跟老男孩学Linux运维

Learn Linux Operation with Old Boy
Shell Programming

Shell编程实战

老男孩 ◎著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

跟老男孩学 Linux 运维：Shell 编程实战 / 老男孩著 . —北京：机械工业出版社，2017.1
（Linux/Unix 技术丛书）

ISBN 978-7-111-55607-7

I. 跟… II. 老… III. Linux 操作系统 IV. TP316.85

中国版本图书馆 CIP 数据核字（2016）第 313248 号

跟老男孩学 Linux 运维：Shell 编程实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：杨绣国 张梦玲 责任校对：董纪丽

印 刷：北京文昌阁彩色印刷有限责任公司 版 次：2017 年 1 月第 1 版第 1 次印刷

开 本：186mm × 240mm 1/16 印 张：26.25

书 号：ISBN 978-7-111-55607-7 定 价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

前言

为什么要写这本书

目前全球正处于互联网 + 的时代，越来越多的传统企业都在通过互联网提供产品和服务，比如，互联网 + 教育、互联网 + 金融、互联网 + 电商、互联网 + 出租车、互联网 + 保险等，可以看到，几乎所有的产品、服务都能在网上找到。而支撑互联网的幕后英雄其实就是 Linux（包括移动互联网在内），掌握 Linux 运维技术已经成为每一个 IT 技术人员的必备技能！

互联网 + 的时代下企业的网站流量呈爆炸式增长，如果你是运维人员，很可能要面对几十台、几百台、上千台甚至上万台的服务器设备，而对于企业来说，如何提高 IT 运维的管理效率、降低成本也成了最大问题。要解决这个问题，必须在 Linux 运维工作中，做好运维服务的标准化、规范化、流程化和自动化，而这里面的前三项其实是在为最后一项“IT 运维自动化”做铺垫。

要实现 IT 运维自动化就需要学会编程语言，目前 Linux 系统下最流行的运维自动化语言就是 Shell 和 Python（Python 相关图书，作者正在写作中）。在这两者之中，Shell 又几乎是所有 IT 企业都必须使用的运维自动化编程语言，特别是在运维工作中的服务监控、业务快速部署、服务启动停止、数据备份及处理、日志分析等环节里，Shell 必不可少。当然 Python 也是一门很好的自动化编程语言，它和 Shell 是互补的，Shell 更适合系统底层，而 Python 则更适合处理复杂的业务逻辑，以及开发复杂的运维软件工具，实现通过 Web 访问等。

在长期的运维工作以及深度教学中，老男孩发现很多 Linux 入门人员很害怕 Shell 编程，觉得 Shell 不好掌握，甚至是已经工作的企业运维人员对 Shell 编程也是一知半

解，不能熟练运用。而市面上的 Shell 图书大多如出一辙，理论多，实战少。因此在众多学员和网友的关注和提议下，老男孩决定写一本比较与众不同的偏重实战的 Shell 编程书籍，相信本书一定会让众多读者受益，提升个人在企业工作中的效率，达到加薪升职的目的。

本书是老男孩 Linux 运维实战系列的第二本书，第一本是《跟老男孩学习 Linux 运维：Web 集群实践》(已由机械工业出版社出版)，第三本是《跟老男孩学习 Linux 运维：三剑客命令实战》(预计书名)，此书将在几个月后和大家见面。更多 Linux 运维实战系列的图书在持续写作中，敬请期待。

读者对象

- ❑ 热衷于 IT 运维自动化的人员
- ❑ Linux 系统管理员和运维工程师
- ❑ 互联网网站开发及数据库管理人员
- ❑ 网络管理员和项目实施工程师
- ❑ Linux 相关售前售后技术工程师
- ❑ 开设 Linux 相关课程的大中专院校
- ❑ 对 Linux 及 Shell 编程感兴趣的人员

如何阅读本书

本书是一本较完整的 Shell 编程实战型图书，并非大而全，但处处可以体现实战二字，大多内容取于企业实战，并结合老男孩十几年的运维工作和教学工作进行了梳理。本书从脉络上可分为五大部分：

第一部分为 Shell 编程基础篇（第 1 章 ~ 第 4 章），着重介绍新手如何学好 Shell 编程，涉及的内容包括 Shell 编程的入门介绍、基础知识、运行原理、编程语法、编程习惯、变量知识以及变量的深入实践。读者学完此部分，将会具备一个学好 Shell 编程的坚实基础。

第二部分为初中级的实战知识和技能篇（第 5 章 ~ 第 8 章），着重讲解变量的多种数值运算、条件测试与比较、if 条件判断语句、Shell 函数等相关的知识，并给出了企业实战技巧和案例。本部分是学好 Shell 编程的重中之重，读者必须掌握。

第三部分为 Shell 中高级实战知识和技能篇（第 9 章 ~ 第 13 章），着重讲解 case 条件语句、while 循环和 until 型循环、for 循环和 select 循环、条件与循环控制及状态返回值、Shell 数组等知识，以及相应的实战技巧和案例。本部分同样是学好 Shell 编程的重中之重，读者必须掌握。

第四部分为高效 Shell 编程必备知识篇（第 14 章 ~ 第 16 章），着重讲解 Shell 脚本开发规范与编码习惯、Shell 脚本的调试知识和技巧、Shell 脚本开发环境的配置调整和优化等。

第五部分为 Shell 特殊应用及企业面试、实战案例篇（第 17 章 ~ 第 19 章），着重讲解 Linux 信号及 trap 命令的企业应用实践、Expect 自动化交互式程序的应用实践，以及能体现全书所讲技术的面试题和企业实战案例，让真正的 Shell 全自动化运维成为可能。

最后一章补充讲解了大家易感困惑的子 Shell 知识及应用实践内容。

勘误和支持

由于作者所授的培训课程排期很紧，课程较多，全书内容基本上都是利用早晨和夜里的时间完成写作的。限于作者的水平和能力，加之编写的时间仓促，书中难免有疏漏和不当之处，恳请读者批评指正。你可以将书中的错误发布在专门为本书准备的博客地址评论处（http://oldboy.blog.51cto.com/2561410/1865956 或微博 http://weibo.com/oldboy8）。同时不管你遇到何种问题，都可以加入我为本书提供的 QQ 交流群 204041129（验证信息：Shell 书籍），我将尽力为你提供最满意的解答。书中所需的工具及源文件也将发布在的博客网站上（书中大部分章节结尾都给出了相关网址及二维码），我也会将相应功能的更新及时发布出来。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱 oldboy@oldboyedu.com，很期待能够听到你们的真挚反馈。

致谢

感谢犹金毅、何清等为本书贡献第 20 章的重要底稿内容及对本书的写作给予的支持。

感谢孔令飞为本书第 19 章贡献有趣的 girlLove 案例内容及对本书的写作给予的支持。

感谢老男孩 IT 教育的每一位在校学员，是你们自觉努力的学习，使得我有较多的时间持续写作，特别是运维 30-31 期 150 位学员参与了本书的校稿。感谢你们对老男孩

老师的支持。

感谢老男孩IT教育里每一个班级的助教、班主任、班长及班干部，感谢你们替我分担老男孩IT教育众多学员的答疑、辅导、批改作业及班级管理工作。

感谢我的同事——老男孩教育Python学院的Alex老师、武老师，云计算与自动化架构班的赵班长老师，Linux+Python高薪运维班的李泳谊、张耀等老师，以及其他未提及名字的众多老师，正是你们辛勤努力的工作，让我得以有时间完成此书。

一如既往地感谢中网志腾的郭威总经理和数码创天的王斐总经理及梁露女士，感谢你们提供优质的DELL服务器资源，使得本书得以高效顺利地完成！

感谢森华易腾的陆锦云女士及其同事，感谢你们提供的优质IDC机房带宽支持，使得本书得以顺利完成！

感谢机械工业出版社华章公司的编辑杨绣国，感谢你的支持、包容和鼓励，正是你的鼓励和帮助引导我顺利完成全部书稿。

感谢没有提及名字的所有学生、网友以及关注老男孩的每一位友人、朋友。

最后要感谢我的父母、家人，正是你们的支持和体谅，让我有无限信心和力量去写作，并最终完成此书！

谨以此书，献给支持老男孩IT教育的每一位朋友、学员及众多热爱Linux运维技术的朋友。

老男孩老师

北京，2016年11月

目录

第1章

如何才能学好 Shell 编程

1.1 为什么要学习 Shell 编程

Shell 脚本语言是实现 Linux/UNIX 系统管理及自动化运维所必备的重要工具，Linux/UNIX 系统的底层及基础应用软件的核心大都涉及 Shell 脚本的内容。每一个合格的 Linux 系统管理员或运维工程师，都需要能够熟练地编写 Shell 脚本语言，并能够阅读系统及各类软件附带的 Shell 脚本内容。只有这样才能提升运维人员的工作效率，适应日益复杂的工作环境，减少不必要的重复工作，从而为个人的职场发展奠定较好的基础。那么，Shell 脚本编程的学习是否容易呢？学习 Shell 编程到底需要什么样的 Linux 基础呢？

1.2 学好 Shell 编程所需的基础知识

本节首先来探讨一下在学习 Shell 编程之前需要掌握的基础知识，需要说明的是，并不是必须具备这些基础知识才可以学习 Shell 编程，而是，如果具备了这些基础知识，那么就可以把 Shell 编程学得更好，领悟得更深。如果只是想简单地了解 Shell 脚本语言，那么就无须掌握太多的系统基础知识，只需要会一些简单的命令行操作即可。

学好 Shell 编程并通过 Shell 脚本轻松地实现自动化管理企业生产系统的必备基础如下：

1）能够熟练使用 vim 编辑器，熟悉 SSH 终端及“.vimrc”等的配置。

在 Linux 下开发 Shell 脚本最常使用的编辑器是 vim，因此如果能够熟练使用并配置好 vim 的各种高级功能设置，就可以让开发 Shell 脚本达到事半功倍的效果。这部分内容在本书的第 16 章有相应的讲解，读者在开始编写脚本之前可以考虑先看看第 16 章并搭建出高效的 Shell 开发环境。

说明：在本书的第 16 章讲解 Shell 脚本开发环境的配置调整和优化时，提到了高效搭建 Shell 开发环境的方法，之所以把这部分内容安排在第 16 章，是希望读者能体验一下比较原始的 Shell 开发过程，然后再来掌握搭建高效的开发环境的方法，老男孩从教学的角度认为这是一个比较好的过程，读者可以根据自身的情况来决定要不要提前学习第 16 章，搭建好高效的 Shell 开发环境。

2）要有一定的 Linux 命令基础，至少需要掌握 80 个以上 Linux 常用命令，并能够熟练使用它们（Linux 系统的常用命令请参见本书的附录）。

和其他的开发语言（例如 Python）不同，Shell 脚本语言很少有可以直接使用的外部函数库，老男孩就将 Linux 系统的命令看作 Shell 的函数库，因此，对 Linux 系统常用命令的掌握程度就直接决定了运维人员对 Shell 脚本编程的掌握高度。一些 Shell 类图书在开篇花费大量章节来讲解 Linux 基础命令也许就是因为这点，本书主要侧重于 Shell 编程企业案例实战讲解，因此不会进行大且全的介绍，也不会过多地讲解 Linux 的常用命令，而是采用小而美的实战策略，本书结尾会以附录的形式给出常用的 Linux 基础命令的相关知识。此外，如果读者想学习 Linux 基础命令，可以关注老男孩即将出版的新书——《跟老男孩学习 Linux 运维：常用命令实战》[㊀]，或者其他相关图书。

3）要熟练掌握 Linux 正则表达式及三剑客命令（grep、sed、awk）。

Linux 正则表达式及三剑客命令（grep、sed、awk）是 Linux 系统里所有命令中最核心的 3 个命令，每个命令加上正则表达式的知识后，功能都会变得异常强大。如果能够掌握它们，就可以在编写 Shell 脚本时轻松很多。如读者想学习这部分知识，可以关注老男孩即将出版的新书——《跟老男孩学习 Linux 运维：三剑客命令实战》[㊁]，或者其他相关图书。

4）熟悉常见的 Linux 网络服务部署、优化、日志分析及排错。

学习 Shell 编程最直接的目的就是在工作中对系统及服务等进行自动化管理，因此，如果不熟悉工作中的网络服务，就会很难使用 Shell 编程处理这些服务；如果不掌

㊀《跟老男孩学习 Linux 运维：常用命令实战》（预计书名）也将由机械工业出版社出版，时间预计为 2017 年。

㊁《跟老男孩学习 Linux 运维：三剑客命令实战》（预计书名）也将由机械工业出版社出版，时间预计为 2017 年。

握网络服务等知识，就会让 Shell 开发者的能力大打折扣，甚至学习到的仅仅是 Shell 的语法及简单的基础，那么想要学好 Shell 编程的想法也就落空了。需要掌握的基础网络服务包括但不限于：Crond、Rsync、Inotify、Nginx、PHP、MySQL、Keepalived、Memcached、Redis、NFS、Iptables、SVN、Git，老男孩 IT 教育的老师在教学的过程中也是先讲解 Linux 常用命令和系统网络服务，然后再讲解 Shell 编程，目的就是不要让学员仅仅掌握 Shell 的语法皮毛，而是让他们能在学完 Shell 编程之后，自动搭建中型集群架构等，有关基础网络服务的知识可以参考机械工业出版社的《跟老男孩学习 Linux 运维：Web 集群实战》一书，或者其他相关图书。

1.3　如何才能学好 Shell 编程之“老鸟”经验谈

学好 Shell 编程的核心：多练→多思考→再练→再思考，坚持如此循环即可！

从老男孩 IT 教育毕业的一名学生[㊀]曾在工作多年后返校分享了一篇“如何学好 Shell 编程”的讲稿，经过老男孩的整理后和读者分享如下。

（1）掌握 Shell 脚本基本语法的方法

最简单有效的方法就是将语法敲 n+1 遍。为什么不是 n 遍呢？因为这里的 n 指的是你刚开始为掌握语法而练习的那些天（21 天法则），而 1 则是指在确定掌握语法后每天都要写一写、想一想，至少是要看一看，保持一个与 Shell 脚本接触的热度。

（2）掌握 Shell 脚本的各种常见语法

要掌握各类条件表达式、if 多种判断、for 循环的不同用法、While 多种读文件的循环等，这样做不是为了什么都学会，而是为了能够看懂别人写的代码。掌握常见的各种语法，也就是要经常写，而且要持续写一段时间（让动作定型，在大脑和肌肉里都打上深刻烙印），各种语法都要用。

（3）形成自己的脚本开发风格

当掌握了各种常见的语法之后，就要选定一种适合自己的语法，形成自己的开发风格，例如：if 语句的语法就只用一种，条件表达式的语法只用一种，函数的写法也只用一种，有些语法需要根据场景去选择，除非你是像师傅（老男孩）一样要教学育人。否则，没有必要什么语法都掌握。在解决问题的前提下，掌握一种语法，然后将其用精、用透就是最好的，切记横向贪多，要多纵深学习。

（4）从简单做起，简单判断，简单循环

初学者一定要从简单做起，最小化代码学习，简单判断，简单循环，简单案例练习，所有的大程序都是由多个小程序组成的，因此，一开始没必要写多大的程序，免得给自己带来过多的挫败感，形成编程恐惧症。可先通过小的程序培养兴趣及成就感，到

㊀ 说明：该分享人是老男孩的早期学员，毕业后曾任职于一家近千人公司的运维经理岗位。目前就职于小米科技公司，担任资深工程师。

碰到大的程序时，即使遇到困难也能坚持下去了。

（5）多模仿，多放下参考资料练习，多思考

多找一些脚本例子来仔细分析一下，或者是系统自带的，或者是别人写的（本书就包含大量例子），不要只看，看着会并不是真的会。当你闭上眼睛的时候，还能完整地回忆起来，甚至还能完整口述或手写出来才是真的会。

（6）学会分析问题，逐渐形成编程思维

在编写程序或脚本时，先将需求理解透，对大的需求进行分解，逐步形成小的程序或模块，然后再开发，或者先分析最终需求的基础实现，最后逐步扩展批量实现。例如师傅（老男孩）在编写批量关闭不需要自启动服务的脚本时，就采用了这种分析方法，思路如下：

1）掌握关闭一个服务的命令，即“chkconfig 服务名 off”。

2）批量处理时，会有多个服务名，那么就要用到多条以上的命令。

3）仔细分析以上命令，会发现需要处理的所有命令中，只有“服务名”不同，其他地方都一样，那么自然就会想到用循环语句来处理。

如果是你，能想到这些吗？若是想到了，则表示你已经形成了初级的编程思维了，恭喜你。

如果你能够通过分析将一个大的需求细分为各个小的单元，然后利用函数、判断、循环、命令等实现每一个小的单元，那么最后把所有程序组合起来就是一个大的脚本程序了。

如果达到了上述的水平，你就算会编程了，对于领导提出的需求，就能够进行合理的分解，只要在机器上多进行调试，相信一定能写出来。

（7）编程变量名字要规范，采用驼峰语法表示

oldboyAgeName 用的就是驼峰表示法。记住，在学习的初期，不要去看大的脚本，要从小问题和小的方面着手，当你觉得小的判断、循环等在你的脑子里瞬间就能出来时，再开始去看和写大的脚本，进行深入练习。

师傅（老男孩）常说，新手初期最好的学习方法就是多敲代码，并针对问题进行分解练习，多敲代码就是让自己养成一个编程习惯，使肌肉、视觉和思维形成记忆，分解问题实际上就是掌握软件的设计和实现思想。

对于最高的编程境界，我个人的理解是：能把大问题进行完整的分析、分解且高效解决。

完整性：就是指预先考虑到各种可能性，将问题分解后，合理模块化并实现。

高效率：例如，在求“1+2+3...+100”的和时，考虑使用算法“（1+100）×100/2”，而不是逐个去加。

（8）不要拿来主义，特别是新手

好多网友看书或学习视频时，喜欢要文档、要代码，其实，这是学习的最大误区。

有了文档和代码，你会变得非常懒惰，心里面会觉得已经学会了，而实际上并没有学会。因此无论是看书还是学习视频，都要自己完成学习笔记及代码的书写，这本身就是最重要的学习过程，在学习上要肯于花时间和精力，而不是投机取巧。如果你至今都没有学好 Linux 运维，那么可以想一想是不是也犯了这个错误？

1.4　学完本书后可以达到何种 Shell 编程高度

如果读者具备了前文提到 Linux 基础知识，认真地阅读并按照书中的内容去勤加练习，相信很快便可熟练掌握 Shell 编程，搞定企业场景中的绝大多数 Shell 编程问题，本书介绍了大量的核心互联网运维场景企业案例，相信对大家的工作会很有帮助。

如果再配合老男孩的 Shell 脚本教学视频，定能使你如虎添翼，相关视频一共有 14 部（数百课时），观看地址为：http://edu.51cto.com/pack/view/id-546.html，读者也可以扫描下面的二维码，注册付费后开始学习。

第2章

Shell 脚本初步入门

在解释“Shell 脚本”这个名词之前，我们先来看看什么是 Shell。

2.1 什么是 Shell

Shell 是一个**命令解释器**，它的作用是解释执行用户输入的命令及程序等，用户每输入一条命令，Shell 就解释执行一条。这种从键盘一输入命令，就可以立即得到回应的对话方式，称为交互的方式。

Shell 存在于操作系统的最外层，负责与用户直接对话，把用户的输入解释给操作系统，并处理各种各样的操作系统的输出结果，然后输出到屏幕返回给用户。输入系统用户名和密码并登录到 Linux 后的所有操作都是由 Shell 解释与执行的。

图 2-1 针对命令解释器 Shell 在操作系统中所处的位置给出了基本图解。

图 2-1　Shell 在操作系统中所处位置的基本图解

提示：Shell 的英文是贝壳的意思，从图 2-1 中可以看出，命令解释器（Shell）就像贝壳一样包住了系统核心。

2.2　什么是 Shell 脚本

理解了 Shell 之后，再理解 Shell 脚本就简单了。当命令或程序语句不在命令行下执行，而是通过一个程序文件来执行时，该程序就被称为 Shell 脚本。如果在 Shell 脚本里内置了很多条命令、语句及循环控制，然后将这些命令一次性执行完毕，这种通过文件执行脚本的方式称为非交互的方式。Shell 脚本类似于 DOS 系统下的批处理程序（早期扩展名一般为“*.bat”）。用户可以在 Shell 脚本中敲入一系列的命令及命令语句组合。这些**命令、变量和流程控制语句**等有机地结合起来，就形成了一个功能强大的 Shell 脚本。

下面是在 Windows 下利用 bat 批处理程序开发的备份企业网站及数据库数据的脚本范例。

范例 2-1：在 Windows 下利用 bat 批处理程序备份网站及数据库数据的脚本。

```
@echo off
set date=%date:~0,4%-%date:~5,2%-%date:~8,2%                    #<== 定义时间变量。
mysqldump -uroot -poldboy -A -B  > D:\bak\"%date%".sql   #<== 备份数据库数据。
rar.exe a -k -r -s -m1 D:\bak\"%date%".sql.rar D:\bak\"%date%".sql
#<== 打包备份出来的数据库数据。
del D:\bak\*.sql #<== 删除未打包的无用数据库数据。
rar.exe a -k -r -s -m1 D:\bak\"%date%"htdocs.rar D:\work\PHPnow\htdocs
#<== 打包站点目录下的数据。
```

范例 2-2：清除 /var/log 下 messages 日志文件的简单命令脚本。

把所有命令放在一个文件里，堆积起来后就形成了脚本，下面就是一个由最简单的命令堆积而成的 Shell 脚本。需要注意的是，必须使用 root 身份来运行这个脚本。

```
# 清除日志脚本，版本 1。
cd /var/log
cat /dev/null>messages
echo "Logs cleaned up."
```

提示：/var/log/messages 是 Linux 系统的日志文件，很重要。

范例 2-2 所示的脚本其实是有一些问题的，具体如下：

1）如果不是 root 用户，则无法执行脚本清理日志，并且会提示系统的权限报错信息。

2）没有任何流程控制语句，简单地说就是只进行顺序操作，没有成功判断和逻辑严密性。

范例 2-3：写一个包含命令、变量和流程控制的语句来清除 /var/log 下 messages 日志文件的 Shell 脚本。

```
# !/bin/bash
# 清除日志脚本，版本 2
LOG_DIR=/var/log
ROOT_UID=0      #<==$UID 为 0 的用户，即 root 用户
# 脚本需要使用 root 用户权限来运行，因此，对当前用户进行判断，对不合要求的用户给出友好提示，并终止程序运行。
if [ "$UID" -ne "$ROOT_UID" ] #<== 如果当前用户不是 root，则不允许执行脚本。
  then
    echo "Must be root to run this script." #<== 给出提示后退出。
    exit 1 #<== 退出脚本。
fi
# 如果切换到指定目录不成功，则给出提示，并终止程序运行。
cd $LOG_DIR || {
    echo "Cannot change to necessary directory."
    exit 1
}
# 经过上述两个判断后，此处的用户权限和路径应该就是对的了，只有清空成功，才打印成功提示。
cat /dev/null>messages && {
    echo "Logs cleaned up."
    exit 0  # 退出之前返回 0 表示成功，返回 1 表示失败。
}
echo "Logs cleaned up fail."
exit 1
```

初学者如果想要快速掌握 Shell 脚本的编写方法，最有效的思路就是采用电子游戏中过关的方式，比如，对于范例 2-3 的脚本可以设计成如下几关：

第一关，必须是 root 才能执行脚本，否则给出友好提示并终止脚本运行。

第二关，成功切换目录（cd /var/log），否则给出友好提示并终止脚本运行。

第三关，清理日志（cat /dev/null > messages），若清理成功，则给出正确提示。

第四关，通关或失败，分别给出相应的提示（echo 输出）。

2.3 Shell 脚本在 Linux 运维工作中的地位

Shell 脚本语言很适合用于处理**纯文本类型**的数据，而 Linux 系统中几乎所有的配置文件、日志文件（如 NFS、Rsync、Httpd、Nginx、LVS、MySQL 等），以及绝大多数的启动文件都是**纯文本**类型的文件。因此，学好 Shell 脚本语言，就可以利用它在 Linux 系统中发挥巨大的作用。

图 2-2 形象地展示了 Shell 脚本在运维工作中的地位。

图 2-2　Shell 脚本在运维工作中的地位形象图

2.4　脚本语言的种类

2.4.1　Shell 脚本语言的种类

Shell 脚本语言是**弱类型语言（无须定义变量的类型即可使用）**，在 Unix/Linux 中主要有两大类 Shell：一类是 Bourne shell，另一类是 C shell。

