开启启动

cd 到 torque-2.4.6/contrib/init.d 目录下

```
sudo -s
cp debian.pbs_mom /etc/init.d/pbs_mom && update-rc.d
   pbs_mom defaults
cp debian.pbs_server /etc/init.d/pbs_server && update-rc.d
    pbs_server defaults
cp debian.pbs_sched /etc/init.d/pbs_sched && update-rc.d
   pbs_sched defaults
cp debian.trqauthd /etc/init.d/trqauthd && update-rc.d
   trqauthd defaults
```

注，请检查 DAEMON 是否为`/usr/local/torque/sbin/$NAME`，不是的话请修改。

13、重启计算机

```
echo "sleep 30" | qsub
qstat
```

输出：

```
Job ID                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2.Node1                    STDIN            mxio                   0 Q batch
```

成功搞定。

6.11.2 重要说明

第二次安装开机自起没有配置成功。需要手动开启服务。进入 root 权限，依次输入如下命令：

`service pbs_server restart`

`service pbs_mom server restart`

`pbs_sched`

`trqauthd`

6.11.3 PBS 命令

PBS 提供 4 条命令用于作业管理。

(1) qsub 命令—用于提交作业脚本

命令格式：

```
qsub [-a date_time] [-c interval] [-C directive_prefix]
[-e path] [-I] [-j join] [-k keep] [-l resource_list] [-m mail_options]
[-M user_list][-N name] [-o path] [-p priority] [-q destination] [-r c]
[-S path_list] [-u user_list][-v variable_list] [-V]
[-W additional_attributes] [-z]
[script]
```

参数说明：因为所采用的选项一般放在 pbs 脚本中提交，所以具体见 PBS 脚本选项。

例：qsub aaa.pbs 提交某作业，系统将产生一个作业号

(2) qstat 命令—用于查询作业状态信息

命令格式：`qatat␣[-f][-a][-i]␣[-n][-s]␣[-R]␣[-Q][-q][-B][-u]`

参数说明：

- -f jobid 列出指定作业的信息
- -a 列出系统所有作业
- -i 列出不在运行的作业
- -n 列出分配给此作业的结点
- -s 列出队列管理员与 scheduler 所提供的建议
- -R 列出磁盘预留信息
- -Q 操作符是 destination id，指明请求的是队列状态
- -q 列出队列状态，并以 alternative 形式显示
- -au userid 列出指定用户的所有作业
- -B 列出 PBS Server 信息
- -r 列出所有正在运行的作业
- -Qf queue 列出指定队列的信息
- -u 若操作符为作业号，则列出其状态。

若操作符为 destination id，则列出运行在其上的属于 user_list 中用户的作业状态。

例：qstat -f 211 查询作业号为 211 的作业的具体信息。

(3) qdel 命令—用于删除已提交的作业

命令格式：

`qdel␣[-W␣间隔时间]␣作业号`

命令行参数：

例：qdel -W 15 211

15 秒后删除作业号为 211 的作业

(4) qmgr 命令—用于队列管理

qmgr -c "create queue batch queue_type=execution"

qmgr -c "set queue batch started=true"

qmgr -c "set queue batch enabled=true"

qmgr -c "set queue batch resources_default.nodes=1"

qmgr -c "set queue batch resources_default.walltime=3600"

qmgr -c "set server default_queue=batch"

6.11.4 PBS 脚本文件

PBS 脚本文件由脚本选项和运行脚本两部分组成。

(1) PBS 作业脚本选项（若无-C 选项，则每项前面加‘#PBS’）

- -a date_time： date_time 格式为：[[[[CC]YY]MM]DD]hhmm[.SS] 表示经过 date_time 时间后作业才可以运行。
- -c interval： 定义作业的检查点间隔，如果机器不支持检查点，则忽略此选项。
- -C directive_prefix：在脚本文件中以 directive_prefix 开头的行解释为 qsub 的命令选项。（若无此选项，则默认为’#PBS’）
- -e path： 将标准错误信息重定向到 path
- -I： 以交互方式运行
- -j join： 将标准输出信息与标准错误信息合并到一个文件 join 中去。
- -k keep：定义在执行结点上保留标准输出和标准错误信息中的哪个文件。keep 为 o 表示保留前者，e 表示后者，oe 或 eo 表示二者都保留，n 表示皆不保留。若忽略此选项，二者都不保留。
- -l resource_list： 定义资源列表。以下为几个常用的资源种类。
  - cput=N： 请求 N 秒的 CPU 时间; N 也可以是 hh:mm:ss 的形式。
  - mem=N[K|M|G][B|W]： 请求 N kilo|mega|gigabytes|words 大小的内存。
  - nodes=N:ppn=M： 请求 N 个结点，每个结点 M 个处理器。
- -m mail_options： mail_option 为 a：作业 abort 时给用户发信；为 b：作业开始运行发信；为 e：作业结束运行时发信。若无此选项，默认为 a。
- -M user_list： 定义有关此作业的 mail 发给哪些用户。
- -N name： 作业名，限 15 个字符，首字符为字母，无空格。
- -o path： 重定向标准输出到 path。
- -p priority： 任务优先级，整数，[-1024，1023]，若无定义则为 0。
- -q destination： destination 有三种形式：queue , @server,queue@server。
- -r y|n： 指明作业是否可运行，y 为可运行，n 为不可运行。
- -S shell： 指明执行运行脚本所用的 shell，须包含全路径。
- -u user_list： 定义作业将在运行结点上以哪个用户名来运行。
- -v variable_list： 定义 export 到本作业的环境变量的扩展列表。
- -V： 表明 qsub 命令的所有环境变量都 export 到此作业。
- -W additional_attributes： 作业的其它属性。
- -z： 指明 qsub 命令提交作业后，不在终端显示作业号。

(2) 运行脚本同 LINUX 下一般的运行脚本文件。[注]：脚本文件中的 mpirun_rsh 命令行中的节点列表文件要用环境变量表示 $PBS_NODEFILE，这个环境变量表示由 pbs 自动分配给作业的节点列表；节点数为命令行中指定的进程数。格式如下：mpirun_rsh –np 进程数–hostfile $PBS_NODEFILE 可执行程序名

6.11.5 设置最大队列数

```
set value: max_running
# qmgr
Max open servers: 4
Qmgr: set queue sort max_running = 20
Qmgr: print server
.................
Qmgr: quit
check new setting:
# qmgr -c "print server" | grep max_run
set queue short max_running = 20
set queue long max_running = 20
set queue infinite max_running = 20
set queue cert max_running = 20
```

```
qmgr -c "set queue batch max_running=20"
```

6.12 Linux 作业管理系统——LSF

https://en.wikipedia.org/wiki/Platform_LSF

Platform Load Sharing Facility (or simply LSF) is a workload management platform, job scheduler, for distributed high performance computing. It can be used to execute batch jobs on networked Unix and Windows systems on many different architectures.[1][2] LSF was based on the Utopia research project at the University of Toronto.[3]

In 2007, Platform released Platform Lava, which is a simplified version of LSF based on an old version of LSF release, licensed under GNU General Public License v2.[4] The project was discontinued in 2011, succeeded by OpenLava.

In January, 2012, Platform Computing was acquired by IBM.[5]

https://www.ibm.com/support/knowledgecenter/en/SSWRJV/product_welcome_spectrum_lsf.html

6.12.1 常用命令

- bhosts
- bjobs -u all, bjobs -l
- bqueues

第七章 操作配置

7.1 使用注意

7.1.1 U 盘移动硬盘退出

要从侧边栏，safely remove，退出。在侧边栏退出才是真的退出，退出时注意看数据灯。

7.2 快捷键

1. 退出终端：ctrl+D
2. 终止当前命令：ctrl+C
3. 显示或隐藏隐藏文件：ctrl+H
   ctrl 是 control 的缩写，是控制的意思；h 是 hide 的首字母；翻译过来就是：控制隐藏的意思。
4. 呼出终端：ctrl +alt +t
   终端的英文是：terminal，这个 t 是英文首字母
5. 锁定屏幕：ctrl+ alt+l
   lock screen 的首字母

7.3 root 身份登入

输入 root 登录，结果提示没有 root 这个用户。查询资料得知 Ubuntu 默认没有开启的，首先需要指定一个密码来开启。用原来的用户名和密码登录，然后输入 sudo passwd root，按照提示给 root 指定一个密码。最后用 logout 登出当前账户，再用 root 和刚刚设置的密码就可以了。

很多时候，无需用 root 身份登陆系统，只需临时开启超级用户权限就可。临时开启超级用户权限很简单，命令 `sudo␣-i`，就行。

7.4 gedit 中文乱码的问题

在中文支持配置还不完整的 Ubuntu 14.04 中，使用 gedit 打开带有中文字符的文件有时会出现乱码的情况，这是由于 gedit 对字符编码匹配不正确导致的，解决方法如下：

在终端中输入如下命令，然后重新打开 gedit 即可：

gsettings set org.gnome.gedit.preferences.encodings auto-detected "['GB18030', 'GB2312', 'GBK', 'UTF-8', 'BIG5', 'CURRENT', 'UTF-16']"

gsettings set org.gnome.gedit.preferences.encodings shown-in-menu "['GB18030', 'GB2312', 'GBK', 'UTF-8', 'BIG5', 'CURRENT', 'UTF-16']"

7.5 修改 host 文件

Google 被封杀了，修改 host 文件是能正常使用 Google 的简单方法。思路很简单，找到 Linux 下的 host 文件，编辑它，将网上搜索到的指向 Google 的 host 粘贴进去即可。

- `cd␣/etc`
- `sudo␣gedit␣hosts`
- 最后一行加入网上搜索到的 host
- 保存，退出

更快捷的命令是 `sudo␣gedit␣/etc/hosts`

7.6 Linux 下查看文件编码及修改编码

7.6.1 查看文件编码

在 Linux 中查看文件编码可以通过以下几种方式：

1、查看文件编码 file 命令

file ip.txt

ip.txt: UTF-8 Unicode text, with escape sequences

2、在 Vim 中可以直接查看文件编码

:set fileencoding 即可显示文件编码格式。

如果你只是想查看其它编码格式的文件或者想解决用 Vim 查看文件乱码的问题，那么你可以在 ~/.vimrc 文件中添加以下内容：

set encoding=utf-8 fileencodings=ucs-bom,utf-8,cp936

这样，就可以让 vim 自动识别文件编码（可以自动识别 UTF-8 或者 GBK 编码的文件），其实就是依照 fileencodings 提供的编码列表尝试，如果没有找到合适的编码，就用 latin-1(ASCII) 编码打开。

3、enca 查看文件编码 enca filename

filename: Universal transformation format 8 bits; UTF-8

CRLF line terminators

需要说明一点的是，enca 对某些 GBK 编码的文件识别的不是很好，识别时会出现：Unrecognized encoding

7.6.2 文件编码转换

1、在 Vim 中直接进行转换文件编码, 比如将一个文件转换成 utf-8 格式

:set fileencoding=utf-8

2、enconv 转换文件编码，比如要将一个 GBK 编码的文件转换成 UTF-8 编码，操作如下

```
enconv -L zh_CN -x UTF-8 filename
```

3、iconv 命令用于转换指定文件的编码，默认输出到标准输出设备，亦可指定输出文件。

用法：iconv [选项...] [文件...]

有如下选项可用：

- 输入/输出格式规范
  - -f, –from-code= 名称原始文本编码
  - -t, –to-code= 名称输出编码
- 信息
  - -l, –list 列举所有已知的字符集
- 输出控制
  - -c 从输出中忽略无效的字符
  - -o, –output=FILE 输出文件
  - -s, –silent 关闭警告
  - –verbose 打印进度信息
  - -?, –help 给出该系统求助列表
  - –usage 给出简要的用法信息
  - -V, –version 打印程序版本号

例子：iconv -f utf-8 -t gb2312 aaa.txt >bbb.txt

这个命令读取 aaa.txt 文件，从 utf-8 编码转换为 gb2312 编码，其输出定向到 bbb.txt 文件。

7.6.3 文件名编码转换

因为现在用 linux, 原来在 windows 里的文件都是用 GBK 编码的。所以 copy 到 linux 下是乱码，文件内容可以用 iconv 来转换可是好多中文的文件名还是乱码，找到个可以转换文件名编码的命令，就是 convmv。

convmv 命令详细参数例如 convmv -f GBK -t UTF-8 *.mp3

不过这个命令不会直正的转换，你可以看到转换前后的对比。如果要直正的转换要加上参数 –notest

convmv -f GBK -t UTF-8 –notest *.mp3

-f 参数是指出转换前的编码，-t 是转换后的编码。这个千万不要弄错了。不然可能还是乱码哦。还有一个参数很有用。就是 -r 这个表示递归转换当前目录下的所有子目录。

* 需要安装 convmv-1.10-1.el5.noarch.rpm

三、更好的傻瓜型命令行工具 enca，它不但能智能的识别文件的编码，而且还支持成批转换。

1. 安装

sudo apt-get install enca

2. 查看当前文件编码

`enca -L zh_CN ip.txt` Simplified Chinese National Standard; GB2312 Surrounded by/intermixed with non-text data

3. 转换

命令格式如下　enca -L 当前语言 -x 目标编码文件名

例如要把当前目录下的所有文件都转成 utf-8

`enca -L zh_CN -x utf-8 *`

检查文件的编码　`enca -L zh_CN file`

将文件编码转换为"UTF-8" 编码　`enca -L zh_CN -x UTF-8 file`

如果不想覆盖原文件可以这样　`enca -L zh_CN -x UTF-8 < file1 > file2`

7.7 字体

查看字体

1. 查看所有字体：`fc-list`
2. 查看中文字体：`fc-list␣:lang=zh-cn`

安装字体

方法一：

ubuntu11.10 中有个字体查看器，用它打开字体，有个安装按钮，直接点击就 ok。

方法二：

在软件中心安装 Fontmtraix，用这个软件装。

方法三：

1. 在自己的用户目录下，如我的/home/phileas 也可以在地址栏输入`~/`，就直接转到用户目录下，建立隐藏文件夹`./fonts`。
2. 将字体文件`***.ttf` 放入该文件夹。
3. 在终端输入`sudo mkfontscale`。
4. 在终端输入`sudo mkfontdir`。
5. 在终端输入`fc-cache -f -v`。
6. 重启电脑。

7.8 中文字号制与点数制

1886 年全美活字铸造协会以派卡（pica）为基准制定派卡点数制，规定 1pica=12point（点，俗称“磅”），即：

$$\boxed{1\text{ 英寸 (in)} = 2.54\text{ 厘米 (cm)} = 6\text{ 派卡 (pc)} = 72\text{ 点 (pt，点是意译，磅是音译)}}$$

$$\boxed{1\text{ 点} = 0.013837\text{ 英寸} = 0.35146\text{ 毫米}}$$

派卡是英文印刷中正文文字的标准大小，当文字大小（高度）是 1 派卡（12 磅）时，1 英寸的宽度上正好可以排下 10 个英文文字。

20 世纪初派卡点数制传入我国，并得到逐步推广。在实用中对常用点数以号数命名而产生了号数制，二者换算如下（以 pt 代表“点”）：

初号 = 42pt
小初号 = 36pt
一号 = 28pt
二号 = 21pt
小二号 = 18pt
三号 = 15.75pt
四号 = 14pt
小四号 = 12pt
五号 = 10.5pt
小五号 = 9pt
六号 = 7.875pt
七号 = 5.25pt

7.9 Win7+Ubuntu 双系统时间不一致

[转]http://blog.sina.com.cn/s/blog_55546df90100xkf3.html

装了 ubuntu 和 win7 双系统，发现每次进入 win7 后时间总是不对，总是比当地时间晚 8 个小时，每次在 win7 下调整好之后，但是再一次进入 win7 系统后，时间又变回去了。网上搜索一下原因，原来是两个系统读取时间的机制不一样。现在来具体的说一下原因和解决办法：

原因所在：

UTC 即 Universal Time Coordinated，协调世界时

GMT 即 Greenwich Mean Time，格林尼治平时

Windows 与 Mac/Linux 缺省看待系统硬件时间的方式是不一样的：

Windows 把系统硬件时间当作本地时间 (local time)，即操作系统中显示的时间跟 BIOS 中显示的时间是一样的。

Linux/Unix/Mac 把硬件时间当作 UTC，操作系统中显示的时间是硬件时间经过换算得来的，比如说北京时间是 GMT+8，则系统中显示时间是硬件时间 +8。

这样，当 PC 中同时有多系统共存时，就出现了问题。假如你的 ubuntu 设置的时区都为北京时间东八区，当前系统时间为 9：00AM。则此时硬件中存储的实际是 UTC 时间 1:00AM。这时你重启进入 Windows 后，你会发现 windows 系统中显示的时间是 1:00AM，比 ubuntu 中慢了八个小时。同理，你在 Windows 中更改或用网络同步了系统时间后，再到 Ubuntu 中去看，系统就会快了 8 小时。在实行夏令时的地区，情况可能会更复杂些。

解决方法显然有两种。第一种是在 windows 下修改，第二种是在 Ubuntu 下修改。觉得在 windows 下修改很麻烦，也没兴趣。这里只讲 Ubuntu 下的修改方法。

ubuntu 默认开启 UTC, 即协调世界时，而 win7 是使用这种计时方式，这将导致的结果就是 Windows 和 Ubuntu 时间计算有差异。可以使用以下方法得到一致的时间：

sudo gedit /etc/default/rcS

找到这一行：UTC=yes

把 yes 改为 no

7.10 Ubuntu 开机让数字小键盘灯起来

在 Windows 开机后，数字键盘灯是亮着的，但是当切换到 Ubuntu 系统后登录用户名和密码时，如果你设定的有数字，都要先打开数字键盘区 NUMLOCK 键，然后再输入了，很不方便。

- `sudo␣apt-get␣install␣numlockx`
- `sudo␣gedit␣/etc/lightdm/lightdm.conf`
- 最后一行加入：greeter-setup-script=/usr/bin/numlockx on
- 保存，退出

7.11 Ubuntu 安装完找不到 Windows 了

sudo updata-grub

7.12 Chromium 浏览器 flash 插件问题

一个 Pepper Flash Player For Chromium 的安装器已经被 Ubuntu 14.04 的官方源收录。

Flash Player For Linux 自 11.2 起已经停止更新，目前 Linux 平台下面的 Flash Player 只能依靠 Google Chrom 的 PPAPI（Pepper Flash Player）进行更新（Chrome Only）（Adobe 仅维护这个版本），其它浏览器包括 Chromium 都只能使用 Flash Player 11.2。

但由于 Chromium 宣布将抛弃旧的标准（NPAPI），导致原本的 Flash Player 将无法在 Chromium 运行，所以决定在 Chromium 中使用 Pepper Flash Player ，这个 Pepper Flash Player 是通过下载 Google Chrome 然后提取出来给 Chromium 使用的。

目前这个安装器已经收录于 Ubuntu 14.04 官方源（从 Debian 源中导入）。Ubuntu 14.04 用户可以通过以下命令安装 Pepper Flash Player For Chromium ：

sudo apt-get install pepperflashplugin-nonfree

sudo update-pepperflashplugin-nonfree –install

如果你想使用 Beta 版的 Google Chrome 中的 Pepper Flash Player ，那么可以把第二个命令改为：

sudo update-pepperflashplugin-nonfree –install –beta –unverified

如果想使用非稳定版的 Google Chrome 中的 Pepper Flash Player，那么可以把第二个命令改为：

sudo update-pepperflashplugin-nonfree –install –unstable –unverified

如果你想卸载这个 Flash Player ，那么请执行以下命令：

sudo update-pepperflashplugin-nonfree –uninstall

7.13 Chromium 浏览器插件管理

在地址栏里输入：`chrome://plugins/`，就能访问浏览器的插件页。

7.14 右键打开当前路径下的终端

```
sudo apt-get install nautilus-open-terminal
```

`nautilus -q` 就可以右键打开当前路径下的终端。

7.15 pdf 的剪切、合并、裁边

Evince 和 Okular 是 Linux 下两款不错的 PDF 阅读器。PDF Mod 和 PDF-Shuffler 则是两款图形化 pdf 剪切、合并软件，但遗憾的是，有些 PDF 它们就是合并不了。然后有些时候，pdf 白边过多，要进行裁边也是件麻烦的事。

7.15.1 pdf 合并

```
pdfunite 1.pdf 2.pdf 3.pdf 4.pdf all-1234.pdf
```

最后为合并输出文件。简单好用。

7.15.2 pdftk

如果 PDF 是一张电子纸，pdftk 就是一个印戳涂抹器、打孔机、浆糊、显影液、和一个 X 光玻璃。Pdftk 是一个简单的 PDF 万用工具，使用它，你可以：合并 PDF 文档、分割 PDF、旋转 PDF 页面、解密 PDF 密码、加密 PDF、使用 FDF Data 或者 XFDF 来填写 PDF 窗体、添加水印或者标签、显示 PDF 信息、修改 PDF 信息、附加文件到 PDF 页面或者 PDF 文档、解压 PDF 附件、压缩 pdf 附件、分解 PDF 文档成单页形式、解压和重新压缩 PDF 流、修复受损的 PDF 文档、分解 PDF 到文本。

pdftk 让你轻松管理你的 PDF 文档，并且是免费的，可以在 Windows、Linux MACOSX、FreeBSD 和 Solaris 上使用。

pdftk 是款命令行工具，使用举例如下：

- 合并 PDF：

```
pdftk 1.pdf 2.pdf 3.pdf cat output 123.pdf
```

或者 (使用通配符):

```
pdftk *.pdf cat output combined.pdf
```

- 把多个 PDF 的不同页面组合成一个新的 PDF 文档:

```
pdftk A=one.pdf B=two.pdf cat A1-7 B1-5 A8 output combined.pdf
```

- 旋转 PDF 第一页 90 度:

```
pdftk in.pdf cat 1E 2-end output out.pdf
```

- 选择所有 PDF 页面 180 度:

```
pdftk in.pdf cat 1-endS output out.pdf
```

- 使用 128 强度加密 PDF:

```
pdftk mydoc.pdf output mydoc.128.pdf owner_pw foopass
```

同上，同时给 PDF 加上访问密码:

```
pdftk mydoc.pdf output mydoc.128.pdf owner_pw foo user_pw baz
```

同上，但是运行打印:

```
pdftk mydoc.pdf output mydoc.128.pdf owner_pw foo user_pw baz allow printing
```

- 解除 PDF 文档密码 (foopass 替换成 pdf 的密码)：注意：前提是你得知道 pdf 的密码所以此功能只是解除不需要输密码

```
pdftk secured.pdf input_pw foopass output unsecured.pdf
```

- 合并两个 PDF 文档，其中一个是加密的，但最终文档不加密:

```
pdftk A=secured.pdf mydoc.pdf input_pw A=foopass cat output combined.pdf
```

- 解压 PDF 流，以便文本编辑:

```
pdftk mydoc.pdf output mydoc.clear.pdf uncompress
```

- 压缩 PDF:

```
pdftk mydoc.pdf output mydoc.clear.pdf compress
```

- 修复 PDF 文档:

```
pdftk broken.pdf output fixed.pdf
```

- 分解成单页 (将文件拆分成一个个的单页):

```
pdftk mydoc.pdf burst
```

- 只拆分出首页:

```
pdftk mypdf.pdf cat 1 output mypdf_1.pdf
```

- 报告 PDF 信息，输出到文本:

```
pdftk mydoc.pdf dump_data output report.txt
```

- 切割 pdf:

```
pdftk input.pdf cat 1-4 output page1to4.pdf
```

以上命令就可以将 input.pdf 的 1 至 3 页输出为 page1to4.pdf 这个文件。

pdftk 使用起来很简单，manpage 也写的很详细。不过目前好像对中文文件名的支持有点问题。

7.15.3 pdf 裁边

我在用的是 pdfcrop，用法很简单：`pdfcrop test.pdf`，就自动裁边了。网友还推荐 briss，感觉不错，但没用过。

7.15.4 pdf 转 eps

```
pdftops -eps 2des.pdf a.eps
```

7.15.5 eps 转 pdf

```
epstopdf a.eps
```

7.15.6 pdf 转图片

The convert program is a member of the ImageMagick suite of tools. Use it to convert between image formats as well as resize an image, blur, crop, despeckle, dither, draw on, flip, join, re-sample, and much more. This is also useful if you do not have PDF reader installed (Gnome and KDE does have in built PDF reader) or required for your webbased project.

Type the following command to convert foo.pdf to foo.png (foo1.png, foo2.png.. etc if you have multiple pages in a pdf file):

$ convert foo.pdf foo.png

You can specify a different file type by changing the file extension of the second file, type:

$ convert foo.pdf foo.jpg

7.16 合并或复制用户到新的系统

http://www.cyberciti.biz/faq/howto-move-migrate-user-accounts-old-to-new-server/

机器换代，把老机器的用户及相关信息复制到新的机器上要比一个个新建用户，然后让用户自行修改密码要省很多事。

首先最重要的是三个文件：`/etc/passwd`、`/etc/group`、`/etc/shadow`；这三个文件分别存放了用户信息、用户组信息和用户密码。有时候可能还涉及到`/etc/gshadow`、`/var/spool/mail` 文件，没有涉及这些功能就不管它了。其次是用户目录，home 目录：`/home`。这些构成了用户的所有信息。

第一步：处理 passwd、group、shadow 三个文件。

比起将这三个文件直接覆盖新机器中的做法，将这三个文件中的用户信息筛选出来添加到新机器的文件中，后者更稳妥些。

筛选方法如下：

（1）Linux 的用户和用户组是通过分配一个具体的整数数值作为 ID 来识别的。首先明确用户和用户组的 ID，RHEL/CentOS/Fedora Core 默认的用户和用户组的 ID 范围是：

500 ∼ 65534（/etc/libuser.conf）；Debian 和 Ubuntu Linux 默认的用户和用户组的 ID 范围是：1000 ∼ 29999（/etc/adduser.conf）。这个并不是固定的，视具体情况而定。

（2）几个文件的处理命令如下：

```
awk -v LIMIT=500 -F: '($3>=LIMIT) && ($3!=65534)' passwd > passwd.mig
awk -v LIMIT=500 -F: '($3>=LIMIT) && ($3!=65534)' group > group.mig
awk -v LIMIT=500 -F: '($3>=LIMIT) && ($3!=65534) {print $1}' passwd | tee - |egrep -f - shadow > shadow.mig
```

passwd、group 文件处理方法一样，shadow 文件处理方法稍有复杂些。

第二步：将原系统中的 home 下用户目录拷贝到新系统 home 目录下。需要注意的是，在拷贝过程中目录及文件权限的属性很可能发生了改变。这样导致的问题是，即使建立了用户目录，用户登录系统也无法使用，图形界面也直接崩溃。为了方便，我开放了用户目录和文件的所有权限，即任何人都有读写以及执行的权限。切换到 home 目录下，使用如下命令：

```
chmod 777 * -R
```

第三步：在新机器中添加用户信息。

```
cat passwd.mig >> /etc/passwd
cat group.mig >> /etc/group
cat shadow.mig >> /etc/shadow
```

需要注意的是，第一个 ID（1000）或许会发生冲突，注意稍微修改下。

至此，完整的复制了用户信息并保留了图形界面。

第八章 Gvim 的使用

8.1 常用配置

```
set nu
set fileencodings=utf-8,gbk,default,latin1
set encoding=utf-8
let $LANG="zh_CN.UTF-8"
```

8.2 常用命令

8.2.1 一般模式 (命令模式)

8.2.1.1 打开文件、保存、关闭文件

- vi filename： 打开 filename 文件
- w： 保存文件
- w vpser.net： 保存至 vpser.net 文件
- q： 退出编辑器，如果文件已修改则用下面的命令
- q!： 退出编辑器，且不保存
- wq： 退出编辑器，且保存文件

8.2.1.2 常用光标移动命令

- 注意区分大小写
- h：左； j： 下； k： 上； l： 右；这很符合书写习惯，用下就知道了
- n<hjkl> 移动相应字符
- 0：零，移动到这一行最前面的字符处
- $: 移动到这一行最后面的字符处
- G：移动到文件的最后一行
- nG：n 表示数字，移动到这个文件的第 n 行
- n<Enter>：n 表示数字，光标向下移动 n 行，等效于 nj
- 空格键：向右
- Backspace : 向左
- Enter： 移动到下一行首
- -： 移动到上一行首

8.2.1.3 搜索与替换

- /word：从光标位置开始，向下搜索一个名为“word”的字符
- ?word：从光标位置开始，向上搜索一个名为“word”的字符
- n：表示“next”
- N：表示“previous”
- %s/word1/word2/g：表示将 word1 替换为 word2，g 表示全局替换
- %s/word1/word2/gc：其中多的一个 c 表示替换之前有个用户提示，需要确认

8.2.1.4 删除、复制与粘贴

- x,X：小写 x 表示向后删除字符；大写 X 表示向前删除字符
- nx：表示向后删除 n 个字符
- dd：表示删除光标所在的整行
- ndd：表示包括光标所在行向下删除 n 行
- d1G：表示删除当前行到第 1 行所有的数据
- dG：删除光标所在位置到最后一行所有的数据
- yy：复制光标所在的一整行，也可以用 ayy 复制，a 为缓冲区，a 也可以替换为 a 到 z 的任意字母，可以完成多个复制任务。
- nyy：n 表示数字，复制光标所在的向下 n 行，也可以用 anyy 复制，a 为缓冲区，a 也可以替换为 a 到 z 的任意字母，可以完成多个复制任务。
- p,P：p 将已复制的数据粘贴到光标下一行，P 相反，如果使用了前面的自定义缓冲区，建议使用“ap”或“aP”进行粘贴。
- J：将光标所在的行与下一行结合在一起
- u,U：小写 u 撤销上一步操作，大写 U 撤销对当前行的所有操作
- Ctrl+r：重做上一个操作，恢复
- .：重复前一个动作
- ：复制单个字符：首先进入正常模式（一般模式）然后按 v ，进入 visual 方式，然后就可以移动方向键选中文本，然后按 y，就拷贝完成。

8.2.2 编辑模式

- i,I：i 从当前光标所在处插入，I 从当前所在行的第一个非空格处开始插入
- a,A：a 从当前光标所在处下一个字符开始插入，a 从当前所在行的最后一个非空格处开始插入。A 是从所在行尾插入。
- o,O：o 在当前光标所在的下一行插入一个行行，O 相反
- r,R：r 会替换光标所在的那一个字符，R 会一直替换光标所在的字符
- J ：合并光标所在行及下一行为一行（命令模式）

8.3 vim 中使用系统粘贴板

用 vim 这么久了，始终也不知道怎么在 vim 中使用系统粘贴板，通常要在网上复制一段代码都是先 gedit 打开文件，中键粘贴后关闭，然后再用 vim 打开编辑，真的不爽；上次论坛上有人问到了怎么在 vim 中使用系统粘贴板，印象里回复很多，有好几页的回复却没有解决问题，今天实在受不了了又在网上找办法，竟意外地找到了，贴出来分享一下。

如果只是想使用系统粘贴板的话直接在输入模式按 Shift+Inset 就可以了，下面讲一下 vim 的粘贴板的基础知识，有兴趣的可以看看，应该会有所收获的。vim 帮助文档里与粘贴板有关的内容如下：

1. vim 有 12 个粘贴板，分别是 0、1、2、...、9、a、“、＋；用:reg 命令可以查看各个粘贴板里的内容。在 vim 中简单用 y 只是复制到 “(双引号) 粘贴板里，同样用 p 粘贴的也是这个粘贴板里的内容；

2. 要将 vim 的内容复制到某个粘贴板，需要退出编辑模式，进入正常模式后，选择要复制的内容，然后按"Ny 完成复制，其中 N 为粘贴板号 (注意是按一下双引号然后按粘贴板号最后按 y)，例如要把内容复制到粘贴板 a，选中内容后按"ay 就可以了，有两点需要说明一下：* “号粘贴板（临时粘贴板）比较特殊，直接按 y 就复制到这个粘贴板中了，直接按 p 就粘贴这个粘贴板中的内容；* + 号粘贴板是系统粘贴板，用"+y 将内容复制到该粘贴板后可以使用 Ctrl ＋ V 将其粘贴到其他文档（如 firefox、gedit）中，同理，要把在其他地方用 Ctrl ＋ C 或右键复制的内容复制到 vim 中，需要在正常模式下按"+p；

3. 要将 vim 某个粘贴板里的内容粘贴进来，需要退出编辑模式，在正常模式按"Np，其中 N 为粘贴板号，如上所述，可以按"5p 将 5 号粘贴板里的内容粘贴进来，也可以按"+p 将系统全局粘贴板里的内容粘贴进来。

第三部分

数值方法

第九章 常微分方程 $\frac{dy}{dx}=f(x,y)$ 的数值解法

参考文献：《数值分析》（第 4 版）李庆扬王能超易大义编，p336 第 9 章常微分方程初值问题数值解法

我们为什么要去解方程？因为方程是我们做的东西功能的抽象或者是自然规律的抽象，只有这样才方便我们具体分析，只有这样才具有普遍性，而不是一个特例，拿到别处就没用了。所以具体应用时，这个方程反映的就是我们系统，以 $\frac{dy}{dx}=f(x,y)$ 为例，x 通常表示是我们的输入，而输入是我们可以控制的，y 是输出。我们很关心一个 x 的值（输入）对应 y 的值（输出）是多少。这也就说明了我们解方程的目的，目的就是找出 x 与 y 的关系，即，这个方程中自变量与应变量的关系。

这里我们先回顾下当初上第一堂课老师就讲的方程解的表示方法。有解析解，用一个函数就将所有的解抽象表示出来，这样的解当然是最理想的解，因为我想知道哪个 x 的值对应的 y 是多少，将 x 的值代进去算就知道了。但是实际上是很少能有具体的解析解。还有就是数值解，没有函数表达式，直接求出一系列的 x 的值对应的 y 值。而我们实际生活中真正关心的也是这些具体数值的对应。所以数值解法很实用。

初值问题

$$\begin{cases}\dfrac{dy}{dx}=f(x,y), \quad a\le x\le b\\ y(a)=c\end{cases}$$

是常微分方程定解问题的最简单最常用的形式，这里讨论的就是它。参考资料：[?]p336 第 9 章常微分方程初值问题数值解法

（1）数值解的存在唯一性

只要函数 $f(x,y)$ 适当光滑，例如，满足利普希茨（Lipschitz）条件

$$|f(x,y)-f(x,\tilde{y})|\le L|y-\tilde{y}|$$

理论上就可以保证上面初值问题的解 $y=y(x)$ 存在并且唯一。

L 是 $f(x,y)$ 关于 y 的利普希茨常数。L 的具体数值如何确定？

（2）数值解法的思想

所谓数值解法，就是寻求解 $y(x)$ 在一系列离散节点

$$x_1<x_2<\cdots<x_n<x_{n+1}<\cdots$$

上的近似值

$$y_1,y_2,\cdots,y_n,y_{n+1},\cdots$$

相邻两个节点的间距 $h_n=x_{n+1}-x_n$ 称为**步长**。今后如不特别说明，总是假定 $h_i=h(i=1,2,\cdots)$ 为定数，这时节点为 $x_n=x_0+nh,n=0,1,2,\cdots$。

这里的初值问题的数值解法有个基本特点，它们都采取“步进式”，即求解过程顺着节点排列的次序一步一步地向前推进。描述这类算法，只要给出用已知信息 $y_n,y_{n-1},y_{n-2},\cdots$

计算 y_{n+1} 的递推公式。

首先要对方程 $\begin{cases}\dfrac{dy}{dx}=f(x,y), & a\le x\le b\\ y(a)=c\end{cases}$ 离散化，建立求数值解的递推公式。一类是计算 y_{n+1} 时只用到前一点的值 y_n，称为单步法。另一类是用到 y_{n+1} 前面 k 点的值 $y_n, y_{n-1}, \cdots, y_{n-k+1}$，称为 k 步法。其次，要研究公式的局部截断误差和阶，数值解 y_n 与精确解 $y(x_n)$ 的误差估计及收敛性，还有递推公式的计算稳定性等问题。

9.1 简单数值方法与基本概念

9.1.1 欧拉法

$$\begin{cases}\dfrac{dy}{dx}=f(x,y), & a\le x\le b\\ y(a)=c\end{cases}$$

的解是 $y=y(x)$，$y=y(x)$ 在 xy 平面上是一条线。从几何观点看，$f(x,y)$ 则成了 $y=y(x)$ 的斜率。已知初值 x_0, y_0，由公式 $f(x,y)$ 就可以求得 $y=y(x)$ 在 (x_0,y_0) 点处的斜率。如图，