1. Bourne shell

Bourne shell 又包括 Bourne shell（sh）、Korn shell（ksh）、Bourne Again Shell（bash）三种类型。

- Bourne shell（sh）由 AT&T 的 Steve Bourne 开发，是标准的 UNIX Shell，很多 UNIX 系统都配有 sh。
- Korn shell（ksh）由 David Korn 开发，是 Bourne shell（sh）的超集合，并且添加了 csh 引入的新功能，是目前很多 UNIX 系统标准配置的 Shell，这些系统上的 /bin/sh 往往是指向 /bin/ksh 的符号链接。
- Bourne Again Shell（bash）由 GNU 项目组开发，主要目标是与 POSIX 标准保持一致，同时兼顾对 sh 的兼容，bash 从 csh 和 ksh 借鉴了很多功能，是各种 Linux 发行版默认配置的 Shell，Linux 系统上的 /bin/sh 往往是指向 /bin/bash 的符号链接。尽管如此，bash 和 sh 还是有很多的不同之处：一方面，bash 扩展了一些命令和参数；另一方面，bash 并不完全和 sh 兼容，它们有些行为并不一致，但在大多数企业运维的情况下区别不大，特殊场景可以使用 bash 替代 sh。

2. C shell

C shell 又包括 csh、tcsh 两种类型。

csh 由 Berkeley 大学开发，随 BSD UNIX 发布，它的流程控制语句很像 C 语言，支持很多 Bourne shell 所不支持的功能，例如：作业控制、别名、系统算术、命令历史、

命令行编辑等。

tcsh 是 csh 的增强版，加入了命令补全等功能，在 FreeBSD、Mac OS X 等系统上替代了 csh。

以上介绍的这些 Shell 中，较为通用的是标准的 Bourne shell（sh）和 C shell（csh）。其中 Bourne shell（sh）已经被 Bourne Again shell（bash）所取代。

可通过以下命令查看 CentOS 6 系统的 Shell 支持情况。

```
[root@oldboy ~]# cat /etc/shells
/bin/sh         #<== 这是 Linux 里常用的 Shell，指向 /bin/bash。
/bin/bash       #<== 这是 Linux 里常用的 Shell，也是默认使用的 Shell。
/sbin/nologin   #<== 这是 Linux 里常用的 Shell，用于禁止用户登录。
/bin/dash
/bin/tcsh
/bin/csh
```

Linux 系统中的主流 Shell 是 bash，bash 是由 Bourne Shell（sh）发展而来的，同时 bash 还包含了 csh 和 ksh 的特色，但大多数脚本都可以不加修改地在 sh 上运行，如果使用了 sh 后发现结果和预期有差异，那么可以尝试用 bash 替代 sh。

2.4.2 其他常用的脚本语言种类

1. PHP 语言

PHP 是网页程序语言，也是脚本语言。它是一款更专注于 Web 页面开发（前端展示）的语言，例如：wordpress、dedecms、discuz 等著名的开源产品都是用 PHP 语言开发的。用 PHP 程序语言也可以处理系统日志、配置文件等，还可以调用 Linux 系统命令，但是，很少有人这么用。

2. Perl 语言

Perl 脚本语言比 Shell 脚本语言强大很多，在 2010 年以前很流行，它的语法灵活、复杂，在实现不同的功能时可以用多种不同的方式，缺点是不易读，团队协作困难，但它仍不失为一种很好的脚本语言，**存世的大量相关程序软件**（比如，xtrabackup 热备工具、MySQL MHA 集群高可用软件等）中都有 Perl 语言的身影。当下的 Linux 运维人员几乎不需要了解 Perl 语言了，最多可了解一下 Perl 语言的安装环境。当然了想要二次开发用 Perl 编写软件人员例外，Perl 语言已经成为历史了。

3. Python 语言

Python 是近几年非常流行的语言，它不但可以用于脚本程序开发，也可以实现 Web 页面程序开发（例如：CMDB 管理系统），甚至还可以实现软件的开发（例如：大名鼎鼎的 OpenStack、SaltStack 都是 Python 语言开发的）、游戏开发、大数据开发、移动端开发。

现在越来越多的公司都要求运维人员会 Python 自动化开发。老男孩 IT 教育持续引领着国内 Linux 培训界的风向标，早在 2012 年以前就已经开设了 Python 自动化运维开

发实战课程（课程表见 http://oldboy.blog.51cto.com/2561410/1123127），并于 2015 年开设了 Python 全栈开发工程师课程，课程表见 http://oldboy.blog.51cto.com/2561410/1749122。Python 语言目前是全球第四大开发语言，未来的发展前景很好，每一个运维人员在掌握了 Shell 编程之后，都应该深入学习 Python 语言，以提升职场竞争力。

2.4.3　Shell 脚本语言的优势

Shell 脚本语言的优势在于处理偏操作系统底层的业务，例如：Linux 系统内部的很多应用（有的是应用的一部分）都是使用 Shell 脚本语言开发的，因为有 1000 多个 Linux 系统命令为它做支撑，特别是 Linux 正则表达式及三剑客 grep、awk、sed 等命令。

对于一些常见的系统脚本，使用 Shell 开发会更简单、更快速，例如：让软件一键自动化安装、优化，监控报警脚本，软件启动脚本，日志分析脚本等，虽然 PHP/Python 语言也能够做到这些，但是，考虑到掌握难度、开发效率、开发习惯等因素，它们可能就不如 Shell 脚本语言流行及有优势了。对于一些常规的业务应用，使用 Shell 更符合 Linux 运维**简单、易用、高效的三大基本原则**。

PHP 语言的优势在于小型网站系统的开发；Python 语言的优势在于**开发较复杂的运维工具软件**、Web 界面的管理工具和 Web 业务的开发（例如：CMDB 自动化运维平台、跳板机、批量管理软件 SaltStack、云计算 OpenStack 软件）等。我们在开发一个应用时应根据业务需求，结合不同语言的优势及自身擅长的语言来选择，扬长避短，从而达到高效开发及易于自身维护等目的。

2.5　常用操作系统默认的 Shell

在常用的操作系统中，Linux 下默认的 Shell 是 Bourne Again shell（bash）；Solaris 和 FreeBSD 下默认的是 Bourne shell（sh）；AIX 下默认的是 Korn Shell（ksh）。

这里重点讲 Linux 系统环境下的 Bourne Again shell（bash）。

下面来看一个企业面试题：CentOS Linux 系统默认的 Shell 是什么？这题的答案就是 bash。

通过以下两种方法可以查看 CentOS Linux 系统默认的 Shell。

方法 1：

```
[root@oldboy ~]# echo $SHELL
/bin/bash
```

方法 2：

```
[root@oldboy ~]# grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

提示：结尾的 /bin/bash 就是用户登录后的 Shell 解释器。

注意：本书写作的环境为 Linux 系统，具体版本为 CentOS 6.x x86_64，绝大部分已写好的脚本程序都不需要经过任何修改，就可以直接应用于其他的 Linux 系统中。对于一些 UNIX 系统，因为默认不是 bash 解释器，所以需要根据解释器版本进行调整，本书的全部内容都是以 bash 及和 bash 兼容的 sh 解释器为基础编写的。

2.6 Shell 脚本的建立和执行

2.6.1 Shell 脚本的建立

在 Linux 系统中，Shell 脚本（bash Shell 程序）通常是在编辑器 vi/vim 中编写的，由 UNIX/Linux 命令、bash Shell 命令、程序结构控制语句和注释等内容组成。这里推荐用 Linux 自带的功能更强大的 vim 编辑器来编写，可以事先做一个别名 alias vi='vim'，并使其永久生效，这样以后习惯输入 vi 的读者也就可以直接调用 vim 编辑器了，设置方法如下：

```
[root@oldboy ~]# echo "alias vi='vim'" >>/etc/profile
[root@oldboy ~]# tail -1 /etc/profile
alias vi='vim'
[root@oldboy ~]# source /etc/profile
```

1. 脚本开头（第一行）

一个规范的 Shell 脚本在第一行会指出由哪个程序（解释器）来执行脚本中的内容，这一行内容在 Linux bash 的编程一般为：

```
#!/bin/bash
或
#!/bin/sh #<==255 个字符以内。
```

其中，开头的“#!”字符又称为幻数（其实叫什么都无所谓，知道它的作用就好），在执行 bash 脚本的时候，内核会根据“#!”后的解释器来确定该用哪个程序解释这个脚本中的内容。

注意，这一行必须位于每个脚本顶端的**第一行**，如果不是第一行则为脚本注释行，例如下面的例子。

```
[oldboy@oldboy ~]$ cat test.sh
#!/bin/bash
```

```
echo "oldboy start"
#!/bin/bash                          #<== 写到这里就是注释了。
#!/bin/sh                            #<== 写到这里就是注释了。
echo "oldboy end"
```

2. bash 与 sh 的区别

早期的 bash 与 sh 稍有不同，它还包含了 csh 和 ksh 的特色，但大多数脚本都可以不加修改地在 sh 上运行，比如：

```
[root@oldboy ~]# ll /bin/sh
lrwxrwxrwx. 1 root root 4 3月  19 20:54 /bin/sh -> bash
[root@oldboy ~]# ll /bin/bash
-rwxr-xr-x 1 root root 940416 10月 16 21:56 /bin/bash
```

提示：sh 为 bash 的软链接，大多数情况下，脚本的开头使用“#!/bin/bash”和“#!/bin/sh”是没有区别的，但更规范的写法是在脚本的开头使用“#!/bin/bash”。

下面的 Shell 脚本是系统自带的软件启动脚本的开头部分。

```
[root@oldboy ~]# head -1 /etc/init.d/sshd
#!/bin/bash
[root@oldboy ~]# head -1 /etc/init.d/ntpd
#!/bin/bash
[root@oldboy ~]# head -1 /etc/init.d/crond
#!/bin/sh
```

提示：如果使用 /bin/sh 执行脚本出现异常，那么可以再使用 /bin/bash 试一试，但是一般不会发生此类情况。

一般情况下，在安装 Linux 系统时会自动安装好 bash 软件，查看系统的 bash 版本的命令如下。

```
[root@oldboy ~]# cat /etc/redhat-release
CentOS release 6.8 (Final)       #<== 这里显示的是作者写作的 Linux 的环境版本。
[root@oldboy ~]# bash --version
GNU bash, version 4.1.2(1)-release (x86_64-redhat-linux-gnu)
                                 #<== 这里显示的是 bash 的版本。
Copyright (C) 2009 Free Software Foundation, Inc.
                                 #<== 下面几行是自由软件提示的相关信息。
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law.
```

如果读者使用的是较老版本的 Shell，那么建议将其升级到最新版本的 Shell，特别是企业使用，因为近两年老版本的 bash 被暴露出存在较严重的安全漏洞。

例如：bash 软件曾经爆出了严重漏洞（破壳漏洞），凭借此漏洞，攻击者可能会接管计算机的整个操作系统，得以访问各种系统内的机密信息，并对系统进行更改等。任何人的计算机系统，如果使用了 bash 软件，都需要立即打上补丁。检测系统是否存在漏洞的方法为：

```
[root@oldboy ~]# env x='() { :;}; echo be careful' bash -c "echo this is a test"
this is a test
```

如果返回如下两行，则表示需要尽快升级 bash 了，不过，仅仅是用于学习和测试就无所谓了。

```
be careful
this is a test
```

升级方法为：

```
[root@oldboy ~]# yum -y update bash
[root@oldboy ~]# rpm -qa bash
bash-4.1.2-40.el6.x86_64
```

提示：如果没有输出 be careful，则不需要升级。

下面是 Linux 中常用脚本开头的写法，不同语言的脚本在开头一般都要加上如下标识内容：

```
1 #!/bin/sh
2 #!/bin/bash
3 #!/usr/bin/awk
4 #!/bin/sed
5 #!/usr/bin/tcl
6 #!/usr/bin/expect      #<==expect 解决交互式的语言开头解释器。
7 #!/usr/bin/perl        #<==perl 语言解释器。
8 #!/usr/bin/env python #<==python 语言解释器。
```

CentOS 和 Red Hat Linux 下默认的 Shell 均为 bash。因此，在写 Shell 脚本的时候，脚本的开头即使不加“ #!/bin/bash”，它也会交给 bash 解释。如果写脚本不希望使用系统默认的 Shell 解释，那么就必须要指定解释器了，否则脚本文件执行后的结果可能就不是你所要的。建议读者养成好的编程习惯，不管采用什么脚本，最好都加上相应的**开头解释器语言标识**，遵守 Shell 编程规范。

如果在脚本开头的第一行不指定解释器，那么就要用对应的解释器来执行脚本，这样才能确保脚本正确执行。例如：

如果是 Shell 脚本，就用 bash test.sh 执行 test.sh。

如果是 Python 脚本，就用 python test.py 执行 test.py。

如果是 expect 脚本，就用 expect test.exp 执行 test.exp。

提示：其他的脚本程序大都是类似的执行方法。

3. 脚本注释

在 Shell 脚本中，跟在 # 后面的内容表示注释，用来对脚本进行注释说明，注释部分不会被当作程序来执行，仅仅是给开发者和使用者看的，系统解释器是看不到的，更不会执行。注释可自成一行，也可以跟在脚本命令的后面与命令在同一行。开发脚本时，如果没有注释，那么团队里的其他人就会很难理解脚本对应内容的用途，而且若时间长了，自己也会忘记。因此，我们要尽量养成为所开发的 Shell 脚本书写关键注释的习惯，书写注释不光是为了方便别人，更是为了方便自己，避免影响团队的协作效率，以及给后来接手的人带来维护困难。特别提示一下，注释尽量不要用中文，在脚本中最好也不要有中文。

2.6.2　Shell 脚本的执行

当 Shell 脚本运行时，它会先查找系统环境变量 ENV，该变量指定了环境文件（加载顺序通常是 /etc/profile、~/.bash_profile、~/.bashrc、/etc/bashrc 等），在加载了上述环境变量文件后，Shell 就开始执行 Shell 脚本中的内容（更多 Shell 加载环境变量的知识请见第 3 章）。

Shell 脚本是从上至下、从左至右依次执行每一行的命令及语句的，即执行完了一个命令后再执行下一个，如果在 Shell 脚本中遇到子脚本（即脚本嵌套）时，就会先执行子脚本的内容，完成后再返回父脚本继续执行父脚本内后续的命令及语句。

通常情况下，在执行 Shell 脚本时，会向系统内核请求启动一个新的进程，以便在该进程中执行脚本的命令及子 Shell 脚本，基本流程如图 2-3 所示。

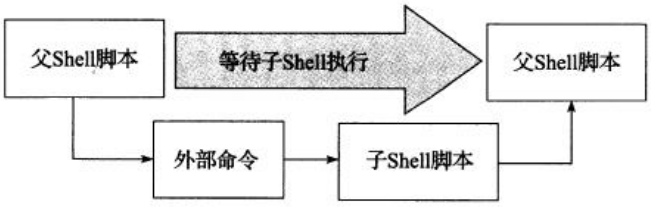


图 2-3　Shell 脚本的基本执行流程

特殊技巧：设置 Linux 的 crond 任务时，最好能在定时任务脚本中重新定义系统环境变量，否则，一些系统环境变量将不会被加载，这个问题需要注意！

Shell 脚本的执行通常可以采用以下几种方式。

1）bash script-name 或 sh script-name：这是当脚本文件本身没有可执行权限（即文件权限属性 x 位为 – 号）时常使用的方法，或者脚本文件开头没有指定解释器时需要使用的方法。**这也是老男孩推荐使用的方法。**

2）path/script-name 或 ./script-name：指在当前路径下执行脚本（脚本需要有执行权限），需要将脚本文件的权限先改为可执行（即文件权限属性加 x 位），具体方法为 chmod +x script-name。然后通过脚本绝对路径或相对路径就可以直接执行脚本了。

在企业生产环境中，不少运维人员在写完 Shell 脚本之后，由于忘记为该脚本设置执行权限，然后就直接应用了，结果导致脚本没有按照自己的意愿手动或定时执行，对于这一点，避免出现该问题的方法就是用第 1 种方法替代第 2 种。

3）source script-name 或 . script-name：这种方法通常是使用 source 或“.”（点号）读入或加载指定的 Shell 脚本文件（如 san.sh），然后，依次执行指定的 Shell 脚本文件 san.sh 中的所有语句。这些语句将在当前父 Shell 脚本 father.sh 进程中运行（其他几种模式都会启动新的进程执行子脚本）。因此，使用 source 或“.”可以将 san.sh 自身脚本中的变量值或函数等的返回值传递到当前父 Shell 脚本 father.sh 中使用。这是它和其他几种方法最大的区别，也是值得读者特别注意的地方。

source 或“.”命令的功能是：在当前 Shell 中执行 source 或“.”加载并执行的相关脚本文件中的命令及语句，而不是产生一个子 Shell 来执行文件中的命令。注意“.”和后面的脚本名之间要有空格。

如果读者学过 PHP 开发就会明白，source 或“.”相当于 include 的功能。HTTP 服务软件 Apache、Nginx 等配置文件里都支持这样的用法。

4）sh<script-name 或 cat scripts-name|sh：同样适用于 bash，不过这种用法不是很常见，但有时也可以有出奇制胜的效果，例如：不用循环语句来实现精简开机自启动服务的案例，就是通过将所有字符串拼接为命令的形式，然后经由管道交给 bash 操作的案例（见《跟老男孩学习 Linux 运维：Web 集群实战》第 3 章）。

范例 2-4：创建模拟脚本 test.sh，并输入如下内容。

```
[oldboy@oldboy ~]$ cat >test.sh #<== 编辑 test.sh 脚本文件。
echo 'I am oldboy'
```

输入“echo 'I am oldboy'”内容后按回车键，然后再按 Ctrl+d 组合键结束编辑。此操作为特殊编辑方法，这里是作为 cat 用法的扩展知识（通过使用来记忆是个好习惯）。

现在使用第 1 种方法实践，命令如下：

```
[oldboy@oldboy ~]$ cat test.sh
echo 'I am oldboy'
[oldboy@oldboy ~]$ sh test.sh    #<== 使用第 1 种方式的 sh 命令执行 test.sh 脚本文件。
I am oldboy
[oldboy@oldboy ~]$ bash test.sh #<== 使用第 1 种方式的 bash 命令执行 test.sh 脚本文件。
I am oldboy
```

这里使用第 1 种方法的 bash 和 sh，均可以执行脚本并得到预期的结果。

使用第 2 种方法实践，命令如下：

```
[oldboy@oldboy ~]$ ls -l test.sh
-rw-rw-r-- 1 oldboy oldboy 19 Apr 30 02:46 test.sh
[oldboy@oldboy ~]$ ./test.sh    #<== 使用第 2 种方式“./”在当前目录下执行 test.sh
脚本文件，细心的读者可以发现，这个地方无法自动补全，这是因为没有权限所导致的。
-bash: ./test.sh: Permission denied    #<== 提示：强制执行会提示权限拒绝，此处是
                                            因为没有执行权限。
```

虽然没有权限的 test.sh 脚本不能直接被执行，但是可以用 source 或“.”（点号）来执行，如下：

```
[oldboy@oldboy ~]$ . test.sh          #<== 请注意开头的“.”后面有空格。
I am oldboy
[oldboy@oldboy ~]$ source test.sh
I am oldboy
```

提示：“.”或 source 命令的功能相同，都是读入脚本并执行脚本。

给 test.sh 添加可执行权限，命令如下：

```
[oldboy@oldboy ~]$ chmod u+x test.sh
[oldboy@oldboy ~]$ ./test.sh
I am oldboy
```

可以看到，给 test.sh 加完可执行权限后就能执行了。前面也提到了，这种方法在使用前每次都需要给定执行权限，但容易被忘记，且多了一些步骤，增加了复杂性。

使用第 3 种方法实践时，会将 source 或“.”执行的脚本中的变量值传递到当前的 Shell 中，如下：

```
[oldboy@oldboy ~]$ echo 'userdir=`pwd`' >testsource.sh #<== 第一行的内容通常用
                                                          echo 处理更方便
[oldboy@oldboy ~]$ cat testsource.sh
userdir=`pwd`  #<== 定义了一个命令变量，内容是打印当前路径。注意，打印命令用反引号
[oldboy@oldboy ~]$ sh testsource.sh    #<== 采用 sh 命令执行脚本
```

```
[oldboy@oldboy ~]$ echo $userdir
                    #<== 此处为空，并没有出现当前路径 /home/oldboy 的输出，这是为什么
```

根据上面的例子可以发现，通过 sh 或 bash 命令执行过的脚本，若在脚本结束之后，在当前 Shell 窗口中查看 userdir 变量的值，会发现值是空的。现在以同样的步骤改用 source 或“.”执行，然后再看看 userdir 变量的值：

```
[oldboy@oldboy ~]$ source testsource.sh  #<== 采用 source 执行同一脚本
[oldboy@oldboy ~]$ echo $userdir
/home/oldboy #<== 此处输出了当前路径 /home/oldboy，这又是为什么呢
```

来了解一下系统 NFS 服务的脚本是如何使用“.”的：

```
# Source function library.
. /etc/init.d/functions                    #<== 通过 "." 加载系统函数库 functions
```

说明：操作系统及服务自带的脚本是我们学习的标杆和参考（虽然有时感觉这些脚本也不是十分规范）。

结论：通过 source 或“.”加载执行过的脚本，由于是在当前 Shell 中执行脚本，因此在脚本结束之后，脚本中的变量（包括函数）值在当前 Shell 中依然存在，而 sh 和 bash 执行脚本都会启动新的子 Shell 执行，执行完后退回到父 Shell。因此，变量（包括函数）值等无法保留。在进行 Shell 脚本开发时，如果脚本中有引用或执行其他脚本的内容或配置文件的需求时，最好用“.”或 source 先加载该脚本或配置文件，处理完之后，再将它们加载到脚本的下面，就可以调用 source 加载的脚本及配置文件中的变量及函数等内容了。

以下采用第 4 种方法来实践：

```
[root@oldboy ~]# ls -l oldboy.sh
-rw-r--r--. 1 root root 28 Nov 18 15:52 oldboy.sh
[root@oldboy ~]# cat oldboy.sh
echo "I am oldboy teacher."
[root@oldboy ~]# sh<oldboy.sh              #<== 尽量不要使用这种方法。
I am oldboy teacher.
[root@oldboy ~]# cat oldboy.sh|bash        #<== 这种方法在命令行拼接字符串命令后，需
                                               要执行时就会用到
I am oldboy teacher.
```

提示：代码中提到的两种执行方法相当于 sh scripts-name，效率很高，但是初学者用得少。

范例 2-5：已知如下命令及返回结果，请问 echo $user 的返回的结果为（ ）。

```
[oldboy@test ~]$ cat test.sh
user=`whoami`
[oldboy@test ~]$ sh test.sh
[oldboy@test ~]$ echo $user
```

参考的选择项如下：

a）当前用户

b）oldboy

c）空（无内容输出）

这是某互联网公司 Linux 运维职位的笔试题。在这里 c）是正确答案，原因前面已经讲过了，即使用 sh 执行脚本会导致当前 Shell 无法获得变量值。

通过上述面试题可得出如下的结论：

- 儿子 Shell 脚本会直接继承父亲 Shell 脚本的变量、函数（就好像是儿子随父亲姓，基因也会继承父亲的）等，反之则不可以。
- 如果希望反过来继承（就好像是让父亲随儿子姓，让父亲的基因也继承儿子的），就要用 source 或“.”在父亲 Shell 脚本中事先加载儿子 Shell 脚本。

2.6.3　Shell 脚本开发的基本规范及习惯

Shell 脚本的开发规范及习惯非常重要，虽然这些规范不是必须要遵守的，但有了好的规范和习惯，可以大大提升开发效率，并能在后期降低对脚本的维护成本。当多人协作开发时，大家有一个互相遵守的规范就显得更重要了。即使只是一个人开发，最好也采取一套固定的规范，这样脚本将会更易读、更易于后期维护，最重要的是要让自己养成一个一出手就很专业和规范的习惯。下面来看看有哪些规范，这些规范在第 14 章也会提及，以便于大家进一步巩固。

1）Shell 脚本的第一行是指定脚本解释器，通常为：

```
#!/bin/bash
或
#!/bin/sh
```

2）Shell 脚本的开头会加版本、版权等信息：

```
# Date:    16:29 2012-3-30
# Author:  Created by oldboy
# Blog:http://oldboy.blog.51cto.com
# Description: This scripts function is.....
# Version: 1.1
```

说明：以上两点在 Linux 系统场景中不是必需的，只属于优秀规范和习惯，第 16 章有自动加载此内容的方法，读者可以做进一步了解。

可修改“~/.vimrc”配置文件配置 vim 编辑文件时自动加上以上信息的功能。

3）在 Shell 脚本中尽量不用中文（不限于注释）。

尽量用英文注释，防止本机或切换系统环境后中文乱码的困扰。如果非要加中文，请根据自身的客户端对系统进行字符集调整，如：export LANG="zh_CN.UTF-8"，并在脚本中，重新定义字符集设置，和系统保持一致。

4）Shell 脚本的命名应以 .sh 为扩展名。

例如：script-name.sh

5）Shell 脚本应存放在固定的路径下。

例如：/server/scripts

以下则是 Shell 脚本代码书写的良好习惯。

1）成对的符号应尽量一次性写出来，然后退格在符号里增加内容，以防止遗漏。这些成对的符号包括：

```
{}、[]、''、``、""
```

说明：这部分也可以配置 .vimrc 实现自动添加，但是老男孩不推荐这样做，因为养成良好的习惯很重要。

2）中括号（[]）两端至少要有 1 个空格，因此，键入中括号时即可留出空格 [ ]，然后再退格键入中间的内容，并确保两端都至少有一个空格，即先键入一对中括号，然后退 1 格，输入两个空格，再退 1 格，双中括号（[[]]）的写法也是如此。

3）对于流程控制语句，应一次性将格式写完，再添加内容。

比如，一次性完成 if 语句的格式，应为：

```
if 条件内容
  then
    内容
fi
```

一次性完成 for 循环语句的格式，应为：

```
for
do
    内容
done
```

提示：while 和 until，case 等语句也是一样。

4）通过缩进让代码更易读，比如：

```
if 条件内容
  then
    内容
fi
```

5）对于常规变量的字符串定义变量值应加双引号，并且等号前后不能有空格，需要强引用的（指所见即所得的字符引用），则用单引号（''），如果是命令的引用，则用反引号（``）。例如：

```
OLDBOY_FILE="test.txt"
```

6）脚本中的单引号、双引号及反引号必须为英文状态下的符号，其实所有的 Linux 字符及符号都应该是英文状态下的符号，这点需要特别注意。

说明：好的习惯可以让我们避免很多不必要的麻烦，提升工作效率。

有关 Shell 开发规范及习惯的更多内容，感兴趣的读者可参考本书的第 14 章。很多开发习惯也可以通过配置 vim 的功能来实现，例如实现自动缩进、自动补全成对符号、自动加入起始解释器及版权信息等，这部分的知识可参考本书第 16 章。对于一些开发规范和习惯，在新手入门学习期间，我们不建议将其搞得太傻瓜化、智能化，这会让我们产生惰性，所以有关 Shell 的开发规范及习惯的知识放在第 14 章来讲解，有关搭建高效的 Shell 开发环境的知识放在第 16 章讲解！

第3章

Shell 变量的核心基础知识与实践

3.1 什么是 Shell 变量

1. 什么是变量

在小学或初中时，我们开始接触**数学方程式**，例如：**已知** $x=1$，$y=x+1$，那么 y 等于多少？

在上述问题中，等号左边的 x 和 y 当时被称为未知数，但在 Shell 编程里它们是变量名，等号右边的 1 和 $x+1$ 则是变量的内容（变量的值）。注意，这里的等号符号被称为赋值，而不是等号。

通过上面的例子可以得出一个变量概念的小结论：简单地说，变量就是用一个固定的字符串（也可能是字符、数字等的组合）代替更多、更复杂的内容，该内容里可能还会包含变量、路径、字符串等其他的内容。

变量是暂时存储数据的地方及数据标记，所存储的数据存在于内存空间中，通过正确地调用内存空间中变量的名字就可以取出与变量对应的数据。使用变量的最大好处就是使程序开发更为方便，当然，在编程中使用变量也是必须的，否则就很难完成相关的程序开发工作。

下面是定义变量和打印变量的示例：

```
[root@oldboy ~]# oldboy="I am oldboy"    #<== 定义变量，名字为 oldboy，对应的内容
                                              为“I am oldboy”。
[root@oldboy ~]# echo $oldboy            #<== 打印变量的值。
I am oldboy
```

变量的赋值方式为：先写变量名称，紧接着是“=”这个字符，最后是值，中间无任何空格，通过 echo 命令加上 $oldboy 即可输出 oldboy 变量的值，变量的内容一般要加双引号，以防止出错，特别是当值里的内容之间有空格时。

2. Shell 变量的特性

默认情况下，在 bash Shell 中是不会区分变量类型的，例如：常见的变量类型为整数、字符串、小数等。这和其他强类型语言（例如：Java/C 语言）是有区别的，当然，如果需要指定 Shell 变量的类型，也可以使用 declare 显示定义变量的类型，但在一般情况下没有这个需求，Shell 开发者在开发脚本时需要自行注意 Shell 脚本中变量的类型，这对新手来说是个重点也是个难点，别害怕，跟着老男孩走，一切都不是事。

3. 变量类型

变量可分为两类：环境变量（全局变量）和普通变量（局部变量）。

环境变量也可称为全局变量，可以在创建它们的 Shell 及其派生出来的任意子进程 Shell 中使用，环境变量又可分为自定义环境变量和 bash 内置的环境变量。

普通变量也可称为局部变量，只能在创建它们的 Shell 函数或 Shell 脚本中使用。普通变量一般由开发者在开发脚本程序时创建。

3.2　环境变量

环境变量一般是指用 export 内置命令导出的变量，用于定义 Shell 的运行环境，保证 Shell 命令的正确执行。Shell 通过环境变量来确定**登录用户名、命令路径、终端类型、登录目录**等，所有的环境变量都是系统全局变量，可用于所有子进程中，这包括编辑器、Shell 脚本和各类应用。

环境变量可以在命令行中设置和创建，但用户退出命令行时这些变量值就会丢失，因此，如果希望永久保存环境变量，可在用户家目录下的 .bash_profile 或 .bashrc（非用户登录模式特有，例如远程 SSH）文件中，或者全局配置 /etc/bashrc（非用户登录模式特有，例如远程 SSH）或 /etc/profile 文件中定义。在将环境变量放入上述的文件中后，每次用户登录时这些变量都将被初始化。

按照系统规范，所有**环境变量的名字均采用大写形式**。在将环境变量应用于用户进程程序之前，都应该用 export 命令导出定义，例如：正确的环境变量定义方法为 export OLDGIRL=1。

有一些环境变量，比如 HOME、PATH、SHELL、UID、USER 等，在用户登录之前就已经被 /bin/login 程序设置好了。通常环境变量被定义并保存在用户家目录下的 .bash_profile 文件或全局的配置文件 /etc/profile 中，具体的环境变量说明参见表 3-1。

表 3-1 部分 bash 环境变量展示（执行 env 命令后获得）

变量名	含 义
_=	上一条命令的最后一个参数
BASH=/bin/bash	调用 bash 实例时使用的全路径名
BASH_VERSINFO=（[0]="3"[1]="2"[2]="25"[3]="1"[4]="release"[5]="x86_64-redhat-linux-gnu"）	使用 2.0 以上版本时，展开为版本信息
BASH_VERSION='3.2.25(1)-release'	当前 bash 实例的版本号
COLORS=/etc/DIR_COLORS	颜色变量
COLUMNS=132	设置该变量，就给 Shell 编辑模式和选择的命令定义了编辑窗口的宽度
DIRSTACK=()	代表目录栈的当前内容
EUID=0	在 Shell 启动时被初始化的当前用户的有效 ID
GROUPS=()	当前用户所属的组
HISTFILE=/root/.bash_history	历史记录文件的全路径
HISTFILESIZE=50	历史文件能包含的最大行数
HISTSIZE=50	记录在命令行历史文件中的命令行数
HOME=/root	当前用户家目录
HOSTNAME=oldboy	当前主机名称
HOSTTYPE=x86_64	当前操作系统类型
IFS=$'\t\n'	内部字段分隔符，一般是空格符、制表符和换行符，用于划分由命令替换、循环结构中的表和所读取的输入产生的词的字段
INPUTRC=/etc/inputrc	readline 启动文件的文件名，取代默认的~/.inputrc
JAVA_HOME=/application/jdk1.6.0_10	JAVA HOME 环境变量
LANG=zh_CN.UTF-8	字符集
LOGNAME=root	登录用户名称
MACHTYPE=x86_64-redhat-linux-gnu	包含一个描述正在运行 bash 的系统串
MAILCHECK=60	这个参数定义 Shell 将隔多长时间（以秒为单位）检查一次由参数 MAILPATH 或 MAILFILE 指定的文件，看看是否有邮件到达。默认值为 600s
MAIL=/var/spool/mail/root	邮件全路径
OLDPWD=/root	前一个当前工作目录
OPTIND=1	下一个由 getopts 内置命令处理的参数的序号
OSTYPE=linux-gnu	自动设置成一个串，该串描述正在运行 bash 的操作系统，默认值由系统来决定
PATH=/usr/lib64/qt-3.3/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/bin:/server/script/shangxian:/root/bin	全局 PATH 路径，命令搜索路径。一个由冒号分隔的目录列表，Shell 用它来搜索命令。默认路径由系统来决定，并且由安装 bash 的管理员来设置

（续）

变量名	含　义
PIPESTATUS=([0]="0"[1]="0")	一个数组，包含一列最近在管道执行的前台作业的进程退出的状态值
PPID=1112	父进程的进程 ID
PS1='[\u@\h \W]\$ '	主提示符串，默认值是 $
PS2='>'	次提示符串，默认值是 >
PS4='+'	当开启追踪时使用的是调试提示符串，默认值是 +，追踪可用 set –x 开启
PWD=/home	当前用户家目录
RESIN_HOME=/application/resin-3.1.6	这是通过 export 人为设置的环境变量，java 环境会用
SHELL=/bin/bash	登录 Shell 类型
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor	包含一列开启的 Shell 选项
SHLVL=1	每启动一个 bash 实例就将其加 1
TERM=vt100	终端设置
TMOUT=3600	退出前等待超时的秒数
UID=0	当前用户的 UID，在 Shell 启动时初始化
USER=root	当前用户的用户名，在 Shell 启动时初始化

在查看设置的变量时，有 3 个命令可以显示变量的值：set、env 和 declare（替代早期的 typeset）。set 命令输出所有的变量，包括全局变量和局部变量；env 命令只显示全局变量；declare 命令输出所有的变量、函数、整数和已经导出的变量。set -o 命令显示 bash Shell 的所有参数配置信息。

范例 3-1：set、env 和 declare 输出。

```
[root@oldboy ~]# env|tail
SHLVL=1
HOME=/root
LOGNAME=root
CVS_RSH=ssh
MODULESHOME=/usr/share/Modules
LESSOPEN=||/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
BASH_FUNC_module()=() {  eval `/usr/bin/modulecmd bash $*`
}
_=/bin/env
[root@oldboy ~]# declare|tail
_module_not_yet_loaded ()
{
    comm -23 <(_module_avail|sort) <(tr : '\n' <<<${LOADEDMODULES}|sort)
}
```

```
module ()
{
    eval `/usr/bin/modulecmd bash $*`
}
[root@oldboy ~]# set|tail
_module_not_yet_loaded ()
{
    comm -23 <(_module_avail|sort) <(tr : '\n' <<<${LOADEDMODULES}|sort)
}
module ()
{
    eval `/usr/bin/modulecmd bash $*`
}
[root@oldboy ~]# set -o|head
allexport       off
braceexpand     on
emacs           on
errexit         off
errtrace        off
functrace       off
hashall         on
histexpand      on
history         on
ignoreeof       off
```

3.2.1 自定义环境变量

1. 设置环境变量

如果想要设置环境变量，就要在给变量赋值之后或在设置变量时使用 export 命令，具体设置见下文的示例。其实，除了 export 命令，带 -x 选项的 declare 内置命令也可以完成同样的功能（注意：此处不要在变量名前面加 $）。

export 命令和 declare 命令的格式如下：

```
①export 变量名=value
②变量名=value ; export 变量名
③declare -x 变量名=value
```

提示：以上为设置环境变量的 3 种方法。

范例 3-2：定义环境变量并赋值的方法。

```
export NAME=oldboy
declare -x NAME=oldboy
NAME=oldboy ;export NAME
```

以下是自定义全局环境变量的示例：

```
[root@oldboy script]# cat /etc/profile|grep OLD
export OLDBOY='oldboy' #<== 编辑 /etc/profile，然后输出此行并保存
[root@oldboy script]# source /etc/profile      #<== 或 . /etc/profile 使其生效
[root@oldboy script]# echo $OLDBOY             #<== 在变量前加 $ 符号并打印变量值
oldboy
[root@oldboy script]# env|grep OLDBOY          #<== 查看定义结果
OLDBOY=oldboy
```

下面来看看让环境变量永久生效的常用设置文件。

（1）用户的环境变量配置

配置如下：

```
[root@oldboy scripts]# ls /root/.bashrc        #<== 推荐在此文件中优先设置
/root/.bashrc
[root@oldboy scripts]# ls /root/.bash_profile
/root/.bash_profile
```

提示：对于用户的环境变量设置，比较常见的是用户家目录下的 .bashrc 和 .bash_profile。

（2）全局环境变量的配置

常见的全局环境变量的配置文件如下：

```
/etc/profile
/etc/bashrc                                    #<== 推荐在此文件中优先设置
/etc/profile.d/
```

若要在登录后初始化或显示加载内容，则把脚本文件放在 /etc/profile.d/ 下即可（无须加执行权限）。

2. 设置登录提示的两种方式

第一种是在 /etc/motd 里增加提示的字符串，如下：

```
[root@oldboy ~]# cat /etc/motd                 #<== 文件里仅为字符串内容
welcome to oldboy linux Shell training.
```

登录后显示内容如下：

```
Last login : Fri Nov  7 15 :36 :56 2016 from 10.0.0.100
welcome to oldboy linux Shell training.
```

第二种是在 /etc/profile.d/ 下面增加如下脚本。

```
[root@oldboy ~]# cat /etc/profile.d/oldboy.sh #<== 这里是脚本的内容
```

```
echo " Here is oldboy training "
```

登录后显示的内容如下：

```
Last login : Fri Nov  7 15 :36 :56 2016 from 10.0.0.100
Here is oldboy training
```

以下是在生产场景下（在 Java 环境中），自定义环境变量的示例。

```
export JAVA_HOME=/application/jdk
export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH:$HOME/bin
export RESIN_HOME=/application/resin
```

提示：上述的环境变量设置通常放在 /etc/profile 全局环境变量里。
如果是写 Java 的脚本，那么最好是把上述 Java 环境配置放入脚本内重新定义，特别是作为定时任务执行的脚本。

3.2.2 显示与取消环境变量

1. 通过 echo 或 printf 命令打印环境变量

下面我们先来看看常见的系统环境变量。

- $HOME：用户登录时进入的目录。
- $UID：当前用户的 UID（用户标识），相当于 id-u。
- $PWD：当前工作目录的绝对路径名。
- $SHELL：当前 SHELL。
- $USER：当前用户。

.....

范例 3-3：通过 echo 和 printf 命令打印环境变量。

```
[oldboy@oldboy ~]$ echo $HOME
/home/oldboy
[oldboy@oldboy ~]$ echo $UID
504
[oldboy@oldboy ~]$ echo $PWD
/home/oldboy
[oldboy@oldboy ~]$ echo $SHELL
/bin/bash
[oldboy@oldboy ~]$ echo $USER
oldboy
[root@oldboy ~]# printf "$HOME\n"
#<==printf 是一个更复杂的格式化打印内容的工具，一般不需要
/root
```

提示：在写 Shell 脚本时可以直接使用系统默认的环境变量，一般情况下是不需要重新定义的，在使用定时任务等执行 Shell 脚本时建议在脚本中重新定义。

2. 用 env 或 set 显示默认的环境变量

用 env（printenv）显示默认环境变量的示例如下：

```
[oldboy@oldboy ~]$ env
HOSTNAME=oldboy
SHELL=/bin/bash
HISTSIZE=1000
LC_ALL=C
MAIL=/var/spool/mail/oldboy
PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/home/oldboy/bin
INPUTRC=/etc/inputrc
PWD=/home/oldboy
LANG=zh_cn.gb18030
SHLVL=1
HOME=/home/oldboy
LOGNAME=oldboy
中间和结尾省略若干代码
```

用 set 也可以显示环境变量（包括局部变量），如下：

```
[root@oldboy ~]# set
APACHEERR=hello
BASH=/bin/bash
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="1" [2]="2" [3]="1" [4]="release" [5]="x86_64-redhat-
linux-gnu")
BASH_VERSION='4.1.2(1)-release'
COLORS=/etc/DIR_COLORS
COLUMNS=103
中间和结尾省略若干代码
```

在 3.2.1 节还提到了一个相关的命令 declare，大家还记得吗？

3. 用 unset 消除本地变量和环境变量

用 unset 消除本地变量和环境变量的示例如下：

```
[oldboy@oldboy ~]$ echo $USER
oldboy
[oldboy@oldboy ~]$ unset USER
[oldboy@oldboy ~]$ echo $USER
```

```
#<== 此处为输出的空行
```

可以看到变量的内容显示为空。

环境变量的知识小结：

- 变量名通常要大写。
- 变量可以在自身的 Shell 及子 Shell 中使用。
- 常用 export 来定义环境变量。
- 执行 env 默认可以显示所有的环境变量名称及对应的值。
- 输出时用“$ 变量名”，取消时用“unset 变量名”。
- 书写 crond 定时任务时要注意，脚本要用到的环境变量最好先在所执行的 Shell 脚本中重新定义。
- 如果希望环境变量永久生效，则可以将其放在用户环境变量文件或全局环境变量文件里。

3.2.3 环境变量初始化与对应文件的生效顺序

在登录 Linux 系统并启动一个 bash shell 时，默认情况下 bash 会在若干个文件中查找环境变量的设置。这些文件可统称为系统环境文件。bash 检查的环境变量文件的情况取决于系统运行 Shell 的方式。系统运行 Shell 的方式一般有 3 种：

1）通过系统用户登录后默认运行的 Shell。

2）非登录交互式运行 Shell。

3）执行脚本运行非交互式 Shell。

当用户登录 Linux 系统时，Shell 会作为登录 Shell 启动。此时的登录 Shell 加载环境变量的顺序如图 3-1 所示。

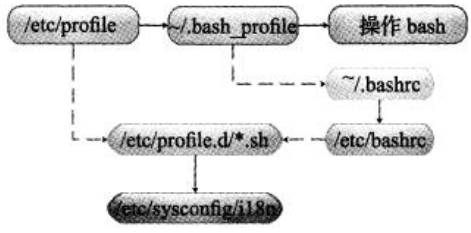


图 3-1 登录 Shell 读取环境变量文件的流程

用户登录系统后首先会加载 /etc/profile 全局环境变量文件，这是 Linux 系统上默认的 Shell 主环境变量文件。系统上每个用户登录都会加载这个文件。

当加载完 /etc/profile 文件后，才会执行 /etc/profile.d 目录下的脚本文件，这个目录下的脚本文件有很多，例如：系统的字符集设置（/etc/sysconfig/i18n）等，在后文开发跳板机案例里，我们也把脚本的起始加载放到这个目录下，以便用户登录后即刻运行脚本。

之后开始运行 $HOME/.bash_profile（用户环境变量文件），在这个文件中，又会去找 $HOME/.bashrc（用户环境变量文件），如果有，则执行，如果没有，则不执行。在 $HOME/.bashrc 文件中又会去找 /etc/bashrc（全局环境变量文件），如果有，则执行，如

果没有，则不执行。

如果用户的 Shell 不是登录时启动的（比如手动敲下 bash 时启动或者其他不需要输入密码的登录及远程 SSH 连接情况 ），那么这种非登录 Shell 只会加载 $HOME/.bashrc（用户环境变量文件），并会去找 /etc/bashrc（全局环境变量文件）。因此如果希望在非登录 Shell 下也可读到设置的环境变量等内容，就需要将变量设定等写入 $HOME/.bashrc 或者 /etc/bashrc，而不是 $HOME/.bash_profile 或 /etc/profile。

3.3　普通变量

3.3.1　定义本地变量

本地变量在用户当前 Shell 生存期的脚本中使用。例如，本地变量 oldboy 的取值为 bingbing，这个值只在用户当前 Shell 生存期中有意义。如果在 Shell 中启动另一个进程或退出，那么变量 oldboy 的值将会无效。

1. 普通变量定义

为普通变量的定义赋值，一般有以下 3 种写法：

```
变量名 =value        #<== 赋值时不加引号
变量名 ='value'      #<== 赋值时加单引号
变量名 ="value"      #<== 赋值时加双引号
```

2. 在 Shell 中定义变量名及为变量内容赋值的要求

变量名一般是由字母、数字、下划线组成的，可以以字母或下划线开头，例如：oldboy、oldboy123、oldboy_training。

变量的内容可以用单引号或双引号引起来，也可不加引号，但是这三者的含义是不同的，具体参见后文说明。

3. 普通变量的定义及输出的示例

范例 3-4：采用不同的方式对普通变量进行定义，并一一打印输出。

```
a=192.168.1.2
b='192.168.1.2'
c="192.168.1.2"
echo "a=$a"
echo "b=$b"
echo "c=${c}"
```

提示：

1）$ 变量名表示输出变量，可以用 $c 和 ${c} 两种用法。

2）请在命令行实践以上内容，然后看一看返回的结果有何不同。

思考：在命令行 Shell 下输入以上内容后会输出什么结果呢？请在看答案之前，先想一想上面 a、b、c 变量值的输出各是什么，最好自己实践一下。

答案：

```
a=192.168.1.2
b=192.168.1.2
c=192.168.1.2
```

可见，将连续的普通字符串的内容赋值给变量，不管用不用引号，或者不管用什么引号，它的内容是什么，打印变量时就会输出什么。

范例 3-5：接着上述范例的结果，再在 Linux 命令行下继续输入如下内容，想一想 a、b、c 的输出又各是什么结果？

```
a=192.168.1.2-$a
b='192.168.1.2-$a'
c="192.168.1.2-$a"
echo "a=$a"
echo "b=$b"
echo "c=${c}"
```

提示：建议先思考结果是什么，然后再在命令行实践以上内容，看看和思考的结果有何不同。

参考答案：

```
a=192.168.1.2-192.168.1.2
b=192.168.1.2-$a
c=192.168.1.2-192.168.1.2-192.168.1.2
```

4. 变量定义的基本技巧总结

这里以范例 3-5 为例：

```
a=192.168.1.2-$a
```

第一种定义 a 变量的方式是不加任何引号直接定义变量的内容，当内容为简单连续的数字、字符串、路径名时，可以这样用，例如：a=1，b=oldboy 等。不加引号时，值里有变量的会被解析后再输出，上述变量定义中因为 $a 的值被解析为 192.168.1.2（范例 3-3 执行的影响），因此新的 a 值就是 192.168.1.2-192.168.1.2。

```
b='192.168.1.2-$a'
```

第二种定义 b 变量的方式是通过单引号定义。这种定义方式的特点是：输出变量内容时单引号里是什么就输出什么，即使内容中有变量和命令（命令需要反引起来）也

会把它们原样输出。这种方式比较适合于定义显示纯字符串的情况，即不希望解析变量、命令等的场景，因此，对于这里的 b 的值，定义时看到的是什么就输出什么，即 192.168.1.2-$a。

```
c="192.168.1.2-$a"
```

第三种定义 c 变量的方式是通过双引号定义变量。这种定义方式的特点是：输出变量内容时引号里的变量及命令会经过解析后再输出内容，而不是把双引号中的变量名及命令（命令需要反引起来）原样输出。这种方式比较适合于字符串中附带有变量及命令且想将其解析后再输出的变量定义。

老男孩经验：

数字内容的变量定义可以不加引号，其他没有特别要求的字符串等定义最好都加上双引号，如果真的需要原样输出就加单引号，定义变量加双引号是最常见的使用场景。

---

5. 把一个命令的结果作为变量的内容赋值的方法

对需要获取命令结果的变量内容赋值的常见方法有两种：

```
变量名 =`ls`          #<== 把命令用反引号引起来，不推荐使用这种方法，因为容易和单引号混淆
变量名 =$(ls)         #<== 把命令用 $() 括起来，推荐使用这种方法
```

范例 3-6：用两种方法把命令的结果赋值给变量。

```
[oldboy@oldboy ~]$ ls
test.sh
[oldboy@oldboy ~]$ CMD=`ls`       #<== 其中 `` 为键盘上 Tab 键上面的那个键输出的字符
[oldboy@oldboy ~]$ echo $CMD
test.sh
[oldboy@oldboy ~]$ CMD1=$(pwd)
[oldboy@oldboy ~]$ echo $CMD1
/home/oldboy                      #<== 打印当前用户所在的目录
```

提示：生产场景中把命令的结果作为变量的内容进行赋值的方法在脚本开发时很常见。

范例 3-7：按天打包网站的站点目录程序，生成不同的文件名（此为企业实战案例）。

```
[root@oldboy scripts]# CMD=$(date +%F)  #<== 将当前日期（格式为 2016-09-10）赋值
                                            给 CMD 变量
[root@oldboy scripts]# echo $CMD                          #<== 输出变量的值
2016-09-10
[root@oldboy scripts]# echo $(date +%F).tar.gz #<== 直接输出时间命令的结果
```