有：

$$\tilde{y}_1=y_0+hf(x_0,y_0)\sim y_1$$

近似有：

$$y_{n+1}=y_n+hf(x_n,y_n)$$

这就是欧拉公式。

如果将步长 h 取的很小，与初始值相邻的第一个值 (x_1,y_1) 求得的近似值 $(x_1,\tilde{y}_1)$ 可以说是非常精确的。但之后一的求解是将近似值 $(x_1,\tilde{y}_1)$ 作为初始值进行计算，很明显，

正在越来越偏离准确值，非常大的误差正在形成。这一点在后面的具体例子中可以明显看出。

9.1.2 后退欧拉法（隐式）

对 $\frac{dy}{dx}=f(x,y)$ 从 x_n 到 x_{n+1} 积分，得

$$y(x_{n+1})=y(x_n)+\int_{x_n}^{x_{n+1}} f(t,y(t))dt$$

将右端的积分用右矩形公式 $hF(x_{n+1},y(x_{n+1}))$ 代替，得

$$y_{n+1}=y_n+hf(x_{n+1},y_{n+1})$$

这个公式称为后退欧拉公式。

后退欧拉公式与欧拉公式有着本质的区别，欧拉公式是关于 y_{n+1} 的一个直接的计算公式，这类公式称作是**显式的**。后退欧拉公式右端含有未知的 y_{n+1}，实际上是关于 y_{n+1} 的一个函数方程，这类公式称作是**隐式的**。

隐式方程通常用迭代法求解，而迭代过程的实质是逐步显式化。

9.1.2.1 迭代法对后退欧拉公式求解

求解过程

1、先用欧拉公式给出迭代初值 $y_{n+1;0}$（前面已经说过，欧拉公式给出的第一个相邻值还是非常精确的）

$$y_{n+1;0}=y_n+hf(x_n,y_n)$$

其中下标中分号“;”后的数字表示迭代次数。

2、将迭代初值 $y_{n+1;0}$ 代入后退欧拉公式

$$y_{n+1;1}=y_n+hf(x_{n+1},y_{n+1;0})$$

3、如此反复进行，得

$$y_{n+1;k+1}=y_n+hf(x_{n+1};y_{n+1;k}) \qquad (k=0,1,\cdots)$$

看到这个迭代过程，很容易产生一个问题：这样迭代求出的解是不是趋进于精确解，即是否收敛？下面就来讨论。

解的收敛性

由于 $f(x,y)$ 对 y 满足利普希茨（Lipschitz）条件

$$|f(x,y)-f(x,\tilde{y})|\le L|y-\tilde{y}|$$

L 是 $f(x,y)$ 关于 y 的利普希茨常数。L 的具体数值如何确定？

将 $y_{n+1;k+1}=y_n+hf(x_{n+1};y_{n+1;k})$ 减去 $y_{n+1}=y_n+hf(x_{n+1},y_{n+1})$ 即，将迭代 k 后的后退欧拉公式减去还未迭代的后退欧拉公式，有

$$|y_{n+1;k+1}-y_{n+1}|=h|f(x_{n+1},y_{n+1;k+1})-f(x_{n+1},y_{n+1})|\le hL|y_{n+1,k}-y_{n+1}|$$

这个公式可以看出，只要 $hL<1$，每次迭代出的解都比上一次迭代出的解趋近与精确解，

即，后退欧拉法的迭代公式收敛到精确解。

9.1.3 梯形公式（隐式）

对 $\frac{dy}{dx}=f(x,y)$ 从 x_n 到 x_{n+1} 积分，得

$$y(x_{n+1})=y(x_n)+\int_{x_n}^{x_{n+1}} f(t,y(t))dt$$

为了得到比后退欧拉法精度更高的计算公式，右端积分用梯形求积公式近似（梯形是比矩形更准确些）

$$y_{n+1}=y_n+\frac{h}{2}[f(x_n,y_n)+f(x_{n+1},y_{n+1})]$$

这个公式称为梯形公式。

梯形公式也是隐式的，也用迭代法求解，仍用欧拉公式提供迭代初值，其迭代公式为

$$\begin{cases} y_{n+1;0}=y_n+hf(x_n,y_n) \\ y_{n+1;k+1}=y_n+\frac{h}{2}[f(x_n,y_n)+f(x_{n+1},y_{n+1;k})] \\ (k=0,1,2\cdots) \end{cases}$$

梯形迭代公式解的收敛性讨论：

与后退欧拉公式迭代求解差不多，只不过这里是将未迭代的公式减去迭代 k 后的公式，

$$y_{n+1}-y_{n+1;k+1}=\frac{h}{2}[f(x_{n+1,y_{n+1}})-f(x_{n+1},y_{n+1;k})]$$

利用利普希茨条件 $|f(x,y)-f(x,\tilde{y})|\le L|y-\tilde{y}|$，有

$$|y_{n+1}-y_{n+1;k+1}|\le\frac{hL}{2}|y_{n+1}-y_{n+1;k}|$$

由此可知，如果选取 h 充分小，使得 $\frac{hL}{2}<1$，则当 $k\to\infty$ 时，有 $y_{n+1;k+1}\to y_{n+1}$，即梯形迭代公式是收敛的。

9.1.4 改进欧拉法

梯形方法虽然提高了精度，但其算法复杂，在应用梯形迭代公式进行实际计算时，没迭代一次，都要重新计算函数 $f(x,y)$ 的值，而迭代又要反复进行若干次，计算量很大，而且往往难以预测。为了控制计算量，通常只迭代一两次就转入下一步的计算，这就简化了算法。

思路如下：

先用欧拉公式求得一个初步的近似值 $\tilde{y}_{n+1}$，称之为**预测值**。预测值 $\tilde{y}_{n+1}$ 的精度可能很差，在用梯形公式将它校正一次，即迭代一次得 y_{n+1}，这个结果称为**校正值**。这样建立的预测-校正系统称为**改进的欧拉公式：**

$$\begin{aligned} &\text{预测} \quad && \tilde{y}_{n+1}=y_n+hf(x_n,y_n)=y_p \\ &\text{校正} \quad && y_{n+1}=y_n+\frac{h}{2}[f(x_n,y_n)+f(x_{n+1},\tilde{y}_{n+1})] \end{aligned}$$

即：

$$y_{n+1} = \frac{y_n}{2} + \frac{h}{2}f(x_n, y_n) + \frac{y_n}{2} + \frac{h}{2}f(x_{n+1}; \tilde{y}_{n+1})$$

由此改写成平均化形式为

$$\begin{cases} y_p = y_n + hf(x_n, y_n) \\ y_c = y_n + hf(x_{n+1}, y_p) \\ y_{n+1} = \frac{1}{2}(y_p + y_c) \end{cases}$$

其中 p 表示 prediciton，c 表示 correction。

9.1.5 单步法的局部截断误差与阶

初值问题 $\begin{cases} \dfrac{dy}{dx} = f(x, y), \quad a \le x \le b \\ y(a) = c \end{cases}$ 单步法的一般形式为：

$$y_{n+1} = y_n + h\varphi(x_n, y_n, y_{n+1}, h)$$

其中多元函数 φ 与 $f(x, y)$ 有关。

当 φ 含有 y_{n+1} 时，方法是隐式的。若不含 y_{n+1} 则为显式方法，该**显式单步法**表示为：

$$y_{n+1} = y_n + h\varphi(x_n, y_n, h)$$

$\varphi(x, y, h)$ 称为增量函数，例如欧拉法有 $\varphi(x, y, h) = f(x, y)$。

对一般显式单步法局部截断误差定义如下：

定义 1

设 $y(x)$ 是初值问题 $\begin{cases} \dfrac{dy}{dx} = f(x, y), \quad a \le x \le b \\ y(a) = c \end{cases}$ 的准确解，假设在 x_n 前各步没有误差，即 y_n 是准确解 $y(x_n)$。x_n 下一步的数值解为：$y_{n+1} = y_n + h\varphi(x_n, y_n, h)$，将精确解 $y(x_{n+1})$ 减去数值解 y_{n+1} 就得到计算这步的误差：

$$T_{n+1} = y(x_{n+1}) - y(x_n) - h\varphi(x_n, y(x_n), h)$$

T_{n+1} 称为显式单步法 $y_{n+1} = y_n + h\varphi(x_n, y_n, h)$ 的局部截断误差。从定义可以看出，T_{n+1} 之所以称为局部截断误差是因为它反映的是显式单步法计算一步时的误差。

T_{n+1} 的表达式是很好理解的，$y(x_{n+1}) - y(x_n)$ 表示的是从 x_n 步到 x_{n+1} 精确解间的差值。而 $h\varphi(x_n, y_n, h)$ 表示的是从 x_n 步到 x_{n+1} 数值解间的差值（前提假设是 x_n, y_n 是精确解，x_n, y_n 可以理解为这步数值计算的初值）。由此这步精确解间的差值和这步数值解间的差值之差反映的当然是这步数值计算的误差，而反映的误差只限于这一步计算，故称为局部误差。

但是实际计算中 y_n 并不是精确解，这个误差定义也就是初值 x_0, y_0 的下一步 x_1, y_1 严格满足。……例子中算算步误差是否在这个误差定义的范围内……

定义 2

设 $y(x)$ 是初值问题 $\begin{cases}\dfrac{dy}{dx}=f(x,y), & a\le x\le b\\ y(a)=c\end{cases}$ 的准确解，若存在最大整数 p 使显式单步法的局部截断误差满足

$$T_{n+1}=y(x+h)-y(x)-h\varphi(x,y,h)=O(h^{p+1})$$

则称显式单步法具有 p 阶精度。若将上式写成

$$T_{n+1}=\psi(x_n,y(x_n))h^{p+1}+O(h^{p+2})$$

则 $\psi(x_n,y(x_n))h^{p+1}$ 称为局部截断误差主项。这样的形式对误差的表示是很有意义的，就 T_{n+1} 而言，其本身就是个小量，这个小量中，又存在更高阶的小量 $O(h^{p+2})$，故实际讨论中，可以将这个更高阶小量 $O(h^{p+2})$ 略去，余下的局部截断误差主项 $\psi(x_n,y(x_n))h^{p+1}$ 已经就能很好的描述误差了。

9.1.5.1 欧拉法的局部截断误差

欧拉公式为：

$$y_{n+1}=y_n+hf(x_n,y_n)$$

利用定义 1，有：

$$T_{n+1}=y(x_{n+1})-y(x_n)-hf(x_n,y(x_n))$$

将 $y(x_{n+1})=y(x_n+h)$ 在 x_n 用泰勒展开：

$$\begin{aligned}y(x_{n+1})=y(x_n+h)&=y(x_n)+y'(x_n)(x_n+h-x_n)+\frac{y''(x_n)}{2!}(x_n+h-x_n)^2+O(h^3)\\&=y(x_n)+hy'(x_n)+\frac{h^2}{2}y''(x_n)+O(h^3)\end{aligned}$$

将 $f(x_n,y(x_n))$ 改写 $y'(x_n)$[1]，于是，有：

$$T_{n+1}=\frac{h^2}{2}y''(x_n)+O(h^3)$$

根据定义 2，局部截断误差主项为 $\dfrac{\hbar^2}{2}y''(x_n)$。就整个 T_{n+1} 而言，将高阶项 $O(h^3)$ 略去（T_{n+1} 本来就是小量），则 $T_{n+1}=O(h^2)$。根据定义 $p=1$，即欧拉法是 1 阶方法。

9.1.5.2 后退欧拉法局部截断误差

后退欧拉公式为：

$$y_{n+1}=y_n+hf(x_{n+1},y_{n+1})$$

利用定义 1，有：

$$T_{n+1}=y(x_{n+1})-y(x_n)-hf(x_{n+1},y(x_{n+1}))$$

[1] $f(x_n,y(x_n))$=$y'(x_n)$，$f(x,y)$ 就是 $y(x)$ 的斜率，看原方程就知道了。

将 $y(x_{n+1}) = y(x_n+h), f(x_{n+1}, y(x_{n+1})) = y'(x_{n+1}) = y'(x_n+h)$ 在 x_n 处泰勒展开，有：

$$\begin{aligned} y(x_{n+1}) = y(x_n+h) &= y(x_n) + y'(x_n)(x_n+h-x_n) + \frac{y''(x_n)}{2!}(x_n+h-x_n)^2 + O(h^3) \\ &= y(x_n) + hy'(x_n) + \frac{h^2}{2}y''(x_n) + O(h^3) \end{aligned}$$

$$f(x_{n+1}, y(x_{n+1})) = y'(x_{n+1}) = y'(x_n+h) = y'(x_n) + hy''(x_n) + O(h^2)$$

代入 T_{n+1} 中，有：

$$\begin{aligned} T_{n+1} &= y(x_n) + hy'(x_n) + \frac{h^2}{2}y''(x_n) + O(h^3) - y(x_n) - h[y'(x_n) + hy''(x_n) + O(h^2)] \\ &= -\frac{h^2}{2}y''(x_n) + O(h^3) \end{aligned}$$

这里 $O(h^2)$ 直接抹去了，是因为有更高阶的小量 $O(h^3)$ 在。

按照定义 2，$p=1$，是 1 阶方法，局部截断误差主项为 $-\frac{h^2}{2}y''(x_n)$。

9.1.5.3 梯形法局部截断误差

梯形公式为：

$$y_{n+1} = y_n + \frac{h}{2}[f(x_n, y_n) + f(x_{n+1}, y_{n+1})]$$

利用定义 1，有：

$$T_{n+1} = y(x_{n+1}) - y(x_n) - \frac{h}{2}[y'(x_n) + y'(x_{n+1})]$$

将 $y(x_{n+1}) = y(x_n+h), f(x_{n+1}, y(x_{n+1})) = y'(x_{n+1}) = y'(x_n+h)$ 在 x_n 处泰勒展开，有：

$$y(x_n+h) = y(x_n) + hy' + h^2\frac{y''(x_n)}{2!} + h^3\frac{y'''(x_n)}{3!} + O(h^4)$$

$$f(x_{n+1}, y(x_{n+1})) = y'(x_{n+1}) = y'(x_n+h) = y'(x_n) + hy''(x_n) + \frac{h^2}{2}y'''(x_n) + O(h^3)$$

这里为什么展开的级数和上面两个不一样，那是因为必须凑成定义 2 的形式，从上式看的到 $y'(x_n), y''(x_n)$ 都相消为 0 了。

代入 T_{n+1} 中，有：

$$\begin{aligned} T_{n+1} =& y(x_n) + hy' + h^2\frac{y''(x_n)}{2!} + h^3\frac{y'''(x_n)}{3!} + O(h^4) - y(x_n) \\ & - \frac{h}{2}[y'(x_n) + y'(x_n) + hy''(x_n) + \frac{h^2}{2}y'''(x_n) + O(h^3)] \\ =& \frac{h^3}{6}y'''(x_n) - \frac{h^3}{4}y'''(x_n) + O(h^3) \\ =& -\frac{h^3}{12}y'''(x_n) + O(h^4) \end{aligned}$$

根据定义 2，梯形法是二阶的，其局部误差主项为 $-\frac{h^3}{12}y'''(x_n)$。

9.2 龙格—库塔法

9.2.1 显式龙格-库塔法的一般形式

方程

$$\begin{cases}\dfrac{dy}{dx}=f(x,y), \quad a\le x\le b\\ y(a)=c\end{cases}$$

等价积分形式为

$$y(x_{n+1})=y(x_n)+\int_{x_n}^{x_{n+1}}f(x,y(x))dx$$

若要是公式阶数提高，就必须使右端积分的数值求积公式精度提高，它必然要增加求积节点，为此将上式右端用求积公式表示为

$$\int_{x_n}^{x_{n+1}}f(x,y(x))dx\approx h\sum_{i=1}^{r}c_if(x_n+\lambda_ih,y(x_n+\lambda_ih))$$

这个公式应该是龙格库塔法思想的核心。这个公式给人感觉是在由 x_n 到 x_{n+1} 步中又分出个 r 步计算，增加步数当然能提高精度了。

为了得到便于计算的显式方法，类似改进欧拉法

$$y_{n+1}=y_n+\frac{h}{2}[f(x_n,y_n)+f(x_n+h,y_n+hf(x_n,y_n))]$$

将 y_{n+1} 表示为

$$y_{n+1}=y_n+h\varphi(x_n,y_n,h)$$

其中

$$\varphi(x_n,y_n,h)=\sum_{i=1}^{r}c_iK_i,\qquad K_1=f(x_n,y_n),\qquad K_i=f(x_n+\lambda_ih,y_n+h\sum_{j=1}^{i-1}\mu_{ij}K_j)\qquad i=2,\cdots,r$$

这里 c_i,λ_i,μ_{ij} 均为常数。这个公式称为 r 级显式龙格-库塔法（Runge-Kutta），简称 R-K 法。

当 $r=1$，$\varphi(x_n,y_n,h)=f(x_n,y_n)$，是欧拉法，此时方法的阶为 $p=1$。

9.2.2 二阶显式 R-K 法

当 $r=2$ 时，由 r 级显式龙格-库塔法，有

$$\begin{cases}y_{n+1}=y_n+h(c_1K_1+c_2K_2)\\ K_1=f(x_n,y_n)\\ K_2=f(x_n+\lambda_2h,y_n+\mu_{21}hK_1)\end{cases}$$

这里 $c_1,c_2,\lambda_2,\mu_{21}$ 均为待定常数，不同系数有不同表达式。可见改进欧拉法就是一种二阶显式龙格-库塔法。适当选取这些系数，是公式阶数 p 尽量高，通过局部截断误差来确定这些系数。

根据局部截断误差的定义，其局部截断误差为

$$T_{n+1} = y(x_{n+1}) - y(x_n) - h[c_1 f(x_n, y_n) + c_2 f(x_n + \lambda_2 h, y_n + \mu_{21} h f_n)]$$

其中 $y_n = y(x_n), f_n = f(x_n, y_n)$。

为得到 T_{n+1} 的阶 p，将上式各项在 (x_n, y_n) 处做泰勒展开[2]，由于 $f(x, y)$ 是二元函数，用二元泰勒展开，各项展开式为

$$y(x_{n+1}) = y(x_n + h) = y_n + hy' + \frac{h^2}{2} y''_n + \frac{h^3}{3!} y'''_n + O(h^4)$$

其中

$$\begin{cases} y'_n = f(x_n, y_n) = f_n \\ y''_n = \frac{\partial}{\partial x} y'_n = \frac{\partial}{\partial x} f(x_n, y_n) = f'_x(x_n, y_n) + f'_y(x_n, y_n) \cdot y'_n = f'_x(x_n, y_n) + f'_y(x_n, y_n) \cdot f_n \\ y'''_n = f''_{xx}(x_n, y_n) + f''_{xy}(x_n, y_n) y'_n + f''_{yx}(x_n, y_n) f_n + f_n^2 f''_{yy}(x_n, y_n) \\ \qquad + f'_y(x_n, y_n)[f'_x(x_n, y_n) + f'_y(x_n, y_n) \cdot f(x_n, y_n)] \\ \quad = f''_{xx}(x_n, y_n) + 2 f_n f'_{xy}(x_n, y_n) + f_n^2 f''_{yy}(x_n, y_n) + f'_y(x_n, y_n)[f'_x(x_n, y_n) + f_n f'_y(x_n, y_n)] \end{cases}$$

注意推导时多次用到第一个式子 $y'_n = f(x_n, y_n) = f_n$ 替换，还有 $f''_{xy}(x_n, y_n)$ 和 $f''_{yx}(x_n, y_n)$ 是相等的，求导顺序不影响结果。

$$f(x_n + \lambda_2 h, y_n + \mu_2 1 h f_n) = f_n + f'_x(x_n, y_n)\lambda_2 h + f'_y(x_n, y_n)\mu_{21} h f_n + O(h^2)$$

观察 $f(x_n + \lambda_2 h, y_n + \mu_{21} h f_n)$ 发现 $y(x_{n+1})$ 展开到 4 阶时，y'''_n 项中的 $f'_y f'_x + f'_y f$ 不能通过选择参数消掉，也就说 $r = 2$ 的龙格-库塔公式不能使局部误差提高到 $O(h^4)$。实际上要使 h^3 项为零，需增加 3 个方程，要确定 4 个参数 $c1, c_1, \lambda_2$ 及 μ_{21}，这是不可能的。故 $r = 2$ 的显式龙格-库塔法的阶只能是 $p = 2$，而不能得到三阶公式。

$y(x_{n+1})$ 最多只能展开到 2 阶，即：

$$y(x_{n+1}) = y(x_n + h) = y_n + hy' + \frac{h^2}{2} y''_n + O(h^3)$$

代入到 T_{n+1}，有

$$\begin{aligned} T_{n+1} =& y_n + hy' + \frac{h^2}{2} y''_n + O(h^3) - y(x_n) \\ & - h\{c_1 f(x_n, y_n) + c_2[f_n + f'_x(x_n, y_n)\lambda_2 h + f'_y(x_n, y_n)\mu_{21} h f_n + O(h^2)]\} \\ =& y_n + hf_n + \frac{h^2}{2}[f'_x(x_n, y_n) + f'_y(x_n, y_n) \cdot f_n] + O(h^3) - y(x_n) \\ & - h\{c_1 f(x_n, y_n) + c_2[f_n + f'_x(x_n, y_n)\lambda_2 h + f'_y(x_n, y_n)\mu_{21} h f_n + O(h^2)]\} \\ =& (1 - c_1 - c_2) f_n h + \left(\frac{1}{2} - c_2\lambda_2\right) f'_x(x_n, y_n) h^2 - \left(\frac{1}{2} - c_2\mu_{21}\right) f'_y(x_n, y_n) f_n h^2 + O(h^3) \end{aligned}$$

要使 $p = 2$ 阶，必须使

$$1 - c_1 - c_2 = 0, \qquad \frac{1}{2} - c_2\lambda_2 = 0, \qquad \frac{1}{2} - c_2\mu_{21} = 0$$

[2]为什么不像后退欧拉法和梯形法，化成 $y(x_n + h)$ 的形式，在 x_n 处泰勒展开？那是因为寻找处阶 p 的目标是化成局部截断误差定义 2 的形式，这里的项 $f(x_n + \lambda_2 h, y_n + \mu_{21} h f_n)$ 是没办法搞成这种形式。后退欧拉法和梯形法只需 $y(x_n)$ 各级导数形式就达成这个目标了，不需要用 $f(x_n, y_n)$ 二元泰勒展开，展开也是多此一举，会相消掉。

即

$$c_2\lambda_2=\frac{1}{2},\qquad c_2\mu_{21}=\frac{1}{2},\qquad c_1+c_2=1$$

它的解不是唯一的，令 $c_2=a\neq 0$，则

$$c_1=1-a,\qquad \lambda_2=\mu_{21}=\frac{1}{2a}$$

这样得到的公式称为二阶龙格-库塔法。

如果取 $a=\frac{1}{2}$，则 $c_1=c_2=\frac{1}{2},\lambda_2=\mu_{21}=1$，这就是改进欧拉法。

若取 $a=1$，则 $c_2=1,c_1=0,\lambda_2=\mu_{21}=\frac{1}{2}$，得到计算公式

$$\begin{cases} y_{n+1}=y_n+hK_2 \\ K_1=f(x_n,y_n) \\ K_2=f(x_n+\frac{h}{2},y_n+\frac{h}{2}K_1) \end{cases}$$

称为中点公式，相当与数值积分的中矩形公式，也可以表示为

$$y_{n+1}=y_n+hf(x_n+\frac{h}{2},y_n+\frac{h}{2}f(x_n,y_n)$$

9.2.3 三阶显式龙格-库塔法

$r=3$ 可以得到三阶显式龙格-库塔法，其表达式为

$$\begin{cases} y_{n+1}=y_n+h(c_1K_1+c_2K_2+c_3K_3) \\ K_1=f(x_n,y_n) \\ K_2=f(x_n+\lambda_2h,y_n+\mu_{21}hK_1) \\ K_3=f(x_n+\lambda_3h,y_n+\mu_{31}hK_1+\mu_{32}hK_2) \end{cases}$$

其中 c_1,c_2,c_3 及 $\lambda_2,\mu_{21},\lambda_3,\mu_{31},\mu_{32}$ 均为待定参数。该公式的局部截断误差为

$$T_{n+1}=y(x_{n+1})-y(x_n)-h[c_1K_1+c_2K_2+c_3K_3]$$

将 K_2,K_3 按二元函数泰勒展开，使 $T_{n+1}=O(h^4)$，可得待定参数满足方程

$$\begin{cases} c_1+c_2+c_3=1 \\ \lambda_2=\mu_{21} \\ \lambda_3=\mu_{31}+\mu_{32} \\ c_2\lambda_2+c_3\lambda_3=\frac{1}{2} \\ c_2\lambda_2^2+c_3\lambda_3^2=\frac{1}{3} \\ c_3\lambda_2\mu_{32}=\frac{1}{6} \end{cases}$$

这是 8 个未知数 6 个方程的方程组，解也不是唯一的，可以得到很多公式，这些公式统称为三阶龙格-库塔公式。

常见的三阶龙格-库塔公式为

$$\begin{cases} y_{n+1} = y_n + \frac{h}{6}(K_1 + 4K_2 + K_3) \\ K_1 = f(x_n, y_n) \\ K_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_1) \\ K_3 = f(x_n + h, y_n - hK_1 + 2hK_2) \end{cases}$$

9.2.4 四阶龙格-库塔法

类似二阶、三阶龙格-库塔法推导过程可以推导出各种四阶龙格-库塔公式，常用公式为

$$\begin{cases} y_{n+1} = y_n + \dfrac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ K_1 = f(x_n, y_n) \\ K_2 = f(x_n + \dfrac{h}{2}, y_n + \dfrac{h}{2}K_1) \\ K_3 = f(x_n + \dfrac{h}{2}, y_n + \dfrac{h}{2}K_2) \\ K_4 = f(x_n + h, y_n + hK_3) \end{cases}$$

其截断误差为 $O(h^5)$。二阶的推导已经比较烦了，四阶推导可以想象有多烦。有的地方采用如下表达式

$$\begin{cases} y_1 = y_0 + \frac{1}{6}(K_0 + 2 \times K_1 + 2 \times K_2 + K_3) \\ K_0 = h \times f(x_0, y_0) \\ K_1 = h \times f(x_0 + \frac{h}{2}, y_0 + \frac{K_0}{2}) \\ K_2 = h \times f(x_0 + \frac{h}{2}, y_0 + \frac{K_1}{2}) \\ K_3 = h \times f(x_0 + h, y_0 + K_2) \end{cases}$$

9.3 各种方法的应用举例及对比

求解初值问题

$$\begin{cases} y' = y - \frac{2x}{y} \quad (0 \le x \le 1) \\ y(0) = 1 \end{cases}$$

9.3.1 精确解

初值问题

$$\begin{cases} y' = y - \frac{2x}{y} \quad (0 \le x \le 1) \\ y(0) = 1 \end{cases}$$

精确解为

$$y=\sqrt{1+2x} \qquad 0\le x\le 1$$

取步长为 0.1 其结果为

x_n	$y(x_n)$
0.0	1.000000
0.1	1.095445
0.2	1.183216
0.3	1.264911
0.4	1.341641
0.5	1.414214
0.6	1.483240
0.7	1.549193
0.8	1.612452
0.9	1.673320
1.0	1.732051

图 9.1: 精确解

图 9.2: 精确解

Fortran 代码如下:

```
!the exact solution of y'=y-2x/y (0<=x<=1) is y=sqrt(1+2x)
program exact_solution
    implicit none
    real :: x , y
    open ( unit = 10 , file = 'exact_solutions.ods' )
    do x = 0 , 1.0 , 0.1 !attention 1 must be 1.0 , the data type must be consistent!
        y = sqrt ( 1 + 2 * x )
        write ( 10 , 100 ) x , y
    enddo
    100 format( 1x , f10.6 , f10.6 )
end program exact_solution
```

9.3.2 欧拉法求解

对于解初值问题

$$\begin{cases} y'=y-\frac{2x}{y} & (0\le x\le 1) \\ y(0)=1 \end{cases}$$

的欧拉公式的具体形式为

$$y_{n+1}=y_n+h(y_n-\frac{2x_n}{y_n})$$

取步长为 $h=0.1$ 其结果为

Fortran 代码如下:

```
!the euler's_method of y'=y-2x/y (0<=x<=1)
program euler_method
```

x_n	精确解 (y_n)	欧拉法解 (y)
0.0	1.000000	1.000000
0.1	1.095445	1.100000
0.2	1.183216	1.191818
0.3	1.264911	1.277438
0.4	1.341641	1.358213
0.5	1.414214	1.435133
0.6	1.483240	1.508966
0.7	1.549193	1.580338
0.8	1.612452	1.649784
0.9	1.673320	1.717780
1.0	1.732051	1.784771

图 9.3: 精确解与欧拉法解

图 9.4: 精确解与欧拉法解

```
    implicit none
    real :: x , y , h
    h = 0.1
    x = 0.0
    y = 1.0
    open ( unit = 10 , file = 'euler_method.ods' )
    write ( 10 , 100 ) x , y
    do x = 0.0 , 0.9 , 0.1
        y = y + h * ( y - 2 * x / y )
        write ( 10 , 100 ) x + 0.1 , y
    enddo
    100 format ( 1x , f10.6 , f10.6 )
end program euler_method
```

9.3.3 改进欧拉法求解

对于解初值问题

$$\begin{cases} y' = y - \frac{2x}{y} \quad (0 \le x \le 1) \\ y(0) = 1 \end{cases}$$

的改进欧拉公式的具体形式为

$$\begin{cases} y_p = y_n + h(y_n - \frac{2x_n}{y_n}) \\ y_c = y_n + h(y_p - \frac{2x_{n+1}}{y_p}) \\ y_{n+1} = \frac{1}{2}(y_p + y_c) \end{cases}$$

取步长 $h = 0.1$ 结果为

Fortran 代码如下:

```
!the imroved_euler_method of y'=y-2x/y (0<=x<=1)
program improved_euler_method
    implicit none
    real :: x , y , yp , yc , h
    h =  0.1
    x =  0.0
```

x_n	精确解 (y_n)	欧拉法解 (y)	改进欧拉法解 (y)
0.0	1.000000	1.000000	1.000000
0.1	1.095445	1.100000	1.095909
0.2	1.183216	1.191818	1.184096
0.3	1.264911	1.277438	1.266201
0.4	1.341641	1.358213	1.343360
0.5	1.414214	1.435133	1.416402
0.6	1.483240	1.508966	1.485955
0.7	1.549193	1.580338	1.552514
0.8	1.612452	1.649784	1.616474
0.9	1.673320	1.717780	1.678166
1.0	1.732051	1.784771	1.737867

图 9.5: 精确解、欧拉法解、改进欧拉法解

图 9.6: 精确解、欧拉法解、改进欧拉法解

```
    y =  1.0
    open ( unit = 10 , file = 'improved_euler_method.ods' )
    write ( 10 , 100 ) x , y
    do x = 0.0 , 0.9 , 0.1
        yp = y + h * ( y - 2.0 * x / y )
        yc = y + h * ( yp - 2.0 * ( x + 0.1 ) / yp )
        y = 1.0 / 2.0 * ( yp + yc ) !notes: the data type must be consistent! 1.0/2.0 can not be 1/2.
        write ( 10 , 100 ) x + 0.1 , y
    enddo
    100 format ( 1x , f10.6 , f10.6 )
end program improved_euler_method
```

9.3.4 四阶龙格-库塔法求解

对于解初值问题

$$\begin{cases} y' = y - \frac{2x}{y} \quad (0 \le x \le 1) \\ y(0) = 1 \end{cases}$$

的四阶龙格-库塔公式的具体形式为

$$\begin{cases} y_{n+1} = y_n + \dfrac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ K_1 = y_n - \dfrac{2x_n}{y_n} \\ K_2 = y_n + \dfrac{h}{2}K_1 - \dfrac{2x_n + h}{y_n + \dfrac{h}{2}K_1} \\ K_3 = y_n + \dfrac{h}{2}K_2 - \dfrac{2x_n + h}{y_n + \dfrac{h}{2}K_2} \\ K_4 = y_n + hK_3 - \dfrac{2(x_n + h)}{y_n + hK_3} \end{cases}$$

Fortran 代码如下：

```
!the 4 order runge_kutta of y'=y-2x/y (0<=x<=1)
program runge_kutta
    implicit none
    real :: x , y , h , k1 , k2 , k3 , k4
    h = 0.1
    x = 0.0
    y = 1.0
    open ( unit = 10 , file = 'runge_kutta.ods' )
    write ( 10 ,100 ) x , y
    do x = 0.0 , 0.9 , 0.1
        k1 = y - 2.0 * x / y
        k2 = y + h / 2.0 * k1 - ( 2.0 * x + h ) / ( y + h / 2.0 * k1)
        k3 = y + h / 2.0 * k2 - ( 2.0 * x + h ) / ( y + h / 2.0 * k2 )
        k4 = y + h * k3 - 2.0 * ( x + h ) / ( y + h * k3 )
        y = y + h / 6.0 * ( k1 + 2.0 * k2 + 2.0 * k3 + k4 )
        write ( 10 ,100 ) x + 0.1 , y
    enddo
    100 format ( 1x , f10.6 , f10.6 )
end program runge_kutta
```

取步长 $h = 0.1$ 结果为

四阶龙格库塔法的局部截断误差是 $O(h^5)$，步长为 $h = 0.1$ 时，$h^5 = 0.00001$。

这个四阶龙格库塔的程序只适用于解这个问题，不具有普遍性，在四阶龙格库塔法程序章节中，将给出具有普遍性的四阶龙格库塔法的程序，而且适用于微分方程组。

9.4 四阶龙格库塔法程序

参考资料：《数值分析》（第 4 版），李庆扬，王能超，易大义编；清华大学出版社

龙格-库塔法的推导基于泰勒展开，因而它要求所求的解具有较好的光滑性质。反之，如果解的光滑性差，那么实用四阶龙格-库塔法求得的数值解，其精度可以能反而不如改进的欧拉法。

一阶常微分方程的一般形式为：

$$\begin{cases} \dfrac{dy}{dx}=f(x,y), \quad a\le x\le b \\ y(a)=c \end{cases}$$

在求解常微分方程时，通常会给出初始条件 x_0 及 y_0。采用龙格库塔法求解常微分方程时，如果 $x_1=x_0+h$，则对应的 y_1 为：

$$y_1=y_0+\frac{h}{6}(K_1+2\times K_2+2\times K_3+K_4)$$

其中 4 个参数为：

$$\begin{aligned} K_1&=f(x_0,y_0) \\ K_2&=f(x_0+\frac{h}{2},y_0+\frac{h}{2}K1) \\ K_3&=f(x_0+\frac{h}{2},y_0+\frac{h}{2}K2) \\ K_4&=f(x_0+h,y_0+hK_3) \end{aligned}$$

方程确定了，它的解也确定了[3]。数值解是近似解，不同的步长有不同的精度，但是不管取何种步长，得出的解都是向精确解靠拢的。所以，当我们对同一个 x 值，用不同步长的到的 y 值相差很少时，说明这是时的解与精确解也相差很小，如果满足我们的精度需求，我们就可以取这时的 y 值。

9.4.1 四阶龙格库塔法求解一阶微分方程组

前面都是单个方程 $y'=f$ 的数值解法，只要把 y 和 f 理解为向量，那么，所提供的各种计算公式即可应用到一阶方程组的情形。（还真没理解到?）

考察一阶方程组

$$y'_i=f_i(x,y_1,y_2,\cdots,y_N) \qquad (i=1,2,\cdots,N)$$

的初值问题，初始条件给为

$$y_i(x_0)=y_{i;0} \qquad (i=1,2,\cdots,N)$$

[3]当然了，这个方程是有解的；还有就是有了初始条件或边界条件是唯一解，没这个是通解

若采用向量的记号，记

$$\vec{y}=\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \qquad \vec{y_0}=\begin{pmatrix} y_{1;0} \\ y_{2;0} \\ \vdots \\ y_{N;0} \end{pmatrix} \qquad \vec{f}=\begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$$

则上述方程组的初值问题可表示为

$$\begin{cases} \vec{y}'=\vec{f}(x,\vec{y}) \\ \vec{y}(x_0)=\vec{y_0} \end{cases}$$

求解这一初值问题的四阶龙格-库塔公式为

（1）

$$\vec{y}_{n+1}=\vec{y}_n+\frac{h}{6}(\vec{k}_1+2\vec{k}_2+2\vec{k}_3+\vec{k}_4)$$