```
2016-09-10.tar.gz
[root@oldboy scripts]# echo `date +%F`.tar.gz
2016-09-10.tar.gz
[root@oldboy scripts]# tar zcf etc_$(date +%F).tar.gz /etc
                                                    #<== 将时间作为压缩包名打包
tar: 从成员名中删除开头的“/”
tar: 从硬连接目标中删除开头的“/”
[root@oldboy scripts]# ls -l etc_2016-09-10.tar.gz   #<== 打包结果，包名中包含
                                                          有当前日期
-rw-r--r-- 1 root root 9700163 9月  10 18:39 etc_2016-09-10.tar.gz
[root@oldboy scripts]# H=$(uname -n)    #<== 获取主机名并赋值给 H 变量
[root@oldboy scripts]# echo $H
oldboy
[root@oldboy scripts]# tar zcf $H.tar.gz /etc/services   #<== 将主机名作为压缩包名
                                                          打包文件
tar: 从成员名中删除开头的“/”
[root@oldboy scripts]# ls -l oldboy.tar.gz      #<== 打包结果，包名中包含有主机名
-rw-r--r-- 1 root root 127303 9月  10 18:40 oldboy.tar.gz
```

局部（普通）变量定义及赋值的经验小结

常规普通变量定义：

- 若变量内容为连续的数字或字符串，赋值时，变量内容两边可以不加引号，例如 a=123。
- 变量的内容很多时，如果有空格且希望解析内容中的变量，就加双引号，例如 a="/etc/rc.local $USER"，此时输出变量会对内容中的 $USER 进行解析然后再输出。
- 希望原样输出变量中的内容时就用单引号引起内容进行赋值，例如：a='$USER'。

希望变量的内容是命令的解析结果的定义及赋值如下：

- 要使用反引号将赋值的命令括起来，例如：a=`ls`；或者用 $() 括起来，例如：a=$(ls)。

变量的输出方法如下：

- 使用“$ 变量名”即可输出变量的内容，常用“echo $ 变量名”的方式，也可用 printf 代替 echo 输出更复杂的格式内容。

变量定义的技巧及注意事项：

- 注意命令变量内容前后的字符 ``（此字符为键盘 Tab 键上面的那个反引号，不是单引号），例如：“CMD=`ls`”。
- 在变量名前加 $ 可以取得该变量的值，使用 echo 或 printf 命令可以显示变量的值，$A 和 ${A} 的写法不同，但效果是一样的。
- 用 echo 等命令输出变量的时候，也可用单引号、双引号、反引号，例如：echo $A、echo "$A"、echo '$A'，它们的用法和前面变量内容定义的总结是一致的。

- $dbname_tname，当变量后面连接有其他字符的时候，必须给变量加上大括号{}，例如：$dbname_tname 就要改成 ${dbname}_tname。

有关上述变量问题输出的小故事

故事 1：老男孩正在给面授班讲课，发了一段内容，结果引起群里网络班学员的强烈反应，下面是对话内容。

老男孩（31333741）12:42:54

金庸新著

XXX- 学员　12:43:39

老师，金庸又写啥小说了？ #<== 看到了吧，这引起了误解

老男孩（31333741）　12:42:54

这是一本小说，作者为**金庸新**，而非金庸，但是给读者造成的感觉是{金庸}新著。

$dbname_tname 变量就类似于这个金庸新著，会引起歧义，因此要改成 ${dbname}_tname，Shell 就会认为只有 dbname 是变量了。

老男孩（31333741）　12:44:45

如果真的是金庸新著，就要像这样用大括号分隔开，${金庸}新著。

故事 2：老男孩运维班 20 期的李同学在他媳妇看电视剧时发现了这个金庸新著，于是他将电视剧停下来，还截图发给了我。

可见形象的比喻学习对学生的影响非常深远！养成将所有字符串变量用大括号括起来的习惯，在编程时将会减少很多问题。不过老男孩也并不是一直都这么做，因为多输入内容会造成效率不高，但是金庸新著的问题确实要多注意。

3.3.2　变量定义及变量输出说明

有关 Shell 变量定义、赋值及变量输出加单引号、双引号、反引号与不加引号的简

要说明见表 3-2。

表 3-2 单引号、双引号、反引号与不加引号的知识说明

名 称	解 释
单引号	所见即所得，即输出时会将单引号内的所有内容都原样输出，或者描述为单引号里面看到的是什么就会输出什么，这称为强引用
双引号（默认）	输出双引号内的所有内容；如果内容中有命令（要反引下）、变量、特殊转义符等，会先把变量、命令、转义字符解析出结果，然后再输出最终内容，推荐使用，这称为弱引用
无引号	赋值时，如果变量内容中有空格，则会造成赋值不完整。而在输出内容时，会将含有空格的字符串视为一个整体来输出；如果内容中有命令（要反引下）、变量等，则会先把变量、命令解析出结果，然后输出最终内容；如果字符串中带有空格等特殊字符，则有可能无法完整地输出，因此需要改加双引号。一般连续的字符串、数字、路径等可以不加任何引号进行赋值和输出，不过最好是用双引号替代无引号的情况，特别是对变量赋值时
反引号	`` 一般用于引用命令，执行的时候命令会被执行，相当于 $()，赋值和输出都要用 `` 将命令引起来

提示：这里仅为 Linux Shell 下的结论，对于 awk 语言会有点特别，有关情况下文会有测试讲解。

老男孩的建议：

- 在脚本中定义普通字符串变量时，应尽量把变量的内容用双引号括起来。
- 单纯数字的变量内容可以不加引号。
- 希望变量的内容原样输出时需要加单引号。
- 希望变量值引用命令并获取命令的结果时就用反引号或 $()。

以下是单引号、双引号与不加引号的实战演示。

范例 3-8：对由反引号引起来的 `date` 命令或 $（date）进行测试。

```
[root@oldboy ~]# echo 'today is date'
today is date     #<== 单引号引起内容时，你看到什么就会显示什么
[root@oldboy ~]# echo 'today is `date`'
today is `date`   #<== 单引号引起内容时，你看到什么就会显示什么，内容中有命令时即使通过
                       反引号引起来也没有用
[root@oldboy ~]# echo "today is date"
today is date
[root@oldboy ~]# echo "today is `date`"
today is Sun Sep 11 15:12:30 CST 2016
#<== 对输出内容加双引号时，如果里面是变量或用反引号引起来的命令，则会先把变量或命令解析成
     具体内容再显示
[root@oldboy ~]# echo "today is $(date)"   #<== $() 的功能和反引号 `` 相同
today is Sun Sep 11 15:12:30 CST 2016
[root@oldboy ~]# echo today is $(date)     #<== 带空格的内容不加引号，同样可以正确
                                               地输出，但不建议这么做
```

```
today is Sun Sep 11 15:12:30 CST 2016
#<== 对于连续的字符串等内容输出一般可以不加引号，但加双引号比较保险，所以推荐使用。
```

范例 3-9：变量定义后，在调用变量输出打印时加引号测试。

```
[root@oldboy ~]# OLDBOY=testchars                    #<== 创建一个不带引号的变量并赋值。
[root@oldboy ~]# echo $OLDBOY                        #<== 不加引号，显示变量解析后的内容。
testchars
[root@oldboy ~]# echo '$OLDBOY'                      #<== 加单引号，显示变量本身。
$OLDBOY
[root@oldboy ~]# echo "$OLDBOY"                      #<== 加双引号，显示变量内容，引号内可以
                                                         是变量、字符串等。
testchars
```

范例 3-10：使用三剑客命令中的 grep 过滤字符串时给过滤的内容加引号。

```
[root@oldboy ~]# cat grep.log                        #<== 待测试的内容。
testchars
oldboy
[root@oldboy ~]# grep "$OLDBOY" grep.log             #<== 将 $OLDBOY 解析为结果后进行过滤。
testchars
[root@oldboy ~]# grep '$OLDBOY' grep.log             #<== 将 $OLDBOY 本身作为结果进行过滤。
[root@oldboy ~]# grep $OLDBOY grep.log               #<== 将 $OLDBOY 解析为结果后进行过滤，同
双引号的情况，但不建议这样使用，没有特殊需要时应一律加双引号。
testchars
```

范例 3-11：使用 awk 调用 Shell 中的变量，分别针对加引号、不加引号等情况进行测试。

首先在给 Shell 中的变量赋值时不加任何引号，这里使用 awk 输出测试结果。

```
[root@oldboy ~]# ETT=123                             #<== 定义变量 ETT 并赋值 123，没加引号。
[root@oldboy ~]# awk 'BEGIN {print "$ETT"}'
#<== 加双引号引用 $ETT，却只输出了本身，这个就不符合前文的结论了。
$ETT
[root@oldboy ~]# awk 'BEGIN {print $ETT}'
          #<== 不加引号的 $ETT，又输出了空的结果，这个就不符合前文的结论了。
[root@oldboy ~]# awk 'BEGIN {print '$ETT'}'
#<== 加单引号引用 $ETT，又输出了解析后的结果，这个就不符合前文的结论了。
123
 [root@oldboy ~]# awk 'BEGIN {print "'$ETT'"}'
123
```

以上的结果正好与前面的结论相反，这是 awk 调用 Shell 变量的特殊用法。

然后在给 Shell 中的变量赋值时加单引号，同样使用 awk 输出测试结果。

```
[root@oldboy ~]# ETT='oldgirl'   #<== 定义变量 ETT 并赋值 oldgirl，加单引号。
[root@oldboy ~]# awk 'BEGIN {print "$ETT"}'
$ETT                                 #<== 加双引号引用 $ETT，则输出本身。
[root@oldboy ~]# awk 'BEGIN {print $ETT}'
#<== 对 $ETT 不加引号，则输出空的结果。
```

```
[root@oldboy ~]# awk 'BEGIN {print '$ETT'}'
#<== 加单引号引用 $ETT，也是输出空的结果，这个和前文的不加引号定义、赋值的结果又不一样。
[root@oldboy ~]# awk 'BEGIN {print "'$ETT'"}'
oldgirl    #<== 在单引号外再加一层双引号引用 $ETT，则输出解析后的结果。
```

以下在给 Shell 中的变量赋值时加双引号，也使用 awk 输出测试结果。

```
[root@oldboy ~]# ETT="tingting" #<== 定义变量 ETT 并赋值 tingting，加双引号，这个
                                    测试结果同单引号的情况。
[root@oldboy ~]# awk 'BEGIN {print "$ETT"}'     #<== 加双引号引用 $ETT，会输出本身。
$ETT
[root@oldboy ~]# awk 'BEGIN {print $ETT}'       #<== 不加引号的 $ETT，会输出空的结果。

[root@oldboy ~]# awk 'BEGIN {print '$ETT'}'     #<== 加单引号的 $ETT，会输出空的结果。

[root@oldboy ~]# awk 'BEGIN {print "'$ETT'"}'   #<== 在单引号外部再加双引号引用 $ETT，
                                                    会输出正确结果。
tingting
```

以下在给 Shell 中的变量赋值时加反引号引用命令，同样使用 awk 输出测试结果。

```
[root@oldboy ~]# ETT=`pwd`  #<== 定义变量 ETT 并赋值 pwd 命令，加反引号，这个测试结果更特殊。
[root@oldboy ~]# echo $ETT
/root
[root@oldboy ~]# awk 'BEGIN {print "$ETT"}'     #<== 加双引号引用 $ETT，会输出本身。
$ETT
[root@oldboy ~]# awk 'BEGIN {print $ETT}'       #<== 不加引号的 $ETT，会输出空的结果。

[root@oldboy ~]# awk 'BEGIN {print '$ETT'}'     #<== 单引号引用 $ETT，看起来输出了结
                                                    果，却是报错，和外层单引号冲突了。
awk: BEGIN {print /root}
awk:                ^ unterminated regexp
awk: cmd. line:1: BEGIN {print /root}
awk: cmd. line:1:                       ^ unexpected newline or end of string
[root@oldboy ~]# awk 'BEGIN {print "'$ETT'"}'   #<== 在单引号外部再加双引号引用 $ETT，
                                                    会输出正确结果。
/root
```

根据上述范例整理的测试结果见表 3-3，供读者参考。

表 3-3　测试结果

awk \ ETT	ETT=123	ETT='oldgirl'	ETT="tingting"	ETT=`pwd`
awk 加双引号	本身	本身	本身	本身
awk 不加引号	空	空	空	空
awk 加单引号	正确输出	空	空	报语法错
awk 加单引号后再同时加双引号	正确输出	正确输出	正确输出	正确输出

结论：不管变量如何定义、赋值，除了加单引号以外，利用 awk 直接获取变量的输出，结果都是一样的，因此，在 awk 取用 Shell 变量时，我们更多地还是喜欢先用 echo 加符号输出变量，然后通过管道给 awk，进而控制变量的输出结果。举例如下：

```
[root@oldboy ~]# ETT="oldgirl"                    #<== 最常规的赋值语法
[root@oldboy ~]# echo "$ETT"|awk '{print $0}'     #<== 用双引号引用 $ETT
oldgirl
[root@oldboy ~]# echo '$ETT'|awk '{print $0}'     #<== 用单引号引用 $ETT
$ETT
[root@oldboy ~]# echo $ETT|awk '{print $0}'       #<== 不加引号引用 $ETT
oldgirl
[root@oldboy ~]# ETT=`pwd`                        #<== 命令赋值的语法
[root@oldboy ~]# echo "$ETT"|awk '{print $0}'
/root
[root@oldboy ~]# echo '$ETT'|awk '{print $0}'
$ETT
[root@oldboy ~]# echo $ETT|awk '{print $0}'
/root
```

这就符合前面给出的普通情况的结论了。不过，这个例子特殊了一点，有关 awk 调用 Shell 变量的详情，还可以参考老男孩的博客“一道实用 Linux 运维问题的 9 种 Shell 解答 http://oldboy.blog.51cto.com/2561410/760192”。

范例 3-12：通过 sed 指定变量关键字过滤。

```
[root@oldboy ~]# cat sed.log
testchars
oldboy
[root@oldboy ~]# sed -n /"$OLDBOY"/p sed.log    #<== 加双引号测试
testchars
[root@oldboy ~]# sed -n /$OLDBOY/p sed.log      #<== 不加引号测试
testchars
[root@oldboy ~]# sed -n /'$OLDBOY'/p sed.log    #<== 加单引号测试
  #<== 输出本身，但是文件里没有本身匹配的字符串，因此输出为空
```

注意：sed 和 grep 的测试和前面结论是相符的，唯有 awk 有些特殊。

提示：上述内容不需要特意去记，在使用时测试一下就会明白。

关于自定义普通字符串变量的建议

1）内容是纯数字、简单的连续字符（内容中不带任何空格）时，定义时可以不加任何引号，例如：

```
a.OldboyAge=33
b.NETWORKING=yes
```

2）没有特殊情况时，字符串一律用双引号定义赋值，特别是多个字符串中间有空格时，例如：

```
a.NFSD_MODULE="no load"
b.MyName="Oldboy is a handsome boy."
```

3）当变量里的内容需要原样输出时，要用单引号（''），这样的需求极少，例如：

```
a.OLDBOY_NAME='OLDBOY'
```

3.4 变量定义技巧总结

可以多学习和模仿操作系统自带的 /etc/init.d/functions 函数库脚本的定义思路，多学习 Linux 系统脚本中的定义，有经验的读者最终应形成一套适合自己的规范和习惯。

（1）变量名及变量内容定义小结

- 变量名只能为字母、数字或下划线，只能以字母或下划线开头。
- 变量名的定义要有一定的规范，并且要见名知意。

示例：

```
OldboyAge=1          #<== 每个单词的首字母大写的写法
oldboy_age=1         #<== 单词之间用 "_" 的写法
oldboyAgeSex=1       #<== 驼峰语法：首个单词的首字母小写，其余单词首字母大写
OLDBOYAGE=1          #<== 单词全大写的写法
```

- 一般的变量定义、赋值常用双引号；简单连续的字符串可以不加引号；希望原样输出时使用单引号。
- 希望变量的内容是命令的解析结果时，要用反引号 ``，或者用 $() 把命令括起来再赋值。

（2）Shell 定义变量时使用“=”的知识

“a=1”里等号是赋值的意思；比较变量是否相等时也可以用“=”或“==”。

（3）打印输出及使用变量的知识

- 打印输出或使用变量时，变量名前要接 $ 符号；变量名后面紧接其他字符的时候，要用大括号将变量部分单独括起来，以防止出现“**金庸新著**”的问题；在 unset、export、(()) 等场景中使用但不打印变量时不加 $，这个有些例外。
- 打印输出或使用变量时，一般加双引号或不加引号；如果是字符串变量，最好加双引号；希望原样输出时使用单引号。

关于变量命名的更多规范可参考第 14 章。

第4章

Shell 变量知识进阶与实践

4.1 Shell 中特殊且重要的变量

4.1.1 Shell 中的特殊位置参数变量

在 Shell 中存在一些特殊且重要的变量，例如：$0、$1、$#，我们称之为特殊位置参数变量。要从命令行、函数或脚本执行等处传递参数时，就需要在 Shell 脚本中使用位置参数变量。表 4-1 为常用的特殊位置参数变量的说明。

表 4-1 常用的特殊位置参数变量说明

位置变量	作用说明
$0	获取当前执行的 Shell 脚本的文件名，如果执行脚本包含了路径，那么就包括脚本路径
$n	获取当前执行的 Shell 脚本的第 n 个参数值，n=1..9，当 n 为 0 时表示脚本的文件名；如果 n 大于 9，则用大括号括起来，例如 ${10}，接的参数以空格隔开
$#	获取当前执行的 Shell 脚本后面接的参数的总个数
$*	获取当前 Shell 脚本所有传参的参数，不加引号和 $@ 相同；如果给 $* 加上双引号，例如："$*"，则表示将所有的参数视为单个字符串，相当于 "$1 $2 $3"
$@	获取当前 Shell 脚本所有传参的参数，不加引号和 $* 相同；如果给 $@ 加上双引号，例如："$@"，则表示将所有的参数视为不同的独立字符串，相当于 "$1" "$2" "$3" "..."。这是将多参数传递给其他程序的最佳方式，因为它会保留所有的内嵌在每个参数里的任何空白。当 "$@" 和 "$*" 都加双引号时，两者是有区别的；都不加双引号时，两者无区别

1. $1 $2…$9 ${10} ${11}.. 特殊变量实践

范例 4-1：测试 $n（n 为 1..15）的实践。

编写如下的 p.sh 脚本，输入内容为“echo $1”，并执行测试：

```
[root@oldboy scripts]# cat p.sh
echo $1    #<== 脚本功能是打印脚本传递的第一个参数的值。
[root@oldboy scripts]# sh p.sh oldboy   #<== 传入一个 oldboy 字符串参数，赋值给脚本中的 $1。
oldboy     #<== 把传入的 oldboy 参数赋值给脚本中的 $1 并输出，因此输出结果为 oldboy。
[root@oldboy scripts]# sh p.sh oldboy oldgirl  #<== 传入两个字符串参数，但脚本不会接收第二个参数，参数默认是以空格分隔。
oldboy     #<== 只输出了 oldboy，因为脚本里没有加入 $2，因此，无法接收第二个参数 oldgirl 字符串。
[root@oldboy scripts]# sh p.sh "oldboy oldgirl"   #<== 加引号扩起来的内容传参，会作为一个字符串参数。
oldboy oldgirl     #<== 虽然都打印了，但是这些内容是作为一个参数传递给 $1 的。
```

范例 4-2：在脚本中同时加入 $1 和 $2，并进行测试。

```
[root@oldboy scripts]# cat p.sh
echo $1 $2
[root@oldboy scripts]# sh p.sh longge bingbing     #<== 同时传入两个字符串参数。
longge bingbing
[root@oldboy scripts]# sh p.sh "longge bingbing" oldgirl #<== 传入两个字符串参数，对第一个有空格的多个字符串用双引号引起来。
longge bingbing oldgirl
```

范例 4-3：设置 15 个位置参数（$1 ~ $15），用于接收命令行传递的 15 个参数。

```
[root@oldboy ~]# echo \${1..15}          #<== 利用大括号输出 15 个位置参数，学会了该命令就不用手敲代码了。
$1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12 $13 $14 $15
[root@oldboy ~]# echo \${1..15} >n.sh   #<== 利用大括号输出 15 个位置参数并定向到文件 n.sh 里。
[root@oldboy ~]# cat n.sh
$1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12 $13 $14 $15
[root@oldboy scripts]# cat n.sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12 $13 $14 $15 #<== 增加 echo 命令打印所有参数，这是最终的测试代码，前面的都是为了写代码，读者也可以用 vim 编辑录入。
[root@oldboy scripts]# echo {a..z}       #<== 测试打印 26 个字母 a ~ z 并以空格分隔。
a b c d e f g h i j k l m n o p q r s t u v w x y z
[root@oldboy scripts]# sh n.sh  {a..z}  #<== 传入 26 个字母 a ~ z，以空格分隔，作为 26 个参数。
a b c d e f g h i a0 a1 a2 a3 a4 a5 #<== 位置参数的数字大于 9 后，输出的内容就不对了。
```

其实，当我们使用 vim 编辑脚本时，利用 vim 的高亮功能就会看到脚本呈现异常的颜色显示，如图 4-1 所示。

图 4-1　vim 高亮功能呈现脚本的异常

当位置参数数字大于 9 时，需要用大括号将数字括起来，如下：

```
[root@oldboy scripts]# cat n.sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} ${12} ${13} ${14} ${15}
#<== 数字大于 9，必须给数字加大括号才能输出正确内容。
```

图 4-2 是加上括号后的高亮颜色，可以看到，颜色已经是正常的了，vim 的语法高亮显示对编程很有帮助，有关 vim 的开发环境配置，见第 16 章。

图 4-2　长方形线内为正常的颜色显示

以下是有关“$1,$2,$3…”这些位置参数的系统生产场景案例。对此，读者可以多参考 rpcbind、NFS 两个软件启动的脚本，这两个服务的启动脚本简单、规范。若是最小化安装的系统，则表示没有安装 rpcbind、NFS，可以通过执行 yum instlla nfs-utils rpcbind –y 来安装。

在生产场景中，执行 /etc/init.d/rpcbind start 之后，rpcbind 脚本后携带的 start 参数会传给脚本里的“$1”进行判断，脚本中传递参数的关键 case 语句节选如下：

```
case "$1" in #<== 这里的 $1 用于接收执行此脚本命令行的第一个参数，规范用法是用双引号引起来。
    start)   #<== 如果 $1 接收的值匹配 start，则执行下文的 start 函数及内部的指令。
        start   #<== 调用脚本中的 start 函数。
        RETVAL=$?  #<== 这里是记录 start 函数执行的返回值，$? 也是重要的变量，暂时可
                       以忽略，后面有介绍。
        ;;
    stop) #<== 如果 $1 接收的值匹配 stop，则执行下文的 stop 函数及内部的指令。
        stop
        RETVAL=$?
        ;;
    status) #<== 如果 $1 接收的值匹配 status，则执行下文的 status 函数及内部的指令。
        status $prog
        RETVAL=$?
        ;;
……省略部分内容
```

说明：读者只需要关注特殊变量（$1）的内容，case 等其他语句后文会细讲。

2. $0 特殊变量的作用及变量实践

$0 的作用为取出执行脚本的名称（包括路径），下面是该功能的实践。

范例 4-4：获取脚本的名称及路径。

```
[root@oldboy scripts]# cat n.sh
echo $0
```

若不带路径执行脚本，那么输出结果就是脚本的名字，如下：

```
[root@oldboy scripts]# sh n.sh
n.sh #<==$0 获取的值就是脚本的名字，因此这里输出了 n.sh。
```

若使用全路径执行脚本，那么输出结果就是全路径加上脚本的名字，如下：

```
[root@oldboy scripts]# sh /server/scripts/n.sh
/server/scripts/n.sh  #<== 如果执行的脚本中带有路径，那么 $0 获取的值就是脚本的名字加路径。
```

当要执行的脚本为全路径时，$0 也会带着路径。此时如果希望单独获取名称或路径，则可用范例 4-5 的方法。

范例 4-5：dirname 及 basename 命令自身的功能和用法。

```
[root@oldboy scripts]# dirname /server/scripts/n.sh
/server/scripts          #<==dirname 命令的作用是获取脚本的路径。
[root@oldboy scripts]# basename /server/scripts/n.sh
n.sh                     #<==basename 命令的作用是获取脚本的名字。
```

说明：以后读者可以根据需求，用不同的命令获取对应的结果。

范例 4-6：利用 $0 和上述命令（dirname、basename）分别取出脚本名称和脚本路径。

```
[root@oldboy scripts]# cat n.sh
dirname $0
basename $0
[root@oldboy scripts]# sh /server/scripts/n.sh
/server/scripts          #<== 这就是 dirname $0 的输出结果。
n.sh                     #<== 这就是 basename $0 的输出结果。
```

有关“$0”这个位置参数的系统生产场景案例如下，其中采用 rpcbind 系统脚本。