其中

$$\begin{cases} \vec{k}_1=\vec{f}(x_n,\vec{y}_n) \\ \vec{k}_2=\vec{f}(x_n+\frac{h}{2},\vec{y}_n+\frac{h}{2}\vec{k}_1) \\ \vec{k}_3=\vec{f}(x_n+\frac{h}{2},\vec{y}_n+\frac{h}{2}\vec{k}_2) \\ \vec{k}_4=\vec{f}(x_n+h,\vec{y}_n+h\vec{k}_3) \end{cases}$$

或表示为

（2）

$$y_{i;n+1}=y_{i;n}+\frac{h}{6}(K_{i;1}+2K_{i;2}+2K_{i;3}+K_{i;4}) \qquad (i=1,2,\cdots,N)$$

其中

$$\begin{cases} K_{i;1}=f_i(x_n,y_{1;n},y_{2;n},\cdots,y_{N;n}) \\ K_{i;2}=f_i(x_n+\frac{h}{2},y_{1;n}+\frac{h}{2}K_{1;1},y_{2;n}+\frac{h}{2}K_{2;1},\cdots,y_{N;n}+\frac{h}{2}K_{N;1}) \\ K_{i;3}=f_i(x_n+\frac{h}{2},y_{1;n}+\frac{h}{2}K_{1;2},y_{2;n}+\frac{h}{2}K_{2;2},\cdots,y_{N;n}+\frac{h}{2}K_{N;2}) \\ K_{i;4}=f_i(x_n+h,y_{1;n}+hK_{1;3},y_{2;n}+hK_{2;3},\cdots,y_{N;n}+hK_{N;3}) \end{cases}$$

本质上就是要包含，各个应变量，在 hierachy 中，i 由 n,j_1,j_2,j_3 四个数来表征，可以考虑用四维数组表示！

下标多了，容易搞混，一定要搞清楚。

$y_{i;n}$ 是第 i 个因变量 $y_i(x)$ 在节点 $x_n=x_0+nh$ 的近似值。分号“;”属于下标中，不要搞混。下标 i（下标中的第一个数字）表征的是方程组中的方程序号，指的是具体的哪个方程，搞清到底有多少个方程是必要的！

9.4.1.1 便于编程的形式

为了方便编程，将方程组中的方程浓缩为一个一维数组来表示，长度就是方程组的个数。这意味着，矩阵运算将很方便。而且龙格库塔法作为一个子程序，循环利用，我

们只需考虑第一步和第二步，始终是前一步是后一步的初值。

$$y_1(i)=y_0(i)+\frac{h}{6}\Big(K_1(i)+2K_2(i)+2K_3(i)+K_4(i)\Big)\qquad (i=1,2,\cdots,N)$$

可见 N 就是方程组中方程的个数。

其中

$$\begin{cases}K_1(i)=f_i\big(x_0,y_0(1),y_0(2),\cdots,y_0(N)\big)\\ \qquad x_0'=x_0+\frac{h}{2},\quad y_0'(1)=y_0(1)+\dfrac{h}{2}K_1(1),\quad y_0'(2)=y_0(2)+\dfrac{h}{2}K_1(2),\cdots,y_0'(N)=y_0(N)+\frac{h}{2}K_1(N\\ K_2(i)=f_i\big(x_0',y_0'(1),y_0'(2),\cdots,y_0'(N)\big)\\ \qquad x_0''=x_0+\frac{h}{2},\quad y_0''(1)=y_0(1)+\dfrac{h}{2}K_2(1),\quad y_0''(2)=y_0(2)+\dfrac{h}{2}K_2(2),\cdots y_0''(N)=y_0(N)+\frac{h}{2}K_2(N\\ K_3(i)=f_i\big(x_0'',y_0''(1),y_0''(2),\cdots,y_0''(N)\big)\\ \qquad x_0'''=x_0+h,\quad y_0'''(1)=y_0(1)+hK_3(1),\quad y_0'''(2)=y_0(2)+hK_3(2),\cdots,y_0'''(N)=y_0(N)+hK\\ K_4(i)=f_i\big(x_0''',y_0'''(1),y_0'''(2),\cdots,y_0'''(N)\big)\end{cases}$$

经过以上的变形，$K_1(i),K_2(i),K_3(i),K_4(i)$ 基本就有了个一致的造型，就是变量不同，而变量不同，另外赋值就可以了。可见代出初值进方程这个过程是可以重复利用的。知道注意的是，程序中，除了 $f_i(\cdots)$ 是个函数外，y 什么的都是数了。

非数学语言表述就是：

1. 代入初值求得各个方程的值 $f_i(\cdots)$，表示为 $K_{1;i}$
2. 对变量进行处理，又作为初值代入各个方程中，求得其值 $f_i(\cdots)$，只不过这次求的值表示为 $k_{2;i}$。变量处理是，前次 x 值加半个步长 $\frac{h}{2}$，即 $x'_0=x_0+\frac{h}{2}$；而 y 是前次 y_i 加乘以半个步长 $\frac{h}{2}$ 对应的 $K_{1;i}$，即 $y'_i=y_i+\frac{h}{2}K_{1;i}$
3. 又对变量进行处理，又作为初值代入各个方程中，求得其值 $f_i(\cdots)$，这次求的值表示为 $k_{3;i}$。这次变量处理与上次相似，只是 $K_{1;i}$ 变成 $K_{2;i}$。
4. 再次对变量进行处理，又作为初值代入各个方程中，求得其值 $f_i(\cdots)$，这次求的值表示为 $k_{4;i}$。这次变量处理与上两回稍有不同，乘的不是半步长，而是一个步长。

上面的数学描述是针对一维数组的形式。但是只要实现非数学语言表述的求值过程，采用任何形式都是可以的。

具体的程序，参见举例。

9.4.1.2 举例说明

为了理解这一公式的计算过程，考察两个方程的特殊情形：

$$\begin{cases}y_1'=f_1(x,y_1,y_2)\\ y_2'=f_2(x,y_1,y_2)\\ y_1(x_0)=y_{1;0}\\ y_2(x_0)=y_{2;0}\end{cases}$$

四阶龙格-库塔公式的具体形式为

$$\begin{cases} y_{1;n+1} = y_{1;n} + \frac{h}{6}(K_{1;1} + 2K_{1;2} + 2K_{1;3} + K_{1;4}) \\ y_{2;n+1} = y_{2;n} + \frac{h}{6}(K_{2;1} + 2K_{2;2} + 2K_{2;3} + K_{2;4}) \\ y_1(x_0) = y_{1;0} \\ y_2(x_0) = y_{2;0} \end{cases}$$

其中

$$\begin{cases} K_{1;1} = f_1(x_n, y_{1;n}, y_{2;n}) \\ K_{2;1} = f_2(x_n, y_{1;n}, y_{2;n}) \\ K_{1;2} = f_1(x_n + \frac{h}{2}, y_{1;n} + \frac{h}{2}K_{1;1}, y_{2;n} + \frac{h}{2}K_{2;1}) \\ K_{2;2} = f_2(x_n + \frac{h}{2}, y_{1;n} + \frac{h}{2}K_{1;1}, y_{2;n} + \frac{h}{2}K_{2;1}) \\ K_{1;3} = f_1(x_n + \frac{h}{2}, y_{1;n} + \frac{h}{2}K_{1;2}, y_{2;n} + \frac{h}{2}K_{2;2}) \\ K_{2;3} = f_2(x_n + \frac{h}{2}, y_{1;n} + \frac{h}{2}K_{1;2}, y_{2;n} + \frac{h}{2}K_{2;2}) \\ K_{1;4} = f_1(x_n + \frac{h}{2}, y_{1;n} + \frac{h}{2}K_{1;3}, y_{2;n} + hK_{2;3}) \\ K_{2;4} = f_2(x_n + \frac{h}{2}, y_{1;n} + \frac{h}{2}K_{1;3}, y_{2;n} + hK_{2;3}) \end{cases}$$

编程的时候，不分第 n 个和 $n+1$ 个，只有第 0 个和第 1 个值，其他循环套用就是了，即

$$\begin{cases} y_{1;1} = y_{1;0} + \frac{h}{6}(K_{1;1} + 2K_{1;2} + 2K_{1;3} + K_{1;4}) \\ y_{2;1} = y_{2;0} + \frac{h}{6}(K_{2;1} + 2K_{2;2} + 2K_{2;3} + K_{2;4}) \\ y_1(x_0) = y_{1;0} \\ y_2(x_0) = y_{2;0} \end{cases}$$

其中

$$\begin{cases} K_{1;1} = f_1(x_0, y_{1;0}, y_{2;0}) \\ K_{2;1} = f_2(x_0, y_{1;0}, y_{2;0}) \\ K_{1;2} = f_1(x_0 + \frac{h}{2}, y_{1;0} + \frac{h}{2}K_{1;1}, y_{2;0} + \frac{h}{2}K_{2;1}) \\ K_{2;2} = f_2(x_0 + \frac{h}{2}, y_{1;0} + \frac{h}{2}K_{1;1}, y_{2;0} + \frac{h}{2}K_{2;1}) \\ K_{1;3} = f_1(x_0 + \frac{h}{2}, y_{1;0} + \frac{h}{2}K_{1;2}, y_{2;0} + \frac{h}{2}K_{2;2}) \\ K_{2;3} = f_2(x_0 + \frac{h}{2}, y_{1;0} + \frac{h}{2}K_{1;2}, y_{2;0} + \frac{h}{2}K_{2;2}) \\ K_{1;4} = f_1(x_0 + \frac{h}{2}, y_{1;0} + \frac{h}{2}K_{1;3}, y_{2;0} + hK_{2;3}) \\ K_{2;4} = f_2(x_0 + \frac{h}{2}, y_{1;0} + \frac{h}{2}K_{1;3}, y_{2;0} + hK_{2;3}) \end{cases}$$

9.4.2 定步长通用四阶龙格库塔程序

为了有直观的理解，程序直接在举例中说明。对比这几个例子，可以发现，四阶龙格库塔法作为子程序 subroutine rk4(xin , yin , n , h , yout) ，已经无需改动。对于不同的方程组，需要改动的仅是部分主体程序和全新的方程组子程序。

9.4.2.1 一个微分方程求解举例

仍然采用前面的例子:

求解初值问题

$$\begin{cases} y' = y - \frac{2x}{y} & (0 \le x \le 1) \\ y(0) = 1 \end{cases}$$

定步长通用四阶龙格库塔程序为

```
subroutine rk4( xin , yin , n , h , yout )
    implicit none
    real*8 :: yin( 0 : n-1 ) , ya( 0 : n-1 ) , yb( 0 : n-1 ) , yc( 0 : n-1 ) , yout( 0 : n-1 )
    real*8 :: k1( 0 : n-1 ) , k2( 0 : n-1 ) , k3( 0 : n-1 ) , k4( 0 : n-1 )
    real*8 :: h , h2 , h6 , xin , xh
    integer :: i , n
    h2 = h / 2.0d0
    h6 = h / 6.0d0
    xh = xin + h2
    call equation( xin , yin , k1 )
    do i = 0 , n-1
        ya(i) = yin(i) + h2 * k1(i)
    enddo
    call equation( xh , ya , k2 )
    do i = 0 , n-1
        yb(i) = yin(i) + h2 * k2(i)
    enddo
    call equation( xh , yb , k3 )
    do i = 0 , n-1
        yc(i) = yin(i) + h * k3(i)
    enddo
    call equation( xin+h , yc , k4 )
    do i = 0 , n-1
        yout(i) = yin(i) + h6 * ( k1(i) + 2.0d0 * k2(i) + 2.0d0 * k3(i) + k4(i) )
    enddo
end subroutine rk4


subroutine equation( x , y , dydx )
    real*8 :: y( 0: 0 ) , dydx( 0 : 0 )
    real*8 :: x
    dydx(0) = y(0) - 2.0d0 * x / y(0)
end subroutine equation

program runge_kutta1
    real*8 :: y( 0 : 0 ) , dydx( 0 : 0 )
    real*8 :: x0 , y0( 0 : 0 ) , y1( 0 : 0 )  , h
    integer :: n
    x0 = 0d0
    y0 = 1d0
    h = 0.1d0
    n = 1
```

```
    open ( unit = 10 , file = 'out.ods' )
    write ( 10 , * ) x0 ," ", y0
    do x0 =0d0 , 0.9d0 , 0.1d0 !数据类型，在整个程序中一定要写对，不然会出问题的。
        call rk4( x0 , y0 , n , h , y1 )
        y0 = y1 !这个是关键步骤之一，就是再次作为程序初值，没有这步，语法上没错，但算的总不对。因为初值不对，所以一定要注意。
        write( 10 , * ) x0+0.1d0 ," ", y1
    enddo
endprogram runge_kutta1
```

求解结果为:

x_n	精确解	四阶龙格库塔解	加强精度精确解	通用四阶龙格库塔程序解	误差
0.0	1.000000	1.000000	1.00000000000000	1.00000000000000	0
0.1	1.095445	1.095446	1.09544511501033	1.09544553169309	-4.16682760073783E-007
0.2	1.183216	1.183217	1.18321595661992	1.18321674550599	-7.88886070024475E-007
0.3	1.264911	1.264912	1.26491106406735	1.26491222834039	-1.16427303997746E-006
0.4	1.341641	1.341642	1.34164078649987	1.34164235375037	-1.56725050004525E-006
0.5	1.414214	1.414215	1.41421356237310	1.41421557789008	-2.0155169799807E-006
0.6	1.483240	1.483242	1.48323969741913	1.48324222277199	-2.52535285993893E-006
0.7	1.549193	1.549196	1.54919333848297	1.54919645230214	-3.11381917006415E-006
0.8	1.612452	1.612455	1.61245154965971	1.61245534965899	-0.0000038
0.9	1.673320	1.673324	1.67332005306815	1.67332465901626	-4.60594811002579E-006
1.0	1.732051	1.732056	1.73205080756888	1.73205636516556	-5.55759668019462E-006

误差是“加强精度精确解”减“通用四阶龙格库塔程序解”所得的，看到冒出了很多小数位，那时因为最多显示 14 位小数所致。从误差可以看出，四阶龙格库塔法的解是相当精确的。步长是 0.1，局部截断误差 0.1^5=1.0E-004，实际误差比局部截断误差小很多。

9.4.2.2 微分方程组求解举例

用四阶龙格-库塔法求解常微分方程组

$$\begin{cases} \dfrac{dy_0}{dx} = y_1 \\ \dfrac{dy_1}{dx} = -y_0 \\ \dfrac{dy_2}{dx} = -y_2 \\ y_0(0) = -1, \quad y_1(0) = 0, \quad y_2(0) = 1 \end{cases} \qquad 0 \le x \le 1$$

这个问题的解析解是

$$y_0 = -\cos x \qquad y_1 = \sin x \qquad y_2 = e^{-x}$$

解析解的程序为

```
program example3
    real*8 :: x , y0 , y1 , y2 , h
    h = 1.0d0 / 20.0d0
    open ( unit = 10 , file = "example3.ods" )
    do x = 0.0d0 , 1.0d0 , h
        y0 = - cos( x )
        y1 = sin ( x )
        y2 = exp ( -x )
        write ( * ,100 )x, y0 , y1 , y2
        write ( 10 , 100 )x, y0 , y1 , y2
    enddo
    100 format ( 1x , f19.8 ,f19.8 , f19.8 , f19.8 )
end program example3
```

定步长通用四阶龙格库塔程序为

```
subroutine rk4( xin , yin , n , h , yout )
    implicit none
    real*8 :: yin( 0 : n-1 ) , ya( 0 : n-1 ) , yb( 0 : n-1
        ) , yc( 0 : n-1 ) , yout( 0 : n-1 )
    real*8 :: k1( 0 : n-1 ) , k2( 0 : n-1 ) , k3( 0 : n-1
       ) , k4( 0 : n-1 )
    real*8 :: h , h2 , h6 , xin , xh
    integer :: i , n
    h2 = h / 2.0d0
    h6 = h / 6.0d0
    xh = xin + h2
    call equation( xin , yin , k1 )
    do i = 0 , n-1
        ya(i) = yin(i) + h2 * k1(i)
    enddo
    call equation( xh , ya , k2 )
    do i = 0 , n-1
        yb(i) = yin(i) + h2 * k2(i)
    enddo
    call equation( xh , yb , k3 )
    do i = 0 , n-1
        yc(i) = yin(i) + h * k3(i)
    enddo
    call equation( xin+h , yc , k4 )
    do i = 0 , n-1
        yout(i) = yin(i) + h6 * ( k1(i) + 2.0d0 * k2(i) +
           2.0d0 * k3(i) + k4(i) )
    enddo
end subroutine rk4


subroutine equation( x , y , dydx )
    real*8 :: y( 0: 2 ) , dydx( 0 : 2 )
    real*8 :: x
    dydx(0) = y(1)
    dydx(1) = -y(0)
    dydx(2) = -y(2)
end subroutine equation

```

```

program runge_kutta1
    real*8 :: x0 , y0( 0 : 2 ) , y1( 0 : 2 )  , h
    integer :: n
    x0 = 0d0
    y0(0) = -1.0d0
    y0(1) = 0.0d0
    y0(2) = 1.0d0
    h = 0.05d0
    n = 3
    open ( unit = 10 , file = 'runge_kutta3.ods' )
    write ( 10 , 100 ) x0 , y0 !输入到文本文件中
    write ( * , 100 ) x0 , y0  !输入到显示屏上
    do x0 =0d0 , 1.0d0 , h !数据类型，在整个程序中一定要写对，不然会出问题的。
        call rk4( x0 , y0 , n , h , y1 )
        y0 = y1 !这个是关键步骤之一，就是再次作为程序初值，没有这步，语法上没错，但算的总不对。因为初值不对，所以一定要注意。
        write( * , 100 ) x0+h ,  y1 !输入到文本文件中
        write( 10 , 100 ) x0+h , y1 !输入大显示屏上
    enddo
    100 format ( 1x , f19.8 , f19.8 , f19.8 , f19.8 )
endprogram runge_kutta1
```

定步长四阶龙格库塔法求解结果为

x_n	y_0	y_1	y_2
0.00000000	-1.00000000	0.00000000	1.00000000
0.05000000	-0.99875026	0.04997917	0.95122943
0.10000000	-0.99500417	0.09983341	0.90483742
0.15000000	-0.98877108	0.14943812	0.86070798
0.20000000	-0.98006658	0.19866932	0.81873076
0.25000000	-0.96891242	0.24740395	0.77880079
0.30000000	-0.95533649	0.29552019	0.74081823
0.35000000	-0.93937272	0.34289779	0.70468810
0.40000000	-0.92106100	0.38941832	0.67032006
0.45000000	-0.90044711	0.43496551	0.63762817
0.50000000	-0.87758257	0.47942552	0.60653068
0.55000000	-0.85252454	0.52268720	0.57694983
0.60000000	-0.82533563	0.56464245	0.54881165
0.65000000	-0.79608382	0.60518638	0.52204580
0.70000000	-0.76484221	0.64421766	0.49658532
0.75000000	-0.73168889	0.68163873	0.47236657
0.80000000	-0.69670674	0.71735606	0.44932898
0.85000000	-0.65998318	0.75128037	0.42741495
0.90000000	-0.62161000	0.78332688	0.40656968
0.95000000	-0.58168313	0.81341547	0.38674104
1.00000000	-0.54030235	0.84147095	0.36787946

解析求解的结果及两种解误差为

x_n	y_0 解析解	y_1 解析解	y_2 解析解	y_0 误差	y_1 误差	y_2 误差
0.00	-1.00000000	0.00000000	1.00000000	0	0	0
0.05	-0.99875026	0.04997917	0.95122942	0	0	0.00000001
0.10	-0.99500417	0.09983342	0.90483742	0	-0.00000001	0
0.15	-0.98877108	0.14943813	0.86070798	0	-0.00000001	0
0.20	-0.98006658	0.19866933	0.81873075	0	-0.00000001	0.00000001
0.25	-0.96891242	0.24740396	0.77880078	0	-0.00000001	0.00000001
0.30	-0.95533649	0.29552021	0.74081822	0	-0.00000002	0.00000001
0.35	-0.93937271	0.34289781	0.70468809	-0.00000001	-0.00000002	0.00000001
0.40	-0.92106099	0.38941834	0.67032005	-0.00000001	-0.00000002	0.00000001
0.45	-0.90044710	0.43496553	0.63762815	-0.00000001	-0.00000002	0.00000002
0.50	-0.87758256	0.47942554	0.60653066	-0.00000001	-0.00000002	0.00000002
0.55	-0.85252452	0.52268723	0.57694981	-0.00000002	-0.00000003	0.00000002
0.60	-0.82533561	0.56464247	0.54881164	-0.00000002	-0.00000002	0.00000001
0.65	-0.79608380	0.60518641	0.52204578	-0.00000002	-0.00000003	0.00000002
0.70	-0.76484219	0.64421769	0.49658530	-0.00000002	-0.00000003	0.00000002
0.75	-0.73168887	0.68163876	0.47236655	-0.00000002	-0.00000003	0.00000002
0.80	-0.69670671	0.71735609	0.44932896	-0.00000003	-0.00000003	0.00000002
0.85	-0.65998315	0.75128041	0.42741493	-0.00000003	-0.00000004	0.00000002
0.90	-0.62160997	0.78332691	0.40656966	-0.00000003	-0.00000003	0.00000002
0.95	-0.58168309	0.81341550	0.38674102	-0.00000004	-0.00000003	0.00000002
1.00	-0.54030231	0.84147098	0.36787944	-0.00000004	-0.00000003	0.00000002

步长为 $h = 0.05$，局部截断误差为 $h^5 = 0.05^5 = 3.125 \times 10^{-7}$

9.4.3 自适应步长通用四阶龙格库塔程序

9.4.3.1 编程思想

单从每一步看，步长越小，截断误差就越小，但随着步长的缩小，在一定求解范围内所要完成的步数就增加了。步数的增加不但引起计算量的增大，而且可能导致舍入误差的严重积累。因此微分方程的数值解法有个选择步长的问题。

在选择步长时，需要考虑两个问题：

1、怎样衡量和检验计算结果的精度?

2、如何依据所获得的精度处理步长?

以四阶龙格-库塔公式为例，从节点 x_n 出发，先以 h 为步长求出一个近似值，记为 $y_{h;n+1}$，由于公式的局部截断误差万恶诶 $O(h^5)$，故有

$$y(x_{n+1}) - y_{h;n+1} \approx ch^5$$

其中 $y(x_{n+1})$ 是 x_{n+1} 处的精确解；$y_{h;n+1}$ 是 x_{n+1} 的数值近似解，h 表示求解步长。

然后将步长折半，即取 $\frac{h}{2}$ 为步长。从 x_n 跨两步到 x_{n+1}，再求得一个近似值 $y_{\frac{h}{2};n+1}$，每跨一步的截断误差是 $c(\frac{h}{2})^2$，即

$$y(x_{n+1}) - y_{h;n+1} \approx 2c\left(\frac{h}{2}\right)^5$$

比较上面两个截断误差公式，步长折半后，误差大约减少到 $\frac{1}{16}$，即

$$\frac{y(x_{n+1})-y_{\frac{h}{2};n+1}}{y(x_{n+1})-y_{h;n+1}}\approx\frac{1}{16}$$

有

$$y(x_{n+1})-y_{\frac{h}{2};n+1}\approx\frac{1}{16}y(x_{n+1})-\frac{1}{16}y_{h;n+1}\Longrightarrow\frac{15}{16}y(x_{n+1})\approx y_{\frac{h}{2};n+1}-\frac{1}{16}y_{h;n+1}\Longrightarrow y(x_{n+1})\approx\frac{16}{15}y_{\frac{h}{2};n+1}-$$

$$y(x_{n+1})-y_{\frac{h}{2};n+1}\approx\frac{1}{15}\left[y_{\frac{h}{2};n+1}-y_{h,n+1}\right]$$

这样我们就可以通过折半前后两次计算结果的偏差

$$\Delta\equiv y_{\frac{h}{2};n+1}-y_{h,n+1}$$

来判定所选的步长是否合适，具体地说，将区分以下两种情况处理：

1、对于给定的精度 ϵ，如果 $\Delta>\epsilon$，我们反复将步长折半进行计算，直至 $\Delta<\epsilon$ 为止，这时取最终得到的 $y_{\frac{h}{2},n+1}$[4]作为结果；

2、如果 $\Delta<\epsilon$，我们将反复将步长加倍，直到 $\Delta>\epsilon$ 为止，这时再将步长折半一次，就得到所要的结果。

显然通常情况下，我们只需考虑第一种情况就可以了，因为总要进行一次折半步长运算，所以变步长龙格库塔法步长可以选则比定步长龙格库塔法大些，大两倍。因为真实用到的步长是预先定义步长 h 的一半 $\frac{h}{2}$。也无需考虑第二种情况，第一次步长折半算到的值就满足精度的话，就通过，直接算下一步。这里的两种情况是为了选择最佳步长，而我们的目的不是选则最佳步长，而是求得满足精度的值，所以无需考虑第二种情况。

[4]这是经过 n 折半后的步长计算值，此时真实步长是 $(\frac{h}{2})^n$

x_n	精确解 (y_n)	欧拉法解 (y)	改进欧拉法解 (y)	四阶龙格库塔法解（y）
0.0	1.000000	1.000000	1.000000	1.000000
0.1	1.095445	1.100000	1.095909	1.095446
0.2	1.183216	1.191818	1.184096	1.183217
0.3	1.264911	1.277438	1.266201	1.264912
0.4	1.341641	1.358213	1.343360	1.341642
0.5	1.414214	1.435133	1.416402	1.414215
0.6	1.483240	1.508966	1.485955	1.483242
0.7	1.549193	1.580338	1.552514	1.549196
0.8	1.612452	1.649784	1.616474	1.612455
0.9	1.673320	1.717780	1.678166	1.673324
1.0	1.732051	1.784771	1.737867	1.732056

图 9.7: 精确解、欧拉法解、改进欧拉法解、四阶龙格库塔法解

图 9.8: 精确解、欧拉法解、改进欧拉法解、四阶龙格库塔法解

第十章 堆排序

http://blog.csdn.net/morewindows/article/details/6709644/ http://www.cnblogs.com/mengdd/archive/2012/11/30/2796845.html http://www.cnblogs.com/jingmoxukong/p/4303826.html

堆排序（Heap Sort），是一种选择排序，其时间复杂度为 $O(n\log n)$。

堆排序方法对记录数较少的文件并不值得提倡，但对 n 较大的文件还是很有效的。因为其运行时间主要耗费在建初始堆和调整建新堆时进行的反复“筛选”上。

堆排序在最坏的情况下，其时间复杂度也为 $O(n\log n)$。相对于快速排序来说，这是堆排序的最大优点。此外，堆排序仅需一个记录大小的供交换用的辅助存储空间。

10.1 堆的定义

数据结构中的二叉堆，是完全二叉树或者是近似完全二叉树。满足的特性为：

- 父结点的键值总是大于或等于（小于或等于）任何一个子节点的键值；
- 每个结点的左子树和右子树都是一个二叉堆.

当父结点的键值总是大于或等于任何一个子节点的键值时为最大堆。当父结点的键值总是小于或等于任何一个子节点的键值时为最小堆。下图展示一个最小堆：

图 10.1: 小顶堆

由于其它几种堆（二项式堆，斐波纳契堆等）用的较少，一般将二叉堆就简称为堆。

10.2 堆的储存

一般用数组来表示堆。

（1）以下标为“0”开始的 n 个元素的序列 $\{k_0, k_1, \cdots, k_{n-1}\}$ 当且仅当满足下列关系时

- 情形 1：$\begin{cases} k_i \le k_{2i+1} \\ k_i \le k_{2i+2} \end{cases}$，称之为最小化堆或小顶堆；

- 情形 2：$\begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}$，称之为最大化堆或大顶堆。

其中 $i = 0, 1, 2, \cdots, (n/2 - 1)$ 向下取整。

（2）以下标为 “1” 开始的 n 个元素的序列 $\{k_1, k_2, \cdots, k_n\}$ 当且仅当满足下列关系时

- 情形 1：$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases}$，称之为最小化堆或小顶堆；
- 情形 2：$\begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}$，称之为最大化堆或大顶堆。

其中 $i = 1, 2, \cdots, n/2$ 向下取整。

10.3 堆排序的思想

有了堆的定义，就自然过渡到堆排序了。一个无序数组，如果构造成了大顶堆，这就筛选出了这个数组的最大值，即第一项（堆顶值）。此时，我们把这个最大值取出，余下的部分继续构造成大顶堆，继续取出堆顶值，反复进行就实现了原来数组的排序。但是具体操作可以变得更为有效：排序开始，首先输出堆顶元素（因为它是最值），将堆顶元素和最后一个元素交换，这样，第 n 个位置（即最后一个位置）作为有序区，前 $n-1$ 个位置仍是无序区，对无序区进行调整，得到堆之后，再交换堆顶和最后一个元素，这样有序区长度变为 $2\cdots$ 不断进行此操作，将剩下的元素重新调整为堆，然后输出堆顶元素到有序区。每次交换都导致无序区 -1，有序区 $+1$。不断重复此过程直到有序区长度增长为 $n-1$，排序完成。由排序过程可见，若想得到升序，则建立大顶堆，若想得到降序，则建立小顶堆。

在将堆顶元素和最后一个元素交换后形成的无序区重新构建堆也是有规律可循的。这也是堆调整的最关键部分。每个子树根都是该子树分支的最值。当堆的根元素与最后一个元素交换后，重新构建堆，实际就是将处于根位置的新元素不断与子结点比较，下沉到树中的合适位置。每次完成比较，另一支子树就不用考虑，减少 1/2 需要比较的数据量。这是堆的特性所决定的，因为每个父节点都是所处树分枝的最值，无论这个树分枝是大还是小。这是堆调整的核心要点。后面会有具体例子说明。

以上思想可归纳为两个操作：

(1) 根据初始数组去构造初始堆（以大顶堆为例，构建一个完全二叉树，保证所有的父结点都比它的孩子结点数值大）。初始化堆的时候是对所有的非叶子结点进行筛选。最后一个非终端元素的下标是 $n/2$ 向下取整，所以筛选只需要从第 $n/2$ 向下取整个元素开始，从后往前进行调整。每次都是从父结点、左孩子、右孩子中进行比较交换，交换可能会引起孩子结点不满足堆的性质，所以每次交换之后需要重新对被交换的孩子结点进行调整。

(2) 每次交换第一个和最后一个元素，输出最后一个元素（最大值），然后把剩下元素重新调整为大根堆。其中调整建堆的方法是，将根结点值与左、右子树的根结点值

进行比较，若不满足堆的条件，则将左、右子树根结点值中的大者与根结点值进行交换。未被破坏的子树就不用管了，然后顺着被破坏的子树一路调整下去，直至叶子结点，就得到新的堆。（牢牢记住堆的特性，父结点是所处子树的最值。堆中有相同元素也没关系，堆的定义特性在于保证堆顶是最值。即堆只能保证堆中元素的最值。对于排序，有这点就够了。）

当输出完最后一个元素后，这个数组已经是按照从小到大的顺序排列了。

10.4 堆排序实例

（1）以构建大顶堆实现从小到大排序为例。建立初始的堆结构如图

（2）然后，交换堆顶的元素和最后一个元素，此时最后一个位置作为有序区（有序区显示为黄色）

（3）每个子树根都是该子树分支的最大值。而子树根 60 比子树根 12 大，12 所在的子树无需考虑，此时一半的数据量无需比较。10 比 60 小，60 是整个序列的第二大值。交换 10 和 60，此时筛选出了第二大值。但是为了保证余下的排序，仍要继续调整 10 至合适位置，直至重新构建成一个堆。

（4）10 不断下沉至合适位置，每次调整，减少余下的一半的数据去比较。此时蓝色部分重新构建称为堆。

（5）进入第二轮排序，60 与堆中最后一个元素交换位置。

（6）又开始调整堆。

（7）重复此过程。

（8）最后，有序区扩展完成即排序完成。

第十一章 差分进化算法

11.1 背景

https://www.cnblogs.com/tsingke/p/5809453.html

DE 算法-作者网站: http://www1.icsi.berkeley.edu/~storn/code.html

维基百科资料库: https://en.wikipedia.org/wiki/Differential_evolution

差分进化算法（Differential Evolution，DE）是由 Storn 等人于 1995 年提出的，和其它演化算法一样，DE 是一种模拟生物进化的随机模型，通过反复迭代，使得那些适应环境的个体被保存了下来。但相比于进化算法，DE 保留了基于种群的全局搜索策略，采用实数编码、基于差分的简单变异操作和一对一的竞争生存策略，降低了遗传操作的复杂性。同时，DE 特有的记忆能力使其可以动态跟踪当前的搜索情况，以调整其搜索策略，具有较强的全局收敛能力和鲁棒性，且不需要借助问题的特征信息，适于求解一些利用常规的数学规划方法所无法求解的复杂环境中的优化问题。目前，DE 已经在许多领域得到了应用，譬如人工神经元网络、化工、电力、机械设计、机器人、信号处理、生物信息、经济学、现代农业、食品安全、环境保护和运筹学等。

DE 算法主要用于求解连续变量的全局优化问题，其主要工作步骤与其他进化算法基本一致，主要包括变异 (Mutation)、交叉 (Crossover)、选择 (Selection) 三种操作。算法的基本思想是从某一随机产生的初始群体开始，利用从种群中随机选取的两个个体的差向量作为第三个个体的随机变化源，将差向量加权后按照一定的规则与第三个个体求和而产生变异个体，该操作称为变异。然后，变异个体与某个预先决定的目标个体进行参数混合，生成试验个体，这一过程称之为交叉。如果试验个体的适应度值优于目标个体的适应度值，则在下一代中试验个体取代目标个体，否则目标个体仍保存下来，该操作称为选择。在每一代的进化过程中，每一个体矢量作为目标个体一次，算法通过不断地迭代计算，保留优良个体，淘汰劣质个体，引导搜索过程向全局最优解逼近。

11.2 具体步骤

https://pablormier.github.io/2017/09/05/a-tutorial-on-differential-evolution-with-python/

https://blog.csdn.net/qq_37423198/article/details/77856744

目的 对于一个黑箱函数，有一组参数，设参数个数为 N。利用差分进化算法来寻找黑箱函数最小值的参数。

$$f(x_1, x_2, x_3, \cdots, x_N) \tag{11.1}$$

定义 将黑箱函数的一组参数称为个体；多组参数（个体）称为种群。

第○步　初始化种群（产生初始值）：产生大小为 M 的种群，即包含 M 组参数（M 个个体）。M 的选择一般介于参数个数 N 的 5 到 10 倍之间

- 设定每个参数的取值边界范围。

$$(b_{j-min}, b_{j-max}), \quad j = 1, 2, 3, \cdots, N \tag{11.2}$$

- 随机产生 M 个个体（M 组参数）；也可以加入自己设定的初始值。

$$X_i(0) = \{x_{i,1}(0), x_{i,2}(0), x_{i,3}(0), \cdots, x_{i,N}(0)\} \tag{11.3}$$

$$= \begin{pmatrix} x_{1,1}(0) & x_{1,2}(0) & x_{1,3}(0) & \cdots & x_{1,N}(0) \\ x_{2,1}(0) & x_{2,2}(0) & x_{2,3}(0) & \cdots & x_{2,N}(0) \\ x_{3,1}(0) & x_{3,2}(0) & x_{3,3}(0) & \cdots & x_{1,N}(0) \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ x_{M,1}(0) & x_{M,2}(0) & x_{M,3}(0) & \cdots & x_{M,N}(0) \end{pmatrix} \tag{11.4}$$

其中

$$x_{i,j}(0) = b_{j-min} + rand(0,1) * (b_{j-max} - b_{j-min}) \tag{11.5}$$

$$i = 1, 2, 3, \cdots, M \tag{11.6}$$

$$j = 1, 2, 3, \cdots, N \tag{11.7}$$

- 代入到黑箱函数计算，得出最优值。

第一步　变异（按照特定策略修改参数）：对之前产生的 M 个个体逐个进行变异。

- 变异方法：对 i 个个体进行变异，则再随机选择除第 i 个个体外的 3 个个体，表示为 a、b、c，以特定的变异策略计算变异体。存在多种变异策略，以最简单的 “rand1bin” 为例

$$x_{i-mut} = a + mut * (b - c) \tag{11.8}$$

其中 mut 是变异因子（缩放因子），取值一般在 $[0.5, 2]$ 之间。大的变异因子可以扩大搜索范围，但是会降低收敛速度。

- 变异策略：
  - Rand/1: $x_{mut} = x_{r1} + mut * (x_{r2} - x_{r3})$
  - Rand/2: $x_{mut} = x_{r1} + mut * (x_{r2} - x_{r3} + x_{r4} - x_{r5})$
  - Best/1: $x_{mut} = x_{best} + mut * (x_{r2} - x_{r3})$
  - Rand-to-best/1: $x_{mut} = x_{r1} + mut_1 * (x_{r2} - x_{r3}) + mut_2 * (x_{best} - x_{r1})$
  - $\cdots$
- 自适应变异因子：暂略！

第二步　交叉（将修改后的参数和原参数按照特定策略进行取舍，得到新参数）：将变异体和未变异体进行混合，即将两者参数进行混合。

- 交叉策略：对应于每个参数，产一个随机数，将这个随机数与设定交叉因子做对比。如果小于这个因子，就保留变异体中的这个参数，否在就保留原参数。这样就实现

了变异体和未变异体的混合。

$$x_{i,j}^{new} = \begin{cases} x_{i,j}^{mut}, & rand(0,1) \leq cr \\ x_{i,j}^{ori}, & else \end{cases} \tag{11.9}$$

其中 $cr \in [0,1]$ 为交叉概率。这个策略称为二项式交叉策略（Binomial crossover, bin）。还有在两组参数中随机挑选参数的指数交叉策略（Exponential crossover, exp）。

- 自适应交叉因子：暂略！

第三步 选择（对比新参数和原参数的函数值，保留最优值）

至此得到一个全新的种群。将这个全新的种群重复后三个步骤（变异、交叉、选择），进行迭代，直至找到最优解。

第十二章 数值计算库

12.1 Intel MKL

12.1.1 Intel MKL 简介

Intel 数学核心函数库（Math Kernel Library, MKL）是一套高度优化、线程安全的数学例程、函数，面向高性能的工程、科学与财务应用。英特尔 MKL 的集群版本包括 ScaLAPACK 与分布式内存快速傅立叶转换，并提供了线性代数 (BLAS、LAPACK 和 Sparse Solver)、快速傅立叶转换、矢量数学 (Vector Math) 与随机号码生成器支持。

主要包括：

LAPACK (线形代数工具 linear algebra package)

DFTs (离散傅立叶变换 Discrete Fourier transforms)

VML (矢量数学库 Vector Math Library)

VSL (矢量统计库 Vector Statistical Library)

MKL 的主要功能：

1）BLAS 和 LAPACK

在英特尔处理器中部署经过高度优化的基本线性代数例程 BLAS（Basic Linear Algebra Subroutines）和线性代数包 LAPACK（Linear Algebra Package）例程，它们提供的性能改善十分显著。

2）ScaLAPACK

ScaLAPACK 是一个并行计算软件包，适用于分布存储的 MIMD 并行机。ScaLAPACK 提供若干线性代数求解功能，具有高效、可移植、可伸缩、高可靠性的特点，利用它的求解库可以开发出基于线性代数运算的并行应用程序。ScaLAPACK 的英特尔? MKL 实施可提供显著的性能改进，远远超出标准 NETLIB 实施所能达到的程度。

3）PARDISO 稀疏矩阵解算器

利用 PARDISO 直接稀疏矩阵解算器解算大型的稀疏线性方程组，该解算器获得了巴塞尔大学的授权，是一款易于使用、具备线程安全性、高性能的内存高效型软件库。英特尔? MKL 还包含共轭梯度解算器和 FGMRES 迭代稀疏矩阵解算器。

4）快速傅立叶变换 (FFT)

充分利用带有易于使用的新型 C/Fortran 接口的多维 FFT 子程序（从 1 维至 7 维）。英特尔? MKL 支持采用相同 API 的分布式内存集群，支持将工作负载轻松地分布到大量处理器上，从而实现大幅的性能提升。此外，英特尔? MKL 还提供了一系列 C 语言例程（“wrapper”），这些例程可模拟 FFTW 2.x 和 3.0 接口，从而支持当前的 FFTW 用户将英特尔? MKL 集成到现有应用中。

5）矢量数学库（VML）

矢量数学库（Vector Math Library）借助计算密集型核心数学函数（幂函数、三角函数、指数函数、双曲函数、对数函数等）的矢量实施显著提升应用速度。

6）矢量统计库—随机数生成器（VSL）

利用矢量统计库（Vector Statistical Library）随机数生成器加速模拟，从而实现远远高于标量随机数生成器的系统性能提升。

12.1.2 命名规则

【注】这里只是简单的介绍，一切以 MKL 的技术手册为准，在技术手册的头几个章节就有说明。技术手册网上有，MKL 安装后也有份本地文件。这个在下面的使用心得中有详细介绍。

Naming Conventions

Argument types. Types of all arguments of Fortran routines in this Quick Reference follow these conventions:

- IMPLICIT COMPLEX (C)
- IMPLICIT DOUBLE PRECISION (D)
- IMPLICIT INTEGER (I-N)
- IMPLICIT DOUBLE COMPLEX (Z)
- CHARACTER*1 compq, compz, diag, eigsrc, howmny, initv, job, norm, order, range, side, trans, transa, transb, uplo, vect LOGICAL select

Arrays. Names of Fortran arrays are typeset in UPPERCASE letters, and names of all other Fortran arguments are in lowercase letters. Code examples and data types for C interface are also given in lowercase.

Names of routines. The first letter of a routine name identifies the type of the result:

- s SINGLE
- d DOUBLE PRECISION
- c COMPLEX
- z DOUBLE COMPLEX
- i INTEGER

Function names for C interface are given in lowercase courier mixed with UpperCase courier.

To specify the group of routine names that differ in the first letter, this Quick Reference uses the question mark (?). For example, ?getrf refers to the routines sgetrf, dgetrf, cgetrf, and zgetrf.

12.1.3 使用心得

MKL 用起来真的是不容易的，曲曲折折，当然了，这跟自己的 Linux 基础知识不够扎实有关。MKL 是优化了很多现有的优秀的数学函数库。我本来想在 C++ 直接调用这些库。后来觉得里面的 BLAS 原生就是用的 Fortran 写的，自己对 Fortran 熟悉，C++ 还不熟。所以还是一个台阶一个台阶的来吧，不然就函数接口而言估计都够折腾的，事实上 Fortran 95 的函数接口都折腾我够呛的。因为 BLAS 最初是 Fortran 77 写的。

先说下心得中最核心的一条：纵使你在网上搜到各种说法，万变不离其综的王道是，看 MKL 自带的技术手册，千多页呢，讲的不能再详细了。技术手册网上有，

http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/index.htm

安装后，本地文件也有。建议看本地文件的技术手册，与安装的版本对应，更全面，更直观。本地的技术手册，在安装目录下，Document 目录下找，所有的文档都在那。

第一天，云里雾里，什么都不知道。第二天，学会了看技术手册，是网页版的，从 mkl_documentation.htm 开始。知道了命名规则，及各项参数，成功的用 Fortran 77 函数接口调用了矩阵相乘的子程序。编译的时候很简单，多加个 -mkl 选项就可以了。以我的测试文件为例：`ifort test1_ver1.f90 -mkl`

测试文件如下：

```fortran
program test
    implicit none
    real*8 :: alp = 1.0, bet = 0.0
    integer:: Ns = 3
    integer:: i = 0
    real*8 :: hs(3,3) = &
    &reshape([ 250.0d0, 100.0d0,  10.0d0, &
              & 100.0d0, 300.0d0,  10.0d0, &
              &  10.0d0,  10.d00, 600.0d0  ], [3,3] , order
                =[2,1] )
    real*8 :: a1(3,3) = 0
    call DGEMM('N', 'N', Ns, Ns, Ns, alp, hs, Ns, hs, Ns,
       bet, a1, Ns)
    open( unit = 10 , file = "hs*hs_ver1.ods")
    do i = 1, 3
        print 100 , a1(i,1), a1(i,2), a1(i,3)
        write(10,100) a1(i,1) , a1(i,2) , a1(i,3)
    enddo

    100 format( 3(f10.1,',') )
end program test
```

Fortran 95 的函数接口简单，参量少，用起来方便，可是调用时屡屡出错：

```
In function ‘MAIN__’:
test_ver2.f90:(.text+0x164): undefined reference to ‘dgemm_mkl95_’
```

究其原因，应该是没有链接到合适的库。在 mkl_documentation.htm 旁边有个 mkl_link_line_advisor.htm 文件，这个文件很有用。它能指出合适的链接和编译选项，只是生成链接和编译选项时，需要手动选择系统相关参数。如果系统相关参数不清楚，怎么办呢？在 MKL 的安装目录下，例如我的安装目录`/opt/intel/mkl` 下有个 tools 目录，里面有个很有用的工具 mkl_link_tool。直接运行这个程序就能得到这个程序的相关说明。如果加上参数 -check_mkl_presence 运行，即 `./mkl_link_tool -check_mkl_presence` 就能到自己安装的详细参数。用这些参数在 mkl_link_line_advisor.htm 文件选择合适的选项，就能得到链接和编译选项，如我的：

```
Use this link line:
$(MKLROOT)/lib/intel64/libmkl_blas95_lp64
-L$(MKLROOT)/lib/intel64
-lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lpthread -lm

Compiler options:
 -I$(MKLROOT)/include/intel64/lp64 -I$(MKLROOT)/include
```

这里的链接和选项需要适当的调整，因为它不知道 MKL 的安装路径，这个要自己补充。要将$(MKLROOT) 替换成安装路径，我的安装路径为/opt/intel/mkl。还有就是库文件 libmkl_blas95_lp64 是有后缀的，加上 libmkl_blas95_lp64.a。这些库文件，自己可以在相应目录下看看。编译链接及编译选项都得加进去，于是最终有效的编译是：

```
ifort test1_ver2.f90
/opt/intel/mkl/lib/intel64/libmkl_blas95_lp64.a
-L/opt/intel/mkl/lib/intel64
-lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lpthread -lm
-I/opt/intel/mkl/include/intel64/lp64 -I/opt/intel/mkl/include
```

很长很麻烦，显然不是最终解决方案。

测试文件如下：

```
program test
    use mkl95_precision
    use mkl95_blas
    implicit none
    integer :: i = 0
    real*8 :: hs(3,3) = &
    &reshape([ 250.0d0, 100.0d0,  10.0d0, &
              & 100.0d0, 300.0d0,  10.0d0, &
              &  10.0d0,  10.d00, 600.0d0  ], [3,3] , order
                 =[2,1] )
    real*8 :: a1(3,33) = 0
    call GEMM(hs,hs,a1)
    open( unit = 10 , file = "hs*hs_ver2.ods")
    do i = 1, 3
        print 100 , a1(i,1), a1(i,2), a1(i,3)
        write(10,100) a1(i,1) , a1(i,2) , a1(i,3)
    enddo

    100 format( 3(f10.1,',') )
end program test
```

12.2 FFTW

12.2.1 FFTW 的简介

官方网站：http://www.fftw.org/

下载地址：http://www.fftw.org/download.html

快速傅立叶变换 (FFTW) 是由麻省理工学院开发的免费的软件包，采用 C 语言编写，计算一维和多维离散傅立叶变换。FFTW 的编码生成器采用面向对象设计技术和面向对象语言 Caml 编写；它能自动适应系统硬件，因而可移植性很强。FFTW2.1.5 支持共享存储多线程并行和分布式存储 MPI 并行。FFTW 的运算性能远远领先于目前已有的其它 FFT 软件。FFTW 为任意大小的模式生成一个计划（plan），通过对该计划施行各种运算完成各种模式的转换；内部结构及其复杂性对用户透明；速度快（适合各种机器的内部编译器、代码生成器利用 AST 在运行时生成代码并自我优化，而且不占用编译时间，采用分层存储技术）。FFTW 受到越来越多的科学研究和工程计算工作者的普遍青睐，并为量子物理、光谱分析、音视频流信号处理、石油勘探、地震预报、天气预报、概率论、编码理论、医学断层诊断等领域提供切实可行的大规模 FFT 计算。

12.2.2 安装调用

（1）安装

- 切换到解压目录
- 使用管理员权限或在命令前直接加 sudo
- ./configure
- make
- make check
- make install

（2）调用

```
ifort␣xx.f90␣-lfftw3␣-lm
```

12.2.3 实例

在 linux 下有如下程序：

```
program aab
    use fftw3
    implicit none
    integer i , j , n
    parameter (n=8)
    type(C_PTR) :: plan
    complex(C_DOUBLE_COMPLEX), dimension(n,n) :: in, out
    do i=1,n
        do j=1,n
            in(i,j)=cmplx(cos(i*0.22),sin(j*0.3))
```

```
        enddo
    enddo
    out=0.d0
    plan =fftw_plan_dft_2d(n,n,in,out,FFTW_FORWARD,
       FFTW_ESTIMATE)
    call fftw_execute_dft(plan, in, out)
    write(777,*) real(out)
    write(777,*) aimag(out)
    call fftw_destroy_plan(plan)
    stop
end
```

编译：gfotran -o x x.f90 -lfftw3

运行：./x

12.3 BLAS

12.3.1 BLAS 简介

官网：http://www.netlib.org/blas/

BLAS 系 Basic Linear Algebra Subprograms 的缩写。顾名思义，BLAS 实现基本的线性代数运算，是 NetLib 中一些高级软件包--比如，LAPACK 和 LINPACK– 的“基石”。据说，BLAS 由美国 NSF 和 DOE 资助，在 1970s 开始推出，至今仍在发展。这是一个 PublicDomain 的著名软件包。BLAS 的基本部分由三个级别构成：第一级完成“向量-向量”操作；第二级实现“矩阵-向量”运算；第三级包含“矩阵-矩阵”程序。所有的子程序都经过精心的优化以确保其高效性；程序的书写和语句的选择都相当规范，以便于移植。针对不同的平台，有多种经过低级别优化 (比如汇编级优化) 的 BLAS 库可供直接使用。长期以来，专家们一直呼吁使用 BLAS 来建立和开发自己的线性代数软件包。这样做，不仅可以缩短开发周期，而且便于形成统一的调用接口。当 BLAS 升级时，您可以坐享其成。由于源码是公开的，您不必担心受制于人 (如果您使用商品化的 IMSL 或者 NAG，这种担心就不是多余的)。

12.4 GLS

http://blog.csdn.net/waleking/article/details/8265008/

12.4.1 GLS 安装过程

下载 gsl-1.9.tar.gz http://ftp.club.cc.cmu.edu/pub/gnu/gsl/

安装

- tar -zxvf gsl-1.9.tar.gz
- cd gsl-1.9
- sudo ./configure
- sudo make

- sudo make install

执行最后一个命令之后会有很多现实问题，其中有：

```
Libraries have been installed in:
   /usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the '-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the 'LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the 'LD_RUN_PATH' environment variable
     during linking
   - use the '-Wl,--rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to '/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
```

一个是libgslcblas.so，libgsl.so 的位置很重要，编译后的文件被扔到了这里，比如/usr/local/lib；另一个重要的位置是 gsl_rng.h 这样的头文件所在的文件夹目录，比如/usr/local/include/gsl。如果想要查找他们，可以使用find / -name gsl_rng.h 这样的命令。

12.4.2 使用实例

```c
#include <stdio.h>
#include <gsl_rng.h>
#include <gsl_randist.h>


int main (int argc, char *argv[])
{
  /* set up GSL RNG */
  gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937);
  /* end of GSL setup */


  int i,n;
  double gauss,gamma;


  n=atoi(argv[1]);
  for (i=0;i<n;i++)
```

```
    {
      gauss=gsl_ran_gaussian(r,2.0);
      gamma=gsl_ran_gamma(r,2.0,3.0);
      printf("%2.4f %2.4f\n", gauss,gamma);
    }
  return(0);
}
```

编译:

第一反应是直接 gcc gsl_test.c

显示错误是: gsl_test.c:2: fatal error: gsl_rng.h: 没有那个文件或目录

这个错误是说没有找到头文件，然后加上头文件的位置，

gcc -I/usr/local/include/gsl gsl_test.c

显示错误是:gsl_test.c:(.text+0x12): undefined reference to ‘gsl_rng_mt19937’ ...

这个错误是说没有找到gsl_rng_mt19937 的定义，查看了http://www.daniweb.com/software-development/cpp/threads/289812/cant-link-gsl-properly,这里说需要加上-lgsl，即链接到 gsl，找到 libgsl.so.

使用命令: gcc -I/usr/local/include/gsl -lgsl gsl_test.c

显示错误是://usr/local/lib/libgsl.so: undefined reference to ‘cblas_ctrmv’

查看了http://sourceware.org/ml/gsl-discuss/2003-q2/msg00123.html, 是说错误是由于没有链接 libgslcblas.so 引起的，再加上-lgslcblas 可以解决问题。

使用命令: gcc -I/usr/local/include/gsl -lgsl -lgslcblas gsl_test.c。

或者命令:gcc -I/usr/local/include/gsl -L/usr/local/lib -lgsl -lgslcblas gsl_test.c

这个时候出现了产出物 a.out。

(5) 运行 a.out

./a.out 出现错误“段错误”

(6) 查看源文件，发现还需要输入参数

./a.out 10

结果是: 0.2678 6.9645 3.3488 1.6894 1.9950 2.1575 -4.7934 6.1648 -0.0782 4.0292 1.7871 11.6031 -2.5931 7.7629 0.3634 1.3344 -1.0965 11.1658 0.0142 3.5412

12.5 Lapack

Lapack，Linear Algebra PACKage，是由美国国家科学基金等资助开发的，以 Fortran 语言编写的，开源的线性代数软件包。包含了求解科学与工程计算中最常见的数值线性代数问题，如求解线性方程组、线性最小二乘问题、特征值问题和奇异值问题等。

官网为: http://www.netlib.org/lapack/，最新版可以在这里下载自行编译使用。

Lapack 编译比较简单，参考下载源码中的 README 文件就能编译成功。但是有几点需要注意:

(1) 最终生成的库文件。最终生成的库文件包含三个: liblapack.a、librefblas.a、libtmglib.a，虽然可能会报几个错，但是只要三个文件生成，那基本成功了。

(2) Makefile 文件。Makefile 文件通常不用修改。但是如下两行需要注意。如果你的系统没有安装 Blas（参考前面章节），那请注释掉`lib: lapacklib tmglib`这行，使用下面那行命令。反之注释第二行，使用第一行。

```
#lib: lapacklib tmglib
lib: blaslib variants lapacklib tmglib
```

(3) make.inc 文件。make.inc 文件后缀，inc 是 include 的缩写。顾名思义，是 Makefile 文件的组成部分。有的开发者喜欢把需要用户自行修改的部分放在这里，而 Makefile 无需用户修改。有的则喜欢把需要用户自行修改的部分放在 Makefile 文件，而 make.inc 文件无需修改。这里是前者。用户需要根据自己的编译环境修改 make.inc 文件。如果不太懂，没关系，开发者在 INSTALL 文件下提供了大量的配置模板。笔者用的是 ifort 编译器，直接选用了 make.inc.ifort。无需任何修改，直接编译成功。（当然注意把后缀改成.inc）

(4) 编译。编译的时候要注意把这三个库文件都链接上。如何链接库，请看相关章节。

12.5.1 使用实例

```fortran
program main
    implicit none
    INTEGER :: N, LDA, LDB
    INTEGER :: NRHS
    INTEGER :: INFO
    INTEGER :: IPIV(4)
    REAL(8) :: A(4,4), B(4,1)

    N=4;LDA=4;LDB=4
    NRHS=1

    A=reshape((/1.80,2.88,2.05,-0.89,&
                5.25,-2.95,-0.95,-3.80,&
                1.58,-2.69,-2.90,-1.04,&
                -1.11,-0.66,-0.59,0.80/),(/4,4/))
    B=reshape((/9.52,24.35,0.77,-6.22/),(/4,1/))

    call DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )

    write(*,*) "Solution:"
    write(*,'(f8.3)') B
    write(*,*) "INFO=", INFO

    stop
end program
```

编译命令：

```
ifort test.f90 -L/home/XX -llapack -lrefblas -ltmglib
gfortran mred_3pe.f90 -L/home/xuanleng/Major/Core/Scripts -llapack -lrefblas
```

注意写好链接库文件的目录。关于链接库文件的更多信息详看相关章节。

12.6 anyschroed

第四部分

Programming

第十三章 编程日志

2015.7.31

从今天开始，记录编程中值得记录的错误，提醒以后。

编程数值计算时，一定要注意初始化数据。当计算结果飘了，很可能是数据没有初始化导致数值累加。

最近做计算，输出结果一直为 0，折腾了好几天。也由于是新编程序，担心是算法问题，各种核查很多遍，虽然侧面加深了对理论的理解，但是最终问题却跟这无关。原来是最后一步的 fortran 矩阵运算，matmul(um,rho1)，习惯的写成了，um*rho1，导致输出为 0。用的是 Intel 的 fortran 编译器，这样的错误居然没报错，也是无语。知道 Intel 的 fortran 语法较宽松，可是这样太宽松了。所以啊，以后为了防止这类错误，还是用语法较为严格的 gfortran 也编译看看吧。

2015.8.1

今天用 Intel fortran 编译程序，弹出来说许可证过期了，不能用了。哎 ~ 我都忘记这碴了！习惯了不为软件付费，来个这个还真不习惯。算了，以后还是用 gfortran 吧。

经常遇到如下错误：

Program received signal SIGSEGV: Segmentation fault - invalid memory reference.

一般就是数组下标过界了，一定要注意组元的赋值。

2015.8.6

Intel fortran 编译器许可证过期是个教训。不同编译器，要求不同，Intel fortran 编译器要求松些，直接导致改用 gfortran 编译器出现很多警告和计算错误。所以以后编程一定要保证这两个编译器都通过和计算一致。

商业软件就是商业软件，Intel fortran 足足比开源免费的 gfortran 快个 6 倍。

第十四章 Python

14.1 准备工作

14.1.1 安装模块

首先安装 python 的模块管理器，对于 python2，有`sudo apt-get install python-pip`；对于 python3，有`sudo apt-get install python3-pip`。

安装模块，以 pylab 为例。对于 python2，有`sudo pip install pylab`；对于 python3，有`sudo pip3 install pylab`。

`sudo apt-get install python-matplotlib`

14.1.2 Ubuntu 切换 Python3

因为 Ubuntu 很多底层采用的是 Python2.*，Python3 和 Python2 是互相不兼容的，所以此时不能卸载 Python2。要使用 python3.*，可以直接输入命令 python3 即可。输入 python 默认是 python2.*。

14.1.3 Python 脚本编码设置

在文件开头加入 # -*- coding: UTF-8 -*- 或者 #coding=utf-8 可以设置 Python 脚本编码。

14.1.4 设置脚本解释器的版本

“#!” 是脚本中的约定符号，用来指定解释器的版本。

- 直接指定解释器路径，如：`#!/usr/bin/python`、`#!/usr/bin/python2.7`、或`#!/usr/bin/python3.4`
- 通过 env 命令来指定解释器，如：

  `#!/usr/bin/env python`

  由于在 Ubuntu 中 Shell 输入 python 默认调用的是 python2.*，所以这是指定解释器为 python2.* 类版本。又如

  `#!/usr/bin/env python3`

  由于 python 存在 python2.* 和 python3.* 两类版本而不兼容，所有这是指定解释器为 python3.* 类版本。

14.1.5 脚本的执行

对于脚本语言，执行方法都类似。类似于 Shell 脚本的执行一样，Python 脚本的执行也有两种方法。

（1）作为解释器参数，即直接运行解释器，其参数就是 Shell 脚本的文件名，例如

使用 Python2.* 解释：`python test.py` 或更严格的 `/usr/bin/python2.7 test.py`

使用 Python3.* 解释：`python3 test.py` 或更严格的 `/usr/bin/python3.4 test.py`

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

（2）作为可执行程序。

（a）在脚本所在目录下，首先使脚本具有执行权限，有两种方法：

1. 命令操作：`chmod +x ./test.py`；
2. 图形化操作：选中脚本 test.py 右键勾选可以执行选项。

（b）然后执行程序：`./test.py`

注意，一定要写成./test.py，而不是 test.py。这里的“./”实质指的是相对路径，即当前路径下。如果 test.py 在/bin 目录下，当然也可以采用绝大路径来运行即“/bin/test.py”，只不过相对路径更简便些。运行其它二进制的程序也一样，直接写 test.py，Linux 系统会去 PATH 里寻找有没有叫 test.py 的，而只有/bin、/sbin、/usr/bin、/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.py 是会找不到命令的，要用./test.py 告诉系统说，就在当前目录找。这意味着如果你编的程序经常用，那么可以添加到系统路径中，这样就真可以直接运行了。

通过这种方式运行的脚本，第一行一定要写对，好让系统查找到正确的解释器。

这里的" 系统"，其实就是 shell 这个应用程序，但我故意写成系统，是方便理解，既然这个系统就是指 shell，那么一个使用/bin/sh 作为解释器的脚本是不是可以省去第一行呢？是的。

14.2 Python2.* 与 Python3.* 的主要变化

Python 的 3.0 版本，常被称为 Python 3000，或简称 Py3k。相对于 Python 的早期版本，这是一个较大的升级。为了不带入过多的累赘，Python 3.0 在设计的时候没有考虑向下相容。许多针对早期 Python 版本设计的程式都无法在 Python 3.0 上正常执行。

为了照顾现有程式，Python 2.6 作为一个过渡版本，基本使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。

新的 Python 程序建议使用 Python 3.0 版本的语法。除非执行环境无法安装 Python 3.0 或者程式本身使用了不支援 Python 3.0 的第三方库。目前不支援 Python 3.0 的第三方库有 Twisted, py2exe, PIL 等。

大多数第三方库都正在努力地相容 Python 3.0 版本。即使无法立即使用 Python 3.0，也建议编写相容 Python 3.0 版本的程式，然后使用 Python 2.6，Python 2.7 来执行。

14.2.1 print 语句改成 print() 函数

print 语句没有了，取而代之的是 print() 函数。Python 2.6 与 Python 2.7 部分地支持这种形式的 print 语法。在 Python 2.6 与 Python 2.7 里面，以下三种形式是等价的：

```
print "fish"
print ("fish") #注意print后面有个空格
print("fish") #print()不能带有任何其它参数
```

14.2.2 TypeError: 'str' does not support the buffer interface

Python 3 中套接字编程中遇到 TypeError: 'str' does not support the buffer interface 的解决办法。

In python 3, bytes strings and unicodestrings are now two different types. Since sockets are not aware of string encodings, they are using raw bytes strings, that have a slightly differentinterface from unicode strings. So, now, whenever you have a unicode stringthat you need to use as a byte string, you need toencode() it. And whenyou have a byte string, you need to decode it to use it as a regular(python 2.x) string. Unicode strings are quotes enclosedstrings. Bytes strings are b"" enclosed strings

14.3 基本语法

14.3.1 注释

python 注释采用 # 开头，没有块注释。

14.3.2 标识符

- 标识符由字母、数字、下划线组成；
- 所有标识符可以包括英文、数字以及下划线（_），但不能以数字开头；
- 标识符区分大小写；
- 以下划线开头的标识符有特殊意义。以单下划线开头（_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用 "from xxx import *" 而导入；
- 以双下划线开头的（__foo）代表类的私有成员；以双下划线开头和结尾的（__foo__）代表 python 里特殊方法专用的标识，如 __init__() 代表类的构造函数。

14.3.3 保留字符

Python 中的保留字不能用作常数或变数，或任何其他标识符名称，所有 Python 的关键字只包含小写字母。Python 的保留字有：and、exec、not、assert、finally、or、break、for、pass、class、from、print、continue、global、raise、def、if、return、del、import、try、elif、in、while、else、is、with、except、lambda、yield

14.3.4 缩进

Python 与其他语言最大的区别就是，Python 的代码块不使用大括号（{}）来控制类，函数以及其他逻辑判断。Python 最具特色的就是用缩进来写模块。缩进的空白数量是可

变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行，一般缩进 4 个空格。如下所示：

```
if True:
    print "True"
else:
    print "False"
```

以下代码将会执行错误：

```
     if True:
    print "Answer"
    print "True"
else:
    print "Answer"
  print "False"
```

14.3.5 多行语句

Python 语句中一般以新行作为为语句的结束符。

但是我们可以使用斜杠（\）将一行的语句分为多行显示，如下所示：

```
 total = item_one + \
         item_two + \
         item_three
```

语句中包含 [], {} 或 () 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

14.3.6 引号

Python 接收单引号 (')，双引号 (")，三引号 (''' """) 来表示字符串，引号的开始与结束必须的相同类型的。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用语文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

14.3.7 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是 Python 语法的一部分。书写时不插入空行，Python 解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

14.3.8 同一行显示多条语句

Python 可以在同一行中使用多条语句，语句之间使用分号 (;) 分割，以下是一个简单的实例：

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

14.3.9 多个语句构成代码组

缩进相同的一组语句构成一个代码块，我们称之代码组。像 if、while、def 和 class 这样的复合语句，首行以关键字开始，以冒号 (:) 结束，该行之后的一行或多行代码构成代码组。我们将首行及后面的代码组称为一个子句 (clause)。如下实例：

```
if expression :
    suite
elif expression :
    suite
else :
    suite
```

14.4 变量赋值

变量存储在内存中的值。这就意味着在创建变量时会在内存中开辟一个空间。基于变量的数据类型，解释器会分配指定内存，并决定什么数据可以被存储在内存中。因此，变量可以指定不同的数据类型，这些变量可以存储整数，小数或字符。

Python 中的变量不需要声明，变量的赋值操作既是变量声明和定义的过程。每个变量在内存中创建，都包括变量的标识，名称和数据这些信息。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。等号（=）用来给变量赋值，等号（=）运算符左边是一个变量名，等号（=）运算符右边是存储在变量中的值。例如：

```
#coding=utf-8
#!/usr/bin/python
```

```
3
4 counter = 100 # 赋值整型变量
5 miles = 1000.0 # 浮点型
6 name = "John" # 字符串
7
8 print counter
9 print miles
10 print name
```

以上实例中，100，1000.0 和"John" 分别赋值给 counter，miles，name 变量。执行以上程序会输出如下结果：

```
1 100
2 1000.0
3 John
```

Python 也允许同时为多个变量赋值。例如：

```
1 a = b = c = 1
```

以上实例，创建一个整型对象，值为 1，三个变量被分配到相同的内存空间上。

可以为多个对象指定多个变量。例如：

```
1 a, b, c = 1, 2, "john"
```

以上实例，两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象"john" 分配给变量 c。

14.5 标准数据类型

Python 有五个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Dictionary（字典）

14.5.1 数字

数字数据类型用于存储数值。Python 支持四种不同的数值类型：

- int（有符号整型）
- long（长整型 [也可以代表八进制和十六进制]）
- float（浮点型）
- complex（复数）

具体使用注意事项：

- Python 使用"L" 来显示长整型，如：535633629843L。长整型也可以使用小写"L"，但是还是建议您使用大写"L"，避免与数字"1" 混淆；
- 浮点型可以用科学表示，如：2.1e+3 表示 $2.1\times10^3 = 2100.0$、2.1e-3 表示 $2.1\times10^{-3} = 0.0021$，也可以用大写的 E；
- 复数由实数部分和虚数部分构成，可以用 a + bj 或者 complex(a,b) 表示，复数的实部 a 和虚部 b 都是浮点型，也可以用大写的 J。

14.5.2 字符串

字符串或串 (String) 是由数字、字母、下划线组成的一串字符。一般记为:

```
s="a1a2···an"(n>=0)
```

它是编程语言中表示文本的数据类型。

python 的字串列表有 2 种取值顺序：

- 从左到右索引默认 0 开始的，最大范围是字符串长度少 1
- 从右到左索引默认-1 开始的，最大范围是字符串开头

如果你的实要取得一段子串的话，可以用到变量 [头下标: 尾下标]，就可以截取相应的字符串，其中下标是从 0 开始算起，可以是正数或负数，下标可以为空表示取到头或尾。比如:

```
s = 'ilovepython'
```

s[1:5] 的结果是 love。当使用以冒号分隔的字符串，python 返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。上面的结果包含了 s[1] 的值 l，而取到的最大范围不包括上边界，就是 s[5] 的值 p。

加号（+）是字符串连接运算符，星号（*）是重复操作。如下实例：

```
#coding=utf-8
#!/usr/bin/python

str = 'Hello World!'

print str # 输出完整字符串
print str[0] # 输出字符串中的第一个字符
print str[2:5] # 输出字符串中第三个至第五个之间的字符串
print str[2:] # 输出从第三个字符开始的字符串
print str * 2 # 输出字符串两次
print str + "TEST" # 输出连接的字符串
```

以上实例输出结果：

```
Hello World!
```

```
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

14.5.2.1 转义字符

在需要在字符中使用特殊字符时，python 用反斜杠 (\) 转义字符。如下表：

转义字符	描述
\ (在行尾时)	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格 (Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy 代表的字符，例如：\o12 代表换行
\xyy	十六进制数，yy 代表的字符，例如：\x0a 代表换行
\other	其它的字符以普通格式输出

14.5.2.2 字符串运算符

下表实例变量 a 值为字符串"Hello"，b 变量值为"Python"：

- 操作符; 描述; 实例
- + ; 字符串连接; a + b 输出结果： HelloPython
- *; 重复输出字符串; a*2 输出结果： HelloHello
- [] ; 通过索引获取字符串中字符; a[1] 输出结果 e
- [:]; 截取字符串中的一部分 a[1:4] 输出结果 ell
- in; 成员运算符 - 如果字符串中包含给定的字符返回 True; H in a 输出结果 1
- not in; 成员运算符 - 如果字符串中不包含给定的字符返回 True; M not in a 输出结果 1
- r/R; 原始字符串 - 原始字符串： 所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。原始字符串除在字符串的第一个引号前加上字母"r"

(可以大小写)以外,与普通字符串有着几乎完全相同的语法。;`print r'\n' prints \n` 和 `print R'\n' prints \n`

14.5.2.3 三引号——字符串的抄录

python 中三引号可以将复杂的字符串进行抄录，即 python 三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

```
>>> hi = '''hi
... there'''
>>> hi
'hi \nthere'
>>> print(hi)
hi
there
>>>
```

三引号让程序员从引号和特殊字符串的泥潭里面解脱出来，自始至终保持一小块字符串的格式是所谓的 WYSIWYG（所见即所得）格式的。

14.5.2.4 Unicode 字符串

Python 中定义一个 Unicode 字符串和定义一个普通字符串一样简单：

```
>>> u'Hello World !'
u'Hello World !'
```

引号前小写的"u" 表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 Unicode-Escape 编码。如下例所示：

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

被替换的 ŏ020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符（空格符）。

14.5.2.5 字符串内建函数

以下实例展示了 capitalize() 方法的实例：

```
#!/usr/bin/python

str = "this is string example....wow!!!";

print "str.capitalize() : ", str.capitalize()
```

输出结果如下：

```
str.capitalize() :  This is string example....wow!!!
```

14.5.3 列表

List（列表）是 Python 中使用最频繁的数据类型。列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。列表用 [ ] 标识。是 python 最通用的复合数据类型。列表中的值得分割也可以用到变量 [头下标: 尾下标]，就可以截取相应的列表，从左到右索引默认 0 开始的，从右到左索引默认-1 开始，下标可以为空表示取到头或尾。加号（+）是列表连接运算符，星号（*）是重复操作。如下实例：

```
#coding=utf-8
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

14.5.3.1 更新列表

可以对列表的数据项进行修改或更新，也可以使用 append() 方法来添加列表项，如下所示：

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "
```

```
print list[2];
list[2] = 2001;
print "New value available at index 2 : "
print list[2];
```

输出结果:

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

14.5.3.2 删除列表元素

可以使用 del 语句来删除列表的的元素，如下实例:

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

print list1;
del list1[2];
print "After deleting value at index 2 : "
print list1;
```

输出结果:

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

14.5.3.3 列表操作符

列表对 + 和 * 的操作符与字符串相似。+ 号用于组合列表，* 号用于重复列表。如下所示:

表达式	结果	描述
`len([1, 2, 3])`	3	长度
`[1, 2, 3] + [4, 5, 6]`	`[1, 2, 3, 4, 5, 6]`	组合
`['Hi!'] * 4`	`['Hi!', 'Hi!', 'Hi!', 'Hi!']`	重复
`3 in [1, 2, 3]`	True	元素是否存在于列表中
`for x in [1, 2, 3]: print x`	1 2 3	迭代

14.5.3.4 列表截取

Python 的列表截取与字符串操作类型，如下所示:

```
L = ['spam', 'Spam', 'SPAM!']
```

操作：

表达式	结果	描述
L[2]	'SPAM!'	读取列表中第三个元素
L[-2]	'Spam'	读取列表中倒数第二个元素
L[1:]	['Spam', 'SPAM!']	从第二个元素开始截取列表