```
[root@oldboy scripts]# tail -6 /etc/init.d/rpcbind  #<== 查看结尾 6 行。
        echo $"Usage: $0 {start|stop|status|restart|reload|force-
reload|condrestart|try-restart}"
        #<==$0 的基本生产场景就是，当用户的输入不符合脚本的要求时，就打印脚本的名字及使用帮助。
        RETVAL=2
        ;;
esac
exit $RETVAL
[root@oldboy scripts]# /etc/init.d/rpcbind #<== 不带任何参数执行 rpcbind 脚本。
Usage: /etc/init.d/rpcbind {start|stop|status|restart|reload|force-
reload|condrestart|try-restart}
```

```
    #<== 上文 /etc/init.d/rpcbind 就是 $0 从脚本命令行获取的值，当用户输入不符合脚本设定的
要求时，打印脚本名字及预期的使用帮助。
```

3. $# 特殊变量获取脚本传参个数的实践

范例 4-7：通过 $# 获取脚本传参的个数。

```
[root@oldboy scripts]# cat q.sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
echo $# #<== 此行是打印脚本命令行传参的个数。
[root@oldboy scripts]# sh q.sh {a..z} #<== 传入 26 个字符作为 26 个参数。
a b c d e f g h i #<== 只接收了 9 个变量，所以打印 9 个字符。
26  #<== 传入 26 个字符作为 26 个参数，因此这里的数字为 26，说明传入了 26 个参数。
```

范例 4-8：根据用户在命令行的传参个数判断用户的输入，不合要求的给予提示并退出。

这是一个针对 $0、$1、$# 等多位置参数的综合型企业案例，脚本中可能包含了部分读者没有掌握的技术，这里只需要理解这几个位置参数就可以了，对于其他知识后面会有详细讲解。

首先来看条件表达式判断语句的写法，如下：

```
[root@oldboy scripts]# cat t1.sh
[ $# -ne 2 ] && {          #<== 如果执行脚本传参的个数不等于 2，
  echo "muse two args"     #<== 则给用户提示正确的用法。
  exit 1 #<== 由于不满足要求，因此退出脚本，返回值为 1。
}
echo oldgirl  #<== 满足了参数个数的传参要求后，就执行判断后的程序脚本，即打印 oldgirl。
[root@oldboy scripts]# sh t1.sh
muse two args #<== 如果不加参数执行脚本，即不符合脚本要求，则直接给出提示。
[root@oldboy scripts]# sh t1.sh  arg1 arg2
oldgirl        #<== 当参数满足要求后，打印 oldgirl 字符串。
```

然后是 if 判断语句的写法，如下：

```
[root@oldboy scripts]# cat t2.sh
if [ $# -ne 2 ] #<== 如果执行脚本传参的个数不等于 2，
  then
    echo "USAGE:/bin/sh $0 arg1 arg2" #<== 则给用户提示正确用法，注意此处的 $0，打印
                                          脚本名字及路径。
    exit 1 #<== 若不满足要求，则退出脚本，返回值为 1。
fi
echo $1 $2 #<== 若参数满足要求，则打印 $1 和 $2 获取到的传参的字符串。
[root@oldboy scripts]# sh t2.sh          #<== 若不加参数执行脚本，则直接给出提示。
USAGE:/bin/sh t2.sh arg1 arg2            #<== t2.sh 就是脚本中 $0 获取的值。
[root@oldboy scripts]# sh t2.sh oldboy oldgirl
oldboy oldgirl #<== 若参数满足要求，则打印 $1 和 $2 获取的字符串，即 oldboy 和 oldgirl。
```

4. $* 和 $@ 特殊变量功能及区别说明

首先，请翻到本章的开头再重新温习一下 $* 和 $@ 的作用，然后再来看范例。

范例 4-9：利用 set 设置位置参数（同命令行脚本的传参）。

```
[root@oldboy scripts]# set -- "I am" handsome oldboy. #<== 通过 set 设置三个字符串参数，“--”表示清除所有的参数变量，重新设置后面的参数变量。
[root@oldboy scripts]# echo $#    #<== 输出参数的个数。
3  #<== 共三个参数。
[root@oldboy scripts]# echo $1    #<== 打印第一个参数值。
I am
[root@oldboy scripts]# echo $2    #<== 打印第二个参数值。
handsome
[root@oldboy scripts]# echo $3    #<== 打印第三个参数值。
oldboy.
```

测试 $* 和 $@，注意，此时不带双引号：

```
[root@oldboy scripts]# echo $*    #<== 打印 $*。
I am handsome oldboy.
[root@oldboy scripts]# echo $@    #<== 打印 $@。
I am handsome oldboy.
[root@oldboy scripts]# for i in $*;do echo $i;done #<== 使用 for 循环输出 $* 测试。
I #<== ($*) 不加双引号，因此会输出所有参数，然后第一个参数 "I am" 也拆开输出了。
am
handsome
oldboy.
[root@oldboy scripts]# for i in $@;do echo $i;done #<== 使用 for 循环输出 $@ 测试。
I #<== ($@) 不加双引号，因此会输出所有参数，然后第一个参数 "I am" 也拆开输出了。
am
handsome
oldboy.
```

测试 "$*" 和 "$@"，注意，此时带有双引号：

```
[root@oldboy scripts]# echo "$*"
I am handsome oldboy.
[root@oldboy scripts]# echo "$@"
I am handsome oldboy.
[root@oldboy scripts]# for i in "$*";do echo $i;done
#<== 在有双引号的情况下 "$*"，参数里引号中的内容当作一个参数输出了！
I am handsome oldboy.
[root@oldboy scripts]# for i in "$@";do echo $i;done
#<== 在有双引号的情况下，每个参数均以独立的内容输出。
I am #<== 有双引号算一个参数。
handsome
oldboy.
#<== 这才真正符合我们传入的参数需求，set -- "I am" handsome oldboy.
[root@oldboy scripts]# for i;do echo $i;done #<== 去掉 in 变量列表，相当于有引号的 in "$@"。
I am
handsome
oldboy.
```

```
    #<== 这才真正符合我们传入的参数需求，set -- "I am" handsome oldboy.
    [root@oldboy 02]# for i in $*;do echo $i;done  #<== ($*) 不加双引号，因此会输出
所有参数，然后第一个参数 "I am" 也拆开输出了。
    I
    am
    handsome
    oldboy.
    [root@oldboy scripts]# shift      #<== 用 shift 将位置参数移位 (左移)。
    [root@oldboy scripts]# echo $#
    2
    [root@oldboy scripts]# echo $1    #<== 这里就打印原来 $2 的值了。
    handsome
    [root@oldboy scripts]# echo $2    #<== 这里就打印原来 $3 的值了。
    oldboy.
```

有关 set 和 eval 命令的使用案例（特殊位置变量用法）见 http://oldboy.blog.51cto.com/2561410/1175971。

4.1.2　Shell 进程中的特殊状态变量

表 4-2 针对 Shell 进程的特殊状态变量进行了说明。

表 4-2　Shell 进程的特殊状态变量说明

位置变量	作用说明
$?	获取执行上一个指令的执行状态返回值（0 为成功，非零为失败），这个变量最常用
$$	获取当前执行的 Shell 脚本的进程号（PID），这个变量不常用，了解即可
$!	获取上一个在后台工作的进程的进程号（PID），这个变量不常用，了解即可
$_	获取在此之前执行的命令或脚本的最后一个参数，这个变量不常用，了解即可

提示：查找上述知识的方法为使用 man bash 命令，然后搜关键字“Special Parameters”。

1. $? 特殊变量功能实践

范例 4-10：执行命令后获取返回值（切换到 oldboy 用户下进行测试）。

```
[oldboy@oldboy ~]$ pwd #<== 执行 pwd 命令，然后用 “echo $?” 查看执行命令的状态返回值。
/home/oldboy
[oldboy@oldboy ~]$ echo $?
0  #<== 返回 0，表示上一个命令的执行是成功的。
[oldboy@oldboy ~]$ ls /root #<== 列表 root 目录的内容，
ls: cannot open directory /root: Permission denied  #<== 提示权限不够。
[oldboy@oldboy ~]$ echo $?
2 #<== 返回值为非 0，表示上一个命令 (ls /root) 执行错误。注意：对于不同的错误，返回值是
      不同的。
[oldboy@oldboy ~]$ rm -fr /root #<== 删除 /root 目录及其子目录。
```

```
rm: cannot remove `/root': Permission denied   #<== 提示权限不够。
[oldboy@oldboy ~]$ echo $?
1 #<== 返回值为 1（非 0）。
[oldboy@oldboy ~]$ oldboy                      #<== 执行一个不存在的命令。
-bash: oldboy: command not found               #<== 提示命令找不到。
[oldboy@oldboy ~]$ echo $?
127 #<== 返回值为 127（非 0）。
```

不同命令的执行结果中，“$?”的返回值不尽相同，但在工作场景中，常用的就是 0 和非 0 两种状态，0 表示成功运行，非 0 表示运行失败。

范例 4-11：根据返回值来判断软件的安装步骤是否成功。

若使用源码编译安装软件，可以在每个步骤的结尾获取“$?”来判断命令执行成功与否。例如：编译 Nginx Web 服务软件，执行 make 命令之后，新手不太容易确定编译是否正确，这时就可以使用“echo $?”命令查看其返回值是否为 0。下面是 Nginx Web 的基本安装过程，其中就是通过获取命令的返回值来确定命令的执行状况的。

```
[root@oldboy tools]# yum install pcre-devel openssl-devel -y
[root@oldboy tools]# wget -q http://nginx.org/download/nginx-1.10.1.tar.gz
[root@oldboy tools]# tar xf nginx-1.10.1.tar.gz
[root@oldboy tools]# cd nginx-1.10.1
[root@oldboy nginx-1.10.1]# ./configure --prefix=/application/nginx-1.10.1
--user=nginx --group=nginx --with-http_ssl_module  --with-http_stub_status_
module
……省略部分配置过程
  nginx http proxy temporary files: "proxy_temp"
  nginx http fastcgi temporary files: "fastcgi_temp"
  nginx http uwsgi temporary files: "uwsgi_temp"
  nginx http scgi temporary files: "scgi_temp"
[root@oldboy nginx-1.10.1]# echo $?
0
[root@oldboy nginx-1.10.1]# make
……省略部分编译过程
sed -e "s|%%PREFIX%%|/application/nginx-1.10.1|" \
                -e "s|%%PID_PATH%%|/application/nginx-1.10.1/logs/nginx.pid|" \
                -e "s|%%CONF_PATH%%|/application/nginx-1.10.1/conf/nginx.conf|" \
                -e "s|%%ERROR_LOG_PATH%%|/application/nginx-1.10.1/logs/error.log|" \
                < man/nginx.8 > objs/nginx.8
make[1]: Leaving directory `/home/oldboy/tools/nginx-1.10.1'
[root@oldboy nginx-1.10.1]# echo $?
0
test -d '/application/nginx-1.10.1/logs' \
                || mkdir -p '/application/nginx-1.10.1/logs'
make[1]: Leaving directory `/home/oldboy/tools/nginx-1.10.1'
[root@oldboy nginx-1.10.1]# echo $?
0
```

对于新手来说，在安装服务时，可以通过获取执行命令的返回值来确定命令的执行

状态，从而快速确定命令是否执行成功。不过，有经验的技术人员不需要获取返回值，通过命令的最后过程输出就可以快速判断是否成功。

范例 4-12：通过获取“$?”的返回值确定网站备份是否正确。

> **提示：当对服务器的数据进行备份时，我们会在执行完关键命令，例如 tar 或 cp 后，通过获取返回值来判断命令是否成功，备份数据是否完整。**

```
[root@oldboy ~]# cd /etc/
[root@oldboy etc]# tar zcf /opt/services.tar.gz ./services     #<== 打包备份
[root@oldboy etc]# echo $?   #<== 检查备份后的 $? 是否为 0，如果为 0 则表示上一 tar 命
                                  令执行成功，工作中会写成 Shell 脚本。
0
```

范例 4-13：通过脚本控制命令及脚本执行后的返回值。

```
[root@oldboy scripts]# cat test4.sh
[ $# -ne 2 ] && {          #<== 若参数个数不等于 2，
echo "must be two args." #<== 则输出提示。
exit 119                 #<== 终止程序运行并以指定的 119 状态值退出程序，赋值给当前
                              Shell 的“$?”变量。
}
echo oldgirl
[root@oldboy scripts]# sh test4.sh #<== 执行脚本。
must be two args.
[root@oldboy scripts]# echo $?
119 #<== 返回值为 119，这个就是脚本中 exit 传过来的返回值。
[root@oldboy scripts]# sh test4.sh a1 a2 #<== 若满足参数要求，
oldgirl #<== 则跳过不合要求的提示及以 119 状态退出的两条命令。
[root@oldboy scripts]# echo $?
0 #<== 返回值为 0，这个就是 echo oldgirl 正确执行后，“$?”的结果。
```

在企业场景下，“$?”返回值的用法如下：

1）判断命令、脚本或函数等程序是否执行成功。

2）若在脚本中调用执行“exit 数字”，则会返回这个数字给“$?”变量。

3）如果是在函数里，则通过“return 数字”把这个数字以函数返回值的形式传给“$?”。

范例 4-14：查看系统脚本的应用情况，脚本名为 /etc/init.d/rpcbind。

这里利用 sed 打印 /etc/init.d/rpcbind 脚本的第 50-73 行，然后分析脚本里“$?”的使用情况！

```
[root@oldboy scripts]# sed -n '63,73p' /etc/init.d/rpcbind
stop() {
        echo -n $"Stopping $prog: "
        killproc $prog  #<== 这是停止 rpcbind 的命令。
        RETVAL=$? #<== 将上述命令的返回值“$?”赋值给 RETVAL 变量，用于后面的判断。
```

```
        echo
        [ $RETVAL -eq 0 ]&&{ #<== 这里就是判断，如果返回值为 0，则执行下面的指令。
                rm -f /var/lock/subsys/$prog
                rm -f /var/run/rpcbind*
        }
        return $RETVAL  #<== 如果返回值不等于 0，则跳过条件表达式的判断，在这里直接作
                            为返回值传给执行 stop 函数的脚本。
}
```

提示：有关特殊位置和进程的状态变量，可以多参考这个简单却功能强大的脚本。

2. $$ 特殊变量功能及实践

范例 4-15：获取脚本执行的进程号（PID）。

```
[root@oldboy scripts]# cat test_pid.sh  #<== 编写一个简单的脚本。
echo $$ >/tmp/a.pid #<== 获取 $$ 的值，并重定向到 /tmp/a.pid 里。
sleep 300           #<== 休息 300 秒，模拟守护进程不退出。
[root@oldboy scripts]# ps -ef|grep test_pid|grep -v grep
[root@oldboy scripts]# sh test_pid.sh & #<== 在后台运行脚本，& 符号表示在后台运行。
[1] 10397                               #<== 这是脚本的进程号。
[root@oldboy scripts]# ps -ef|grep test_pid|grep -v grep
root      10397 10292  0 15:57 pts/0    00:00:00 sh test_pid.sh #<== 这是脚本
的进程号。
[root@oldboy scripts]# cat /tmp/a.pid
10397 #<== 这是 $$ 对应的值。
```

提示：到这里大家应该明白了吧，$$ 就是获取当前执行的 Shell 脚本的进程号。

范例 4-16：实现系统中多次执行某一个脚本后的进程只有一个（此为 $$ 的企业级应用）。

说明：有时执行定时任务脚本的频率比较快，并不知道上一个脚本是否真的执行完毕，但是，业务要求同一时刻只能有一个同样的脚本在运行，此时就可以利用 $$ 获取上一次运行的脚本进程号，当程序重新运行时，根据获得的进程号，清理掉上一次的进程，运行新的脚本命令，脚本如下：

```
[root@oldboy scripts]# cat pid.sh
# !/bin/sh
pidpath=/tmp/a.pid     #<== 定义 pid 文件。
if [ -f "$pidpath" ]   #<== 如果 pid 文件存在，则执行 then 后面的命令。
  then
    kill `cat $pidpath` >/dev/null 2>&1  #<== 杀掉与前一个进程号对应的进程。
    rm -f $pidpath     #<== 删除 pid 文件。
fi
```

```
echo $$ >$pidpath      #<== 将当前 Shell 进程号记录到 pid 文件里。
sleep 300
```

执行结果如下：

```
[root@oldboy scripts]# ps -ef|grep pid.sh|grep -v grep
[root@oldboy scripts]# sh pid.sh &  #<== 后台运行脚本。
[1] 10617
[root@oldboy scripts]# ps -ef|grep pid.sh|grep -v grep  #<== 查看启动的脚本进程。
root      10617 10462  0 16:20 pts/1    00:00:00 sh pid.sh  #<== 只有一个。
[root@oldboy scripts]# sh pid.sh &  #<== 多次运行脚本，每次都会将上一次运行的杀掉。
[2] 10624
[root@oldboy scripts]# sh pid.sh &  #<== 多次运行脚本，每次都会将上一次运行的杀掉。
[3] 10628
[1]   Terminated              sh pid.sh
[root@oldboy scripts]# ps -ef|grep pid.sh|grep -v grep
root      10628 10462  0 16:20 pts/1    00:00:00 sh pid.sh  #<== 发现无论运行
多少次脚本，都只有一个进程。
[2]-  Terminated              sh pid.sh
[root@oldboy scripts]# ps -ef|grep pid.sh|grep -v grep
root      10628 10462  0 16:20 pts/1    00:00:00 sh pid.sh #<== 发现无论运行多
少次脚本，都只有一个进程。
```

提示：这是一个生产案例的简单模拟，脚本用于执行启动或定时任务时，相同的脚本中只能有一个在运行，当新脚本运行时，必须关闭未运行完或未退出的上一次的同名脚本进程。

3. $_ 特殊变量功能说明及实践

$_ 的作用是获得上一条命令的最后一个参数值，此功能用得不多，了解即可。

范例 4-17： $_ 参数的示例。

```
[root@oldboy scripts]# /etc/init.d/rpcbind start oldboy
[root@oldboy scripts]# echo $_ #<== 打印上一条命令的最后一个参数值，即 oldboy。
oldboy
[root@oldboy scripts]# /etc/init.d/rpcbind stop oldgirl
Stopping rpcbind:                                          [  OK  ]
[root@oldboy scripts]# echo $_ #<== 打印上一条命令的最后一个参数值，即 oldgirl。
oldgirl
```

4. $! 特殊变量功能说明及实践

$! 的功能类似于 $$，只不过作用是获取上一次执行脚本的 pid，对此，了解即可。

范例 4-18： $! 的功能示例。

```
[root@oldboy scripts]# ps -ef|grep pid.sh|grep -v grep
[root@oldboy scripts]# sh pid.sh & #<== 后台运行 pid.sh 脚本。
```

```
[1] 10760
[root@oldboy scripts]# echo $!   #<== 获取前一次执行脚本 pid.sh 的进程号。
10760
[root@oldboy scripts]# ps -ef|grep pid.sh|grep -v grep
root     10760 10462  0 16:44 pts/1    00:00:00 sh pid.sh
```

4.2 bash Shell 内置变量命令

bash Shell 包含一些内置命令。这些内置命令在目录列表里是看不见的，它们由 Shell 本身提供。常用的内部命令有：echo、eval、exec、export、read、shift。下面简单介绍几个最常用的内置命令的格式和功能，想要了解更多的内置命令请参考老男孩的 Linux 命令相关图书，或者本书所带案例中的内置命令的使用。

（1）echo 在屏幕上输出信息

命令格式：echo args #<== 可以是字符串和变量的组合。

功能说明：将 echo 命令后面 args 指定的字符串及变量等显示到标准输出。

常用参数见表 4-3。

表 4-3 echo 的参数等信息说明

echo 参数选项	说　明
-n	不换行输出内容
-e	解析转义字符（见下面的字符）
转义字符：	
\n	换行
\r	回车
\t	制表符（tab）
\b	退格
\v	纵向制表符

范例 4-19：echo 的参数应用示例。

```
[root@oldboy etc]# echo oldboy;echo oldgirl
oldboy
oldgirl
[root@oldboy etc]# echo -n oldboy;echo oldgirl #<==-n 不换行输出。
oldboyoldgirl
[root@oldboy etc]# echo "oldboy\toldgirl\noldboy\toldgirl"
oldboy\toldgirl\noldboy\toldgirl
[root@oldboy etc]# echo -e "oldboy\toldgirl\noldboy\toldgirl" #<== 加上 -e 解析以“\”开头的字符。
oldboy  oldgirl #<==oldboy 和 oldgirl 之间的空隙就是 \t 的作用。遇到 \n 后重新开启一行。
oldboy  oldgirl
[root@oldboy etc]# printf "oldboy\toldgirl\noldboy\toldgirl\n" #<==printf 的转义字符能力与 echo 类似。
```

```
oldboy  oldgirl
oldboy  oldgirl
[root@oldboy etc]# echo -e "1\b23"#<== 加上 -e 解析以“\”开头的字符，\b 退格。
23
[root@oldboy etc]# printf "1\b23\n" #<==printf 的转义字符功能与 echo 类似。
23
```

有关 echo 命令的用法将会贯穿全书，更多内容请见后文案例讲解，printf 与 echo 的功能类似，但是 printf 更强大，当需要特殊复杂的格式时才考虑使用 printf，本书用 printf 的地方不多，仅在结尾的有趣案例 girlLove 工具程序中用得较多。

（2）eval（后文有案例讲解）

命令格式：eval args

功能：当 Shell 程序执行到 eval 语句时，Shell 读入参数 args，并将它们组合成一个新的命令，然后执行。

范例 4-20：set 和 eval 命令的使用（含特殊位置变量用法）方法。

```
[root@oldboy etc]# cat noeval.sh
echo \$$#   #<==$# 表示传参的个数。
[root@oldboy etc]# sh noeval.sh arg1 arg2 #<== 传入两个参数。
$2  #<== 传入两个参数，因此 $#  为 2，于是 echo  \$$#  就变成了 echo  $2，最后输出 $2，并没
        有打印 arg2。
[root@oldboy etc]# cat eval.sh
eval "echo \$$# " #<== 加上 eval 命令，使得打印的特殊位置参数，重新解析输出，而不是输出 $2 本身。
[root@oldboy etc]# sh eval.sh arg1 arg2
arg2  #<== 输出了 $2。
```

案例见 http://oldboy.blog.51cto.com/2561410/1175971。

（3）exec

命令格式：exec 命令参数

功能：exec 命令能够在不创建新的子进程的前提下，转去执行指定的命令，当指定的命令执行完毕后，该进程（也就是最初的 Shell）就终止了，示例如下：

```
[root@oldboy ~]# exec date
2016 年 09 月 12 日 星期一 13:29:22 CST
[oldboy@oldboy ~]$ #<== 退到普通用户模式下了。
```

当使用 exec 打开文件后，read 命令每次都会将文件指针移动到文件的下一行进行读取，直到文件末尾，利用这个可以实现处理文件内容。

范例 4-21：exec 的功能示例。

```
[root@oldboy etc]# seq 5 >/tmp/tmp.log
[root@oldboy etc]# cat exec.sh
exec </tmp/tmp.log          #<== 读取 log 内容。
while read line             #<== 利用 read 一行行读取处理。
```

```
do
    echo $line              #<== 打印输出。
done
echo ok
```

执行结果如下：

```
[root@oldboy etc]# sh exec.sh
1
2
3
4
5
ok
```

（4）read

命令格式：read 变量名表

功能：从标准输入读取字符串等信息，传给 Shell 程序内部定义的变量。

此命令将在后文详细讲解。

（5）shift

命令格式：shift-Shift positional parameters

功能：shift 语句会按如下方式重新命名所有的位置参数变量，即 $2 成为 $1、$3 成为 $2 等，以此类推，在程序中每使用一次 shift 语句，都会使所有的位置参数依次向左移动一个位置，并使位置参数 $# 减 1，直到减到 0 为止。

范例 4-22：shift 的功能介绍。

```
[root@oldboy 02]# help shift
shift: shift [n]
    Shift positional parameters.
    Rename the positional parameters $N+1,$N+2 ... to $1,$2 ...  If N is
    not given, it is assumed to be 1.
    Exit Status:
    Returns success unless N is negative or greater than $# .
```

shift 命令的主要作用是将位置参数 $1、$2 等进行左移，即如果位置参数是 $3、$2、$1，那么执行一次 shift 后，$3 就变成了 $2，$2 变成了 $1，$1 就消失了。

范例 4-23：shift 命令的使用示例。

```
[root@oldboy script]# cat n.sh
echo $1 $2
if [ $# -eq 2 ];then
    shift
    echo $1
fi
[root@oldboy script]# sh n.sh 1 2
```

```
1 2 #<== 这是 echo $1 $2 的结果。
2   #<== 这里是 echo $1 的结果，但是输出的是传参时 $2 的值。
```

应用场景：当我们写 Shell 希望像命令行的命令通过参数控制不同的功能时，就会先传一个类似 -c 的参数，然后再接内容。

```
[root@shell scripts]# sh shift.sh -c oldboy
-c oldboy  #<== 对应 $1 $2 的输出。
oldboy     #<== 对应 $1 的输出，因为执行了 shift，因此第二个参数 $2 的内容，就变成了 $1，
               所以输出了 oldboy。
```

以下是系统案例 ssh-copy-id -i /root/.ssh/id_dsa.pub：

```
ID_FILE="${HOME}/.ssh/id_rsa.pub"
if [ "-i" = "$1" ]; then
  shift
  # check if we have 2 parameters left, if so the first is the new ID file
  if [ -n "$2" ]; then
    if expr "$1" : ".*\.pub" > /dev/null ; then
      ID_FILE="$1"
    else
      ID_FILE="$1.pub"
    fi
    shift          # and this should leave $1 as the target name
  fi
```

作用：方便。

（6）exit

命令格式：exit-Exit the shell

功能：退出 Shell 程序。在 exit 之后可以有选择地指定一个数位作为返回状态。

4.3　Shell 变量子串知识及实践

4.3.1　Shell 变量子串介绍

Shell 变量子串的常用操作见表 4-4，读者可以在执行 man bash 命令之后，搜索“Parameter Expansion”找到相应的帮助知识，对于 Shell 新手来说，此部分内容可以暂时忽略，在学完本书后再回来学习。

表 4-4　Shell 变量子串说明

ID	表达式	说　明
1	${parameter}	返回变量 $parameter 的内容
2	${#parameter}	返回变量 $parameter 内容的长度（按字符），也适用于特殊变量
3	${parameter:offset}	在变量 ${parameter} 中，从位置 offset 之后开始提取子串到结尾

（续）

ID	表达式	说　明
4	${parameter:offset:length}	在变量 ${parameter} 中，从位置 offset 之后开始提取长度为 length 的子串
5	${parameter#word}	从变量 ${parameter} **开头**开始删除**最短**匹配的 word 子串
6	${parameter##word}	从变量 ${parameter} **开头**开始删除**最长**匹配的 word 子串
7	${parameter%word}	从变量 ${parameter} **结尾**开始**删除最短匹配**的 word 子串
8	${parameter%%word}	从变量 ${parameter} **结尾**开始**删除最长匹配**的 word 子串
9	${parameter/pattern/string}	使用 string 代替第一个匹配的 pattern
10	${parameter//pattern/string}	使用 string 代替所有匹配的 pattern

4.3.2 Shell 变量子串的实践

准备：定义 OLDBOY 变量，赋值内容为“I am oldboy”，操作代码如下：

```
[root@oldboy scripts]# OLDBOY="I am oldboy"      #<== 这里的 OLDBOY 变量，就是表 4-4 中 parameter 变量的具体化示例。
[root@oldboy scripts]# echo ${OLDBOY}            #<== 带大括号打印变量 OLDBOY。
I am oldboy
[root@oldboy scripts]# echo $OLDBOY              #<== 直接打印变量 OLDBOY。
I am oldboy
```

范例 4-24：返回 OLDBOY 变量值的长度。

通过在变量名前加 #，就可以打印变量值的长度：

```
[root@oldboy scripts]# echo ${#OLDBOY}
11                          #<==I am oldboy. 这些字符加起来正好是 11.
```

范例 4-25：Shell 的其他打印变量长度的方法。

此为范例 4-24 的拓展，代码如下：

```
[root@oldboy scripts]# echo $OLDBOY|wc -L        #<== 输出变量值，然后通过管道交给 wc 计算长度。
11
[root@oldboy scripts]# expr length "$OLDBOY"     #<== 利用 expr 的 length 函数计算变量长度。
11
[root@oldboy scripts]# echo "$OLDBOY"|awk '{print length($0)}'
#<== 利用 awk 的 length 函数计算变量长度，也可无“($0)”这几个字符。
11
```

提示：上述计算变量长度的方法中，变量的字串方式是最快的，即 ${#OLDBOY} 的方式。

范例 4-26：利用 time 命令及 for 循环对几种获取字符串长度的方法进行性能比较。

（1）变量自带的获取长度的方法（echo ${#char}）

```
[root@oldboy tmp]# time for n in {1..10000};do char=`seq -s "oldboy"
100`;echo ${#char} &>/dev/null;done
real    0m40.702s #<== 变量自带的获取长度的方法用时最少，效率最高。
user    0m19.040s
sys     0m17.156s
```

（2）利用管道加 wc 的方法（echo ${char}|wc -L）

```
[root@oldboy tmp]# time for n in {1..10000};do char=`seq -s "oldboy"
100`;echo ${char}|wc -L &>/dev/null;done
real    1m53.716s #<== 使用了管道加 wc -L 计算，结果倒数第二，仅次于管道加 awk 统计的。
user    0m46.709s
sys     0m53.930s
```

（3）利用 expr 自带的 length 方法（expr length "${char}"）

```
[root@oldboy tmp]# time for n in {1..10000};do char=`seq -s "oldboy"
100`;expr length "${char}"&>/dev/null;done
real    1m15.936s #<== 好于使用管道和 wc 的计算方法，但是比变量自带的获取长度的方法要差一些。
user    0m34.375s
sys     0m31.338s
```

（4）利用 awk 自带的 length 函数方法

```
[root@oldboy tmp]# time for n in {1..10000};do char=`seq -s "oldboy"
100`;echo $char|awk '{print length($0)}'&>/dev/null;done
real    2m1.099s  #<== 使用了管道还有 awk 的函数计算，结果最差。
user    0m49.129s
sys     0m58.034s
```

可以看到，这几种方法的速度相差几十到上百倍，一般情况下调用外部命令来处理的方式与使用内置操作的速度相差较大。在 Shell 编程中，应尽量使用内置操作或函数来完成。

有关获取字符串长度的几种统计方法的性能比较如下：

- 变量自带的计算长度的方法的效率最高，在要求效率的场景中尽量多用。
- 使用管道统计的方法的效率都比较差，在要求效率的场景中尽量不用。
- 对于日常简单的脚本计算，读者可以根据自己所擅长的或易用的程度去选择。

关于计算字符串的长度，有一个企业面试案例，面试题目如下：

请编写 Shell 脚本以打印下面语句中字符数小于 6 的单词。

I am oldboy linux,welcome to our training.

说明：该面试题涉及范例 4-24 和范例 4-25 的计算变量长度的知识，该面试题的答案可参考本书后文数组知识范例 13-4，那里会有多种方法的精彩讲解。

范例 4-27：截取 OLDBOY 变量的内容，从第 2 个字符之后开始截取，默认截取后面字符的全部，第 2 个字符不包含在内，也可理解为删除前面的多个字符。

```
[root@oldboy scripts]# echo ${OLDBOY}
I am oldboy
[root@oldboy scripts]# echo ${OLDBOY:2}
am oldboy  #<== 相当于从 I 后面的空格开始计算，截取到了结尾。
```

范例 4-28：截取 OLDBOY 变量的内容，从第 2 个字符之后开始截取，截取 2 个字符。

```
[root@oldboy scripts]# echo ${OLDBOY:2:2}
am
提示：这个功能类似于 cut 命令 -c 参数的功能。
[root@oldboy scripts]# echo ${OLDBOY}|cut -c 3-4 #<== 输出变量的内容，管道交给 cut
                                                    截取第 3 ~ 4 个位置的字符。
am
```

范例 4-29：从变量 $OLDBOY 内容的开头开始删除最短匹配“a*C”及“a*c”的子串。

```
[root@oldboy scripts]# OLDBOY=abcABC123ABCabc
[root@oldboy scripts]# echo $OLDBOY
abcABC123ABCabc
[root@oldboy scripts]# echo ${OLDBOY#a*C}  #<== 从开头开始删除最短匹配“a*C”的子串。
123ABCabc                                  #<== 从开头开始删除了 abcABC。
[root@oldboy scripts]# echo ${OLDBOY# a*c}
ABC123ABCabc                               #<== 从开头开始删除了 abc。
```

范例 4-30：从变量 $OLDBOY 开头开始删除最长匹配“a*C”及“a*c”的子串。

```
[root@oldboy scripts]# OLDBOY=abcABC123ABCabc
[root@oldboy scripts]# echo $OLDBOY
abcABC123ABCabc
[root@oldboy scripts]# echo ${OLDBOY## a*c} #<== 从开头开始删除最长匹配“a*c”的子串。
        #<== 结果为空了，说明都匹配了，全部删除了。
[root@oldboy scripts]# echo ${OLDBOY## a*C} #<== 从开头开始删除最长匹配“a*C”的子串。
abc     #<== 结果为 abc，说明匹配了 abcABC123ABC 并删除了这些字符。
```

范例 4-31：从变量 $OLDBOY 结尾开始删除最短匹配“a*C”及“a*c”的子串。

```
[root@oldboy scripts]# OLDBOY=abcABC123ABCabc
[root@oldboy scripts]# echo $OLDBOY
abcABC123ABCabc
[root@oldboy scripts]# echo ${OLDBOY%a*C} #<== 从结尾开始删除最短匹配“a*C”的子串。
abcABC123ABCabc  #<== 原样输出，因为从结尾开始“a*C”没有匹配上任何子串，因此，没有删
                      除任何字符。
[root@oldboy scripts]# echo ${OLDBOY%a*c} #<== 从结尾开始删除最短匹配“a*c”的子串。
abcABC123ABC     #<== 从结尾开始删除最短匹配“a*c”，即删除了结尾的 abc 三个字符。
```

范例 4-32：从变量 $OLDBOY 结尾开始删除最长匹配“a*C”及“a*c”的子串。

```
[root@oldboy scripts]# OLDBOY=abcABC123ABCabc
```

```
[root@oldboy scripts]# echo $OLDBOY
abcABC123ABCabc
[root@oldboy scripts]# echo ${OLDBOY%%a*C}      #<== 从结尾开始删除最长匹配“a*C”
                                                    的子串。
abcABC123ABCabc #<== 原样输出，因为从结尾开始“a*C”没有匹配上任何子串，因此，没有删除任何字符。
[root@oldboy scripts]# echo ${OLDBOY%%a*c}      #<== 从结尾开始删除最长匹配“a*c”
                                                    的子串。
                #<== 从结尾开始删除最长匹配“a*c”的字符串，即删除全部字符。
```

有关上述匹配删除的小结：

- ❑ # 表示从开头删除匹配最短。
- ❑ ## 表示从开头删除匹配最长。
- ❑ % 表示从结尾删除匹配最短。
- ❑ %% 表示从结尾删除匹配最长。
- ❑ a*c 表示匹配的字符串，* 表示匹配所有，a*c 匹配开头为 a、中间为任意多个字符、结尾为 c 的字符串。
- ❑ a*C 表示匹配的字符串，* 表示匹配所有，a*C 匹配开头为 a、中间为任意多个字符、结尾为 C 的字符串。

范例 4-33：使用 oldgirl 字符串代替变量 $OLDBOY 匹配的 oldboy 字符串。

```
[root@oldboy scripts]# OLDBOY="I am oldboy,yes,oldboy"
[root@oldboy scripts]# echo $OLDBOY
I am oldboy,yes,oldboy
[root@oldboy scripts]# echo ${OLDBOY/oldboy/oldgirl} #<== 替换匹配的第一个字符串。
I am oldgirl,yes,oldboy    #<== 真的只替换了第一个。
[root@oldboy scripts]# echo ${OLDBOY//oldboy/oldgirl} #<== 替换匹配的所有字符串。
I am oldgirl,yes,oldgirl   #<== 真的替换了所有匹配 oldboy 的字符串。
```

有关替换的小结：

- ❑ 一个“/”表示替换匹配的第一个字符串。
- ❑ 两个“/”表示替换匹配的所有字符串。

4.3.3　变量子串的生产场景应用案例

范例 4-34：去掉下面所有文件的文件名中的“_finished”字符串。

```
[root@oldboy test]# ls -l *.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_1_finished.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_2_finished.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_3_finished.jpg
```

```
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_4_finished.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_5_finished.jpg
```

要想批量改名，首先得知道单个的文件应如何改名。单个文件的改名命令为：

```
mv stu_102999_1_finished.jpg stu_102999_1.jpg
```

下面利用变量赋值和替换的方式实现上述 mv 命令的改名要求。

```
[root@oldboy test]# f=stu_102999_1_finished.jpg #<== 取一个文件名赋值给变量 f，
                                                    这个就是要修改的源文件。
[root@oldboy test]# echo ${f//_finished/} #<== 利用变量的子串替换功能把变量 f 里的 _
                                                finished 替换为空。
stu_102999_1.jpg #<== 这个就是要修改的目标文件。
[root@oldboy test]# mv $f `echo ${f//_finished/}` #<== 使用 mv 命令执行修改操作，注意
                                                      目标命令要用反引号括起来。
[root@oldboy test]# ls -l *.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_1.jpg #<== 查看改过的结果，确
                                                           实全部都改完了。
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_2_finished.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_3_finished.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_4_finished.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_5_finished.jpg
```

学会处理了一个，就可以进行批量处理了，批量处理就是利用循环而已，虽然这里还没有学过循环，但是没关系，先来看一下如何实现吧。

```
[root@oldboy test]# for f in `ls *fin*.jpg`;do mv $f `echo ${f//_finished/}`;done
#<== 其实就是使用 for 循环，循环上面是进行单个处理的 mv 命令。mv $f `echo ${f//_finished/}`
[root@oldboy test]# ls -l *.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_1.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_2.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_3.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_4.jpg
-rw-r--r-- 1 root root 0 Jul 30 20:33 stu_102999_5.jpg
#<== 大功告成，怎么样，很简单吧？
```

这里给大家展示的是利用变量的子串替换来实现的改名方法，其实还有更简单的实现方法，更多批量改名的案例，请参看老男孩的博客《Linux 下批量修改文件名精彩解答案例分享》，地址为 http://oldboy.blog.51cto.com/2561410/711342。

4.4 Shell 特殊扩展变量的知识与实践

4.4.1 Shell 特殊扩展变量介绍

Shell 的特殊扩展变量说明见表 4-5，读者可以执行 man bash 命令，然后搜索“Parameter Expansion”查找相关的帮助内容。

表 4-5　Shell 的特殊扩展变量说明

表达式	说　明
${parameter:-word}	如果 parameter 的变量值为空或未赋值，则会返回 word 字符串并替代变量的值 用途：如果变量未定义，则返回备用的值，防止变量为空值或因未定义而导致异常
${parameter:=word}	如果 parameter 的变量值为空或未赋值，则设置这个变量值为 word，并返回其值。位置变量和特殊变量不适用 用途：基本同上一个 ${parameter:-word}，但该变量又额外给 parameter 变量赋值了
${parameter:?word}	如果 parameter 变量值为空或未赋值，那么 word 字符串将被作为标准错误输出，否则输出变量的值。 用途：用于捕捉由于变量未定义而导致的错误，并退出程序
${parameter:+word}	如果 parameter 变量值为空或未赋值，则什么都不做，否则 word 字符串将替代变量的值

在表 4-5 中，每个表达式内的冒号都是可选的。如果省略了表达式中的冒号，则将每个定义中的“为空或未赋值”部分改为“未赋值”，也就是说，运算符仅用于测试变量是否未赋值。更多内容，请执行 man bash 命令查看帮助。

4.4.2　Shell 特殊扩展变量的实践

1. ${parameter:−word} 功能实践

${parameter:-word} 的作用是如果 parameter 变量值为空或未赋值，则会返回 word 字符串替代变量的值。

范例 4-35：${parameter:-word} 用法功能示例 1。

```
[root@oldboy test]# echo $test  #<== 变量未设置，所以输出时为空。

[root@oldboy test]# result=${test:-UNSET}       #<== 若 test 没值，则返回 UNSET。
[root@oldboy test]# echo $result #<== 打印 result 变量，返回 UNSET，因为 test 没有赋值。
UNSET
[root@oldboy test]# echo ${test} #<== 注意，此时打印 test 变量还是为空。
```

结论：对于 ${test:-UNSET}，当 test 变量没值时，就返回变量结尾设置的 UNSET 字符串。

范例 4-36：${parameter:-word} 用法功能示例 2。

```
[root@oldboy test]# test=oldboy #<== 给 test 变量赋值 oldboy 字符串。
[root@oldboy test]# echo $test
oldboy
[root@oldboy test]# result=${test:-UNSET}       #<== 重新定义 result。
[root@oldboy test]# echo $result
#<== 因为 test 已赋值，因此，打印 result 就输出了 test 的值 oldboy，而不是原来的 UNSET。
oldboy
提示：这个变量的功能可以用来判断变量是否已定义。
[root@oldboy test]# result=${test-UNSET}        #<== 定义时忽略了冒号。
```

```
[root@oldboy test]# echo $result
oldboy                                          #<== 打印结果和带冒号时没有变化。
```

结论：当 test 变量有值时，就打印 result 变量，返回 test 变量的内容。

2. ${parameter:=word} 功能实践

${parameter:=word} 的作用是：如果 parameter 变量值为空或未赋值，就设置这个变量值为 word，并返回其值。位置变量和特殊变量不适用。

范例 4-37：${parameter:=word} 用法功能示例。

```
[root@oldboy test]# unset result                #<== 撤销 result 变量定义。
[root@oldboy test]# echo $result

[root@oldboy test]# unset test                  #<== 撤销 test 变量定义。
[root@oldboy test]# echo $test

[root@oldboy test]# result=${test:=UNSET}       #<== 重新对变量 result 进行定义。
[root@oldboy test]# echo $result
UNSET
[root@oldboy test]# echo $test
#<== 注意，这里的 test 原来是没有定义的，现在已经被赋值 UNSET 了，这是和“:- ”表达式的区别。
UNSET
[root@oldboy test]# result=${test=UNSET}        #<== 定义时忽略了冒号。
[root@oldboy test]# echo $result
UNSET #<== 打印结果和带冒号时没有变化。
[root@oldboy test]# echo $test
UNSET #<== 打印结果和带冒号时没有变化。
```

当变量（result）值里的变量（test）值没有定义时，会给变量（result）赋值“:=”后面的内容，同时会把“:=”后面的内容赋值给变量（result）值里没有定义的变量（test）。

这个变量的功能可以解决变量没有定义的问题，并确保没有定义的变量始终有值。

3. ${parameter:?word} 功能实践

{parameter:?word} 的作用是：如果 parameter 变量值为空或未赋值，那么 word 字符串将被作为标准错误输出，否则输出变量的值。

范例 4-38：${parameter:?word} 用法功能示例。

```
[root@oldboy test]# echo ${key:?not defined}
#<==key 变量没有定义，因此，把“not defined”作为标准错误输出。
-bash: key: not defined    #<== 错误提示，只不过是事先定义好的错误输出。
[root@oldboy test]# echo ${key?not defined}     #<== 去掉冒号定义，并输出，结果一致。
-bash: key: not defined
[root@oldboy test]# key=1 #<== 给变量赋值 1。
[root@oldboy test]# echo ${key:?not defined}    #<== 因为 key 有值了，所以，打印 key
                                                    的值 1。
1
[root@oldboy test]# echo ${key?not defined}     #<== 去掉冒号定义，并输出，结果一致。
```

```
1
[root@oldboy test]# unset key    #<== 取消 key 的定义。
[root@oldboy test]# echo ${key:?not defined}
-bash: key: not defined          #<== 又打印错误提示了。
```

本例的用法可以用于设定由于变量未定义而报错的具体内容，如："not defined"。

4. ${parameter:+word} 功能实践

${parameter:+word} 的作用是：如果 parameter 变量值为空或未赋值，则什么都不做，否则 word 字符串将替代变量的值。

范例 4-39：${parameter:+word} 用法功能示例。

```
[root@oldboy test]# oldboy=${oldgirl:+word} #<==oldgirl 变量未定义。
[root@oldboy test]# echo $oldboy #<== 因为 oldgirl 变量未定义，所以打印 oldboy 变
                                      量为空。

[root@oldboy test]# oldgirl=19 #<==oldgirl 变量赋值为 19。
[root@oldboy test]# oldboy=${oldgirl:+word} #<== 注意，这里一定要重新定义 oldboy。
[root@oldboy test]# echo $oldboy
#<== 因为 oldgirl 变量有值，所以打印 oldboy 变量输出为 ":+ " 后面的内容。
word
```

此功能可用于测试变量（oldgirl 的位置）是否存在，如果 oldboy 的值为 word，则证明 oldgirl 变量有值。

4.4.3　Shell 特殊扩展变量的生产场景应用案例

范例 4-40（生产案例）：实现 Apache 服务启动脚本 /etc/init.d/httpd（请重点看加粗的部分）。

```
# !/bin/bash
#
# httpd        Startup script for the Apache HTTP Server
#
..skip…
# Start httpd in the C locale by default.
HTTPD_LANG=${HTTPD_LANG-"C"} #<== 如果 HTTPD_LANG 变量没有定义或为空，则打印 HTTPD_
                                  LANG 变量返回 C 值。
#<== 这一步定义方法的目的是防止变量值为空或未定义。使用的是 (${parameter-word} 用法，
     此处省略了冒号 )。
# This will prevent initlog from swallowing up a pass-phrase prompt if
# mod_ssl needs a pass-phrase from the user.
INITLOG_ARGS=""
# Set HTTPD=/usr/sbin/httpd.worker in /etc/sysconfig/httpd to use a server
# with the thread-based "worker" MPM; BE WARNED that some modules may not
# work correctly with a thread-based MPM; notably PHP will refuse to start.