14.5.3.5 列表内置函数

- cmp(list1, list2)：比较两个列表的元素
- len(list)：列表元素个数
- max(list)：返回列表元素最大值
- min(list)：返回列表元素最小值
- list(seq)：将元组转换为列表

14.5.3.6 列表方法

- list.append(obj)：在列表末尾添加新的对象
- list.count(obj)：统计某个元素在列表中出现的次数
- list.extend(seq)：在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
- list.index(obj)：从列表中找出某个值第一个匹配项的索引位置
- list.insert(index, obj)：将对象插入列表
- list.pop(obj=list[-1])：移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
- list.remove(obj)：移除列表中某个值的第一个匹配项
- list.reverse()：反向列表中元素
- list.sort([func])：对原列表进行排序

14.5.4 元组

Tuple（元组）是另一个数据类型，类似于 List（列表）。元组用"()" 标识。内部元素用逗号隔开。但是元素不能二次赋值，相当于只读列表。

```
#coding=utf-8
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple # 输出完整元组
print tuple[0] # 输出元组的第一个元素
```

```
print tuple[1:3] # 输出第二个至第三个的元素
print tuple[2:] # 输出从第三个开始至列表末尾的所有元素
print tinytuple * 2 # 输出元组两次
print tuple + tinytuple # 打印组合的元组
```

以上实例输出结果:

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

以下是元组无效的，因为元组是不允许更新的。而列表是允许更新的:

```
#coding=utf-8
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
#tuple[2] = 1000 # 元组中是非法应用
list[2] = 1000 # 列表中是合法应用
```

14.5.4.1 连接元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例:

```
#coding=utf-8
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# 以下修改元组元素操作是非法的。
# tup1[0] = 100;

# 创建一个新的元组
tup3 = tup1 + tup2;
print tup3;
```

输出结果:

```
(12, 34.56, 'abc', 'xyz')
```

14.5.4.2 删除元组

元组中的元素值是不允许删除的，但我们可以使用 del 语句来删除整个元组，如下实例:

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;
del tup;
print "After deleting tup : "
print tup;
```

以上实例元组被删除后，输出变量会有异常信息，输出如下所示:

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

14.5.4.3 元组运算符

与字符串一样，元组之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

表达式	结果	描述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	复制
3 in (1, 2, 3)	True	元素是否存在
for x in (1, 2, 3): print x	1 2 3	迭代

14.5.4.4 元组索引、截取

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素，如下所示:

元组: L = ('spam', 'Spam', 'SPAM!')

表达式	结果	描述
L[2]	'SPAM!'	读取第三个元素
L[-2]	'Spam'	反向读取；读取倒数第二个元素
L[1:]	('Spam', 'SPAM!')	截取元素

14.5.4.5 无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

以上实例允许结果：

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

14.5.4.6 元组内置函数

Python 元组包含了以下内置函数：

- cmp(tuple1, tuple2)：比较两个元组元素。
- len(tuple)：计算元组元素个数。
- max(tuple)：返回元组中元素最大值。
- min(tuple)：返回元组中元素最小值。
- tuple(seq)：将列表转换为元组。

14.5.5 字典

字典 (dictionary) 是除列表以外 python 之中最灵活的内置数据结构类型。列表是有序的对象结合，字典是无序的对象集合。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。字典用"{ }" 标识。字典由索引 (key) 和它对应的值 value 组成。如下示例：

```
#coding=utf-8
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one'] # 输出键为'one' 的值
print dict[2] # 输出键为 2 的值
print tinydict # 输出完整的字典
print tinydict.keys() # 输出所有键
print tinydict.values() # 输出所有值
```

输出结果为:

```
This is one
This is two
{'code': 6734, 'name': 'john', 'dept': 'sales'}
dict_keys(['code', 'name', 'dept'])
dict_values([6734, 'john', 'sales'])
```

14.5.5.1 字典键的特性

字典值可以没有限制地取任何 python 对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住:

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];
```

输出结果:

```
dict['Name']:  Manni
```

2）键必须不可变，所以可以用数，字符串或元组充当，所以用列表就不行，如下实例:

```
#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];
```

输出结果:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

14.5.5.2 修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry


print "dict['Age']: ", dict['Age'];
print "dict['School']: ", dict['School'];
```

输出结果：

```
dict['Age']:  8
dict['School']:  DPS School
```

14.5.5.3 删除字典

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用 del 命令，如下实例：

```
#coding=utf-8
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典

print "dict['Age']: ", dict['Age'];
print "dict['School']: ", dict['School'];
```

但这会引发一个异常，因为用 del 后字典不再存在：

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

14.5.5.4 字典内置函数 & 方法

Python 字典包含了以下内置函数：

- cmp(dict1, dict2) 比较两个字典元素。
- len(dict) 计算字典元素个数，即键的总数。
- str(dict) 输出字典可打印的字符串表示。

- type(variable) 返回输入的变量类型，如果变量是字典就返回字典类型。

Python 字典包含了以下内置方法：

- radiansdict.clear() 删除字典内所有元素
- radiansdict.copy() 返回一个字典的浅复制
- radiansdict.fromkeys() 创建一个新字典，以序列 seq 中元素做字典的键，val 为字典所有键对应的初始值
- radiansdict.get(key, default=None) 返回指定键的值，如果值不在字典中返回 default 值
- radiansdict.has_key(key) 如果键在字典 dict 里返回 true，否则返回 false
- radiansdict.items() 以列表返回可遍历的 (键, 值) 元组数组
- radiansdict.keys() 以列表返回一个字典所有的键
- radiansdict.setdefault(key, default=None) 和 get() 类似, 但如果键不已经存在于字典中，将会添加键并将值设为 default
- radiansdict.update(dict2) 把字典 dict2 的键/值对更新到 dict 里
- radiansdict.values() 以列表返回字典中的所有值

14.5.6 数据类型转换

数据类型的转换只需要将数据类型作为函数名即可。以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

- int(x [,base])：将 x 转换为一个整数
- long(x [,base])：将 x 转换为一个长整数
- float(x)：将 x 转换到一个浮点数
- complex(real [,imag])：创建一个复数
- str(x)：将对象 x 转换为字符串
- repr(x)：将对象 x 转换为表达式字符串
- eval(str)：用来计算在字符串中的有效 Python 表达式, 并返回一个对象
- tuple(s)：将序列 s 转换为一个元组
- list(s)：将序列 s 转换为一个列表
- set(s)：转换为可变集合
- dict(d)：创建一个字典。d 必须是一个序列 (key,value) 元组。
- frozenset(s)：转换为不可变集合
- chr(x)：将一个整数转换为一个字符
- unichr(x)：将一个整数转换为 Unicode 字符
- ord(x)：将一个字符转换为它的整数值
- hex(x)：将一个整数转换为一个十六进制字符串
- oct(x)：将一个整数转换为一个八进制字符串

14.5.6.1 不同类型的数字

对于初学者，各种不同的数值类型令人迷惑。请看下面 4 个不同的值：5、5.0、'5' 和'5.0'。虽然它们看起来相似，但内部表示截然不同。

5 是一个整数，可直接用于算术运算。5.0 是一个浮点数，也可用于算术运算，但包含小数部分。

'5' 和'5.0' 都是字符串，分别包含 1 个和 3 个字符。字符串可显示到屏幕上或用于基于字符的操作（如删除空白或计算字符数）。字符串不能用于算术运算。当然，字符串可用于拼接，虽然结果可能让人觉得有点不合情理。例如：

```
>>> 3 * '5'
'555'
>>> 3 * '5.0'
'5.05.05.0'
```

14.6 输入与输出

14.6.1 读取键盘输入

Python 提供了两个内置函数从标准输入读入一行文本，默认的标准输入是键盘。如下：

- `raw_input()`
- `input()`

14.6.1.1 raw_input 函数

`raw_input([prompt])` 函数从标准输入读取一个行，并返回一个字符串，按「Enter」完成输入：

```
#!/usr/bin/python

str = raw_input("Enter your input: ");
print "Received input is : ", str
```

这将提示你输入任意字符串，然后在屏幕上显示相同的字符串。当我输入"Hello Python!"，它的输出如下：

```
Enter your input: Hello Python!
Received input is :  Hello Python!
```

14.6.1.2 input() 函数

input([prompt]) 函数和 raw_input([prompt]) 函数基本可以互换，但是 input 会假设输入是一个有效的 Python 表达式，并返回运算结果。例如：

```
#!/usr/bin/python

str = input("Enter your input: ");
print "Received input is : ", str
```

这会产生如下的对应着输入的结果：

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

14.6.1.3 返回值的算术运算

由于函数 input() 只是返回字符串，因此如果需要的是数字（如用于算术运算），就必须使用 Python 的数值转换函数。例如，请看下面的程序：

```
age = input('How old are you today? ')
age10 = int(age) + 10
print('In 10 years you will be ' +  str(age10) + ' years
    old.')
```

假设运行该程序时用户输入 22，变量 age 将指向字符串'22'，因为 Python 不会自动将看起来像数字的字符串转换为整数或浮点数。如果要将字符串用于算术运算，必须先将其转换为数字。为此，可使用函数 int(s)（如果需要的是整数）或 float(s)（如果需要的是浮点数）。

这里要指出的最后一个技巧是，在 print 语句中，必须将变量 age10（它指向一个整数）转换为字符串，这样才能打印它。如果忘记这样做，Python 将显示错误消息，指出不能将数字与字符串相加。

14.6.2 输出

在 Python2.* 中，输出是 print 语句；而在 Python3.* 中输出是 print() 函数。这在 Python2.* 和 Python3.* 区别章节中有相关说明。这里按照 Python3.* 的标准来说明。

用 print() 在括号中加上字符串，就可以向屏幕上输出指定的文字。比如输出'hello, world'，用代码实现如下：

```
print('hello, world')
```

print() 函数也可以接受多个字符串，用逗号“,”隔开，就可以连成一串输出：

```
print('The quick brown fox', 'jumps over', 'the lazy dog'
    )
```

print() 会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：

```
The quick brown fox jumps over the lazy dog
```

print() 也可以打印整数，或者计算结果：

```
>>> print(300)
300
>>> print(100 + 200)
300
```

考虑到 print() 会依次打印每个字符串，我们可以把计算 100 + 200 的结果打印得更漂亮一点：

```
>>> print('100 + 200 =', 100 + 200)
100 + 200 = 300
```

其中，对于 100 + 200，Python 解释器自动计算出结果 300。但是，’100 + 200 =’ 是字符串而非数学公式，Python 把它视为字符串，直接输出。

14.6.3 格式化输出

Python 中的格式化输出是以字符串的形式进行格式化输出。尽管格式化输出可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符%s 的字符串中。在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。如下实例：

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

以上实例输出结果：

```
My name is Zara and weight is 21 kg!
```

python 字符串格式化符号：

符号	描述
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f 和%e 的简写
%G	%f 和%E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令：

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号 (+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零 ('0')，在十六进制前面显示'0x' 或者'0X'(取决于用的是'x' 还是'X'
0	显示的数字前面填充'0' 而不是默认的空格
%	'%%' 输出一个单一的'%'
(var)	映射变量 (字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数 (如果可用的话)

14.6.4 格式化输出举例

使用 print 输出各型的

- 字符串
- 整数
- 浮点数
- 出度及精度控制

例如：

```
h=int(5.32)
print '-'%h
print 'd'%h
```

输出' 5'，5 前面有空格保持两位数；输出'05' ，5 前面有 0 保持两位数。

```
strHello = 'Hello Python'
print strHello
```

输出：Hello Python，直接出字符串。

14.6.4.1 格式化输出整数

python print 也支持参数格式化，与 C 言的 printf 似，

```
strHello = "the length of (%s) is %d" %('Hello World',len('Hello World'))
print strHello
```

输出：the length of (Hello World) is 11

14.6.4.2 格式化输出 16 制整数

nHex = 0x20

- %x — hex 十六进制
- %d — dec 十进制
- %o — oct 八进制

```
print "nHex = %x,nDec = %d,nOct = %o" %(nHex,nHex,nHex)
```

输出结果：nHex = 20,nDec = 32,nOct = 40，使用整数的各个制打印同一个数。

14.6.4.3 格式化输出浮点数

```
import math
#default
print "PI = %f" % math.pi

#width = 10,precise = 3,align = left
print "PI = .3f" % math.pi

#width = 10,precise = 3,align = rigth
print "PI = %-10.3f" % math.pi

#前面填充字符
print "PI = d" % int(math.pi)
```

输出结果

```
PI = 3.141593
PI =       3.142
PI = 3.142
```

```
PI = 000003
```

14.6.4.4 格式化输出字符串

```
#precise = 3
print "%.3s " % ("jcodeer")
#precise = 4
print "%.*s" % (4,"jcodeer")
#width = 10,precise = 3
print ".3s" % ("jcodeer")
```

输出结果:

```
jco
jcod
        jco
```

14.6.5 python 读入 csv 的三种方式

读数据到 python 有好几种方法，我们以读取 iris.csv 为例，将其中的数值部分提取出来。

第一种方法是列表理解，文件读取到 lines 之后用一个嵌套的列表理解就可以将数值存为一个 list。

第二种方法是使用 numpy 库，它内带的 loadtxt 函数，读取的数据都认作是字符串，所以在第二行取我们需要的部分，并转为数值 array。

第三种方法是使用 pandas 库，它内带 read_csv 函数，读取数据会自动判断数值还是字符串，而且会自动保存好变量名，只需要用 ix 方法就可以类似 R 一样取出需要的子集，它存为 dataframe 对象。

这三种方法中最后一种最简单，不过花费时间比较长一点，第一种最麻烦，不过用时最短。这个可以通过 ipython 中的 magic 函数 timeit 来看。

14.7 文件及目录操作

- file 对象方法: file 对象提供了操作文件的一系列方法。
- os 对象方法: 提供了处理文件及目录的一系列方法。

14.7.1 打开和关闭文件

Python 提供了必要的函数和方法进行默认情况下的文件基本操作，可以用 file 对象做大部分的文件操作。

14.7.1.1 open 函数

先用 Python 内置的 open() 函数打开一个文件，创建一个 file 对象，相关的辅助方法才可以调用它进行读写。语法如下：

```
file object = open(file_name [, access_mode][, buffering])
```

各个参数的细节：

- file_name：file_name 变量是一个包含了你要访问的文件名称的字符串值。
- access_mode：access_mode 决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读 (r)。
- buffering：如果 buffering 的值被设为 0，就不会有寄存。如果 buffering 的值取 1，访问文件时会寄存行。如果将 buffering 的值设为大于 1 的整数，表明了这就是的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

不同模式打开文件的完全列表：

- 模式：描述
- r： 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
- rb： 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
- r+： 打开一个文件用于读写。文件指针将会放在文件的开头。
- rb+： 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
- w：打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
- wb： 以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
- w+： 打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
- wb+： 以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
- a：打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
- ab： 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
- a+： 打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
- ab+： 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

14.7.1.2 file 对象的属性

一个文件被打开后，就有一个 file 对象，可以得到有关该文件的各种信息。以下是和 file 对象相关的所有属性的列表：

- 属性：描述
- file.closed： 返回 true 如果文件已被关闭，否则返回 false。
- file.mode： 返回被打开文件的访问模式。
- file.name： 返回文件的名称。
- file.softspace： 如果用 print 输出后，必须跟一个空格符，则返回 false。否则返回 true。

如下实例：

```
#coding=utf-8
#!/usr/bin/python

# 打开一个文件
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

以上实例输出结果：

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

14.7.1.3 close() 方法

file 对象的 close() 方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。当一个文件对象的引用被重新指定给另一个文件时，Python 会关闭之前的文件。用 close() 方法关闭文件是一个很好的习惯。

语法：

```
 fileobject.close();
```

例子：

```
#coding=utf-8
#!/usr/bin/python

# 打开一个文件
```

```
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# 关闭打开的文件
fo.close()
```

以上实例输出结果:

```
Name of the file:  foo.txt
```

14.7.2 读取和写入文件

14.7.2.1 write() 方法

write() 方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python 字符串可以是二进制数据，而不是仅仅是文字。write() 方法不在字符串的结尾不添加换行符 ('\n')。

语法:

```
fileobject.write(string);
```

在这里，被传递的参数是要写入到已打开文件的内容。例如:

```
#coding=utf-8
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n
    ");

# 关闭打开的文件
fo.close()
```

上述方法会创建 foo.txt 文件，并将收到的内容写入该文件，并最终关闭文件。如果你打开这个文件，将看到以下内容:

```
Python is a great language.
Yeah its great!!
```

14.7.2.2 read() 方法

read() 方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python 字符串可以是二进制数据，而不是仅仅是文字。

语法:

```
fileobject.read([count]);
```

在这里，被传递的参数是要从已打开文件中读取的字节计数。该方法从文件的开头开始读入，如果没有传入 count，它会尝试尽可能多地读取更多的内容，很可能是直到文件的末尾。

例如：

```
#coding=utf-8
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# 关闭打开的文件
fo.close()
```

输出结果：

```
Read String is :  Python is
```

14.7.2.3 文件位置

tell() 方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后。

seek(offset [,from]) 方法改变当前文件的位置。offset 变量表示要移动的字节数。from 变量指定开始移动字节的参考位置。如果 from 被设为 0，这意味着将文件的开头作为移动字节的参考位置。如果设为 1，则使用当前的位置作为参考位置。如果它被设为 2，那么该文件的末尾将作为参考位置。

例如：

```
#coding=utf-8
#!/usr/bin/python

# 打开一个文件
fo = open("/tmp/foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str

# 查找当前位置
position = fo.tell();
print "Current file position : ", position

# 把指针再次重新定位到文件开头
position = fo.seek(0, 0);
str = fo.read(10);
```

```
print "Again read String is : ", str
# 关闭打开的文件
fo.close()
```

实例输出结果：

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

14.7.3 重命名和删除文件

Python 的 os 模块提供了帮你执行文件处理操作的方法，比如重命名和删除文件。要使用这个模块，你必须先导入它，然后可以调用相关的各种功能。

14.7.3.1 rename() 方法

rename() 方法需要两个参数，当前的文件名和新文件名。

语法：

```
os.rename(current_file_name, new_file_name)
```

例如：下例将重命名一个已经存在的文件 test1.txt。

```
#coding=utf-8
#!/usr/bin/python
import os

# 重命名文件test1.txt到test2.txt。
os.rename( "test1.txt", "test2.txt" )
```

14.7.3.2 remove() 方法

用 remove() 方法删除文件，需要提供要删除的文件名作为参数。

语法：

```
os.remove(file_name)
```

例如：删除一个已经存在的文件 test2.txt。

```
#coding=utf-8
#!/usr/bin/python
import os

```

```
# 删除一个已经存在的文件test2.txt
os.remove("text2.txt")
```

14.7.4 创建和删除目录

所有文件都包含在各个不同的目录下，Python 也能轻松处理。os 模块有许多方法能创建，删除和更改目录。

14.7.4.1 mkdir() 方法

可以使用 os 模块的 mkdir() 方法在当前目录下创建新的目录们。你需要提供一个包含了要创建的目录名称的参数。

语法：

```
os.mkdir("newdir")
```

例如：在当前目录下创建一个新目录 test

```
#coding=utf-8
#!/usr/bin/python
import os

# 创建目录test
os.mkdir("test")
```

14.7.4.2 chdir() 方法

可以用 chdir() 方法来改变当前的目录。chdir() 方法需要的一个参数是你想设成当前目录的目录名称。

语法：

```
os.chdir("newdir")
```

例如：

```
#coding=utf-8
#!/usr/bin/python
import os

# 将当前目录改为"/home/newdir"
os.chdir("/home/newdir")
```

14.7.4.3 getcwd() 方法

getcwd() 方法显示当前的工作目录。

语法：

```
os.getcwd()
```

例如：

```
#coding=utf-8
#!/usr/bin/python
import os

# 给出当前的目录
os.getcwd()
```

14.7.4.4 rmdir() 方法

rmdir() 方法删除目录，目录名称以参数传递。需要注意的是，在删除这个目录之前，它的所有内容应该先被清除。

语法：

```
os.rmdir('dirname')
```

例如：

```
#coding=utf-8
#!/usr/bin/python
import os

# 删除”/tmp/test” 目录
os.rmdir( "/tmp/test"  )
```

14.8 运算符

Python 语言支持以下类型的运算符：

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符
- 位运算符
- 成员运算符

- 身份运算符

14.8.1 算术运算符

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -10
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除 - x 除以 y	b / a 输出结果 2
%	取模 - 返回除法的余数	b% a 输出结果 0
**	幂 - 返回 x 的 y 次幂	a**b 为 10 的 20 次方，输出结果
//	取整除 - 返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果

14.8.2 比较运算符

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述
==	等于 - 比较对象是否相等
!=	不等于 - 比较两个对象是否不相等
<>	不等于 - 比较两个对象是否不相等
>	大于 - 返回 x 是否大于 y
<	小于 - 返回 x 是否小于 y。所有比较运算符返回 1 表示真，返回 0 表示假。这分别与特殊
>=	大于等于 - 返回 x 是否大于等于 y
<=	小于等于 - 返回 x 是否小于等于 y

14.8.3 赋值运算符

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

14.8.4 位运算符

按位运算符是把数字看作二进制来进行计算的。Python 中的按位运算法则如下：以下假设变量 a 为 60，变量 b 为 13：

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ，二进制解释：0000 1100
|	按位或运算符	(a | b) 输出结果 61 ，二进制解释：0011 1101
~	按位取反运算符	(~a) 输出结果 -61 ，二进制解释：1100 0011，在一个有符号二进制数的补码
«	左移动运算符	a « 2 输出结果 240 ，二进制解释：1111 0000
»	右移动运算符	a » 2 输出结果 15 ，二进制解释：0000 1111

14.8.5 逻辑运算符

Python 语言支持逻辑运算符，以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
and	布尔" 与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。	(a and b) 返回 true。
or	布尔" 或" - 如果 x 是 True，它返回 True，否则它返回 y 的计算值。	(a or b) 返回 true。
not	布尔" 非" - 如果 x 为 True，返回 False。如果 x 为 False，它返回 True。	not(a and b) 返回 false

14.8.6 成员运算符

Python 还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	x 在 y 序列中, 如果 x 在 y 序列中
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	x 不在 y 序列中, 如果 x 不在 y 序

以下实例演示了 Python 所有成员运算符的操作：

```
#!/usr/bin/python

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print "Line 1 - a is available in the given list"
else:
   print "Line 1 - a is not available in the given list"

if ( b not in list ):
   print "Line 2 - b is not available in the given list"
else:
   print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):
   print "Line 3 - a is available in the given list"
```

```
else:
   print "Line 3 - a is not available in the given list"
```

以上实例输出结果：

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

14.8.7 身份运算符

身份运算符用于比较两个对象的存储单元。

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y, 如果 id(x) 等于 id(y) , is 返回结
is no	t is not 是判断两个标识符是不是引用自不同对象	x is not y, 如果 id(x) 不等于 id(y). is no

以下实例演示了 Python 所有身份运算符的操作：

```
#!/usr/bin/python

a = 20
b = 20

if ( a is b ):
   print "Line 1 - a and b have same identity"
else:
   print "Line 1 - a and b do not have same identity"

if ( id(a) == id(b) ):
   print "Line 2 - a and b have same identity"
else:
   print "Line 2 - a and b do not have same identity"

b = 30
if ( a is b ):
   print "Line 3 - a and b have same identity"
else:
   print "Line 3 - a and b do not have same identity"

if ( a is not b ):
   print "Line 4 - a and b do not have same identity"
else:
   print "Line 4 - a and b have same identity"
```

以上实例输出结果：

```
Line 1 - a and b have same identity
Line 2 - a and b have same identity
Line 3 - a and b do not have same identity
Line 4 - a and b do not have same identity
```

14.8.8 运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
+ -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
» «	右移，左移运算符
&	位'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

14.9 条件语句

条件语句是通过一条或多条语句的执行结果（True 或者 False）来决定执行的代码块。Python 程序语言指定任何非 0 和非空（null）值为 true，0 或者 null 为 false。编程中 if 语句用于控制程序的执行，基本形式为：

```
if 判断条件:
    执行语句……
else:
    执行语句……
```

其中" 判断条件" 成立时（非零），则执行后面的语句，而执行内容可以多行，以缩进来区分表示同一范围。else 为可选语句，当需要在条件不成立时执行内容则可以执行相关语句，具体例子如下：

```
# coding=utf8
# 例1: if 基本用法

flag = False
name = 'luren'
if name == 'python':          # 判断变量否为'python'
    flag = True           # 条件成立时设置标志为真
    print 'welcome boss'     # 并输出欢迎信息
else:
    print name               # 条件不成立时输出变量名称
```

输出结果为：

```
>>> luren              # 输出结果
```

if 语句的判断条件可以用 > （大于）、<(小于)、== （等于）、>= （大于等于）、<= （小于等于）来表示其关系。当判断条件为多个值是，可以使用以下形式：

```
if 判断条件1:
    执行语句1……
elif 判断条件2:
    执行语句2……
elif 判断条件3:
    执行语句3……
else:
    执行语句4……
```

实例如下：

```
# coding=utf8
# 例2: elif用法

num = 5
if num == 3:              # 判断num的值
    print 'boss'
elif num == 2:
    print 'user'
elif num == 1:
    print 'worker'
elif num < 0:             # 值小于零时输出
    print 'error'
else:
    print 'roadman'       # 条件均不成立时输出
```

输出结果为：

```
>>> roadman       # 输出结果
```

由于 python 并不支持 switch 语句，所以多个条件判断，只能用 elif 来实现，如果判断需要多个条件需同时判断时，可以使用 or （或），表示两个条件有一个成立时判断条件成功；使用 and （与）时，表示只有两个条件同时成立的情况下，判断条件才成功。

```
# coding=utf8
# 例3: if语句多个条件

num = 9
if num >= 0 and num <= 10:     # 判断值是否在0~10之间
    print 'hello'
>>> hello        # 输出结果
```

```
8
9  num = 10
10 if num < 0 or num > 10:     # 判断值是否在小于0或大于10
11     print 'hello'
12 else:
13     print 'undefine'
14 >>> undefine          # 输出结果
15
16 num = 8
17 # 判断值是否在0~5或者10~15之间
18 if (num >= 0 and num <= 5) or (num >= 10 and num <= 15):
19     print 'hello'
20 else:
21     print 'undefine'
22 >>> undefine          # 输出结果
```

当 if 有多个条件时可使用括号来区分判断的先后顺序，括号中的判断优先执行，此外 and 和 or 的优先级低于 >（大于）、<（小于）等判断符号，即大于和小于在没有括号的情况下会比与或要优先判断。

也可以在同一行的位置上使用 if 条件判断语句，如下实例：

```
1 #!/usr/bin/python
2
3 var = 100
4
5 if ( var  == 100 ) : print "Value of expression is 100"
6
7 print "Good bye!"
```

以上代码执行输出结果如下：

```
1 Value of expression is 100
2 Good bye!
```

14.10 循环语句

14.10.1 for 循环

Python for 循环可以遍历任何序列的项目，如一个列表或者一个字符串。for 循环的语法格式如下：

```
1 for iterating_var in sequence:
2     statements(s)
```

实例：

```
#!/usr/bin/python

for letter in 'Python':     # First Example
   print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print 'Current fruit :', fruit

print "Good bye!"
```

以上实例输出结果:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

另外一种执行循环的遍历方式是通过索引，如下实例:

```
#!/usr/bin/python

fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]

print "Good bye!"
```

以上实例输出结果:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

以上实例我们使用了内置函数 len() 和 range()。函数 len() 返回列表的长度，即元素的个数。range 返回一个序列的数。

在 python 中，for ···else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while ···else 也是一样。如下实例:

```
#!/usr/bin/python

```

```
for num in range(10,20):  #to iterate between 10 to 20
    for i in range(2,num): #to iterate on the factors of
        the number
        if num%i == 0:      #to determine the first factor
            j=num/i         #to calculate the second factor
            print '%d equals %d * %d' % (num,i,j)
            break #to move to the next number, the #first FOR
    else:                   # else part of the loop
        print num, 'is a prime number'
```

以上实例输出结果:

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

14.10.2 while 循环

Python 编程中 while 语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。其基本形式为:

```
while 判断条件:
    执行语句……
```

执行语句可以是单个语句或语句块。判断条件可以是任何表达式，任何非零、或非空(null)的值均为 true。当判断条件假 false 时，循环结束。实例:

```
#!/usr/bin/python

count = 0
while (count < 9):
    print 'The count is:', count
    count = count + 1

print "Good bye!"
```

以上代码执行输出结果:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
```

```
5 The count is: 4
6 The count is: 5
7 The count is: 6
8 The count is: 7
9 The count is: 8
10 Good bye!
```

如果条件判断语句永远为 true，循环将会无限的执行下去，如下实例：

```
1 #coding=utf-8
2 #!/usr/bin/python
3
4 var = 1
5 while var == 1 :  # 该条件永远为true，循环将无限执行下去
6    num = raw_input("Enter a number  :")
7    print "You entered: ", num
8
9 print "Good bye!"
```

以上实例输出结果：

```
1 Enter a number  :20
2 You entered:  20
3 Enter a number  :29
4 You entered:  29
5 Enter a number  :3
6 You entered:  3
7 Enter a number between :Traceback (most recent call last):
8   File "test.py", line 5, in <module>
9     num = raw_input("Enter a number :")
10 KeyboardInterrupt
```

注意：以上的无限循环你可以使用 CTRL+C 来中断循环。

像 for 循环一样，while 循环也可以用 else 语句，else 中的语句会在循环正常执行完后执行，如下：

```
1 #!/usr/bin/python
2
3 count = 0
4 while count < 5:
5    print count, " is  less than 5"
6    count = count + 1
7 else:
8    print count, " is not less than 5"
```

以上实例输出结果为：

```
1 0 is less than 5
2 1 is less than 5
3 2 is less than 5
```

```
3 is less than 5
4 is less than 5
5 is not less than 5
```

类似 if 语句的语法，如果你的 while 循环体中只有一条语句，你可以将该语句与 while 写在同一行中，如下所示：

```
#!/usr/bin/python

flag = 1

while (flag): print 'Given flag is really true!'

print "Good bye!"
```

注意：以上的无限循环你可以使用 CTRL+C 来中断循环。

14.10.3 循环嵌套

Python 语言允许在一个循环体里面嵌入另一个循环。

Python for 循环嵌套语法：

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

Python while 循环嵌套语法：

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

可以在循环体内嵌入其他的循环体，如在 while 循环中可以嵌入 for 循环，反之，你可以在 for 循环中嵌入 while 循环。

实例：

以下实例使用了嵌套循环输出 2~100 之间的素数：

```
#coding=utf-8
#!/usr/bin/python

i = 2
while(i < 100):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
```

```
    if (j > i/j) : print i, " 是素数"
    i = i + 1

print "Good bye!"
```

以上实例输出结果:

```
2 是素数
3 是素数
5 是素数
7 是素数
11 是素数
13 是素数
17 是素数
19 是素数
23 是素数
29 是素数
31 是素数
37 是素数
41 是素数
43 是素数
47 是素数
53 是素数
59 是素数
61 是素数
67 是素数
71 是素数
73 是素数
79 是素数
83 是素数
89 是素数
97 是素数
Good bye!
```

14.11 循环控制语句

14.11.1 break 语句

Python break 语句，就像在 C 语言中，打破了最小封闭 for 或 while 循环。break 语句用来终止循环语句，即循环条件没有 False 条件或者序列还没被完全递归完，也会停止执行循环语句。break 语句用在 while 和 for 循环中。如果使用嵌套循环，break 语句将停止执行最深层的循环，并开始执行下一行代码。

Python 语言 break 语句语法:

```
break
```

实例:

```
#!/usr/bin/python

for letter in 'Python':      # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter

var = 10                     # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

14.11.2 continue 语句

Python continue 语句跳出本次循环，而 break 跳出整个循环。continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。continue 语句用在 while 和 for 循环中。

Python 语言 continue 语句语法格式如下：

```
continue
```

实例：

```
#!/usr/bin/python

for letter in 'Python':      # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter

var = 10                     # Second Example
while var > 0:
   var = var -1
   if var == 5:
```

```
        continue
    print 'Current variable value :', var
print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

14.11.3 pass 语句

Python pass 是空语句，是为了保持程序结构的完整性。

Python 语言 pass 语句语法格式如下：

```
pass
```

实例：

```
#!/usr/bin/python

for letter in 'Python':
    if letter == 'h':
        pass
        print 'This is pass block'
    print 'Current Letter :', letter

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
```

```
Current Letter : n
Good bye!
```

14.12 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道 Python 提供了许多内建函数，比如 print()。但你也可以自己创建函数，这被叫做用户自定义函数。

14.12.1 定义函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道 Python 提供了许多内建函数，比如 print()。但你也可以自己创建函数，这被叫做用户自定义函数。

- 函数代码块以 def 关键词开头，后接函数标识符名称和圆括号 ()。
- 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号起始，并且缩进。
- Return[expression] 结束函数，选择性地返回一个值给调用方。不带表达式的 return 相当于返回 None。

语法：

```
def functionname( parameters ):
    "函数_文档字符串"
    function_suite
    return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例：以下为一个简单的 Python 函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

```
def printme( str ):
    "打印传入的字符串到标准显示设备上"
    print str
    return
```

14.12.2 函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从 Python 提示符执行。

如下实例调用了 printme () 函数：

```
#coding=utf-8
#!/usr/bin/python

# Function definition is here
def printme( str ):
    "打印任何传入的字符串"
    print str;
    return;

# Now you can call printme function
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
```

输出结果:

```
我要调用用户自定义函数!
再次调用同一函数
```

14.12.3 函数参数

以下是调用函数时可使用的正式参数类型:

- 必备参数
- 命名参数
- 缺省参数
- 不定长参数

14.12.3.1 必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用 printme() 函数，你必须传入一个参数，不然会出现语法错误:

```
#coding=utf-8
#!/usr/bin/python

#可写函数说明
def printme( str ):
    "打印任何传入的字符串"
    print str;
    return;

#调用printme函数
printme();
```

输出结果:

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
```

```
3     printme();
4 TypeError: printme() takes exactly 1 argument (0 given)
```

14.12.3.2 命名参数

命名参数和函数调用关系紧密，调用方用参数的命名确定传入的参数值。你可以跳过不传的参数或者乱序传参，因为 Python 解释器能够用参数名匹配参数值。用命名参数调用 printme() 函数：

```
1 #coding=utf-8
2 #!/usr/bin/python
3
4 #可写函数说明
5 def printme( str ):
6     "打印任何传入的字符串"
7     print str;
8     return;
9
10 #调用printme函数
11 printme( str = "My string");
```

输出结果：

```
1 My string
```

下例能将命名参数顺序不重要展示得更清楚：

```
1 #coding=utf-8
2 #!/usr/bin/python
3
4 #可写函数说明
5 def printinfo( name, age ):
6     "打印任何传入的字符串"
7     print "Name: ", name;
8     print "Age ", age;
9     return;
10
11 #调用printinfo函数
12 printinfo( age=50, name="miki" );
```

输出结果：

```
1 Name:  miki
2 Age  50
```

14.12.3.3 缺省参数

调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的 age，如果 age 没有被传入：

```
#coding=utf-8
#!/usr/bin/python

#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print "Name: ", name;
   print "Age ", age;
   return;

#调用printinfo函数
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

输出结果：

```
Name:  miki
Age  50
Name:  miki
Age  35
```

14.12.3.4 不定长参数

可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。如下实例：

```
#coding=utf-8
#!/usr/bin/python

# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print "输出："
   print arg1
   for var in vartuple:
      print var
   return;
```

```

# 调用printinfo 函数
printinfo( 10 );
printinfo( 70, 60, 50 );
```

输出结果:

```
输出:
10
输出:
70
60
50
```

14.12.4 匿名函数

用 lambda 关键词能创建小型匿名函数。这种函数得名于省略了用 def 声明函数的标准步骤。

- Lambda 函数能接收任何数量的参数但只能返回一个表达式的值，同时只能不能包含命令或多个表达式。
- 匿名函数不能直接调用 print，因为 lambda 需要一个表达式。
- lambda 函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。
- 虽然 lambda 函数看起来只能写一行，却不等同于 C 或 C++ 的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

如下实例:

```
#coding=utf-8
#!/usr/bin/python

#可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

#调用sum函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

输出结果:

```
Value of total :  30
Value of total :  40
```

14.12.5 return 语句

return 语句 [表达式] 退出函数，选择性地向调用方返回一个表达式。不带参数值的 return 语句返回 None。之前的例子都没有示范如何返回数值，下例便告诉你怎么做：

```
#coding=utf-8
#!/usr/bin/python

# 可写函数说明
def sum( arg1, arg2 ):
    # 返回2个参数的和."
    total = arg1 + arg2
    print "Inside the function : ", total
    return total;

# 调用sum函数
total = sum( 10, 20 );
print "Outside the function : ", total
```

输出结果：

```
Inside the function :  30
Outside the function :  30
```

14.12.6 局部和全局变量

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：全局变量、局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

```
#coding=utf-8
#!/usr/bin/python

total = 0; # This is global variable.
# 可写函数说明
def sum( arg1, arg2 ):
    #返回2个参数的和."
    total = arg1 + arg2; # total在这里是局部变量.
    print "Inside the function local total : ", total
    return total;

#调用sum函数
sum( 10, 20 );
print "Outside the function global total : ", total
```

输出结果:

```
Inside the function local total :  30
Outside the function global total :  0
```

14.12.7 数学常量

常量	描述
pi	数学常量 pi（圆周率，一般以 来表示）
e	数学常量 e，e 即自然常数（自然常数）

14.12.8 数学函数——math 模块

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如 abs(-10) 返回 10
cmp(x, y)	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回 x 的整数部分与小数部分，两部分的数值符号与 x 相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数 x 的四舍五入值，如给出 n 值，则代表舍入到小数点后的位数。
ceil(x)	返回数字的上入整数，如 math.ceil(4.1) 返回 5
floor(x)	返回数字的下舍整数，如 math.floor(4.9) 返回 4
exp(x)	返回 e 的 x 次幂 (ex), 如 math.exp(1) 返回 2.718281828459045
log10(x)	返回以 10 为基数的 x 的对数，如 math.log10(100) 返回 2.0
log(x)	如 math.log(math.e) 返回 1.0,math.log(100,10) 返回 2.0
fabs(x)	返回数字的绝对值，如 math.fabs(-10) 返回 10.0
sqrt(x)	返回数字 x 的平方根，数字可以为负数，返回类型为实数，如 math.sqrt(4) 返回 2+0j

其中有些需要导入 math 模块，通过静态对象调用该方法，如 ceil() 用法:

```
import math

math.ceil( x )
```

三角函数

函数	描述
acos(x)	返回 x 的反余弦弧度值。
asin(x)	返回 x 的反正弦弧度值。
atan(x)	返回 x 的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回 x 的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(x*x + y*y)。
sin(x)	返回的 x 弧度的正弦值。
tan(x)	返回 x 弧度的正切值。
degrees(x)	将弧度转换为角度, 如 degrees(math.pi/2) ，返回 90.0
radians(x)	将角度转换为弧度

14.12.9 随机数函数——random 模块

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

Python 包含以下常用随机数函数：

- choice(seq)：从序列的元素中随机挑选一个元素，比如 random.choice(range(10))，从 0 到 9 中随机挑选一个整数。
- randrange ([start,] stop [,step])：从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为 1
- random()：随机生成下一个实数，它在 [0,1) 范围内。
- seed([x])：改变随机数生成器的种子 seed。如果你不了解其原理，你不必特别去设定 seed，Python 会帮你选择 seed。
- shuffle(lst)：将序列的所有元素随机排序
- uniform(x, y)：随机生成下一个实数，它在 [x,y] 范围内。

注意：这些函数都不能直接访问，需要导入 random 模块，然后通过 random 静态对象调用该方法。

14.13 模块

模块让你能够有逻辑地组织你的 Python 代码段。把相关的代码分配到一个模块里能让你的代码更好用，更易懂。模块也是 Python 对象，具有随机的名字属性用来绑定或引用。简单地说，模块就是一个保存了 Python 代码的文件。模块能定义函数，类和变量。模块里也能包含可执行的代码。

例如：一个叫做 aname 的模块里的 Python 代码一般都能在一个叫 aname.py 的文件中找到。下例是个简单的模块 hello.py。

```
def print_func( par ):
```

```
    print "Hello : ", par
    return
```

14.13.1 import 语句

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[,... moduleN]
```

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。搜索路径是一个解释器会先进行搜索的所有目录的列表。如想要导入模块 hello.py，需要把命令放在脚本的顶端：

```
#coding=utf-8
#!/usr/bin/python

# 导入模块
import hello

# 现在可以调用模块里包含的函数了
hello.print_func("Zara")
```

以上实例输出结果：

```
Hello : Zara
```

一个模块只会被导入一次，不管你执行了多少次 import。这样可以防止导入模块被一遍又一遍地执行。

14.13.2 from…import 语句

Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中。语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，要导入模块 fib 的 fibonacci 函数，使用如下语句：

```
from fib import fibonacci
```

这个声明不会把整个 fib 模块导入到当前的命名空间中，它只会将 fib 里的 fibonacci 单个引入到执行这个声明的模块的全局符号表。

14.13.3 from…import* 语句

使用如下声明，可以把一个模块的所有内容全都导入到当前的命令空间：

```
from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。

14.13.4 定位模块

当你导入一个模块，Python 解析器对模块位置的搜索顺序是：

- 当前目录
- 如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录
- 如果都找不到，Python 会察看默认路径。UNIX 下，默认路径一般为/usr/local/lib/python/

模块搜索路径存储在 system 模块的 sys.path 变量中。变量里包含当前目录，PYTHONPATH 和由安装过程决定的默认目录。

14.13.5 PYTHONPATH 变量

作为环境变量，PYTHONPATH 由装在一个列表里的许多目录组成。PYTHONPATH 的语法和 shell 变量 PATH 的一样。

在 Windows 系统，典型的 PYTHONPATH 如下：

```
set PYTHONPATH=c:\python20\lib;
```

在 UNIX 系统，典型的 PYTHONPATH 如下：

```
set PYTHONPATH=/usr/local/lib/python
```

14.13.6 命名空间和作用域

变量是拥有匹配对象的名字（标识符）。命名空间是一个包含了变量名称们（键）和它们各自相应的对象们（值）的字典。

一个 Python 表达式可以访问局部命名空间和全局命名空间里的变量。如果一个局部变量和一个全局变量重名，则局部变量会覆盖全局变量。

每个函数都有自己的命名空间。类的方法的作用域规则和通常函数的一样。

Python 会智能地猜测一个变量是局部的还是全局的，它假设任何在函数内赋值的变量都是局部的。

因此，如果要给全局变量在一个函数里赋值，必须使用 global 语句。

global VarName 的表达式会告诉 Python，VarName 是一个全局变量，这样 Python 就不会在局部命名空间里寻找这个变量了。

例如，我们在全局命名空间里定义一个变量 money。我们再在函数内给变量 money 赋值，然后 Python 会假定 money 是一个局部变量。然而，我们并没有在访问前声明一个局部变量 money，结果就是会出现一个 UnboundLocalError 的错误。取消 global 语句的注释就能解决这个问题。

```
#coding=utf-8
#!/usr/bin/python

Money = 2000
def AddMoney():
    # 想改正代码就取消以下注释:
    # global Money
    Money = Money + 1

print Money
AddMoney()
print Money
```

14.13.7 dir() 函数

dir() 函数一个排好序的字符串列表，内容是一个模块里定义过的名字。

返回的列表容纳了在一个模块里定义的所有模块，变量和函数。如下一个简单的实例：

```
#coding=utf-8
#!/usr/bin/python

# 导入内置math模块
import math

content = dir(math)

print content;
```

以上实例输出结果：

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan'
    ,
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

在这里，特殊字符串变量`__name__` 指向模块的名字，`__file__` 指向该模块的导入文件名。

14.13.8 globals() 和 locals() 函数

根据调用地方的不同，globals() 和 locals() 函数可被用来返回全局和局部命名空间里的名字。

如果在函数内部调用 locals()，返回的是所有能在该函数里访问的命名。

如果在函数内部调用 globals()，返回的是所有在该函数里能访问的全局名字。

两个函数的返回类型都是字典。所以名字们能用 keys() 函数摘取。

14.13.9 reload() 函数

当一个模块被导入到一个脚本，模块顶层部分的代码只会被执行一次。

因此，如果你想重新执行模块里顶层部分的代码，可以用 reload() 函数。该函数会重新导入之前导入过的模块。语法如下：

```
reload(module_name)
```

在这里，module_name 要直接放模块的名字，而不是一个字符串形式。比如想重载 hello 模块，如下：

```
reload(hello)
```

14.13.10 Python 中的包

包是一个分层次的文件目录结构，它定义了一个由模块及子包，和子包下的子包等组成的 Python 的应用环境。包的存在可以让我们一次性导入一个目录下的所有函数。

考虑一个在 Phone 目录下的 pots.py 文件。这个文件有如下源代码：

```
#coding=utf-8
#!/usr/bin/python

def Pots():
    print "I'm Pots Phone"
```

同样地，我们有另外两个保存了不同函数的文件：

- Phone/Isdn.py 含有函数 Isdn()
- Phone/G3.py 含有函数 G3()

现在，在 Phone 目录下创建`__init__.py`：

```
Phone/__init__.py
```

当要导入 Phone 时，为了能够使用所有函数，你需要在`__init__.py` 里使用显式的导入语句，如下：

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

把这些代码添加到__init__.py 之后，导入 Phone 包的时候这些类就全都是可用的了。

```
#coding=utf-8
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

以上实例输出结果：

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

如上，为了举例，我们只在每个文件里放置了一个函数，但其实可以放置许多函数。你也可以在这些文件里定义 Python 的类，然后为这些类建一个包。

14.14 面向对象

14.14.1 面向对象技术简介

- 类 (Class)：用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量用于处理类及其实例对象的相关的数据。
- 方法重载：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重载。
- 实例变量：定义在方法中的变量，只作用于当前实例的类。
- 继承：即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 Dog 类型的对象派生自 Animal 类，这是模拟" 是一个（is-a）" 关系（例图，Dog 是一个 Animal）。
- 实例化：创建一个类的实例，类的具体对象。
- 方法：类中定义的函数。

- 对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

14.14.2 创建类

使用 class 语句来创建一个新类，class 之后为类的名称并以冒号结尾，如下实例：

```
class ClassName:
   '类的帮助信息'   #类文档字符串
   class_suite  #类体
```

类的帮助信息可以通过 ClassName.__doc__ 查看。class_suite 由类成员，方法，数据属性组成。

14.14.2.1 实例

以下是一个简单的 Python 类实例：

```
#coding=utf-8
class Employee:
   '所有员工的基类'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.
         salary
```

- empCount 变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 Employee.empCount 访问；
- 第一种方法 __init__() 方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法。

14.14.2.2 创建实例对象

要创建一个类的实例，你可以使用类的名称，并通 __init__ 方法接受参数。

```
"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
"创建 Employee 类的第二个对象"
emp2 = Employee("Manni", 5000)
```

14.14.2.3 访问属性

可以使用点 (.) 来访问对象的属性。使用如下类的名称访问类变量：

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

14.14.2.4 完整实例

```
#coding=utf-8
#!/usr/bin/python

class Employee:
   '所有员工的基类'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.
         salary

"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
"创建 Employee 类的第二个对象"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

执行以上代码输出结果如下：

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

可以添加，删除，修改类的属性，如下所示：

```
emp1.age = 7  # 添加一个 'age' 属性
emp1.age = 8  # 修改 'age' 属性
del emp1.age  # 删除 'age' 属性
```

也可以使用以下函数的方式来访问属性：

- getattr(obj, name[, default])：访问对象的属性。
- hasattr(obj,name)：检查是否存在一个属性。
- setattr(obj,name,value)：设置一个属性。如果属性不存在，会创建一个新属性。
- delattr(obj, name)：删除属性。

```
hasattr(emp1, 'age')    # 如果存在 'age' 属性返回 True。
getattr(emp1, 'age')    # 返回 'age' 属性的值
setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8
delattr(empl, 'age')    # 删除属性 'age'
```

14.14.2.5 Python 内置类属性

- __dict__：类的属性（包含一个字典，由类的数据属性组成）；
- __doc__：类的文档字符串；
- __name__：类名；
- __module__：类定义所在的模块（类的全名是'__main__.className'，如果类位于一个导入模块 mymod 中，那么 className.__module__ 等于 mymod）；
- __bases__：类的所有父类构成元素（包含了以个由所有父类组成的元组）。

Python 内置类属性调用实例如下：

```
#coding=utf-8
#!/usr/bin/python

class Employee:
   '所有员工的基类'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.
         salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

执行以上代码输出结果如下：

```
Employee.__doc__: 所有员工的基类
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', '
    displayCount': <function displayCount at 0x10a939c80>,
    'empCount': 0, 'displayEmployee': <function
    displayEmployee at 0x10a93caa0>, '__doc__': '\xe6\x89\
    x80\xe6\x9c\x89\xe5\x91\x98\xe5\xb7\xa5\xe7\x9a\x84\xe5
    \x9f\xba\xe7\xb1\xbb', '__init__': <function __init__
    at 0x10a939578>}
```

14.14.2.6 Python 对象销毁 (垃圾回收)

同 Java 语言一样，Python 使用了引用计数这一简单技术来追踪内存中的对象。在 Python 内部记录着所有使用中的对象各有多少引用。一个内部跟踪变量，称为一个引用计数器。当对象被创建时，就创建了一个引用计数，当这个对象不再需要时，也就是说，这个对象的引用计数变为 0 时，它被垃圾回收。但是回收不是" 立即" 的，由解释器在适当的时机，将垃圾对象占用的内存空间回收。

```
a = 40        # 创建对象  <40>
b = a         # 增加引用，  <40> 的计数
c = [b]       # 增加引用.  <40> 的计数

del a         # 减少引用 <40> 的计数
b = 100       # 减少引用 <40> 的计数
c[0] = -1     # 减少引用 <40> 的计数
```

垃圾回收机制不仅针对引用计数为 0 的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充，垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。在这种情况下，解释器会暂停下来，试图清理所有未引用的循环。

实例

析构函数 __del__ , __del__ 在对象消逝的时候被调用，当对象不再被使用时，__del__ 方法运行：

```
#coding=utf-8
#!/usr/bin/python

class Point:
    def __init( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
```

```
        print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # 打印对象的id
del pt1
del pt2
del pt3
```

实例运行结果如下：

```
3083401324 3083401324 3083401324
Point destroyed
```

注意：通常需要在单独的文件中定义一个类。

14.14.2.7 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。

需要注意的地方：继承语法 class 派生类名（基类名）：//... 基类名写作括号里，基本类是在类定义的时候，在元组之中指明的。

在 python 中继承中的一些特点：

- 1：在继承中基类的构造（__init__() 方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
- 2：在调用基类的方法时，需要加上基类的类名前缀，且需要带上 self 参数变量。区别于在类中调用普通函数时并不需要带上 self 参数
- 3：Python 总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。(先在本类中查找调用的方法，找不到才去基类中找)。

如果在继承元组中列了一个以上的类，那么它就被称作" 多重继承" 。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
```

实例：

```
#coding=utf-8
#!/usr/bin/python

class Parent:        # 定义父类
    parentAttr = 100
```

```
    def __init__(self):
        print "调用父类构造函数"

    def parentMethod(self):
        print '调用父类方法'

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print "父类属性 :", Parent.parentAttr

class Child(Parent): # 定义子类
    def __init__(self):
        print "调用子类构造方法"

    def childMethod(self):
        print '调用子类方法 child method'

c = Child()           # 实例化子类
c.childMethod()       # 调用子类的方法
c.parentMethod()      # 调用父类方法
c.setAttr(200)        # 再次调用父类的方法
c.getAttr()           # 再次调用父类的方法
```

以上代码执行结果如下：

```
调用子类构造方法
调用子类方法 child method
调用父类方法
父类属性 : 200
```

可以继承多个类

```
class A:        # 定义类 A
.....

class B:         # 定义类 B
.....

class C(A, B):   # 继承类 A 和 B
.....
```

可以使用 issubclass() 或者 isinstance() 方法来检测：

- issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)；
- isinstance(obj, Class) 布尔函数如果 obj 是 Class 类的实例对象或者是一个 Class 子类的实例对象则返回 true。

14.14.2.8 方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

```
#coding=utf-8
#!/usr/bin/python

class Parent:        # 定义父类
   def myMethod(self):
      print '调用父类方法'

class Child(Parent): # 定义子类
   def myMethod(self):
      print '调用子类方法'

c = Child()          # 子类实例
c.myMethod()         # 子类调用重写方法
```

执行以上代码输出结果如下：

```
调用子类方法
```

14.14.2.9 基础重载方法

下表列出了一些通用的功能，可以在自己的类重写：

- __init__ (self [,args...])：构造函数。简单的调用方法：obj = className(args)
- __del__(self)：析构方法，删除一个对象。简单的调用方法: dell obj
- __repr__(self)：转化为供解释器读取的形式。简单的调用方法: repr(obj)
- __str__(self)：用于将值转化为适于人阅读的形式。简单的调用方法: str(obj)
- __cmp__(self, x)：对象比较。简单的调用方法: cmp(obj, x)

14.14.2.10 运算符重载

Python 同样支持运算符重载，实例如下：

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
```

```
print v1 + v2
```

以上代码执行结果如下所示：

```
Vector(7,8)
```

14.14.2.11 类属性与方法

- 类的私有属性，__private_attrs：两个下划线开头，声明该属性为私有，不能在类地外部被使用或直接访问。在类内部的方法中使用时 self.__private_attrs。
- 类的方法，在类地内部，使用 def 关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数 self，且为第一个参数
- 类的私有方法，__private_method：两个下划线开头，声明该方法为私有方法，不能在类地外部调用。在类的内部调用 slef.__private_methods

实例

```
#coding=utf-8
#!/usr/bin/python

class JustCounter:
    __secretCount = 0  # 私有变量
    publicCount = 0    # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.publicCount
print counter.__secretCount  # 报错，实例不能访问私有变量
```

Python 通过改变名称来包含类名：

```
1
2
2
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print counter.__secretCount  # 报错，实例不能访问私有变量
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python 不允许实例化的类访问私有数据，但你可以使用 object._className__attrName 访问属性，将如下代码替换以上代码的最后一行代码：

```
.......................
print counter._JustCounter__secretCount
```

执行以上代码，执行结果如下：

```
1
2
2
2
```

14.15 模块学习

14.15.1 optparse 模块

Python 有两个内建的模块用于处理命令行参数：一个是 getopt，《Deep in python》一书中也有提到，只能简单处理命令行参数；另一个是 optparse，它功能强大，而且易于使用，可以方便地生成标准的、符合 Unix/Posix 规范的命令行说明。

简单流程

首先，必须 import OptionParser 类，创建一个 OptionParser 对象：

```
from optparse import OptionParser
parser = OptionParser()
```

然后，使用 add_option 来定义命令行参数：

```
parser.add_option("-f", "--file", action = "store", type =
    "string", dest = "fileName")
```

add_option() 参数说明：

- short option string: 为第一个参数，表示 option 的缩写，例如-f;
- long option string: 为第二个参数，表示 option 的全拼，例如–file;

后面的参数皆为命名参数，命名参数为可选参数;

- action=: 表示对此 option 的处理方式，默认值为 store，表示存储 option 的值到解析后的 options 对象的成员中。action 还可以有其他的值：对于 bool 值，使用 store_true 来默认存储 true，使用 store_false 来默认存储 false，store_const 用来存储 const 设置的值到此 option，append 表示增加 option 的参数到 list 中，此时此 option 是一个 list，可能包含多个值，count 表示对 counter 增加一，callback 表示调用指定的函数。所有的 action 值如下：store + store_true + store_false + store_const + append + count + callback
- type=: 表示此 option 的值的类型，默认为 string，可以指定为 string,int,choice, float andcomplex；

- dest=：表示此 option 在经过 optionparser 解析后的 options 对象中成员的名字，默认使用 long option string；
- default=: 表示比 option 的默认值；
- metavar=: 表示显示到 help 中 option 的默认值（显示到 help 的时候并不是 default）；
- const=: 当 action 为 store_const 的时候，需要设置此值；
- choices=: 当设置 type 为 choices 时，需要设置此值；

14.15.2 os 模块

一、os 模块概述

Python os 模块包含普遍的操作系统功能。如果你希望你的程序能够与平台无关的话，这个模块是尤为重要的。(一语中的)

二、常用方法

1、os.name

输出字符串指示正在使用的平台。如果是 window 则用“nt”表示，对于 Linux/Unix 用户，它是“posix”。

2、os.getcwd()

函数得到当前工作目录，即当前 Python 脚本工作的目录路径。

3、os.listdir()

返回指定目录下的所有文件和目录名。

```
>>> os.listdir(os.getcwd())
['Django', 'DLLs', 'Doc', 'include', 'Lib', 'libs', '
    LICENSE.txt', 'MySQL-python-wininst.log', 'NEWS.txt', '
    PIL-wininst.log', 'python.exe', 'pythonw.exe', 'README.
    txt', 'RemoveMySQL-python.exe', 'RemovePIL.exe', '
    Removesetuptools.exe', 'Scripts', 'setuptools-wininst.
    log', 'tcl', 'Tools', 'w9xpopen.exe']
>>>
```

4、os.remove()

删除一个文件。

5、os.system()

运行 shell 命令。

```
>>> os.system('dir')
or
>>> os.system('cmd') #启动dos
```

6、os.sep 可以取代操作系统特定的路径分割符。

7、os.linesep 字符串给出当前平台使用的行终止符

```
>>> os.linesep
'\r\n'                  #Windows使用'\r\n', Linux使用'\n'而Mac使用'\r'。
>>> os.sep
'\\'                    #Windows
>>>
```

8、os.path.split()

函数返回一个路径的目录名和文件名

```
>>> os.path.split('C:\\Python25\\abc.txt')
('C:\\Python25', 'abc.txt')
```

9、os.path.isfile() 和 os.path.isdir() 函数分别检验给出的路径是一个文件还是目录。

```
>>> os.path.isdir(os.getcwd())
True
>>> os.path.isfile('a.txt')
False
```

10、os.path.exists() 函数用来检验给出的路径是否真地存在

```
>>> os.path.exists('C:\\Python25\\abc.txt')
False
>>> os.path.exists('C:\\Python25')
True
>>>
```

11、os.path.abspath(name): 获得绝对路径

12、os.path.normpath(path): 规范 path 字符串形式

13、os.path.getsize(name): 获得文件大小，如果 name 是目录返回 0L

14、os.path.splitext(): 分离文件名与扩展名

```
>>> os.path.splitext('a.txt')
('a', '.txt')
```

15、os.path.join(path,name): 连接目录与文件名或目录

```
>>> os.path.join('c:\\Python','a.txt')
'c:\\Python\\a.txt'
>>> os.path.join('c:\\Python','f1')
'c:\\Python\\f1'
>>>
```

16、os.path.basename(path): 返回文件名

```
>>> os.path.basename('a.txt')
'a.txt'
>>> os.path.basename('c:\\Python\\a.txt')
'a.txt'
>>>
```

17、os.path.dirname(path): 返回文件路径

```
>>> os.path.dirname('c:\\Python\\a.txt')
'c:\\Python'
```

14.15.3 sys 模块

（1）sys.argv 很多人会想，我如何给我的程序在外部传递参数呢？这个，就可以实现。如：

```
Tesy.py
Import sys
Print sys.argv[number]
```

一般情况下，number 为 0 是这个脚本的名字，1，2…则为命令行下传递的参数，如：

Test.py 脚本内容：

```
import sys

print sys.argv[0]
print sys.argv[1]
print sys.argv[2]
print sys.argv[3]
```

那么

```
[root@databak scripts]# python test.py arg1 arg2 arg3
test.py
arg1
arg2
arg3
```

一个有用的实例：

```
#!/usr/bin/env python

import sys

if len(sys.argv) == 3:
    ftype = sys.argv[1]
    fname =  sys.argv[2]
```

```
else:
    print " Claculates the average of 2DES time-domain
        outputs."
    print " USAGE: aver filetype filename"
    print "     eg: aver rp fort.71"
    sys.exit(2)

print ftype
print fname
```

（2）sys.exit(n) 执行至主程序的末尾时, 解释器会自动退出. 但是如果需要中途退出程序, 你可以调用 sys.exit 函数, 它带有一个可选的整数参数返回给调用它的程序. 这意味着你可以在主程序中捕获对 sys.exit 的调用。（注：0 是正常退出，其他为不正常，可抛异常事件供捕获!）

(3)sys.path 大家对模块都有一定了解吧？大家在使用模块的某一个功能前，是不是需要导入呢？答案是需要。那 import,__import__ 命令就不用提干嘛的了吧。那大家在执行 import module_name 的时候，python 内部发生了什么呢？简单的说，就是搜索 module_name。根据 sys.path 的路径来搜索 module.name

```
>>> sys.path
['', '/usr/local/lib/python24.zip', '/usr/local/lib/
    python2.4', '/usr/local/lib/python2.4/plat-freebsd4', '
    /usr/local/lib/python2.4/lib-tk', '/usr/local/lib/
    python2.4/lib-dynload', '/usr/local/lib/python2.4/site-
    packages']
```

大家以后写好的模块就可以放到上面的某一个目录下，便可以正确搜索到了。当然大家也可以添加自己的模块路径：

```
Sys.path.append( “mine module path” )
```

14.16 Python 内置特殊函数

这些函数很具 python 特性，可以让代码更加简洁。

1、filter(function, sequence)：

```
str = ['a', 'b','c', 'd']

def fun1(s): return s if s != 'a' else None

ret = filter(fun1, str)

print ret

## ['b', 'c', 'd']
```

对 sequence 中的 item 依次执行 function(item)，将执行结果为 True 的 item 组成一个 List/String/Tuple（取决于 sequence 的类型）返回。

可以看作是过滤函数。

2、map(function, sequence)

```
str = ['a', 'b','c', 'd']

def fun2(s): return s + ".txt"

ret = map(fun2, str)

print ret

## ['a.txt', 'b.txt', 'c.txt', 'd.txt']
```

对 sequence 中的 item 依次执行 function(item)，见执行结果组成一个 List 返回：

map 也支持多个 sequence，这就要求 function 也支持相应数量的参数输入：

```
def add(x, y): return x+y
 print map(add, range(10), range(10))
##[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3、reduce(function, sequence, starting_value)：def add1(x,y): return x + y

```
print reduce(add1, range(1, 100))

print reduce(add1, range(1, 100), 20)

## 4950  (注: 1+2+...+99)
## 4970  (注: 1+2+...+99+20)

对sequence中的item顺序迭代调用function，如果有
   starting_value，还可以作为初始值调用，例如可以用来对
   List求和：
```

4、lambda：

```
g = lambda s: s + ".fsh"

print g("haha")

print (lambda x: x * 2) (3)

## haha.fsh

## 6
```

这是 Python 支持一种有趣的语法，它允许你快速定义单行的最小函数，类似与 C 语言中的宏，这些叫做 lambda 的函数.

14.17 Python 数值计算库 NumPy 和 SciPy

标准的 Python 中用列表 (list) 保存一组值，可以用来当作数组使用。但由于列表的元素可以是任何对象，因此列表中所保存的是对象的指针。这样为了保存一个简单的列表 [1,2,3]，需要有三个指针和三个整数对象。对于数值运算来说这种结构显然比较浪费内存和 CPU 计算时间。

此外 Python 还提供了 array 模块，它所提供的 array 对象和列表不同，能直接保存数值，和 C 语言的一维数组类似。但是由于它不支持多维数组，也没有各种运算函数，因此也不适合做数值运算。

NumPy 为 Python 提供了快速的多维数组处理的能力，而 SciPy 则在 NumPy 基础上添加了众多的科学计算所需的各种工具包，有了这两个库，Python 就有几乎和 Matlab 一样的处理数据和计算的能力了。

NumPy 和 SciPy 官方网址：http://www.numpy.org、http://www.scipy.org

NumPy 为 Python 带来了真正的多维数组功能，并且提供了丰富的函数库处理这些数组。它将常用的数学函数都进行数组化，使得这些数学函数能够直接对数组进行操作，将本来需要在 Python 级别进行的循环，放到 C 语言的运算中，明显地提高了程序的运算速度。

SciPy 的核心计算部分都是一些久经考验的 Fortran 数值计算库，例如：

- 线性代数使用 LAPACK 库
- 快速傅立叶变换使用 FFTPACK 库
- 常微分方程求解使用 ODEPACK 库
- 非线性方程组求解以及最小值求解等使用 MINPACK 库

14.17.1 文件存取

savetxt() 和 loadtxt() 可以读写保存一维和二维数组的文本文件，如 CSV 格式的文本文件。

14.17.1.1 说明文档

http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html

numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', foote

Save an array to a text file.

Paramters:

- fname : filename or file handle

  If the filename ends in .gz, the file is automatically saved in compressed gzip format. loadtxt understands gzipped files transparently.

- X : array_like

  Data to be saved to a text file.

- fmt : str or sequence of strs, optional A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. 'Iteration %d –%10.5f', in which case delimiter is ignored. For complex X, the legal options for fmt are:
  - a single specifier, fmt='%.4e', resulting in numbers formatted like '(%s+%sj)' % (fmt, fmt)
  - a full string specifying every real and imaginary part, e.g. '%.4e %+.4j %.4e %+.4j %.4e %+.4j' for 3 columns
  - a list of specifiers, one per column - in this case, the real and imaginary part must have separate specifiers, e.g. ['%.3e + %.3ej', '(%.15e%+.15ej)'] for 2 columns
- delimiter : str, optional

  String or character separating columns.
- newline : str, optional

  String or character separating lines. New in version 1.5.0.
- header : str, optional

  String that will be written at the beginning of the file. New in version 1.7.0.
- footer : str, optional

  String that will be written at the end of the file. New in version 1.7.0.
- comments : str, optional

  String that will be prepended to the header and footer strings, to mark them as comments. Default: '# ', as expected by e.g. numpy.loadtxt. New in version 1.7.0.

http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

`numpy.loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None`

Load data from a text file.Each row in the text file must have the same number of values.

Parameters:

- fname : file or str

  File, filename, or generator to read. If the filename extension is .gz or .bz2, the file is first decompressed. Note that generators should return byte strings for Python 3k.
- dtype : data-type, optional

  Data-type of the resulting array; default: float. If this is a record data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.
- comments : str, optional

  The character used to indicate the start of a comment; default: '#'.
- delimiter : str, optional

  The string used to separate values. By default, this is any whitespace.
- converters : dict, optional

A dictionary mapping column number to a function that will convert that column to a float. E.g., if column 0 is a date string: converters = 0: datestr2num. Converters can also be used to provide a default value for missing data (but see also genfromtxt): converters = 3: lambda s: float(s.strip() or 0). Default: None.

- skiprows : int, optional

  Skip the first skiprows lines; default: 0.

- usecols : sequence, optional

  Which columns to read, with 0 being the first. For example, usecols = (1,4,5) will extract the 2nd, 5th and 6th columns. The default, None, results in all columns being read.

- unpack : bool, optional

  If True, the returned array is transposed, so that arguments may be unpacked using x, y, z = loadtxt(...). When used with a record data-type, arrays are returned for each field. Default is False.

- ndmin : int, optional

  The returned array will have at least ndmin dimensions. Otherwise mono-dimensional axes will be squeezed. Legal values: 0 (default), 1 or 2. New in version 1.6.0.

Returns: out : ndarray, Data read from the text file.