# Path to the apachectl script, server binary, and short-form for messages.
```

```
apachectl=/usr/sbin/apachectl
httpd=${HTTPD-/usr/sbin/httpd}
#<== 如果 HTTPD 变量没有定义或为空，则打印 HTTPD_LANG 变量返回 /usr/sbin/httpd 的值。
#<== 此步定义方法的目的是防止变量值为空或未定义。使用的是 (${parameter-word} 用法，冒号
     省略了 )。
prog=httpd
pidfile=${PIDFILE-/var/run/httpd.pid}
lockfile=${LOCKFILE-/var/lock/subsys/httpd}
RETVAL=0
…skip…
exit $RETVAL
```

提示：通过执行 yum install httpd -y 后可以查看 /etc/init.d/httpd 文件。

在企业中，针对目录路径等的处理就可以采用上述变量不存在（即赋指定值）的方式，防止因目录路径不存在而导致的异常。例如：变量如果为 NULL 或没有定义，则赋予一个备用的值，特别是针对变量的删除操作，这种方式会很有用，否则所删除的变量如果不存在，则可能导致未知的危险。

范例 4-41（生产案例）：删除 7 天前的过期数据备份。

如果忘记了定义 path 变量，又不希望 path 值为空值，就可以定义 /tmp 替代 path 空值的返回值，如下：

```
[root@oldboy test]# cat del.sh
find ${path-/tmp} -name "*.tar.gz" -type f -mtime +7|xargs rm -f
[root@oldboy test]# sh -x del.sh
+ xargs rm -f
+ find /tmp -name '*.tar.gz' -type f -mtime +7   #<== 执行时，系统会自动删除 /
                                                      tmp 下的文件。
```

如果忘了定义 path 变量，并且还未做特殊变量定义，那么命令就会出现意外，如下：

```
[root@oldboy tmp]# cat a.sh
find ${path} -name "*.tar.gz" -type f -mtime +7|xargs rm -f
[root@oldboy tmp]# sh -x a.sh
+ xargs rm -f
+ find -name '*.tar.gz' -type f -mtime +7  #<== 这条命令明显没有指定路径，因此将
                                               会导致异常。
```

Linux

第5章

变量的数值计算实践

5.1 算术运算符

如果要执行算术运算，就会离不开各种运算符号，和其他编程语言类似，Shell 也有很多算术运算符，下面就给大家介绍一下常见的 Shell 算术运算符，如表 5-1 所示。

表 5-1 Shell 中常见的算术运算符号

算术运算符	意义（* 表示常用）
+、-	加法（或正号）、减法（或负号）*
*、/、%	乘法、除法、取余（取模）*
**	幂运算 *
++、--	增加及减少，可前置也可放在变量结尾 *
!、&&、||	逻辑非（取反）、逻辑与（and）、逻辑或 (or)*
<、<=、>、>=	比较符号（小于、小于等于、大于、大于等于）
==、!=、=	比较符号（相等、不相等，对于字符串“=”也可以表示相当于）*
<<、>>	向左移位、向右移位
~、|、&、^	按位取反、按位异或、按位与、按位或
=、+=、-=、*=、/=、%=	赋值运算符，例如 a+=1 相当于 a=a+1，a-=1 相当于 a=a-1*

表 5-1 中的算术运算符号均适用于常见的运算命令，那么，这里所说的运算命令又有哪些呢？见表 5-2。

表 5-2 Shell 中常见的算术运算命令

运算操作符与运算命令	意义
(())	用于整数运算的常用运算符，效率很高
let	用于整数运算，类似于“(())”
expr	可用于整数运算，但还有很多其他的额外功能
bc	Linux 下的一个计算器程序（适合整数及小数运算）
$[]	用于整数运算
awk	awk 既可以用于整数运算，也可以用于小数运算
declare	定义变量值和属性，-i 参数可以用于定义整形变量，做运算

在下面的章节中，我们将逐一为大家讲解 Shell 中的各种运算符号及运算命令。

5.2 双小括号“(())”数值运算命令

5.2.1 双小括号“(())”数值运算的基础语法

双小括号“(())”的作用是进行数值运算与数值比较，它的效率很高，用法灵活，是企业场景运维人员经常采用的运算操作符，其操作方法见表 5-3。

表 5-3 双小括号“(())”的操作方法

运算操作符与运算命令	意义
((i=i+1))	此种书写方法为运算后赋值法，即将 i+1 的运算结果赋值给变量 i。注意，不能用“echo ((i=i+1))”的形式输出表达式的值，但可以用 echo $((i=i+1)) 输出其值
i=$((i+1))	可以在“(())”前加 $ 符，表示将表达式运算后赋值给 i
((8>7&&5==5))	可以进行比较操作，还可以加入逻辑与和逻辑或，用于条件判断
echo $((2+1))	需要直接输出运算表达式的运算结果时，可以在“(())”前加 $ 符

5.2.2 双小括号“(())”数值运算实践

范例 5-1：利用“(())”进行简单的数值计算。

```
[root@oldboy ~]# echo $((1+1)) #<== 计算 1+1 后输出。
2 #<== 结果为 2。
[root@oldboy ~]# echo $((6-3)) #<== 计算 6-3 后输出。
3 #<== 结果为 3。
[root@oldboy ~]# ((i=5))
[root@oldboy ~]# ((i=i*2)) #<== 获取 i 值，然后计算 i*2，再赋值给变量 i，此时没有输出。
[root@oldboy ~]# echo $i  #<== 输出时才用 echo，而且要加 $。
10 #<== 因为开始时 i=5，所以，这里经过前一指令的计算后，i 的结果为 5*2=10。
```

范例 5-2：利用“(())”进行稍微复杂一些的综合算术运算。

```
[root@oldboy ~]# ((a=1+2**3-4%3))
#<== 这是一个较复杂的表达式运算并赋值的操作，表达式运算后将结果赋值给 a，先乘除后加减。
```

```
[root@oldboy ~]# echo $a
8 #<== 运算后输出结果为 8。
[root@oldboy ~]# b=$((1+2**3-4%3)) #<== 这是另外一种表达式运算后将结果赋值给变量的
                                        写法，变量放在了括号的外面。
[root@oldboy ~]# echo $b
8
[root@oldboy ~]# echo $((1+2**3-4%3)) #<== 还可以直接运算表达式并将结果输出，注意，
                                          不要落下了 $ 符。
8
[root@oldboy ~]# a=$((100*(100+1)/2)) #<== 利用公式计算 1+2+3+...+100 的和。
[root@oldboy ~]# echo $a
5050
[root@oldboy ~]# echo $((100*(100+1)/2)) #<== 直接输出表达式的结果。
5050
```

范例 5-3：特殊运算符号的运算小示例。

```
[root@oldboy ~]# a=8
[root@oldboy ~]# echo $((a=a+1)) #<== 将 a+1 赋值给 a，然后输出表达式的值。
9
[root@oldboy ~]# echo $((a+=1))  #<== 相当于 a=a+1。
10
[root@oldboy ~]# echo $((a**2))  #<== 计算 a 的平方，** 表示幂运算。
100
```

范例 5-4：利用“(())”双括号进行比较及判断。

```
[root@oldboy ~]# echo $((3<8))  #<==3<8 的结果是成立的，因此，输出了 1，1 表示真。
1  #<== 输出 1 说明上述表达式的结果是对的。
[root@oldboy ~]# echo $((8<3))  #<==8<3 的结果是不成立的，因此，输出了 0，0 表示假。
0  #<== 输出 0 说明上述表达式的结果是错的。
[root@oldboy ~]# echo $((8==8)) #<== 判断是否相等。
1
[root@oldboy ~]# if ((8>7&&5==5))  #<== 如果 8>7 成立并且 5==5 成立，则打印 yes。显
                                       然这两个条件都成立。
> then                           #<== 这是一个简单的命令行 if 语句格式。
> echo yes
> fi
yes                              #<== 因此结果输出了 yes。
```

提示：上面涉及的数字及变量必须为整数（整型），不能是小数（浮点数）或字符串。后面的 bc 和 awk 命令可以用于进行小数（浮点数）运算，但一般用到的较少，下文对此还会讲解。

范例 5-5：在变量前后使用 -- 和 ++ 特殊运算符的表达式。

```
[root@oldboy ~]# a=10
```

```
[root@oldboy ~]# echo $((a++))
#<== 如果 a 在运算符 (++ 或 --) 的前面，那么在输出整个表达式时，会输出 a 的值，因为 a 为 10，
     所以表达式的值为 10。
10
[root@oldboy ~]# echo $a #<== 执行上面的表达式后，因为有 a++，因此 a 会自增 1，因此输
                              出 a 的值为 11。
11
[root@oldboy ~]# a=11
[root@oldboy ~]# echo $((a--))
#<== 如果 a 在运算符 (++ 或 --) 的前面，那么在输出整个表达式时，会输出 a 的值，因为 a 为 11，
     所以表达式的值的为 11。
11
[root@oldboy ~]# echo $a #<== 执行上面的表达式后，因为有 a--，因此 a 会自动减 1，因此
                              a 为 10。
10
[root@oldboy ~]# a=10
[root@oldboy ~]# echo $a
10
[root@oldboy ~]# echo $((--a))
#<== 如果 a 在运算符 (++ 或 --) 的后面，那么在输出整个表达式时，先进行自增或自减计算，因为 a
     为 10，且要自减，所以表达式的值为 9。
9
[root@oldboy ~]# echo $a #<== 执行上面的表达式后，a 自减 1，因此 a 为 9。
9
[root@oldboy ~]# echo $((++a))
#<== 如果 a 在运算符 (++ 或 --) 的后面，输出整个表达式时，先进行自增或自减计算，因为 a 为 9，
     且要自增 1，所以输出 10。
10
[root@oldboy ~]# echo $a  #<== 执行上面的表达式后，a 自增 1，因此 a 为 10。
10
```

执行 echo $((a++)) 和 echo $((a--)) 命令输出整个表达式时，输出的值即为 a 的值，表达式执行完毕后，会对 a 进行 ++、-- 的运算，而执行 echo $((++a)) 和 echo $((--a)) 命令输出整个表达式时，会先对 a 进行 ++、-- 的运算，然后再输出表达式的值，即为 a 运算后的值。

提示：有关 ++、-- 运算的记忆方法：

变量 a 在运算符之前，输出表达式的值为 a，然后 a 自增或自减；变量 a 在运算符之后，输出表达式会先自增或自减，表达式的值就是自增或自减后 a 的值。如果实在理解不了这里的 ++ 和 --，跳过就可以了，不会影响读者学好 Linux 运维，在工作中，使用它们前先测试好结果即可。

范例 5-6：通过“(())”运算后赋值给变量。

```
[root@oldboy ~]# myvar=99
```

```
[root@oldboy ~]# echo $((myvar+1))          #<== "(())" 中的变量 myvar 前也可以加
                                                $ 符号，也可以不加。
100
[root@oldboy ~]# echo $((  myvar + 1  ))    #<== "(())" 内部内容的两端有几个空格无
所谓，变量和运算符号之间有无空格也无所谓，可以有一个或多个，也可以没有。
100
[root@oldboy ~]# myvar=$((myvar+1))         #<== 还可以在 "(())" 表达式前加 $ 符号，
                                                将表达式赋值给变量。
[root@oldboy ~]# echo $myvar
100
```

提示：在 "(())" 中使用变量时可以去掉变量前的 $ 符号。

范例 5-7：包含 "(())" 的各种常见运算符命令行的执行示例。

```
[root@oldboy ~]# echo $((6+2))  #<== 加法
8
[root@oldboy ~]# echo $((6-2))  #<== 减法
4
[root@oldboy ~]# echo $((6*2))  #<== 乘法
12
[root@oldboy ~]# echo $((6/2))  #<== 除法，取商数
3
[root@oldboy ~]# echo $((6%2))  #<== 取模，即余数
0
[root@oldboy ~]# echo $((6**2)) #<== 幂运算
36
```

提示：

- "(())" 表达式在命令行执行时不需要加 $ 符号，直接使用 ((6%2)) 形式即可，但是如果需要输出，就要加 $ 符，例如：echo $((6%2))。
- "(())" 里的所有字符之间没有空格、有一个或多个空格都不会影响结果。

范例 5-8：各种 "(())" 运算的 Shell 脚本示例。

```
[root@oldboy scripts]# cat test.sh
#!/bin/bash
a=6                         #<== 在脚本中定义 a 和 b 两个变量并分别赋值。
b=2
echo "a-b=$(($a-$b))"       #<== 对定义的变量值进行各种符号运算，并通过表达式的形式输出，下同。
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

建议读者手动输入一遍这个脚本，并执行看看结果。这个例子很重要，后文经常会使用到，可保留起来备用。

其执行结果如下，看看能否理解该结果：

```
[root@oldboy scripts]# sh test.sh
a-b=4
a+b=8
a*b=12
a/b=3
a**b=36
a%b=0
```

范例 5-9：把范例 5-8 脚本中的 a、b 两个变量通过命令行脚本传参，以实现混合运算脚本的功能。

这是一个考察实战编程思想的综合实践考试题，将涉及前面提到的特殊位置参数变量的知识。

提示：该范例的答题时间为 5 分钟，建议读者先不要看答案，思考一下。

参考答案 1：

```
[root@oldboy scripts]# cat test.sh
#!/bin/bash
a=$1    #<== 直接把特殊位置参数变量 $1 赋值给 a，
b=$2    #<== 并且把特殊位置参数变量 $2 赋值给 b，这样，脚本传参的内容就会赋值给 a 和 b。
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

执行结果如下：

```
[root@oldboy scripts]# sh test.sh 6 2
a-b=4
a+b=8
a*b=12
a/b=3
a**b=36
a%b=0
[root@oldboy scripts]# sh test.sh 10 5
a-b=5
a+b=15
a*b=50
a/b=2
a**b=100000
a%b=0
```

使用脚本传参的好处是可以进行各种数字间的运算，不像前一个脚本，因为是直接定义变量的，所以只能做 6 和 2 这两个数字的运算，也就是说，使用传参，可以让脚本更具备通用性。

参考答案 2[⊖]：

```
#!/bin/bash
echo "a-b =$(( $1 - $2 ))"
echo "a+b =$(( $1 + $2 ))"
echo "a*b =$(( $1 * $2 ))"
echo "a/b =$(( $1 / $2 ))"
echo "a**b =$(( $1 ** $2 ))"
echo "a%b =$(( $1 % $2 ))"
```

老男孩点评：这个方法虽然可以实现同样的功能，但是对原脚本的改动过大，不过，可以看出，该同学对编程思想已有一定的领悟，只是仍需进一步提高。

范例 5-10：实现输入 2 个数进行加、减、乘、除功能的计算器。

此题超过了本书当前已学的知识范围，读者可以在学完本书后再回来查看此题的解答方案。

参考答案 1：本解答方案利用了 read 命令的读入功能，并对读入的内容是否为整数，传入的符号是否符合加、减、乘、除之一做了判断。

```
#!/bin/bash
#add, subtract, multiply and divide   by  yubing   2013-07-13
print_usage(){                                  #<== 定义一个函数，名字为 print_usage。
    printf "Please enter an integer\n"    #<== 打印符合脚本要求的提示信息。
    exit 1    #<== 以返回值 1 退出脚本，这个在前面讲特殊进程变量时已经讲过这个返回值的
                用法了。
}
read -p "Please input first number: " firstnum #<== 读入第一个数字，本章后文将讲
                                                    解 read。
if [ -n "`echo $firstnum|sed 's/[0-9]//g'`" ];then
#<== 判断是否为整数，删除读入内容的数字部分看是否为空 (-n 功能)，进而判断读入的内容是否为数字。
    print_usage    #<== 如果上述条件变量值不为空，说明不是整数，则调用用户帮助函数。
fi
read -p "Please input the operators: " operators    #<== 继续读入运算符。
if [ "${operators}" != "+" ]&&[ "${operators}" != "-" ] && [ "${operators}"
!= "*" ] && [ "${operators}" != "/" ];then #<== 判断第二个输入内容操作符是否为 +-*/ 任
                                                意运算符之一。
    echo "please use {+|-|*|/}"          #<== 如果操作符不符合要求，则给出提示。
    exit 2      #<== 因为不符合要求，因此以返回值 2 退出脚本，表示出现错误了。
fi
read -p "Please input second number: " secondnum    #<== 读入第二个要运算的数字。
if [ -n "`echo $secondnum|sed 's/[0-9]//g'`" ];then  #<== 同第一个运算的数字，判断
                                                        是否为整数。
```

⊖ 此方法为老男孩教育的学生实现的方法。

```
    print_usage #<== 如果上述条件变量值不为空，说明不是整数，则调用用户帮助函数。
fi
echo "${firstnum}${operators}${secondnum}=$((${firstnum}${operators}${seco
ndnum}))"          #<== 上述条件都符合后，进入运算，输出运算表达式和计算结果。
```

执行结果如下：

```
[root@oldboy scripts]# sh 05_10_jisuan.sh
Please input first number: 6    #<== 输入数字 6。
Please input the operators: -   #<== 输入运算符号减号。
Please input second number: 2   #<== 输入数字 2。
6-2=4
[root@oldboy scripts]# sh 05_10_jisuan.sh
Please input first number: 6
Please input the operators: *
Please input second number: 3
6*3=18
[root@oldboy scripts]# sh 05_10_jisuan.sh
Please input first number: oldboy #<== 当输入非数字时，系统就会提示“请输入数字”，并
                                      终止程序运行。
Please enter an integer           #<== 提示“请输入数字”。
[root@oldboy scripts]# sh 05_10_jisuan.sh
Please input first number: 10
Please input the operators: ^     #<== 运算符号不符合 +-*/ 之一时，也会提示，并终止程
                                      序运行。
please use {+|-|*|/}
```

参考答案 2：本解答方案利用了脚本命令行传参的功能，并对传参的内容是否为整数，传入的符号是否符合加、减、乘、除之一做了判断。

```
[root@oldboy scripts]# cat 05_11_jisuan.sh
#!/bin/bash
#add, subtract, multiply and divide   by  oldboy   2013-07-13
print_usage(){
    printf $"USAGE:$0 NUM1 {+|-|*|/} NUM2\n"
    exit 1
}
if [ $# -ne 3 ] #<== 如果脚本传入的参数个数不等于 3 个（因为要输入两个数字及一个运算符号）。
  then
    print_usage #<== 则调用用户帮助函数。
fi

firstnum=$1  #<== 第一个数字，为了减小对原脚本的改动，这里将 $1 赋值给 firstnum，下同。
secondnum=$3
op=$2

if [ -n "`echo $firstnum|sed 's/[0-9]//g'`" ];then  #<== 是否为整数的判断，同上。
    print_usage
fi
```

```
if [ "$op" != "+" ]&&[ "$op" != "-" ]&&[ "$op" != "*" ]&&[ "$op" != "/" ]
  #<== 判断第二个输入的操作符是否为 +、-、*、/ 任意运算符之一。
  then
    print_usage
fi

if [ -n "`echo $secondnum|sed 's/[0-9]//g'`" ];then  #<== 是否为整数的判断，同上。
    print_usage
fi
echo "${firstnum}${op}${secondnum}=$((${firstnum}${op}${secondnum}))"
```

执行结果如下：

```
[root@oldboy scripts]# sh 05_11_jisuan.sh 6 + 2
6+2=8
[root@oldboy scripts]# sh 05_11_jisuan.sh 6 - 2
6-2=4
[root@oldboy scripts]# sh 05_11_jisuan.sh 6 \* 2    #<==* 号要转义。
6*2=12
[root@oldboy scripts]# sh 05_11_jisuan.sh 6 / 2
6/2=3
[root@oldboy scripts]# sh 05_11_jisuan.sh old / 2   #<== 若输入的参数不符合要求，
                                                        则给予提示。
USAGE:05_11_jisuan.sh NUM1 {+|-|*|/} NUM2
[root@oldboy scripts]# sh 05_11_jisuan.sh 6 / girl  #<== 若输入的参数不符合要求，
                                                        则给予提示。
USAGE:05_11_jisuan.sh NUM1 {+|-|*|/} NUM2
```

参考答案 3：此为高效、简单的方法，只用一个 $1 进行计算，传参时传一个表达式就可以了。

```
[root@oldboy scripts]# cat jisuan1.sh
echo $(($1))
[root@oldboy scripts]# sh jisuan1.sh 6+2
8
[root@oldboy scripts]# sh jisuan1.sh "6 + 2"  #<== 如果有空格，则要加双引号。
8
```

提示：本脚本没有对输入的参数做判断。

5.3　let 运算命令的用法

let 运算命令的语法格式为：let 赋值表达式

let 赋值表达式的功能等同于“((赋值表达式))”。

范例 5-11：给自变量 i 加 8。

```
[root@oldboy scripts]# i=2
[root@oldboy scripts]# i=i+8      #<== 假如开头不用 let 进行赋值。
[root@oldboy scripts]# echo $i    #<== 输出时会发现，打印结果为 i+8，也就是没有计算。
i+8
[root@oldboy scripts]# unset i
[root@oldboy scripts]# i=2
[root@oldboy scripts]# let i=i+8 #<== 采用 let 赋值后再输出。
[root@oldboy scripts]# echo $i
10  #<== 结果为 10
```

提示：let i=i+8 等同于 ((i=i+8))，但后者效率更高。

范例 5-12：监控 Web 服务状态，如果访问两次均失败，则报警（let 应用案例）。

为了给读者呈现真正的实战案例，此题也超越了当下本书已讲解的内容范围，读者若无法完全理解（如果认真看注释，应该是可以弄懂的），那么先留着，等看完全书后再回来细读，这不会影响学习。

以下以简单的企业实战脚本作为参考答案（更专业更规范的企业实战脚本见后文）：

```
[root@oldboy scripts]# cat 05_12_checkurl.sh
CheckUrl() {  #<== 定义函数，名字为 CheckUrl。
timeout=5     #<== 定义 wget 访问的超时时间，超时就退出。
fails=0       #<== 初始化访问网站失败的次数记录变量，若失败达到两次，就发邮件报警。
success=0     #<== 初始化访问网站成功的次数记录变量，如果为 1，则表示成功，退出脚本。
while true    #<== 持续循环检测。
  do
    wget --timeout=$timeout --tries=1 http://oldboy.blog.51cto.com -q -O /dev/null
    #<== 使用 wget 测试访问老男孩的博客地址。
    if [ $? -ne 0 ]  #<== 如果上述 wget 命令执行不成功，即返回值不为 0，则执行 if 语
                        句内的指令。
      then
        let fails=fails+1  #<== 将访问失败的次数加 1，这个就是 let 的用法，可以用
                               ((fails=fails+1)) 代替。
    else
        let success+=1     #<== 返回值不为 0 则不成立，即访问成功，将成功的次数加 1。
    fi
    if [ $success -ge 1 ]  #<== 如果成功的次数大于等于 1
      then
        echo success    #<== 则打印成功标识，这也可以用冒号（:）替代，不输出结果，这是
                            为了观察方便。
        exit 0             #<== 返回 0 值，退出脚本，表示检测成功。
    fi
    if [ $fails -ge 2 ]    #<== 如果失败的次数大于等于 2，则报警。
      then
        Critical="sys is down."
        echo $Critical|tee|mail -s "$Critical" abc@oldboyedu.com
        #<== 输出并发邮件报警，这里需要单独配你自己的邮箱地址，别用作者这里写的。
```

```
        exit 2
    fi
done
}
CheckUrl  #<== 执行函数。
```

提示：实际上 wget 命令有自动重试的功能，--tries=1 参数就是，这里以一个脚本为大家阐述编程思想及 let 的应用案例。

执行结果如下：

```
[root@oldboy scripts]# sh 05_12_checkurl.sh
success
[root@oldboy scripts]# sh 05_12_checkurl.sh    #<== 当无法访问地址时会输出错误。
sys is down.
[root@oldboy scripts]# sh -x 05_12_checkurl.sh #<== 使用 -x 可以跟踪详细的执行过程。
+ CheckUrl
+ timeout=5
+ fails=0
+ success=0
+ true
+ wget --timeout=5 --tries=1 http://oldboy.old.51cto.com -q -O /dev/null
#<== 故意搞错地址。
+ '[' 4 -ne 0 ']'    #<== 返回值不是 0，因此，将失败次数加 1。
+ let fails=fails+1 #<== 访问失败后次数加 1。
+ '[' 0 -ge 1 ']'
+ '[' 1 -ge 2 ']'    #<== 因为没有达到两次失败，因此不报警。
+ true
+ wget --timeout=5 --tries=1 http://oldboy.old.51cto.com -q -O /dev/null
#<== 继续第 2 次访问。
+ '[' 4 -ne 0 ']'
+ let fails=fails+1
+ '[' 0 -ge 1 ']'
+ '[' 2 -ge 2 ']'    #<== 若达到两次失败的阈值，则启动报警装置。
+ Critical='sys is down.'
+ echo sys is down.
sys is down.
+ mail -s 'sys is down.' abc@oldboyedu.com
+ echo sys is down.
+ exit 2
```

5.4　expr 命令的用法

5.4.1　expr 命令的基本用法示例

expr（evaluate（求值）expressions（表达式））命令既可以用于整数运算，也可以用

于相关字符串长度、匹配等的运算处理。

1. expr 用于计算

语法：expr Expression

范例 5-13：expr 命令运算用法实践。

```
[root@oldboy scripts]# expr 2 + 2
4
[root@oldboy scripts]# expr 2 - 2
0
[root@oldboy scripts]# expr 2 * 2 #<==* 号用 \ 来转义。
expr: 语法错误
[root@oldboy scripts]# expr 2 \* 2
4
[root@oldboy scripts]# expr 2 / 2
1
```

要注意，在使用 expr 时：

- 运算符及用于计算的数字左右都至少有一个空格，否则会报错。
- 使用乘号时，必须用反斜线屏蔽其特定含义，因为 Shell 可能会误解星号的含义。

2. expr 配合变量计算

expr 在 Shell 中可配合变量进行计算，但需要用反引号将计算表达式括起来。

范例 5-14：给自变量 i 加 6。

```
[root@oldboy scripts]# i=5
[root@oldboy scripts]# i=`expr $i + 6` #<== 注意用反引号将表达式引起来，变量和数字
                                            符号两边要有空格。
[root@oldboy scripts]# echo $i
11
```

5.4.2 expr 的企业级实战案例详解

1. 判断一个变量值或字符串是否为整数

在 Shell 编程里，由于函数库很少，所以判断字符串是否为整数就不是一件很容易的事情。在这里，老男孩为读者介绍一种简单的可以判断一个字符串是否为整数的方法。

实现原理是，利用以 expr 做计算时变量或字符串必须是整数的规则，把一个变量或字符串和一个已知的整数（非 0）相加，看命令返回的值是否为 0。如果为 0，就认为做加法的变量或字符串为整数，否则就不是整数。下面给出几个示例。

范例 5-15：通过 expr 判断变量或字符串是否为整数。

```
[root@oldboy scripts]# i=5                          #<== 赋值一个数 5 给 i。
[root@oldboy scripts]# expr $i + 6 &>/dev/null #<== 把 i 和整数相加，&>/dev/null
                                                    表示不保留任何输出。
[root@oldboy scripts]# echo $?                      #<== 输出返回值。
0  #<== 返回为 0，则证明 i 的值为整数。
```

```
[root@oldboy scripts]# i=oldboy  #<== 此时赋值一个字符串给 i。
[root@oldboy scripts]# expr $i + 6 &>/dev/null  #<== 同样把 i 和整数相加，不保留
                                                    任何输出。
[root@oldboy scripts]# echo $?   #<== 输出返回值。
2  #<== 返回为非 0，则证明 i 的值不是整数，因为赋值的是 oldboy。
```

结论：利用 expr 做计算，将一个未知的变量和一个已知的整数相加，看返回值是否为 0，如果为 0 就认为做加法的变量为整数，否则就不是整数。

范例 5-16：通过传参判断输出内容是否为整数。