14.17.1.2 单一数据类型的情况

（1）默认格式。默认按照%.18e 格式保存数值，以空格分隔。

```
>>> import numpy as np
>>> a = np.arange(0,12,0.5).reshape(4,-1)
>>> np.savetxt("a.txt",a)
>>> np.loadtxt("a.txt")
array([[  0. ,   0.5,   1. ,   1.5,   2. ,   2.5],
       [  3. ,   3.5,   4. ,   4.5,   5. ,   5.5],
       [  6. ,   6.5,   7. ,   7.5,   8. ,   8.5],
       [  9. ,   9.5,  10. ,  10.5,  11. ,  11.5]])
```

（2）自定义格式。

```
>>> import numpy as np
>>> a = np.arange(0,12,0.5).reshape(4,-1)
>>> np.savetxt("a.txt", a, fmt="%d", delimiter=",")
#保存为整数，以逗号分隔
>>> np.loadtxt("a.txt", delimiter=",")
# 读入的时候也需要指定逗号分隔
array([[  0.,   0.,   1.,   1.,   2.,   2.],
       [  3.,   3.,   4.,   4.,   5.,   5.],
       [  6.,   6.,   7.,   7.,   8.,   8.],
       [  9.,   9.,  10.,  10.,  11.,  11.]])
```

14.17.1.3 混合数据类型的情况

有的 CSV 文件中除了保存数值之外，还保存了一些说明文字。例如第一行和第一列通常为行名和列名。如果需要忽略 CSV 文件的第一行和第一列，可以先将文件读为字符串数组，然后取出需要的部分，转换为数值数值。例如对于下面的 CSV 数据文件：

```
姓名,年龄,体重,身高
张三,30,75,165
李四,45,60,170
王五,15,30,120
```

可以采用如下程序读入其中的数值部分：

```
>>> import numpy as np
>>> tmp = np.loadtxt("test.csv", dtype=np.str, delimiter=",")
>>> data = tmp[1:,1:].astype(np.float)
>>> data
array([[  30.,   75.,  165.],
       [  45.,   60.,  170.],
       [  15.,   30.,  120.]])
```

第十五章 Markdown

[转]http://elegantlatex.org/2014/06/11/markdown-first-start

在 ubuntu 下可以用 ReText 编辑器。

15.1 简介

Markdown 的目标是实现「易读易写」。

可读性，无论如何，都是最重要的。一份使用 Markdown 格式撰写的文件应该可以直接以纯文本发布，并且看起来不会像是由许多标签或是格式指令所构成。Markdown 语法受到一些既有 text-to-HTML 格式的影响，包括 Setext、atx、Textile、reStructuredText、Grutatext 和 EtText，而最大灵感来源其实是纯文本电子邮件的格式。

总之，Markdown 的语法全由一些符号所组成，这些符号经过精挑细选，其作用一目了然。比如：在文字两旁加上星号，看起来就像强调。Markdown 的列表看起来，嗯，就是列表。Markdown 的区块引用看起来就真的像是引用一段文字，就像你曾在电子邮件中见过的那样。

15.2 段落文字编写

15.2.1 空格与断行

- 单个空格与多个空格效果是一样的（单个空格）
- 单个回车视为空格，多个回车视为断行

这点跟 LaTeX 输入是一样的。

15.2.2 斜体与加粗

斜体只需在斜体文字左右加星号（*）或下划线（_）即可，而粗体需要加双星号（**）或双下划线（__）。注意：星号或者下划线应该与斜体或者加粗字体相连，不能有空格。

示例代码：

```
*斜体示例 Italic words and sentences*

_斜体示例 Italic words and sentences_

**黑体示例 Bold Face samples**

__黑体示例 Bold Face samples__
```

15.2.3 引用文字

引文使用 > 开头。

15.2.4 标题

Markdown 标题等级有六级，从一级标题到六级标题分别由 # 到 ###### 表示。

示例代码：

```
# 一级标题示例
## 二级标题示例
### 三级标题示例
...
###### 六级标题，最末等级标题
```

15.2.5 列表

列表有两类，分别为有序列表和无序列表，有序列表使用 1、2、3 …引导，无序列表使用星号（*）、加号（+）或者减号（-）引导，且三者的效果是一样的（同一等级的话）。并且列表之间是可以嵌套的。

示例代码（无序列表）：

```
* Candy.
    + Candy.
    + Gum.
    + Booze.
* Gum.
    - Candy.
    - Gum.
    - Booze.
* Booze.
```

示例代码（有序列表）：

```
1. Red
    1. Red
    2. Green
    3. Blue
2. Green
    + Candy.
    + Gum.
    + Booze.
3. Blue
```

15.3 超链接

Markdown 支持两种形式的链接语法：行内和参考两种形式，两种都是使用括号把文字转成链接。

15.3.1 行内语法

示例代码：

```
This is an [example link with title](http://example.com/ "标题 Title").
```

说明：行内语法用到了方括号和圆括号，方括号表示链接在网页中的显示效果，而圆括号内为链接地址，使用双引号可以加入链接标题，这个标题是指当我们鼠标停留在链接时，网页上浮动显示的文字，链接标题是可选项，比如：

```
This is an [example link without title](http://example.com/).
```

15.3.2 参考语法

参考形式的链接让你可以为链接定一个名称，之后你可以在文件的其他地方（建议放在文末）定义该链接的内容。

示例代码：

```
ElegantLaTeX 的站点链接为 [ElegantLaTeX][1]，其中部分内容来源于 [TeX Stack Exchange][2]，
作为更全面的一个站点，我们推荐 [LaTeX Studio][3].

[1]: http://elegantlatex.org/ "ElegantLaTeX"
[2]: http://tex.stackexchange.com/ "TeX.Stack.Exchange"
[3]: http://www.latexstudio.net/ "LaTeX Studio"
```

15.4 插图

图片的语法和链接很像，区别在开头加一个惊叹号!。同样，图片插入也有两种方式，一种是行内，一种是参考引用。

15.4.1 行内语法

示例代码：

```
我们 ElegantLaTeX 的图标是 ![ElegantLaTeX Logo](http://elegantlatex.qiniudn.com/
wp-content/uploads/2014/05/logo-e1400431970427.png "ElegantLaTeX Logo")
```

15.4.2 参考语法

参考语法的格式和超链接的几乎一致，比如我们要显示百度的 Logo 使用下面的代码：

示例代码：

```
百度的图标为 ![百度 Logo][4]。

[4]: http://www.baidu.com/img/baidu_sylogo1.gif "百度"
```

15.5 代码显示

15.5.1 行内代码

Markdown 中代码显示比较简单，行内代码使用反引号 ‘ 来标记代码区段。

示例代码：

```
在 LaTeX 中调用宏包的命令为‘\usepackage{pkg.name}‘，其中 ‘pkg.name‘ 为宏包名。
```

15.5.2 行间代码

行间代码只要每行都缩进 4 个空格或是一个 tab 就可以了。

```
C++ code

    public class Blog
    {
        public int Id { get; set; }
        public string Subject { get; set; }
    }
```

15.6 \ 的作用

反斜杠 \ 可以显示 *_+-‘ 等在 Markdown 中有特殊意义的字符。

第十六章 Fortran

16.1 哑元

- intent(in) 表示这个参数是输入的；
- intent(out) 表示参数是输出的；
- intent(inout) 表示这个参数同时用于两个方向的数据传递。

哑实结合是在两个程序单元间传递数值的主要手段，主程序中实元 2.0 与过程中哑元 X 结合，就使 X 有值 2.0，也即把主程序中 2.0 的值传递给子程序中的 X，该值可供子程序运算。反之，如果子程序中的变量 Y 在子程序执行完后有值 3.0，它与实元 R 结合后则使调用程序单元中的实元变量 R 得值 3.0。

在 F77 中，不能确切地说明哑元的目的。它们到底是用于把数据传入到过程中的，还是用于把数据传出到调用它的程序单元中的，或是两者兼而有之的，这个概念是含糊的。在 F90 中，为了避免当过程内部变量值变化后返回到引用的程序单元时可能造成的混淆情况，在过程的变量类型的定义中，可以对哑元指定意图说明的 INTENT 属性。哑元按数据传输特性可分为输入输出两用、仅用于输入和仅用于输出。其一般形式为：

- 在类型定义语句中：`类型，INTENT(意图说明符) :: 哑元名表`，如：`real*8, intent(in) :: X`；
- 或用 INTENT 语句：`INTENT(意图说明符) :: 哑元名表`，如：`intent(in) :: X`。

推荐使用第一种形式。意图说明符为以下字符串：

- IN ： 指明哑元仅用于向过程提供数据，过程的执行期间哑元不能被重定义或成为未定义的，相联合的实元可以是常数、变量、数组以及它们的算术表达式；
- OUT ： 指明哑元用于把过程中的数据传回调用过程的程序，与之相结合的实元只允许是变量，不得为常数或算术表达式；
- INOUT：指明哑元既可以用于向过程提供数据，也可用于向调用程序返回数据，与之相结合的实元只允许是变量。

INTENT 属性不能在主程序说明语句中出现，只能在过程的哑元说明语句中使用。它是可选的，可省略。但现代特性的编程中应提倡使用 INTENT 属性，因为这样能增加可读性和可维护性，还能防止编程中的一些错误。因为一旦哑实结合，哑元和实元始终是同一个值，如果过程中给有属性 INTENT(IN) 的哑元重新赋值，也将改变调用程序单元中实元的值，而这是不应该的。这样，如在程序执行部分中误把有 INTENT(IN) 属性的哑元赋值时，操作系统就会提示。

16.2 双冒号

在新的语法里，很多变量类型后书写两个冒号，然后是变量名。

双冒号，表示修饰符的结束。例如：

`Integer , parameter , private :: fcode( 3 , 3 )`

Integer 是变量类型，它有两个修饰符：parameter 和 private，两个冒号表示修饰符结束了，也就是说，如果有修饰符，则必须有双冒号。

语法还规定，如果定义变量时，同时对其赋予初始值，则变量默认具有 save 属性，此时相当于隐藏了 save 这个修饰符，因此也必须有双冒号。例如：`integer :: a = 0`，实际是 `integer , save :: a = 0`，因此不能简写为：`integer a=0`。

因此，也建议在所有定义语句中使用双冒号。

16.3 浮点型变量声明的格式

单精度浮点数的声明方法單（使用 4 个字节）

```
real(kind=4) x_float !Fortran 90
real(4) y_float            !Fortran 77
real*4 z_float             !Fortran 77
```

双精度浮点数的声明方法（使用 8 个字节）

```
real(kind=8) x_double !Fortran 90
real(8) y_double            !Fortran
real*8 z_double             !Fortran
```

16.4 输入输出的格式命令

格式命令，[] 中的可省略

- Aw： 以 w 个字符宽度来输出字符串；
- BN： 定义文本框中的空位为没有东西，在输入时才需要使用；
- BZ： 定义文本框中的空位代表 0，输入时才需要使用；
- Dw.d：以 w 个字符宽来输出指数类型的浮点数，小数部分占 d 个字符宽；
- Ew.d[Ee]：以 w 个字符宽度来输出指数类型的浮点数，小数部分占 d 个字符宽，指数部分占 e 个字符；
- ENw.d[Ee]：以指数类型来输出浮点数，工程计数法
- ESw.d[Ee]：以指数类型来输出浮点数，科学计数法
- Fw.d：以 w 个字符宽来输出浮点数，小数部分占 d 个字符宽；
- Gw.d[Ee]：以 w 个字符宽度来输出任何种类的数据；
- Iw[.m]：以 w 个字符宽来输出整数，最少输出 m 个数字；
- Lw：以 w 个字符宽来输出 T 或 F 的真假值；
- nX：把输出的位置向右跳过 n 个位置；
- /：换行；
- ：：在没有更多数据时结束输出；

- kP：K 值控制输入输出的 SCALE；
- Tn：输出的位置移动到本行第 n 列；
- TLn：输出的位置向左相对移动 n 列；
- TRn：输出的位置向右相对移动 n 列；
- SP：在数值为正时加上“正号”；
- SS：取消 SP；
- 以下 Fortran 90 添加
  - Bw[.m]：把整数转换成二进制来输出，输出会占 w 个字符宽，固定输出 m 个数字，m 值可以不给定；
  - Ow[.m]：把整数转换成八进制来输出，输出会占 w 个字符宽，固定输出 m 个数字，m 值可以不给定；
  - Zw[.m]：把整数转换成十六进制来输出，输出会占 w 个字符宽，固定输出 m 个数字，m 值可以不给定；

16.5 整数转字符串——产生序列文件名

整型转字符，借助 FORTRAN 语言的内部文件完成，即将一个字符串变量当作一个内部文件来看待。给一个实用的例子吧。

假如你有一组文件有 20 个，命名规律是 myFile*.dat，其中 * 是从 1 到 20 递增的整型数，则要用循环依次打开这些文件可以这样写：

```
program main
implicit none

character( len = 2 ) :: cTemp
integer :: k

do k = 1, 20

write( cTemp,'(i2)' ) k
open ( 1, file = 'myFile' // trim(adjustl( cTemp )) // '.dat', status = 'old' )
! ...
close( 1 )

end do

end porgram main
```

这里因为 k 可能是一位数，也可能是两位数，所以字符串变量 cTemp 的长度至少要是 2 个字符长度，才能保证最大整数能装下。trim 和 adjustl 是 FORTRAN 的标准内部函数。adjustl 的作用是将字符串里的内容左对齐，空格置于右端。trim 的作用是将字符串末尾 (即右端) 的空格删掉。这样无论你的 k 是一位数还是两位数，都可以保证 open 路径中不会出现多余的空格。// 是 FORTRAN 的字符串操作符，作用是将字符串连接起来。

同理，如果是字符型数字转整型或实型，方法一样，比如：

```
character( len = 4 ) :: cTemp = '2007'
integer :: year

read( cTemp, '(i4)' ) :: year
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  CHARACTER*3   STR
  WRITE(STR,'(A3)')123
  此时STR作为内部文件使用
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Use a character string variable as an "internal file", write the file
: name you want to this string (like sprintf in C), and use the string.
: character(len=12)::fname
: integer::i
: do i=0,9999,1
: write(unit=fname,fmt="('file',i4.4,'.dat')")i ! like sprintf() in C
: open(unit=10,file=fname)
: ...
: close(unit=10)
: end do
编制有限元程序时，经常要用到序列文件名，如：
file001.txt, file002.txt, ...

贴出源程序，供各位参考：

文件名：  NAMEGEN.F90
*****************************************************************


MODULE NAMEGEN_

CONTAINS


!*****************************************************************

FUNCTION NAMEGEN (NAME, NUM, INDEX, EXT)

!*****************************************************************
!
! 程序说明：
! Generate sorted file name. Attach the number after each filename.
! The conversion of integer value into character is used.
!
! INPUT:
! NAME filename;
```

```
! NUM indicate the maximum file number
!                                          used, e.g.
! "0000","00000";
! INDEX the number of this file;
! EXT extension of the file;
! EXAMPLE:
! NAME='FILE'
! NUM='0000'
! INDEX=18
! EXT='TXT'
! Returned name is: FILE0018.TXT
!
! Xin Zhao, 2002-2-12
!
    ***************************************************************

CHARACTER*108 NAMEGEN,NA,NUM1
CHARACTER*(*) NAME,EXT,NUM
INTEGER*4 INDEX,LENGTH1,LENGTH2
WRITE(NA,'(I4)') INDEX
NA=ADJUSTL (NA)
LENGTH1=LEN_TRIM(NA)
LENGTH2=LEN_TRIM(NUM)
NUM1=NUM(1:(LENGTH2-LENGTH1))//NA
NAMEGEN=TRIM(NAME)//TRIM(NUM1)//'.'//TRIM(EXT)
END FUNCTION

END MODULE
```

16.6 Fortran 格式输出——不换行与换行

`write (file_number,format_spec,advance='no')` 这样写就不会自动换行，而格式中如果加入 '\' 则是换行和\n 一样。

实例：

原来的：

```
    open( unit = 10, file = "real_rho.ods" )
    write( 10, 100 )  t/fs ,   real( y0( 1, 1, 1 ) ) , &
                                  & real( y0( 1, 2, 2 ) ) , &
                                  & real( y0( 1, 3, 3 ) ) , &
                                  & real( y0( 1, 4, 4 ) ) , &
                                  & real( y0( 1, 5, 5 ) ) , &
                                  & real( y0( 1, 6, 6 ) ) , &
                                  & real( y0( 1, 7, 7 ) )
        100 format( 8(f14.7 , ',') )
```

```
    200 format( 8(A8 , ',' ) )
```

改:

```
open(unit = 10, file = "real_rho.ods" )
write(10, 100, advance='no')  t/fs
do j = 1, nsite-1
    write(10, 100, advance='no')  real( y0(1,j,j) )
enddo
write(10, 100) real(y0(1,nsite,nsite))
100 format( 3(f14.7 , ',') )
200 format( 1(A8 , ',' ) )
```

新改的程序适应性更强，不用手动修改输出。

16.7 getarg、iargc、trim 的用法

getarg 用法:`call getarg(NUMBER,VALUE)` 其中 NUMBER 是获取第几个参数, VALUE 是相应的值。iargc 用法: `n=iargc()`, 返回命令行中参数的数量。

getarg 是用来返回你输入的命令行参数的。

`call getarg(n,buffer)`

其中 n 是命令序号，buffer 是相应的命令行参数。运行程序本身的命令是 0 号，跟在它后面的参数是 1，2。。。号。

比如，你写这样一个小程序:

```
character*80 buff
call getarg(0,buff)
write(*,*) buff
call getarg(1, buff)
write(*,*) buff
call getarg(2, buff )
write (*,*) buff
end
```

然后编译它，比如把这个可执行程序命名为 mypro，然后键入命令如下

Linux 系统，键入

./mypro ar1 ar2

可以看到结果是

./mypro

ar1

ar2

Windows 下，则键入

mypro ar1 ar2

可看到结果是

mypro

ar1

ar2

可见，用命令行方式，程序执行命令本身是第 0 个参数，后面跟的第 1，2。。个参量则可以用相应的 getarg 来获得。利用这个 getarg，你可以在外部输入命令时控制程序中的一些东西。

16.8 huge

第十七章 C&C++

17.1 Linux 下的开源编译器

C 与 C++ 用的是不同的编译器

C++ 要用 g++ hello.cpp -o hello

C 可以用 gcc hello.cpp -o hello

#define 和 const 与区别：const 进行数据类型检查，而 define 只是简单的替换

src 是 source 的缩写，这里是源文件的意思。“source”本身是“源”的意思。

17.2 数组的下标

同时用不同语言编程的话，很容易被各自特点搞晕。C&C++ 的数组下标索引默认是从 0 开始的，而 Fortran 默认是从 1 开始的。最要命的是 C&C++ 的数组下标越界还不报错。例如给两个数组赋值。

```
double G[3]={0.0};
double P[3]={0.0};
G[1] = 1.7835;
G[2] = 0.666;
G[3] = 0.3615;
P[1] = 16.94475;
P[2] = 40.78275;
P[3] = 6.07425;
```

这里数组越界了，但编译运行时不会报错。两个数组在内存中是连续储存的。在内存中其真实值为：

```
G[0] = 0.0;
G[1] = 1.7835;
G[2] = 0.666;
P[0] = 0.3615; // 本应赋值给G数组第3个数的，现在却赋值给P数组的第一个数了。
P[1] = 16.94475;
P[2] = 40.78275;
```

严格来说，C&C++ 语言中没有多维数组，通常所说的多维数组其实是数组的数组。谨记这一点，对理解和使用多维数组大有益处。

第十八章 C

18.1 static 关键字解析

http://blog.csdn.net/wu_zf/article/details/7068326

static 声明的变量在 C 语言中有两方面的特征：

- 变量会被放在程序的全局存储区中，这样可以在下一次调用的时候还可以保持原来的赋值。这一点是它与堆栈变量和堆变量的区别。
- 变量用 static 告知编译器，自己仅仅在变量的作用范围内可见。这一点是它与全局变量的区别。

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，即其占用的存储单元不释放，在下一次函数调用时，该变量已有值，就是上一次函数调用结束时的值。这是就应该指定该局部变量为“局部静态变量”，用 static 加以说明。

在 C 语言中，关键字 static 有三个明显的作用：

- 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。
- 在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。
- 在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。

static 全局变量与普通全局变量的区别：

在定义变量时，全局变量之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。两者在存储方式上并无不同。两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用。由于静态全局变量的作用域局限域于一个源文件内，只能为该源文件内的函数使用，因此可以避免其他源文件使用该变量。把普通全局变量改变为静态全局变量是改变了他的作用域，限制了他的使用范围。

static 局部变量和普通局部变量的区别：

普通局部变量所在的函数每次被调用都会被重新定义并分配存储空间，而 static 局部变量不会，他的值始终保存着。static 局部变量只被初始化一次，下一次使用时依旧是上一次的值。

static 函数与普通函数的区别：

static 函数 (即静态函数，在函数定义时加上了 static 关键字) 与普通函数作用域不同，他仅存在于文本文件中。只在当前源文件中使用的函数应该说明为内部函数 (即加上 static 关键字)。内部函数应该在当前源文件中声明和定义。对于可在当前源文件以外的

函数，应该在一个头文件中说明，要使用这个函数的源文件要包含这个头文件。另：程序的普通全局变量存在于堆栈中，全局变量、static 局部变量存在于静态存储区中。

18.2 文件操作

http://www.cnblogs.com/whiteyun/archive/2009/08/08/1541822.html

18.2.1 文件的基本概念

18.2.1.1 文件

所谓“文件”是指一组相关数据的有序集合。这个数据集有一个名称，叫做文件名。文件通常是驻留在外部介质 (如磁盘等) 上的，在使用时才调入内存中来。从不同的角度可对文件作不同的分类。

从用户的角度看，文件可分为普通文件和设备文件两种。

普通文件是指驻留在磁盘或其它外部介质上的一个有序数据集，可以是源文件、目标文件、可执行程序；也可以是一组待输入处理的原始数据，或者是一组输出的结果。对于源文件、目标文件、可执行程序可以称作程序文件，对输入输出数据可称作数据文件。

设备文件是指与主机相联的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也看作是一个文件来进行管理，把它们的输入、输出等同于对磁盘文件的读和写。通常把显示器定义为标准输出文件，一般情况下在屏幕上显示有关信息就是向标准输出文件输出。如前面经常使用的 printf，putchar 函数就是这类输出。键盘通常被指定标准的输入文件，从键盘上输入就意味着从标准输入文件上输入数据。scanf，getchar 函数就属于这类输入。

从文件编码的方式来看，文件可分为 ASCII 码文件和二进制码文件两种。

ASCII 文件也称为文本文件，这种文件在磁盘中存放时每个字符对应一个字节，用于存放对应的 ASCII 码。例如，数 5678 的存储形式为：

ASC 码：	00110101	00110110	00110111	00111000
	↓	↓	↓	↓
十进制码：	5	6	7	8

共占用 4 个字节。ASCII 码文件可在屏幕上按字符显示，例如源程序文件就是 ASCII 文件。由于是按字符显示，因此能读懂文件内容。

二进制文件是按二进制的编码方式来存放文件的。例如，数 5678 的存储形式为：00010110 00101110 只占二个字节。二进制文件虽然也可在屏幕上显示，但其内容无法读懂。C 系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。输入输出字符流的开始和结束只由程序控制而不受物理符号 (如回车符) 的控制。因此也把这种文件称作“流式文件”。在 C 语言中，文件操作都是由库函数来完成的。

18.2.1.2 文件指针

在C语言中用一个指针变量指向一个文件，这个指针称为文件指针。通过文件指针就可对它所指的文件进行各种操作。定义说明文件指针的一般形式为：FILE * 指针变量标识符；其中 FILE 应为大写，它实际上是由系统定义的一个结构，该结构中含有文件名、文件状态和文件当前位置等信息。在编写源程序时不必关心 FILE 结构的细节。

例如：FILE *fp；表示 fp 是指向 FILE 结构的指针变量，通过 fp 即可找存放某个文件信息的结构变量，然后按结构变量提供的信息找到该文件，实施对文件的操作。习惯上也笼统地把 fp 称为指向一个文件的指针。

18.2.1.3 文件的打开与关闭

文件在进行读写操作之前要先打开，使用完毕要关闭。所谓打开文件，实际上是建立文件的各种有关信息，并使文件指针指向该文件，以便进行其它操作。关闭文件则断开指针与文件之间的联系，也就禁止再对该文件进行操作。

18.2.2 fopen 函数

fopen 函数用来打开一个文件，其调用的一般形式为：

文件指针名 = fopen(文件名，使用文件方式)

其中，“文件指针名”必须是被说明为 FILE 类型的指针变量，“文件名”是被打开文件的文件名。“使用文件方式”是指文件的类型和操作要求。“文件名”是字符串常量或字符串数组。例如：

```
FILE *fp;
fp=fopen("a.pdb","r");
```

其意义是在当前目录下打开文件 a.pdb，只允许进行“读”操作，并使 fp 指向该文件。

第十九章 C++

19.1 文件包含处理 #include

19.1.1 文件包含的作用

所谓“文件包含”处理是指一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。C++ 提供了 #include 命令用来实现“文件包含”的操作。如在 file1.cpp 中有以下 #include 命令：#include file2.cpp，它的作用见下图。

“文件包含”命令是很有用的，它可以节省程序设计人员的重复劳动。

#include 命令的应用很广泛，绝大多数 C++ 程序中都包括 #include 命令。现在，库函数的开发者把这些信息写在一个文件中，用户只需将该文件“包含”进来即可（如调用数学函数的，应包含 cmath 文件—），这就大大简化了程序，写一行 #include 命令的作用相当于写几十行、几百行甚至更多行的内容。这种常用在文件头部的被包含的文件称为“标题文件”或“头部文件”。

头文件一般包含以下几类内容：

- 对类型的声明；
- 函数声明；
- 内置 (inline) 函数的定义；
- 宏定义，用 #define 定义的符号常量和用 const 声明的常变量；
- 全局变量定义；
- 外部变量声明，如 entern int a;
- 还可以根据需要包含其他头文件。

不同的头文件包括以上不同的信息，提供给程序设计者使用，这样，程序设计者不需自己重复书写这些信息，只需用一行 #include 命令就把这些信息包含到本文件了，大大地提高了编程效率。由于有了 #include 命令，就把不同的文件组合在一起，形成一个文件。因此说，头文件是源文件之间的接口。

19.1.2 include 命令的两种形式

在 #include 命令中，文件名除了可以用尖括号括起来以外，还可以用双撇号括起来。#include 命令的一般形式为：#include < 文件名 > 或 #include 文件名 。如：#include <iostream> 或 #include iostream都是合法的。二者的区别是：用尖括号时，系统到系统目录中寻找要包含的文件，如果找不到，编译系统就给出出错信息。

有时被包含的文件不一定在系统目录中，这时应该用双撇号形式，在双撇号中指出文件路径和文件名。如果在双撇号中没有给出绝对路径，如 #include file2.c则默认指用户当前目录中的文件。系统先在用户当前目录中寻找要包含的文件，若找不到，再按标准

方式查找。如果程序中要包含的是用户自己编写的文件，宜用双撇号形式。

对于系统提供的头文件，既可以用尖括号形式，也可以用双撇号形式，都能找到被包含的文件，但显然用尖括号形式更直截了当，效率更高。

19.2 关于 C++ 标准库

在 C++ 编译系统中，提供了许多系统函数和宏定义，而对函数的声明则分别存放在不同的头文件中。如果要调用某一个函数，就必须用 #include 命令将有关的头文件包含进来。C++ 的库除了保留 C 的大部分系统函数和宏定义外，还增加了预定义的模板和类。但是不同 C++ 库的内容不完全相同，由各 C++ 编译系统自行决定。不久前推出的 C++ 标准将库的建设也纳入标准，规范化了 C++ 标准库，以便使 C++ 程序能够在不同的 C++ 平台上工作，便于互相移植。新的 C++ 标准库中的头文件一般不再包括后缀.h，例如：#include <string>

但为了使大批已有的 C 程序能继续使用，许多 C++ 编译系统保留了 C 的头文件，即提供两种不同的头文件，由程序设计者选用。如：

#include <iostream.h> //C 形式的头文件

#include <iostream> //C++ 形式的头文件

效果基本上是一样的。建议尽量用符合 C++ 标准的形式，即在包含 C++ 头文件时一般不用后缀。如果用户自己编写头文件，可以用.h 为后缀。

第二十章 LabVIEW

20.1 注意事项

- 多使用快捷键；
- 尽量不要将输入和输出控件显示为图标形式，可以大大增大操作空间；
- 经常添加说明信息；
- 注意 VI 的图标和连线版处理；
- 即时帮助很重要，最好一直打开；
- 编程要避免功能混乱。

20.2 快捷键

- Ctrl+E：前后面板切换；
- Ctrl+S：保存；
- Ctrl+Z：撤销；
- Ctrl+R：运行；
- Ctrl+H：即时帮助，很重要；
- Ctrl+B：删掉悬空的线；
- Shift+ 右键：切换鼠标状态；
- 空格：快速切换鼠标状态

20.3 基本特性

20.3.1 属性节点

20.3.2 事件

20.3.3 子 Vi

- 连线状态可以查看子 Vi 端口名称。

20.4 用户界面设计

界面好坏的最基本指标：

- 是否完成了交互功能
- 是否可以简单直观地输入或获取信息
- 界面的美观度

Data Type	Scalar	1D Array	2D Array	Color
Numeric - Floating Point				Orange
Numeric - Integer				Blue
Boolean				Green
String				Pink

使用 Labview 开发一个项目的步骤：

- 收集需求；
- 设计；
- 编码；
- 测试；
- 发布及维护

建议：先做用户界面设计，在做程序设计

理由：先设计程序结构，再设计界面，难免会朝着最可能简化编码工作方向去做。但是这样的界面往往不是最方便用户使用的界面。

另一方面，使用比较老的文本语言编程，设计用户界面时通常在草稿纸上画出原型。LabVIEW 有独特的优势，可视化编程做的非常方便。有大量现成的控件，控件属性更改非常方便。因此，用户可以通过拖拽的方式，直接用 LabVIEW 来设计界面原型。

好的用户界面共有的特点：

- 一致性
- 使用恰当的数据类型和控件类型；
- 控件的分类排布合理、简洁

一致性：让用户迅速接受并且方便的操作一个程序界面，最关键的一点就是让这个界面保存高度的一致性。

一致性包含以下多个方面的一致：

- 程序内部的一致性；
  由于应用领域、面向的客户群体的不同，不同的软件可以有自己独特的风格。不论一个程序采用哪种风格，它内部不同界面，同一面板上的不同控件等，它们的风格应当保持一致。一个软件采用统一的风格，才会让用户有一种协调的感觉。
- 与约定俗成的习惯保持一致；
  很多设计或操作方法，已经被大家广为接受。它们也许不见得美观或优化，但是一旦习惯养成了，就很难被改变了。
- 与真实事物保持一致；
  有很多程序是对现实世界的模拟或模仿，这样的程序若希望便于被用户接受，最好是尽量与现实世界保持一致。LabVIEW 编写的程序大多是测量、控制等有关，在这些领域，原本也存在着一些相关的仪器或设备。因此软件的界面可以借鉴这些仪器的外观。
- 建立并遵循界面规范。
  使界面保持一致性的最好办法就是在设计开发时遵循一定的规范。这个规范可以由公司内部定义，也可以遵循现有的行业规范。对于开发 Windows 系统风格的程序，可以遵循微软定义的界面规范。对于一般的 LabVIEW 程序，可以遵循 LabVIEW 程序开发规范。

界面元素的关联：

当一个界面上的元素比较多，找到自己想要的信息就要花上一小点时间。用户常常是一眼就看到了一个与自己想要的信息有一点关联的某个元素，他这时候会期望这个元素就有一定的提示信息，帮他加速找到自己想要的东西。因此，我们要在界面上，告诉用户哪些元素是相关的，或不相关的。

有很多手段可以把界面的元素之间的关联现实给用户，比如通过元素的排布、边框、空白、颜色、字体等等方式。我们总是在相关内容的附近去找想要的信息，所以逻辑上相关的控件或项目，应当在屏幕空间上相对临近。单纯的把条目排在一起还是不利于用户查看。可以把它们按功能分成几个不同的区域，比如保存文件与 Project 的操作在功能上相对独立一些，就可以用分隔线，帮它们的项目划分开。对于面板上的控件，功能相关的几个控件可以通过被边框围住、使用分割线、采用不同的间隙等等方法，让用户直观的感觉到他们在功能上的紧密关联。

帮助和反馈信息：

用户界面要照顾到那些不熟悉它的用户。为了方便用户了解界面的使用方法，需要给用户提供足够的帮助信息。对于 LabVIEW，给用户提供提示主要通过以下几个手段：用户手册、在线帮助窗口、提示条、利用控件的标题、选项文字、直接把帮助文字写在界面上。不论何时，都应该尽量使用有意义的控件名称。

用户界面设计的限制：

目的：保障程序的可靠性

手段：限制用户的输入数据和操作

- 限制数据的输入；

- 防止用户的误操作。

20.5 调用 DLL 动态链接库

参考：阮奇桢《我和 LabVIEW——一个 NI 工程师的十年编程经验》，北京航空航天大学出版社。感觉他写的这部分内容前后背景交代的很清楚，看起来舒服。

常用的计算机语言各有其特色，也有其不足。LabVIEW 虽然功能强大，但编程时也时常会发现它缺少某些所需的功能，而这些功能也许恰恰已经有其他编程语言的 DLL、ActiveX 组件等提供。在这种情况下，最高效的方法莫不如让 LabVIEW 调用这些模块，直接利用其他语言或者操作系统已经实现的功能。

20.5.1 DLL

Dynamcis Linkable Library，DLL，动态链接库。从字母上理解，它是一种“程序库”。库内存放的是可供应用程序使用的函数、变量等。

“动态”是与“静态”相对应而来的。这里的动态和静态是指链接库中代码与使用它们的应用程序之间的链接方式。如果采用静态链接库，在生成应用程序时，库中的函数等都会被直接放入最终生成的可执行文件中；而使用动态链接库时，库中的函数等不会放到可执行文件中去，而是仍然保留在 DLL 文件内。当程序被运行时，再链接到动态链接库中的函数和变量等内容。

静态库的局限性比较大，C 语言编写的静态库只能在 C 语言中使用，LabVIEW 无法调用。而动态链接库则可以在多种编程语言中通用，即使用某一种语言编写出来的 DLL 可以在另一种语言编写的程序中使用。例如，使用 C 语言编写的 DLL 可以在 LabVIEW 中使用，反之亦可。

动态链接库的加载方式又分“动态”与“静态”两种。这里的动态和静态是指应用程序运行时，动态链接库代码被载入内存的方式。常用的方式是静态加载，指动态链接库在应用程序启动时随应用程序一起被载入内存。而动态加载方式是指，应用程序启动时并不载入动态链接库，只有在使用到动态链接库中某个函数时，才把动态链接库载入内存。

DLL 最大的优势在于代表共享，只要某一功能以 DLL 的形式提供出来了，其他的应用程序就可以直接使用这一功能，而不必再实现一份相同的代码了。DLL 的使用非常普遍，比如 Windows 操作系统提供给应用程序调用的功能，就是以 DLL 的形式公布出来的。LabVIEW 中若需使用某个系统功能，如读/写注册表等，即可通过调用 Windows 提供的 DLL 函数来完成。Windows 提供的这些完成系统功能的函数也被称为 Windows API（Application Programming Interface）。在常用的几个系统 DLL 中，kernel32.dll 提供了内存管理和进程调度相关的函数，user32.dll 中的函数则主要用于控制用户界面，gdi32.dll 中的函数则负责图形方法的操作。在 32 位操作系统中，这些 Windows API 的 DLL 都被保存在 system32 目录下。很多硬件设备的驱动程序也往往是以 DLL 方式提供的。此外，在互联网上还可以找到各种各样的 DLL。如果需要解析某种文件、使用某些常用的算法等，

都可以先去互联网上搜索一下，是否有相关的 DLL 库可供直接使用。

在 LabVIEW 中，经常会遇到需要使用 DLL 的情况，比如在程序中使用到某个以 DLL 方式提供的第三方驱动程序或算法；再比如，在一个大项目的开发中，出于效率和开发人员喜好等因素的考虑，可能使用 C++ 语言实现软件的运算部分，并把这些功能构建在 DLL 文件中，再使用 LabVIEW 编写程序的界面部分，并通过调用编写好的 DLL 来调用运算部分的功能。

20.5.2 CLN 和 CIN 节点

在 LabVIEW 中，通过“互连接口 → 库与可执行程序 → 调用库函数”节点来调用 DLL 中的函数。调用库函数节点常简称为 CLN 节点，是英文 Call Library Function Node 的缩写。在同一函数选板上，它旁边的一个节点是“代码接口”（Code Interface Node），简称 CIN 节点。在 CLN 节点出现以前，LabVIEW 只能通过 CIN 节点调用 C 语言编写的函数。现在有了 CLN，可以不再考虑使用 CIN 了。CIN 节点不能调用动态链接库中的函数，只能调用按照特定方式编译出来的程序代码。稍有差错，程序就无法正常运行。CIN 所调用的程序模块不通用，而且限制颇多，CLN 节点出现之后，就很少有人再使用 CIN 节点了。

20.5.3 LabVIEW 调用 DLL 中的函数以及手动配置 CLN 节点

这部分在参考书中写的非常详细。这里简单说下。一个 CLN 节点被拖到程序框图上后，需要进行对其配置才能使用。双击 CLN 节点，则弹出其配置对话框，该对话框有 4 个选项卡，如下图所示

- 第一个选项卡是被调用函数的信息，主要有
  - “库名或路径”列表框。用于填写 DLL 文件名和 DLL 的全路径。在系统路径下的 DLL，直接输入文件名即可，否则需要全路径。
  - “在程序框图中指定路径”选项。若没有勾选，则是 LabVIEW 静态调用了这个 DLL；若勾选，则是动态调用。具体说明参见参考书。
  - “函数名”列表框。用于输入需要调用的 DLL 中的函数。如果选用静态加载方式，并已输入了正确的 DLL 文件全路径，这里会列出 DLL 中所有允许被外部调用的函数，用户只要在下拉列表框中选取一个即可。
  - “线程”选项。用于选择被调用的 DLL 函数在何线程内运行。CLN 节点的线程选项只有两项：“在 UI 线程中运行”和“在任一线程中运行”。在程序框图上直接就可以看出一个 CLN 节点选用的是什么线程。如果是“在 UI 线程中运行”，则节点颜色是比较深的桔黄色；如果是“在任一线程中运行”，则节点是比较淡的黄色。详细说明见参考书。
  - “调用规范”。用于指明被调用函数的参数压栈规范。Windows API 一般使用 stdcall，标准 C 的库函数大多使用 C call。
- 第二个选项卡是配置参数选项卡。使用 CLN 节点，最困难部分就是把函数参数的数据类型映射为相应的 LaVIEW 中的数据类型。这也是最关键的。

- 第三个选项卡是设置回掉函数选项卡。可以使用这些回调函数在特定的情形下完成初始化、清理资源等工作。如果为 Reserve 选择了一个回调函数，那么当一个新的线程开始调用这个 DLL 时，这个回调函数首先被调用。可以利用这个函数为新线程使用到的数据做初始化工作。如果一个线程使用了这个 DLL，在线程结束时，它会去调用 Unreserve 中指定的回调函数。Abort 中指定的函数用在 VI 非正常结束时被调用。比如按 Abort 按钮让一个 VI 停止，而不是让它运行完。这里的几个回调函数必须要由 DLL 的开发者按照特定的格式实现。它的原型就是 Prototype for these procedures 中列出的那个。如果你使用的 DLL 不是专为 LabVIEW 设计的，一般不会包含这样的回调函数。
- 第四个是错误处理方式选项卡。这上面说明写得已经很详细了。

20.5.4 把 DLL 中的函数包装成子 VI

在 LabVIEW 中调用 DLL 的函数，最大的困难在于把函数参数的数据类型映射为相应的 LabVIEW 中的数据类型。在手工设置 CLN 节点前，可以优先考虑使用导入共享库工具，以自动生成配置 CLN 节点。这个工具在“工具 → 导入 → 共享库”菜单项中，专门用于把 DLL 中的函数包装成 VI，生成的每个 VI 中最主要的部分就是一个 CLN 节点；它能够自动设置函数的参数。有了这个工具，用户就不再需要费脑筋去考虑数据类型匹配的问题了。

这个工具可能无法直接处理一些非常特殊的数据类型（如字符串数组）和函数（如

回调函数），但在大多数情况下，都能够把 DLL 中的函数包装成可以正确运行的 VI。如果有现成的 DLL 打算在 LabVIEW 中使用，首先应该考虑用这个工具，把 DLL 中所有的函数都包装成 VI，再使用起来，就方便多了。

如果 DLL 是 C++ 编写的，并且使用了 C++ 的类作为接口，这样的 DLL 是没办法在 LabVIEW 中直接调用的。CLN 节点只能调用符合标准 C 语言函数接口的 DLL。若项目中必须使用某个 C++ DLL，则可以在其上再用 C 语言写一个 C 接口 DLL，作为它和 LabVIEW 之间的中间层。LabVIEW 调用这个中间层 DLL 提供的函数，中间层函数再调用 C++ 接口的 DLL 函数。

20.5.4.1 应用举例

转载:http://bbs.ickey.cn/community/forum.php?mod=viewthread&tid=22795&extra%3D

Labview 是一个图形化软件库，不但支持 NI 官网硬件，还支持众多第三方硬件，这些硬件提供的驱动基本上是 DLL 的方式，需要在 labview 里面调用。其实，labview 也可以支持自制 MCU 硬件。只要能提供相应的 DLL 就行，甚至，如果能在 labview 和 IDE 开发环境之间建立接口，一个 labview 子 VI，就可以实现在 labview 运行状态下，配合 MCU 仿真器，实现图形化程序到 MCU 的下载，脱离了 C 代码的编写，开发更加快速很方便！

今天为大家分享一个深度调用 DLL 的方法，为什么是深度调用 DLL。因为，调用的不是 DLL 本身。而是由 DLL 生成的子 VI 函数。下面重点分享如何将 DLL 生成可以被 labveiw 调用的子 VI。

一、要会编写 DLL

动态链接库英文为 DLL，是 Dynamic Link Library 的缩写形式，DLL 是一个包含可由多个程序同时使用的代码和数据的库。编写 DLL 文件需要有 VC++ 功底。比如平时经常开发 MCU 软件，积累了很多 C 驱动代码，如果能够按照规范，就可以生称 DLL 文件。

二、要有相应的 DLL.H 文件

编写了 DLL 文件外，还要有 H 文件才行。通常开发数据采集卡的厂家只会提供 DLL 文件本身，而不会提供编写 DLL 文件用的 C 代码和 H 文件。所以，那些数据采集卡只能采用直接调用 DLL 的方式。而自制硬件中，C 代码和 H 文件是自己编写，所以，可以很容易实现 DLL 转 labview 的子 VI。

三、应用实例讲解:

因为一个 MCU 内容太多，短时间内实现也不现实。就采用一个开源厂家的例子来实现。CH375 是国内厂家江苏沁恒（www.wch.cn）生产的一个 USB 芯片，支持 HOST 主机方式和 SLAVE 设备方式。速度可以实现全速方式，在一些总线设备和 USB 数据采集设备上采用很多。厂家开源了一 CH375DLL.DLL 和一个 CH375DLL.H 文件。下面分享一下过程:

1、打开 labview 界面，建立一个空 VI。在【工具】选项中，选择导入—共享库（dll），如图：

2、在出现的界面中，为共享库创建新 Vi，进入下一步：

3、选择共享库及头文件，这里把 CH375DLL.DLL 和 CH375DLL.H 文件载入：

4、配置包路径和宏定义命令，这里空着不填，进入下一步：

5、全部勾选 DLL 库里面的函数定义文件，下一步：

6、配置好生成的 VI 库的路径和名称：

7、选择错误处理方式，这里有多种方式，可以选择简易错误处理：

8、配置 VI 和控件，这里和 DLL 一样设置如图：

9、最后，是一个生成总结：

10、然后，点下一步就可以了：

11、从这里调入 vi，看到没，调用后，直接进行连线就可以了，不像原来那样看着眼花缭乱了。

20.5.4.2 调用 DLL 时遇见的问题

在使用 DLL 时，最大的困难就是把函数参数的数据类型映射为相应的 LabVIEW 中的数据类型。LaBVIEW 提示：

未定义符号可能会造成函数和参数无法被识别。如要解决该问题，检查头文件并确定是否必须添加预定义符号。单击上一步按钮返回至向导的前一页并添加预定义符号（例如，“NIAPI_stdcall _stdcall” 或 “NIAPIDEfined = 1’’）

归咎原因就是头文件中哦你搞得一些类型定义不符合标准 C 语法，而使解析器无法获得正确的定义。DLL 函数的头文件中可能使用了某个系统定义的数据类型，数据类型的定义在 windows.h 中（windows.h 是 Windows SDK 的一个文件，VC 等开发环境中常常带有 Windows SDK），要正确解析必须得到这些数据类型，也就是找到 windows.h 这个头文件，用户须把 windows.h 文件的全路径加在“包括路径”中。例如 Visual C++ 6.0 编译环境中头文件位于安装目录下 VC98 文件夹下的 Include 文件中。

而“预处理定义”中，当用户需要写一些宏定义，那么就写在这个位置，如添加如

下代码:

ULONG = unsigned long; VOID = void; LONG = long; UCHAR = unsigned char; PUCHAR = unsigned char*; PULONG = unsigned long*; WINAPI; BOOL=bool; USHORT = unsigned short; PUSHORT = unsigned short*;

这样就不会出现问题了。

20.6 NI-DAQmx 数据采集

参考书：

- 《LabVIEW 大学实用教程（第三版）》Jeffrey Travis，Jim Kring 著，乔瑞萍等译。LabVIEW for Everyone (Graphical Programming Made Easy and Fun, Third Edition) 第二章、第十章和第十一章。请详细阅读相关章节。这里只记录比较重要的部分。
- 《LabVIEW 实践教程》Robert H. Bishop，National Instruments 著，乔瑞萍林欣等译

20.6.1 NI-DAQmx 简介与安装

（1）简介

DAQ，Data AcQuisition，是一个术语，即数据采集，是实现测量现实世界信号如电压，并把这些信息发送到计算机用于处理、分析、储存或其他数据操作的过程。而 NI-DAQmx 则是 NI 公司的跨平台 DAQ 设备驱动程序，包含了所有 NI 公司 DAQ 设备的驱动。NI-DAQmx 代替了传统的 NI-DAQ（以前称为 “NI-DAQ”），并且在传统的 NI-DAQ 基础上提供了诸多改进，如：改进了状态模型、多线程驱动程序、意外情况下的健壮性、简化的同步、降低了 LabVIEW 框图的混乱、从简单到高级程序的平滑过渡。NI-DAQmx 优于先前 NI-DAQ 的显著特定就是包含了 DAQ Assistant 以配置通道和完成测量任务。

信号的采集需要对信号有所了解，比如信号类型、一个信号的五种测量角度常见的转换器和信号调节等，这里不详细讲述，参考书中相关章节有叙述。这里需要注意的是，Nyquist-Shannnon 采样定理，采样频率必须大于被采集信号最高频率的两倍。但是为了充分保持信号的形状，采样频率通常至少为信号最高频率的 5～10 倍。

（2）安装

NI-DAQmx 是与 LabVIEW 单独安装的，它们版本之间的兼容性如下图所示，可以在官网查询。安装过程比较简单，跟通常软件安装过程一样，需要注意的是安装过程比较长。

20.6.2 MAX 及创建 NI-DAQmx 虚拟设备

MAX，Measurement & Automation Explorer，是在 NI-DAQmx 和 LabVIEW 之间的实用软件，目前只支持 Windows 平台。它是一个 Windows 的软件接口，用于配置和测试硬件，可以访问 NI 公司的所有设备（包括 DAQ，GPIB，VXI 等）。换句话说，就是一个用于建立各种卡及通道配置参数的配置工具。Windows 设备管理器管理着用户系统中的所有硬件，其中包括 NI 公司的 DAQ 设备。MAX 会读取设备管理器在 Windows 注册表中记录的信息，并为用户系统中的每个 NI DAQ 卡分配逻辑设备号。用户可以通过该设备号访问 LabVIEW 中的卡。

用户可以通过访问计算机中哦你搞得设备管理器找到数据采集设备，如 Data Acquisition Devices 中列出了安装在计算机上的所有 DAQ 卡。选取一个 DAQ 卡，选择 Properties 或双击该 DAQ 卡，可以看到一个多页面的对话框。General 显示了与卡相关的整体信息。

图 20.1: NI-DAQmx 与 LabVIEW 的版本兼容性

可以通过 Resources 指定该卡占用的系统资源，比如中断级、DMA 以及软件可配置卡的基址。Driver 指明了该 DAQ 卡驱动程序的版本及位置。

用户可以在 NI-DAQmx7.5 及新版本中利用 MAX 创建 NI-DAQmx 虚拟设备，这样用户可以建立原型系统，在进行实际数据采集前，测试数据采集应用程序。当需要实际采集时，还可以使用 MAX 可移植配置向导，将 NI-DAQmx 虚拟设备的配置输入导入到物理设备中。创建 NI-DAQmx 虚拟步骤如下：

1. 打开 MAX（Measurement & Automation），可以点击桌面图标打开，也可以从 LabVIEW 打开。
2. 右击“设备和接口（Devices and Interfaces）”弹出快捷菜单并选择“新建（Create New）”。
3. 一个对话框会提示用户选择要添加的设备。选择“仿真 NI-DAQmx 设备或模块化仪器（NI-DAQmx Simulated Device）”并点击“完成（Finish）”。
4. 在“选择设备（Choose Device）”对话框中选择需要的虚拟设备，比如 M series DAQ。
5. 选择 NI PCI-6251 并单击 OK 按钮。

20.6.3 DAQ 助手

DAQ Assistant，DAQ 助手，是一个用来配置测量任务及通道的图形界面接口。DAQ 助手位于 Functions>Measurement I/O>DAQmx-Data Acquisition 选项板中。将 DAQ 助手拖

入框图中就可以将其弹出。当 DAQ 助手置入框图中时，DAQ 助手对话框将会自动出现。一旦打开 DAQ 助手，配置 NI-DAQmx 任务所需步骤与使用 MAX 配置 NI-DAQmx 任务的步骤基本一致。使用 DAQ 助手构建数据采集 VI 的通用过程如下：

- 打开一个新的 VI。
- 在框图中拖入 DAQ 助手。
- 出现 DAQ 助手对话框帮助用户配置测量任务。
- 配置、命名及测试 NI-DAQmx 任务。
- 单击 OK 按钮以返回框图。
- 编辑前面板和框图完成 VI。

如果使用 DAQ 助手配置了 DAQmx 任务，则该任务就是一个本地任务，因此，它不能保存到 MAX 中被其他应用使用。如果想要使任务对于其他应用也可用，可是通过 DAQ 助手生成 NI-DAQmx Task Name 控件使其保存到 MAX 中并在其他应用中使用。DAQmx Task Name Constant 是为了与 DAQ 卡进行通信而在 DAQ VI 中使用的一种 LabVIEW 数据类型。

- 通过右击框图中的 DAQ 助手弹出快捷菜单并在下拉菜单中选择 Convert to NI-DAQmx Task；
- 此时会出现 DAQ 助手以便重新配置任务；
- 退出 DAQ 助手后，则完成转换，该 DAQmx 任务就存在于 MAX 中并且可以被其他应用使用。

如果想要在应用中使用来自 MAX 的 DAQmx 任务，则按如下步骤：

- 首先在 Functions>Measurement I/O>NI DAQmx-data Acquisition 选项板中找到 DAQmx Task Name Constant ，并拖入框图中。
- 任务名称可以通过鼠标手形工具单击 DAQmx Task Name Constant 浏览选择。

20.6.4 NI-DAQmx VI

如果说 DAQ 助手是数据采集的图形化配置界面，那么 NI-DAQmx VI 则是 DAQ 助手配置的底层 VI。在涉及与其他操作联动时，采用 NI-DAQmx VI 的形式更为可靠。NI-DAQmx VI 是一种被称为多态 VI 的特殊 VI，其结果是能够适应不同 DAQ 功能的一组核心 VI，比如模拟输入、模拟输出、数字 I/O 等等。在 LabVIEW 后面板上，右键打开函数（Functions）选项板，单击“测量 I/O（Measure I/O）>DAQmx 数据采集（Measurement I/O>DAQmx-data Acquisition）”访问该选项板。

Windows 版的 LabVIEW DAQ VI 使用的是 Windows 版 32 位动态链接库（DLL）形式的 NI 公司标准 NI-DAQ。LabVIEW 安装程序将 NI-DAQ DLL 安装在 Windows\System32 目录下。nidaq32.dll 文件是用户使用的 DAQ 卡的高级接口，安装在 Windows\System32 目录下。随后，nidaq32.dll 文件将与 Windows 注册表对接以获取由 MAX 定义的配置参数。

NI-DAQmx VI 使用非常简单，主要有如下几个步骤：

- 创建一个任务（或引用一个 MAX DAQmx 任务）。

- 启动任务。
- 读或写，以及根据需要重复。
- 停止任务。
- 清除任务。

需要注意的是：

- 用户不一定必须启动任务，通常读或写操作会自动启动任务。
- 在清除任务之前不必停止任务。如果任务正在运行，清楚操作将先停止任务。

20.7 EMCCD

20.7.1 CDD 类型简介

CCD, Charge-Coupled Device, 电荷耦合器。[1] 简单来说，在一个用于感光的 CCD 中，有一个光敏区域（硅的外延层），和一个由移位寄存器制成的传感区域（狭义上的 CCD）。图像通过透镜投影在一列电容上（光敏区域），导致每一个电容都积累一定的电荷，而电荷的数量则正比于该处的入射光强。用于线扫描相机的一维电容阵列，每次可以扫描一单层的电容；而用于摄像机和一般相机的二维电容阵列，则可以扫描投射在焦平面上的图像。一旦电容阵列曝光，一个控制回路将会使每个电容把自己的电荷传给相邻的下一个电容（传感区域）。而阵列中最后一个电容里的电荷，则将传给一个电荷放大器，并被转化为电压信号。通过重复这个过程，控制回路可以把整个阵列中的电荷转化为一系列的电压信号。在数字电路中，会将这些信号采样、数字化，通常会存储起来；而在模拟电路中，会将它们处理成一个连续的模拟信号（例如把电荷放大器的输出信号输给一个低通滤波器）。

Frame transfer CCD，帧转移 CCD。A frame transfer CCD is a specialized CCD, often used in astronomy and some professional video cameras, designed for high exposure efficiency and correctness.

The normal functioning of a CCD, astronomical or otherwise, can be divided into two phases: exposure and readout. During the first phase, the CCD passively collects incoming photons, storing electrons in its cells. After the exposure time is passed, the cells are read out one line at a time. During the readout phase, cells are shifted down the entire area of the CCD. While they are shifted, they continue to collect light. Thus, if the shifting is not fast enough, errors can result from light that falls on a cell holding charge during the transfer. These errors are referred to as "vertical smear" and cause a strong light source to create a vertical line above and below its exact location. In addition, the CCD cannot be used to collect light while it is being read out. Unfortunately, a faster shifting requires a faster readout, and a faster readout can introduce errors in the cell charge measurement, leading to a higher noise level.

A frame transfer CCD solves both problems: it has a shielded, not light sensitive, area

[1] https://en.wikipedia.org/wiki/Charge-coupled_device

containing as many cells as the area exposed to light. Typically, this area is covered by a reflective material such as aluminium. When the exposure time is up, the cells are transferred very rapidly to the hidden area. Here, safe from any incoming light, cells can be read out at any speed one deems necessary to correctly measure the cells' charge. At the same time, the exposed part of the CCD is collecting light again, so no delay occurs between successive exposures.

ICCD, Intensified Charge-Coupled Device, 增强电子耦合器。

An intensified charge-coupled device (ICCD) is a CCD that is optically connected to an image intensifier that is mounted in front of the CCD.

An image intensifier includes three functional elements: a photocathode, a micro-channel plate (MCP) and a phosphor screen. These three elements are mounted one close behind the other in the mentioned sequence. The photons which are coming from the light source fall onto the photocathode, thereby generating photoelectrons. The photoelectrons are accelerated towards the MCP by an electrical control voltage, applied between photocathode and MCP. The electrons are multiplied inside of the MCP and thereafter accelerated towards the phosphor screen. The phosphor screen finally converts the multiplied electrons back to photons which are guided to the CCD by a fiber optic or a lens.

An image intensifier inherently includes a shutter functionality: If the control voltage between the photocathode and the MCP is reversed, the emitted photoelectrons are not accelerated towards the MCP but return to the photocathode. Thus, no electrons are multiplied and emitted by the MCP, no electrons are going to the phosphor screen and no light is emitted from the image intensifier. In this case no light falls onto the CCD, which means that the shutter is closed. The process of reversing the control voltage at the photocathode is called gating and therefore ICCDs are also called gateable CCD cameras.

Besides the extremely high sensitivity of ICCD cameras, which enable single photon detection, the gateability is one of the major advantages of the ICCD over the EMCCD cameras. The highest performing ICCD cameras enable shutter times as short as 200 picoseconds.

ICCD cameras are in general somewhat higher in price than EMCCD cameras because they need the expensive image intensifier. On the other hand, EMCCD cameras need a cooling system to cool the EMCCD chip down to temperatures around 170 K. This cooling system adds additional costs to the EMCCD camera and often yields heavy condensation problems in the application.

ICCDs are used in night vision devices and in various scientific applications.

EMCCD，Electronic Multiplying Charge Coupled Device，电子倍增耦合器，是一种增强型帧转移 CDD。但它的输出信号质量远高于普通 CCD 且优于其他增强型 CCD，提供出色信噪比的 CCD。[2]EMCCD 是一块基于硅的半导体芯片，具有二维矩阵的光电传感器或像素。EMCCD 技术，有时也被称作“片上增益”技术，是一种微弱光信号增强探测技术。而“片上增益”这一功用的实现就是在移位寄存器的后面加了增益寄存器。该寄

[2]参考文献：[1] 韩露, 熊平. EMCCD 工作原理及性能分析 [J]. 传感器世界,2009,(05):24-28.

存器可以使信号在噪声加入前就被倍增到一定的程度，使放大器的读出噪声并不能影响信号的质量，或者说减小对信号的影响。

20.7.2 EMCCD 和 ICCD 的异同

参考：www.lustervision.com/new-product/emccdxj20150528.shtml

EMCCD 也就是电子倍增 CCD，它和 ICCD 被成为业内最为灵敏的两种 CCD。现在罐子探测领域的高速发展对探测器灵敏成都的要求越发的苛刻，而 EMCCD 技术对于这种越来越苛刻的要求做出了完美的答复。所以 EMCCD 相机也应用领域中较为常见。那么 EMCCD 相机的优点有什么? 我们可以通过比较 ICCD 和 EMCCD 来看。

ICCD 的放大原理是让光先通过光电阴极激发出电子，电子再进入微通道版进行放大，放大后的电信号通过轰击荧光屏激发出荧光，最后再让荧光通过光纤耦湖综合透镜，最终让荧光合到普通 CCD 靶面上成像。ICCD 拥有两个特点：一是想要实现纳秒量级别的快门控制时间，只需要在光电阴极上加脉冲电压，这样就可以做超快时间分辨探测; 二是信号能力因为电子轰击增益而加强。缺点也是非常明显的，在使用时空间分辨率会有一定损失。

EMCCD 技术就是在普通的 CCD 读出寄存器后面增加一个增益寄存器，从而将电子信号进行放大。其中的原理就是利用电子在转移过程中产生的“撞击离子化”效应，从而产生新的电子。暗电流通过制冷后可以起到抑制噪声的作用，这样信号增益提取就可以有效地进行了。EMCCD 相对于 ICCD 的好处就是控件分辨率很好，成像很快，不过缺点就是时间分辨率不如 ICCD。

EMCCD 和 ICCD 的比较：

- EMCCD 不会因为增强器中有几百上千伏的高压起到的高增益环境下，引入的强信号损毁增强器，ICCD 技术就有可能发生这种情况。
- EMCCD 拥有毫秒级别的时间分辨;ICCD 技术可以精确到纳秒级。
- EMCCD 采用的 CCD 芯片可以将背照式峰值量子效率提高到 90%; 而 ICCD 通过的光电转换通过光电阴极实现，最多只能将峰值量子效率提高到 50%，甚至不足。
- EMCCD 的像增强器并没有被出口管制; 而 ICCD 因为成本高、价格高，被限制出口。
- EMCCD 的空间分辨率只受像素大小影响，分辨率比 ICCD 高，适合生命科学领域。总的来说，通过比较 ICCD 和 EMCCD 技术特点，EMCCD 相机的有点还是比较多的，因此 EMCCD 技术也能针对应用领域内越来越高的要求，而给出一个个完美的答复。

20.8 PI-MAX3 型 CCD LabVIEW 参数设置学习

快速测量的示例 vi 都在 `ExamplesXX_ver\Fast Examples` 目录下找到。需要注意的是安装后的目录下的 vi 文件或许有些缺失，最好直接到安装光盘 Examples 目录下寻找。示例 vi 都是以 Ex 开头的。外触发快速测量 CCD 参数初始化 vi，由 ExOpenGlobalcam.vi、

ExOpenGlobalPulser.vi、ExSetupROI.vi、ExSetupExtSuperSyncro.vi 组成。这些都是软件包提供的快速测量示例 vi，需要注意的是这些示例 vi 中的子 vi 有重复，具体使用时需要做些删减。示例 vi 并不是最底层的 vi，是可以查看后面板，供用户学习使用的。而底层 vi，后面板有密码保护，无法查看，但有详细的说明文档。接下来对各示例 vi 进行详细说明。参数设置要注意结合 EMCCD 的用户手册、LightField 软件用户手册（或 WinSpec 软件用户手册）一起研究。

（1）ExOpenGlobalcam.vi、ExOpenGlobalPulser.vi: 打开 CCD，创建全局变量，是快速测量的关键。

快速测量是通过全局句柄（Global Handles）实现的。必须首先运行 ExOpenGlobalcam.vi，再运行 ExOpenGlobalPulser.vi，这样就完成了全局句柄的创建。在整个 LabVIEW 程序运行过程中，只要不关闭 Camera，这两个子 vi 只需运行一次。这两个 vi 很简单，无需做任何修改，直接使用即可。

ExOpenGlobalcam.vi 由多个底层 vi 构成。

- 第一个 vi: CamHandle.vi。全局变量。
- 第二个 vi: InitToolkit.vi。SITK 软件包初始化。
- 第三个 vi: CameraOpen.vi。设置 Camera 编号以及初始默认参数。
- 第四个 vi: ToolKitIsError.vi。检查软件包是否有误。

ExOpenGlobalPulser.vi 由多个底层 vi 构成。

- 第一个 vi: PulserHandle.vi。全局变量。
- 第二个 vi: PulserOpen.vi。
- 第三个 vi: ToolKitIsError.vi。检查软件包是否有误。

（2）ExSetupROI.vi: ROI, Region Of Interest。CCD 曝光参数设置，由多个示例 vi 和底层子 vi 构成。

- 第一个子 vi 是底层 vi: CameraSkips.vi。This Vi will allow the setting of the skip parameters in the camera. This will allow a number of pixels to be “passed over" and allows for a quicker read of the camera.

  参数： Minimum Block Size:2; Number of Minimum blocks:5。

  参数说明: 这部分参数取值比较复杂，建议使用厂家默认值。在 WinSpec 用户手册，Vertical Skips 部分有详细说。在 LightField 用户手册，Appendix D:Reference Topics>Cleaning and Skipping Algorithm 部分有更详细的说明。

  - Minimum Block Size: The number of lines to group on the CCD shift register before discarding (for lines that are to be skipped). Sets the size, in rows, of the skip blocks that immediately precede the data. The default value will generally give good results.
  - Number of Minimum blocks: The number of minimum block sizes to do before active data, after which the grouping increases geometrically. Sets the number of binned "skip" blocks preceding and following the region of interest. The default value will generally give good results.

  WinSpec 设置: Hardware Setup>Clean/Skips>Load Default Values

LightField 设置: Experiment Settings>Sensor>Sensor Cleaning

总之，这部分使用默认值。

- 第二个子 vi 是示例 vi :ExSub1strip.vi。它只由底层子 vi，CameraROI.vi 构成。是用来设置使用单个 CCD 像素区域。还有示例 vi，ExSub2strip.vi，用来设置两个 CCD 像素区域，我们不需要使用。

  参数：x1:1; x2:1600; y1:1; y2:200; x bin:1; y bin: 200。

  参数说明：我们的 CCD 像素面积为 1600 × 200。

图 20.2: 坐标轴定义。来自 WinSpec User's Manual

  - x1 表示 x 轴选用的起始像素位置，x2 表示 x 轴选用终止像素位置。y1，y2 同理，表示 y 轴方向的像素选取。
  - x bin(binning)，表示 x 轴每多少像素求和，y bin 同理。为了增强信号，我们将 y 轴信号全部加起来，即 y bin 设置为 200。为了不降低分辨率，并没有对 x 轴信号进行部分求和，即 x bin 设置为 1。

  WinSpec 设置: Acquisition>Experiment Setup

  - Select Mode: Imaging Mode or Spectroscopy Mode
  - Clicking on **Full** loads the full size of the chip into the edit boxes.

  LightField 设置: Experiment Settings>Regions of Interest

- 第三个子 vi 是示例 vi: ExSubSetup.vi。This Sub VI sets up the skips, cleans, and shutter parameters. This should be changed to suit the camera you are using. 由多个底层 vi 构成。
  - 第一个底层 vi：CameraCleans.vi。用来清除 CCD 芯片电荷的。

    默认参数[3]：Number of Clears:1 ; Number of Strips Per Clear:50; Continous Clears:0。

    参数说明：这部分参数说明要结合 EMCDD 说明书，参考其中 Cleaning 部分，这里摘抄如下。

    - *Number of Cleans* value is ususally set to one(1). These are additional clean cycles that can be required after a start exposure signal is received and the

[3]这里默认参数指的是示例 vi 直接在后面板中设定好的参数

current clean cycle has finished. The maximum value for this entry depends on the camera. 通常设为 1。

- *Number of Strips per Clean* sets the number of rows that will be shifted and discarded per clean cycle. While a large number such as the number of rows in the array may result in the best cleaning of the array, the tradeoff is that there may be a significant delay between the receipt of a start exposure signal and the beginning of the actual exposure. This delay occurs because the current clean cycle must be completed before a start exposure signal received during the cycle will be implemented.Typically, the default setting is much smaller and in time critical experiments, the setting should be 1 or 2. 值越大清理的 CCD 行数越多，但是耗时也越多，需要用户权衡好。
- *Continuous Cleans* is available when the start of exposure is tied to an external trigger.This parameter is a flag that will allow clears to be performed "continuously" while running. This is the number o strips to clear before looking if an external trigger has arrived. This mode is usually used when collecting data with the camera in external trigger mode and the shutter disabled open.

WinSpec 设置: Hardware Setup>Clean/Skips

LightField 设置: Experiment Settings>Sensor>Sensor Cleaning

- 第二个底层 vi：CameraShutter.vi。设置 CCD 的开和关。

默认参数：Shutter State:4；Shutter Compensation Time: Pre Open:0; Shutter Compensation Time: Post Open:0。

参数说明：

- Shutter State 设为 4, 表示 CCD 一直开着。
- Shutter Compensation Time: Pre Open，只支持 Photometrics brand cameras 和 Acton InSpectrum camera，可以不连。
- Shutter Compensation Time: Post Open 的值必须设置，是 CCD 关闭的时间，单位是毫秒。

WinSpec 设置:

LightField 设置: Experiment Settings>Shutter

- 第三个底层 vi：ToolKitIsError.vi。检查 SITK 软件包是否有错误。

- 第四个子 vi 是底层 vi: CameraTrigger.vi。设置的是 CCD 信号触发模式。

默认参数：Timing Mode:3(Strobed/Ext Sync)；Trigger Edge:2。

参数说明：

- Timing Mode:3 表示选用的是外触发模式 “Each exposure requires a trigger (Ext Sync, Strobed Mode)”；
- Trigger Edge:2 表示信号触发沿设定的是下降沿 “Negative(falling) edge”。

WinSpec 设置:

LightField 设置: Experiment Settings>Trigger>Readout Per Trigg

- 第五个子 vi 是底层 vi：CameraADCset.vi。ADC，Analog to Digital Conversion，模拟信号转换为数字信号，模数转换。

  默认参数:ADC Speed:-2; ADC Gain:-1; Read Out port:-1; ADC offset:-1

  参数说明:

  - ADC Speed: For Slowest ADC speed set this to -3; For Fastest ADC speed set this to -2; To use current ADC speed don't hook up or use -1 otherwise put speed in as KHz.
  - ADC Gain: To use current gain don't hook up or set to -1.
  - Read Out port: To use current readout port don't hook up or set to -1.
  - ADC offset: To use current offset don't hook up or set to -1.

  WinSpec 设置:

  LightField 设置: Experiment Settings>Analog to Digital Conversion>Electron Multiplied
- 第六个子 vi 是底层 vi：CameraADCget.vi。获取当前模数转换的信息，示例 vi 中只输出了 ADC Speed 项。

（3）ExSetupExtSuperSyncro.vi: 由一系列子 vi 构成，用来设置 CCD 的触发信号同步。

- 第一个子 vi 是示例 vi: ExSubCheckCamPulser.vi。检查 Camera 和 Pulser 全局句柄是否创建。
- 第二个子 vi 是示例 vi: ExSubSetupCamPulserExtTrigSuperSync.vi。由一系列底层 vi 构成[4]。
  - 第一个底层 vi 是: CameraShutter.vi

    与前面 vi 有重复，删除此处。
  - 第二个底层 vi 是：CameraIntensMode.vi。设置光电倍增管的模式和强度。

    参数：Mode:2; Intensifier Gain: 20

    参数说明:

    - Mode:2 表示光电倍增管一直开着；
    - Intensifier Gain: 20 表示增强值设为 20。为什么是 20 不太清楚。

    WinSpec 设置:

    LightField 设置: Experiment Settings>Analog to Digital Conversion>Electron Multiplied>EM Gain
  - 第三个底层 vi 是：CameraTrigger.vi

    与前面 vi 有重复，删除此处。
  - 第四个底层 vi 是：CameraCleans.vi

    与前面 vi 有重复，删除此处。
  - 第五个底层 vi 是：CameraSkips.vi

    与前面 vi 有重复，删除此处。
  - 第六个底层 vi 是：PulserExternalTrigger.vi。设置外触发模式以及相关参数。

[4]注意：底层 vi 都是有详细说明文档的；底层 vi 中经常用到无符号长整型，但是用的时候可能用的有无符号长整型，虽然不会影响参数传递，但是在接口处有红点提示数据类型不是完全匹配

参数：Threshold:1; Slope:1; Coupling:1; Termination: 0

参数说明：

- Threshold: 设置触发电压阈值，单位伏特 (Volts)。
- Slope: 触发沿，0 表示下降沿触发，表示上升沿触发。
- Coupling: 设置触发是交流（AC）触发还是直流（DC）触发，0 表示 ac，1 表示 dc。
- Termination: 设置阻抗,0 表示 50 欧姆（Ohms），1 表示高阻抗。

Available only in Kinetics mode or DIF acquisition.

- 第七个底层 vi 是：PulserGateWidthDelay.vi。This VI will set the gate width and delay values for repetitive pulsing in microseconds.

  参数：Gate Width: 0.1(μs); Gate Delay: 0.0(μs)

  参数说明：

  - Gate Width: This value is the width of the gate pulse in microseconds.
  - Gate Delay: This value is the delay of the gate pulse in microseconds.

- 第八个底层 vi 是：PulserOnChipAccum.vi。设置读取 CCD 芯片值前的曝光次数。

  参数：Accumulation:1

  参数说明：单脉冲测量关键，1 表示曝光一次读取一次值，大于 1 表示曝光多次后读取一次值。

- 第九个底层 vi 是：PulserSetVar.vi

  默认参数：Parameter ID:1002; Value:1601.0

  参数说明：Parameter ID,1002，表示 Pulse Delay From;Value，1601.0 表示 del from ext trig in, 而 1602 表示 del from t0 out.

- 第十个底层 vi 是：PulserSetVar.vi

  默认参数：Parameter ID:1102; Value:0.0

  参数说明：Parameter ID,1102，表示 Braket Pulse;Value，0.0 表示 turn off braket pulse.

20.9 PI-MAX3 光谱采集 LabVIEW 程序解析

CCD 光谱采集 LabVIEW 程序是由示例程序 ExDataCollectPulserNframe.vi 改写过来的。这里详细说明下。

- 第一个子 vi 是示例 vi: ExSubCheckCamPulser.vi。检查 Camera 和 Pulser 全局句柄是否创建。

  初始化已经检查过了，这里删除？

- 第二个子 vi 是示例 vi: ExSubInitCamPulser.vi。CCD 初始化。由多个底层 vi 构成。
  - 第一个底层 vi 是: CameraExperiment.vi。This function will allow the setup and running of a basic data collection experiment. All advanced camera settings are set

to good defaults.

参数: Exposure(sec):0.0001(s,100μs); Number of Images: 200(**注：这里的严格表述应该为 Number of pulse**)

默认参数: Acquire/Focus Flag: 1; Number of Accumulations:1

- 第二个底层 vi 是: PulserInit.vi。**什么是 Pulser?**
- 第三个底层 vi 是: CameraInitialzie.vi。初始化 CCD，随时准备采集数据。

- 第三个子 vi 是示例 vi: ExSubCreateFileMultImage.vi 由多个底层 vi 构成。

默认参数:Number of Images:1。**和前面有重名需要进一步考察他们的功能**。

  - 第一个底层 vi 是: CameraGetDataDim.vi。This VI will return the X and Y dimension of one frame of data as defined in the camera. **不出意外，根据前面的设置，应该为 1600*1, 要测试下**。
  - 第二个底层 vi 是: ImageCreate.vi: This VI will create an image space to hold an image of dimensions X and Y, number of frames and of the data type specified. A handle to this image is returned to provide access.

    默认参数: x dimension、y dimension: 由前面 vi 获取得到;Data Type:4; Number of Frames: 1

    参数说明:

    - X dimension: Number of data points in the X direction.
    - Y dimension: Number of data points in the Y direction.
    - Data Type: Type of binary representation of the data. 4=32-bit floating point
    - Number of Frames: Number of full frames of data X by Y in size.
    - Image Handle: A handle to image created is returned to provide access to the image. **Handle 到底是什么东西？**
  - 第三个底层 vi 是: FileOpen.vi。**并没有用到**。

- 第四个子 vi 是示例 vi:ExSubStartExpPulser.vi。由多个底层 vi 构成。
  - 第一个底层 vi 是: PulserStart.vi
  - 第二个底层 vi 是: CameraStart.vi

- 第五个子 vi 是示例 vi: ExSubNframeColMultiImageFast.vi。由多个底层 vi 构成。

默认参数:Number of Images:1。**和前面有重名需要进一步考察他们的功能**。

  - 第一个底层 vi 是: CameraCheckData.vi。This function will return the number of complete frames of data available to the calling function.
  - 第二个底层 vi 是: CameraGetData.vi。This VI will retrieve the data from the camera and store it in the data area of the data cluster parameter. The storage space for the data must be pre-allocated by the caller. The dimensions to be allocated may be obtained from the CameraGetDataDim VI.
  - 第三个底层 vi 是: ImageGetLineF32.vi。This VI will return a 1-dimensional line of data from an image in 32-bit Floating Point format.

    默认参数: Y Position:1; Z Position:1

参数说明:

- Data Handle: Handle to an image. This is obtained via the ImageCreate VI.
- Z Position: This value is the 1-based position in the Z-axis of the frame of data from which the line of data is to be returned.
- Y Position: This value is the 1-based position in the Y-axis of the line of data to be returned.

- 第六个子 vi 是底层 vi: ImageDestroy.vi
- 第七个子 vi 是示例 vi: ExSubStopCamPulser.vi。
  - 第一个底层 vi 是: CameraStop.vi
  - 第二个底层 vi 是: PulserStop.vi

第五部分

Parallel computing

参考资料

- 高性能计算之并行编程技术——MPI 并行程序设计

第二十一章 背景知识

21.1 并行语言

并行程序是通过并行语言来表达的，并行语言的产生主要有三种方式：

- 设计全新的并行语言

  设计一种全新的并行语言的优点是可以完全摆脱串行语言的束缚，从语言成分上直接支持并行。这样就可以使并行程序的书写更方便，更自然，相应的并行程序也更容易在并行机上实现。但是，由于并行计算至今还没有象串行计算那样统一的冯诺伊曼模型可供遵循，因此，并行机、并行模型、并行算法和并行语言的设计和开发千差万别，没有一个统一的标准。虽然有多种多样全新的并行语言出现，但至今还没有任何一种新出现的并行语言，成为普遍接受的标准。设计全新的并行语言，实现起来难度和工作量都很大。

- 扩展原来的串行语言的语法成分使它支持并行特征

  一种重要的对串行语言的扩充方式就是标注，即将对串行语言的并行扩充作为原来串行语言的注释，对于这样的并行程序，若用原来的串行编译器来编译，标注的并行扩充部分将不起作用，仍将该程序作为一般的串行程序处理，若使用扩充后的并行编译器来编译，则该并行编译器就会根据标注的要求，将原来串行执行的部分转化为并行执行。对串行语言的并行扩充，相对于设计全新的并行语言，显然难度有所降低，但需要重新开发编译器，使它能够支持扩充的并行部分。一般地，这种新的编译器往往和运行时支持的并行库相结合。如 openMP（Open Multi-Processing）。

- 不改变串行语言仅为串行语言提供可调用的并行库

  仅仅提供并行库，是一种对原来的串行程序设计改动最小的并行化方法。这样，原来的串行编译器也能够使用，不需要任何修改，编程者只需要在原来的串行程序中加入对并行库的调用就可以实现并行程序设计。MPI 并行程序设计，就属于这种方式。

对于这三种并行语言的实现方法，目前最常使用的是第二种和第三种方法特别是第三种方法。