```
[root@oldboy scripts]# cat 05_16_expr.sh
#!/bin/sh
expr $1 + 1 >/dev/null 2>&1
[ $? -eq 0 ] &&echo int||echo chars  #<== 这是一个条件表达式语法，返回值为 0，则输出
                                         int，否则输出 chars。
[root@oldboy scripts]# sh 05_16_expr.sh oldboy  #<== 传入 oldboy 字符。
chars  #<== 返回的是字符
[root@oldboy scripts]# sh 05_16_expr.sh 119     #<== 传入 119。
int  #<== 返回的是数字
```

范例 5-17：通过 read 读入持续等待输入例子（此为范例 5-16 的加强版）。

```
[root@oldboy scripts]# cat judge_int.sh
#!/bin/sh
while true
do
    read  -p "Pls input:" a
    expr $a + 0 >/dev/null 2>&1
    [ $? -eq 0 ] && echo int || echo chars
done
```

执行结果如下：

```
[root@oldboy scripts]# sh judge_int.sh
Pls input:oldgirl
chars
Pls input:120
int
Pls input:
```

范例 5-18：将前文的混合运算小程序改成输入两个参数后进行计算的程序，并且要能判断传参的个数及通过 expr 判断传入的参数是否为整数。

待使用的脚本如下：

```
[root@oldboy scripts]# cat test.sh
#!/bin/bash
a=6   #<== 在脚本中定义 a 和 b 两个变量并分别赋值。
b=2
```

```
echo "a-b=$(($a-$b))"   #<== 对定义的变量值进行四则运算，并通过表达式的形式输出，下同。
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

这道题可以使用游戏过关的思想来编程解决。

编程思路：

第一关，判断参数个数是否为 2，若不是，则给出提示终止运行。

第二关，判断传入的参数是否为整数，若不是，则给出提示终止运行。

第三关，做运算。

参考答案 1：

```
[root@oldboy scripts]# cat 05_18_1.sh
#!/bin/bash
#no.1
[ $# -ne 2 ] &&{                    #<== 对传入参数的个数进行判断，是否等于 2。
    echo $"USAGE $0 NUM1 NUM2"      #<== 若不满足条件，则打印提示。
    exit 1                          #<== 以返回值 1 退出脚本，表示脚本执行错误。
}
#no.2
a=$1                                #<== 获取脚本传参的 $1 并赋值给 a。
b=$2                                #<== 获取脚本传参的 $2 并赋值给 b。
expr $a + $b + 110 &>/dev/null      #<== 将两个参数一起和 110 相加。
if [ $? -ne 0 ] #<== 如果返回值不为 0，则至少有一个不是整数。
  then
    echo "you must input two nums." #<== 打印提示。
    exit 2    #<== 以返回值 2 退出脚本，表示脚本执行错误，为什么不用 1 呢，这里是为了区分前
                面的 1，便于调试。
fi
#no.3
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

执行结果如下：

```
[root@oldboy scripts]# sh 05_18_1.sh
USAGE 05_14_size.sh NUM1 NUM2
[root@oldboy scripts]# sh 05_18_1.sh 8 2
a-b=6
a+b=10
a*b=16
a/b=4
```

```
a**b=64
a%b=0
```

参考答案 2：

```
[root@oldboy scripts]# cat 05_18_2.sh
#!/bin/bash
#no.1
[ $# -ne 2 ] &&{
    echo $"USAGE $0 NUM1 NUM2"
    exit 1
}
#no.2
a=$1
b=$2
expr $a + 1 &>/dev/null                          #<== 将第一个变量和整数相加。
RETVAL_A=$?                                      #<== 获取返回值。
expr $b + 1 &>/dev/null                          #<== 将第二个变量和整数相加。
RETVAL_B=$?                                      #<== 获取返回值。
if [ $RETVAL_A -ne 0 -o $RETVAL_B -ne 0 ] #<== 对两个变量通过 expr 加和过的返回值
                                                     进行判断。
  then
    echo "you must input two nums."
    exit 2
fi
#no.3
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

此外，用 expr match 功能进行整数判断时，可执行 man expr 命令获得帮助，如下：

```
[root@oldboy scripts]# cat t1.sh
if [[ `expr match "$1" "[0-9][0-9]*$"` == 0 ]]
  then
    echo "$1 is not a num"
else
    echo "$1 is a num"
fi
```

测试结果如下：

```
[root@oldboy scripts]# sh t1.sh 11
11 is a num
[root@oldboy scripts]# sh t1.sh oldboy
oldboy is not a num
[root@oldboy scripts]# sh t1.sh 31333741@qq.com
```

```
31333741@qq.com is not a num
```

2. expr 的特殊用法：判断文件扩展命名是否符合要求

范例 5-19：通过 expr 判断文件扩展名是否符合要求。

```
[root@oldboy scripts]# cat expr1.sh
#!/bin/sh
if expr "$1" : ".*\.pub" &>/dev/null
  then
    echo "you are using $1"
else
    echo "pls use *.pub file"
fi
[root@oldboy scripts]# sh expr1.sh id_dsa.pub
you are using id_dsa.pub
[root@oldboy scripts]# sh expr1.sh id_dsa
pls use *.pub file
```

范例 5-20：使用 expr 命令实现系统 ssh 服务自带的 ssh-copy-id 公钥分发脚本。

```
[root@oldboy scripts]# sed -n '10,20p' `which ssh-copy-id`
if [ "-i" = "$1" ]; then
  shift
  # check if we have 2 parameters left, if so the first is the new ID file
  if [ -n "$2" ]; then
    if expr "$1" : ".*\.pub" > /dev/null ; then #<== 判断脚本传入的 $1 是否符合
                                                    .pub 扩展名。
#  man expr 结果。
# STRING : REGEXP
# anchored pattern match of REGEXP in STRING
      ID_FILE="$1"
    else
      ID_FILE="$1.pub"#<== 如果扩展名不符合要求，则自动加上扩展名，以防止用户发送私钥。
    fi
    shift               # and this should leave $1 as the target name
  fi
```

3. 通过 expr 计算字符串的长度

> **说明**：4.3 节讲变量子串时已提到过此功能。

范例 5-21：利用 expr 计算字符串的长度。

```
[root@oldboy scripts]# char="I am oldboy"
[root@oldboy scripts]# expr length "$char" #<== 利用 expr 的 length 函数计算字符
                                                串长度。
11
[root@oldboy scripts]# echo ${#char}        #<== 计算变量子串长度的方法。
```

```
11
[root@oldboy scripts]# echo ${char}|wc -L    #<==wc 方法
11
[root@oldboy scripts]# echo ${char}|awk '{print length($0)}' #<== 利用 awk 的
length 函数来计算字符串的长度。
11
```

范例 5-22：请编写 Shell 脚本，打印下面语句中字符数不大于 6 的单词。

I am oldboy linux welcome to our training

```
[root@oldboy scripts]# cat word_length.sh
for n in I am oldboy linux welcome to our training
do
    if [ `expr length $n` -le 6 ] #<== 利用 expr 的 length 函数计算字符串长度，并输
                                      出长度不大于 6 的字符串。
      then
        echo $n
    fi
done
```

执行结果如下：

```
[root@oldboy scripts]# sh word_length.sh
I
am
oldboy
linux
to
our
```

提示：此题的解题方法众多，后文会有多种解法（见第 13 章），这里仅给出一种关于 expr 的方法，读者也可以替换为其他计算长度的方法。更多的 expr 知识，可通过 man expr 来获得。

5.5　bc 命令的用法

bc 是 UNIX/Linux 下的计算器，因此，除了可以作为计算器来使用，还可以作为命令行计算工具使用。

范例 5-23：将 bc 作为计算器来应用。

```
[root@oldboy scripts]# bc          #<== 执行 bc 后，交互式计算。
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
```

```
For details type `warranty'.
1+1                                                    #<== 输入 1+1，按回车键计算。
2
3*3                                                    #<== 输入 3+3 回车后计算。
9
```

范例 5-24：将 bc 用在命令行下面，以实现运算功能。

```
[root@oldboy scripts]# echo 3+5|bc
8
[root@oldboy scripts]# echo 3.3+5.5|bc
8.8
[root@oldboy scripts]# echo 8.8-5.5|bc
3.3
[root@oldboy scripts]# echo "scale=2;355/113"|bc #<== 使用 scale=2 保留两位小数。
3.14
[root@oldboy scripts]# echo "scale=6;355/113"|bc #<== 使用 scale=6 保留 6 位小数。
3.141592
```

利用 bc 配合变量运算：

```
[root@oldboy scripts]# i=5
[root@oldboy scripts]# i=`echo $i+6|bc` #<== 利用 echo 输出表达式，通过管道交给 bc
                                            计算。此方法效率较低。
[root@oldboy scripts]# echo $i
11
```

提示：根据 bc 所具有的特殊性来看，如果是小数，则选择 bc 运算没有问题（老男孩推荐 awk）；若是整数场景，可用“(())”、let、expr 等。

范例 5-25：通过一条命令计算输出 1+2+3+…+10 的表达式，并计算出结果，请使用 bc 命令计算。输出内容如 1+2+3+4+5+6+7+8+9+10=55。

这里生成 1+2+3+4+5+6+7+8+9+10 表达式的方法有：

```
[root@oldboy scripts]# seq -s "+" 10    #<== seq 是生成数字序列，-s 是指定数字序列
                                            之间的分隔符。
1+2+3+4+5+6+7+8+9+10
[root@oldboy scripts]# echo {1..10}|tr " " "+"
#<=={1..10} 是生成以空格为间隔的数字序列，并交给 tr 将空格替换为 + 号。
1+2+3+4+5+6+7+8+9+10
```

实现本题的多种方法如下：

```
[root@oldboy scripts]# echo `seq -s '+' 10`=`seq -s "+" 10|bc` #<== 使用 bc 计算
1+2+3+4+5+6+7+8+9+10=55
[root@oldboy scripts]# echo "`seq -s '+' 10`="$((`seq -s "+" 10`))#<== 使用“(())”计算
1+2+3+4+5+6+7+8+9+10=55
```

```
[root@oldboy scripts]# echo `seq -s '+' 10`=`seq -s " + " 10|xargs expr`#<== 使用 expr 计算
1+2+3+4+5+6+7+8+9+10=55
[root@oldboy scripts]# echo `seq -s "+" 10`=$(echo $[`seq -s "+" 10`])#<== 使用 $[] 计算
1+2+3+4+5+6+7+8+9+10=55
```

bc 命令的独有特点是除了支持整数运算之外，还支持小数运算。

5.6　awk 实现计算

利用 awk 进行运算的效果也很好，适合小数和整数，特别是命令行计算，尤其是小数，运算很精确，好用。来看个示例，如下：

```
[root@oldboy scripts]# echo "7.7 3.8" |awk '{print ($1-$2)}'
#<==$1 为第一个数字，$2 为第二个数字，用空格隔开，下同。
3.9
[root@oldboy scripts]# echo "358 113" |awk '{print ($1-3)/$2}'
3.14159
[root@oldboy scripts]# echo "3 9" |awk '{print ($1+3)*$2}'
54
```

5.7　declare（同 typeset）命令的用法

下面将要讲解的是使用 typeset 定义整数变量，直接进行计算。这个方法不是很常用，因为需要定义才能生效。示例如下：

```
[root@oldboy scripts]# declare -i A=30 B=7 #<==declare -i 参数可以将变量定义为整形。
[root@oldboy scripts]# A=A+B  #<== 因为已声明是整型，因此可以直接进行运算了。
[root@oldboy scripts]# echo $A
37  #<== 结果为 37（老男孩 37 岁了，还在奋斗。）
```

5.8　$[] 符号的运算示例

关于 $[] 符号运算的示例如下：

```
[root@oldboy scripts]# i=5
[root@oldboy scripts]# i=$[i+6]
[root@oldboy scripts]# echo $i
11
[root@oldboy scripts]# echo $[2*3]
6
[root@oldboy scripts]# echo $[2**3]
8
[root@oldboy scripts]# echo $[3/5]
0
```

```
[root@oldboy scripts]# echo $[3/2]
1
[root@oldboy scripts]# echo $[3%5]
3
[root@oldboy scripts]# echo $[ 3 % 5 ]
3
```

下面是一个解决实际问题的示例：打印数学杨辉三角。

```
#!/bin/bash
if (test -z $1) ;then     #<== 判断传参的值长度是不是为 0，如果没有传入参数，则使用 read 读入。
    read -p "Input Max Lines:" MAX  #<==read 读入一个数值。
else
    MAX=$1                #<== 如果已经传参了，就把传参的 $1 赋值给 MAX。
fi
#<== 上述脚本很巧妙地通过判断，实现了用户既可以传参输入，也可以 read 读入数字。
i=1
while [ $i -le $MAX ]     #<==i 行控制。
do
    j=1
    while [ $j -le $i ]   #<==j 列控制。
    do
        f=$[i-1]          #<==f=i-1 是 $[] 计算写法。
        g=$[j-1]          #<==g=j-1 是 $[] 计算写法。
        if [ $j -eq $i ] || [ $j -eq 1 ] ; then
            declare SUM_${i}_$j=1                #<== 声明变量头尾都是 1。
        else
            declare A=$[SUM_${f}_$j]             #<== 取上一行的 j 列变量。
            declare B=$[SUM_${f}_$g]             #<== 取上一行的 j-1 列变量。
            declare SUM_${i}_$j=`expr $A + $B`   #<== 声明并计算当前变量的值。
        fi
        echo -en $[SUM_${i}_$j]" "               #<== 输出当前变量。
        let j++                                  #<==let 运算用法。
    done
    echo                                         #<== 换行。
    let i++                                      #<==let 运算用法。
done
```

有关用 Shell 脚本实现杨辉三角的细节和 3 个实例请参见老男孩的博文（http://oldboy.blog.51cto.com/2561410/756234），这里不再多提，此题对于运维实战的意义不大，仅在于练习编程能力和思想。

5.9 基于 Shell 变量输入 read 命令的运算实践

5.9.1 read 命令基础

Shell 变量除了可以**直接赋值**或**脚本传参**外，还可以使用 read 命令从标准输入中获

得，read 为 bash 内置命令，可通过 help read 查看帮助。

语法格式：read [参数] [变量名]

常用参数如下。

- -p prompt：设置提示信息。
- -t timeout：设置输入等待的时间，单位默认为秒。

来看几个示例。

范例 5-26：实现 read 的基本读入功能。

```
[root@oldboy scripts]# read -t 10 -p "Pls input one num:" num
#<== 读入一个输入，赋值给 num 变量，注意，num 变量前需要有空格。
Pls input one num:18                #<== 输出数字 18，相当于把 18 赋值给 num 变量。
[root@oldboy scripts]# echo $num #<== 输出变量值。
[root@oldboy scripts]# read -p "please input two number:" a1 a2
#<== 读入两个输入，注意要以空格隔开，分别赋值给 a1 和 a2 变量，a1 变量前后都需要有空格。
please input two number:1 2
[root@oldboy scripts]# echo $a1
1
[root@oldboy scripts]# echo $a2
2
```

提示：read 的读入功能就相当于交互式接受用户输入，然后给变量赋值。

上面 read -p 的功能可以用 echo 和 read 来实现，如下：

```
echo -n "please input two number:"
read a1 a2
```

以上两句和下面的命令相当（-t 排除在外）。

```
read -t 5 -p "please input two number:" a1 a2    #5 秒超时退出
```

范例 5-27：把前面加减乘除计算传参的脚本改成通过 read 方式读入整数变量。

原始脚本如下：

```
#!/bin/bash
a=$1
b=$2
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

解答：

```
[root@oldboy scripts]# cat test_2.sh
#!/bin/bash
read -t 15 -p "please input two number:" a b  #<== 去掉原脚本中 a 和 b 的定义，通
                                                过 read 读入即可。
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
[root@oldboy scripts]# sh test_2.sh
please input two number:10 5
a-b=5
a+b=15
a*b=50
a/b=2
a**b=100000
a%b=0
```

下面是初学者的多种典型错误案例，大家一起来找茬。

典型错误案例 1：

```
#!/bin/bash
a=$1 #<== 将 $1 赋值给 a，脚本传参和 read 读入，保留一种即可，这里该删掉。
b=$2 #<== 将 $2 赋值给 b，脚本传参和 read 读入，保留一种即可，这里该删掉。
read -p "pls input"  #<== 这里的 read 没有用了，而且没有变量可接收用户输入。
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

错误在于：没有搞懂 read 的用法，并且将传参和 read 混用了。

典型错误案例 2：

```
[root@mysql oldboy]# vim a.sh
#!/bin/bash
read -p "pls input": "$1" "$2"  #<==$1 和 $2 本来是有特定功能的变量，不能用其作为变
                                  量来接收 read 读入。
a=$1  #<== 脚本传参和 read 读入，保留一种即可，这里该删掉。
b=$2  #<== 脚本传参和 read 读入，保留一种即可，这里该删掉。
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

错误在于：没有搞懂 read 用法，read 后面接的是普通变量，并且将传参和 read 混用了，传参和 read 可以理解为是两种变量赋值的方法，使用其一即可。

典型错误案例 3：

```
#!/bin/bash
a=$1  #<== 脚本传参和 read 读入，保留一种即可，这里该删掉。
b=$2  #<== 脚本传参和 read 读入，保留一种即可，这里该删掉。
read -p "diyige":$a #<== 作为接收 read 的变量，不该带 $ 符号。
read -p "dierge":$b #<== 作为接收 read 的变量，不该带 $ 符号。
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
echo yigong=${#}
```

5.9.2　以 read 命令读入及传参的综合企业案例

先来思考一下：如果前面范例 5-27 中读入的不是整数或者输入的数字个数不是 2，那么执行脚本会有什么结果？又该如何解决呢？

其实，这里要用到的思想在讲解 expr 命令时已经提到过了，这里再以其他方法进行讲解。

该问题同样可用打游戏过关的思路来解决。

第一关：若用户按要求输入了两个值，则过关，否则 game over。

　　编程提示：可以用变量的子串长度知识，例如：${#OLDBOY}。

第二关：用户输入的内容均为整数，才能过关，否则 game over。

　　编程提示：用 expr 或其他特殊技巧进行判断。

第三关：当读入的参数符合个数和整数条件时，进行计算。

解答过程如下。

以 read 命令进行读入判断。

```
[root@oldboy scripts]# cat read_size01.sh
#!/bin/bash
read -t 15 -p "Please input two number:" a b
#no1
[ ${#a} -le 0 ]&&{ #<== 利用条件表达式，根据变量值的长度是否小于 0，来确定第一个数是否为空。
    echo "the first num is null"
    exit 1
}
[ ${#b} -le 0 ]&&{ #<== 利用条件表达式，根据变量值的长度是否小于 0，来确定第二个数是否为空。
    echo "then second num is null"
    exit 1
}
```

```
#no2 #<== 这里的用法在前面已详细注释过，因此这里不再注释。
expr $a + 1 &>/dev/null
RETVAL_A=$?
expr $b + 1 &>/dev/null
RETVAL_B=$?
if [ $RETVAL_A -ne 0 -o $RETVAL_B -ne 0 ];then
    echo "one of the num is not num,pls input again."
    exit 1
fi

#no.3
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a/b=$(($a/$b))"
echo "a**b=$(($a**$b))"
echo "a%b=$(($a%$b))"
```

执行结果如下：

```
[root@oldboy scripts]# sh read_size01.sh
Please input two number:qq                    #<== 若输入非数字，则会报错。
then second num is null
[root@oldboy scripts]# sh read_size01.sh
Please input two number:qq dd
one of the num is not num,pls input again.
[root@oldboy scripts]# sh read_size01.sh
Please input two number:12 6
a-b=6
a+b=18
a*b=72
a/b=2
a**b=2985984
a%b=0
```

再来思考一下：如何将上述 read 读入改用脚本传参的方法来实现呢？

此题前文已经给过答案，这里给出其他脚本方法，如下：

```
[root@oldboy scripts]# cat read_size02.sh
#!/bin/bash
a=$1                                          #<== 将 $1 赋值给 a。
b=$2                                          #<== 将 $2 赋值给 b。
Usage(){                                      #<== 定义帮助函数。
    echo $"USAGE:sh $0 num1 num2"             #<== 输出帮助，$0 表示脚本名。
    exit 1                                    #<== 以 1 作为返回值退出脚本。
}
if [ $# -ne 2 ];then                          #<== 参数个数不等于 2，打印帮助。
    Usage
fi
```

```
expr $a + 1 >/dev/null 2>&1
[ $? -ne 0 ] && Usage
expr $b + 0 >/dev/null 2>&1  #<== 这个地方使用的加0，是为了防止b为0而导致被除数为0。
[ $? -ne 0 ] && Usage
#no.3
echo "a-b=$(($a-$b))"
echo "a+b=$(($a+$b))"
echo "a*b=$(($a*$b))"
echo "a**b=$(($a**$b))"
if [ $b -eq 0 ]  #<== 这个地方对b做了个判断，防止被除数为0，算是个保险吧，可能用不上。
  then
    echo "your input does not allow to run."
    echo "a/b =error"
    echo "a%b =error"
else
    echo "a/b =$(($a/$b))"
    echo "%b =$(($a%$b))"
fi
```

执行结果如下：

```
[root@oldboy scripts]# sh read_size02.sh
USAGE:sh read_size02.sh num1 num2
[root@oldboy scripts]# sh read_size02.sh 7 oldboy
USAGE:sh read_size02.sh num1 num2
[root@oldboy scripts]# sh read_size02.sh 7 3
a-b=4
a+b=10
a*b=21
a**b=343
a/b =2
a%b =1
[root@oldboy scripts]# sh read_size02.sh 7 0
USAGE:sh read_size02.sh num1 num2
```

下面是几位同学给出的具有一定编程思想的答案，但不建议采用该编程方法，因为不易读、不易改，特别是对于新手来说，还是按照过关的编程方法来解决较好，那样思路更清晰、更易懂。

老男孩不建议的答案1：

```
#!/bin/bash
read -p "please insert values: " a b
expr 1 + $a &>/dev/null
A=$?
expr 1 + $b &>/dev/null
B=$?
if [ ! -n "$a" ] || [ ! -n "$b" ]
  then
```

```
    echo "plese insert two values!"
    exit 1
elif [ "$A"  -ne 0 ] || [ "$B"  -ne 0 ]
  then
    echo "please insert two zhengshu!"
    exit 1
else
    echo "$a-$b =$(( $a - $b ))"
    echo "$a+$b =$(( $a + $b ))"
    echo "$a*$b =$(( $a * $b ))"
    echo "$a/$b =$(( $a / $b ))"
    echo "$a**$b =$(( $a ** $b ))"
    echo "$a%$b =$(( $a % $b ))"
    exit 0
fi
```

老男孩不建议的答案 2：

```
#!/bin/bash
read -p "Input: " a b
if expr 1 + $a >/dev/null 2>&1; then
    if expr 1 + $b >/dev/null 2>&1; then
        echo "a-b =$(( $a - $b ))"
        echo "a+b =$(( $a + $b ))"
        echo "a*b =$(( $a * $b ))"
        echo "a/b =$(( $a / $b ))"
        echo "a**b =$(( $a ** $b ))"
        echo "a%b =$(( $a % $b ))"
    else
        echo "Input Error 2 no: $b"
        exit 1
    fi
else
    echo "Input Error 1 no: $a"
    exit 1
fi
```

老男孩不建议的答案 3：

```
#!/bin/bash
#
read -p "Please input two numbers:" a b
if [ ! -n "$a" ];then
    echo "a lost"
    exit 2
fi
if [ ! -n "$b" ];then
    echo "b lost"
    exit 2
```

```
fi
expr 1 + $a &>/dev/null
if [ $? -eq 0 ];then
    expr 1 + $b &>/dev/null
    if [ $? -eq 0 ];then
        echo "a + b = $(($a+$b))"
        echo "a - b = $(($a-$b))"
        echo "a * b = $(($a*$b))"
        echo "a / b = $(($a/$b))"
        echo "a ** b = $(($a**$b))"
        echo "a % b = $(($a%$b))"
    else
        echo "$b"|grep " " &>/dev/null
        if [ $? -eq 0 ];then
            echo "too many arguments"
            exit 2
        fi
        echo "b is incorect"
        exit 2
    fi
else
    expr 1 + $b &>/dev/null
    if [ $? -ne 0 ];then
        echo "$b"|grep " " &>/dev/null
        if [ $? -eq 0 ];then
            echo "too many arguments"
            exit 2
        fi
        echo "a,b is incorect"
        exit 2
    else
        echo "a is incorect"
        exit 2
    fi
fi
```

第6章

Shell 脚本的条件测试与比较

6.1 Shell 脚本的条件测试

6.1.1 条件测试方法综述

通常，在 bash 的各种条件结构和流程控制结构中都要进行各种测试，然后根据测试结果执行不同的操作，有时也会与 if 等条件语句相结合，来完成测试判断，以减少程序运行的错误。

执行条件测试表达式后通常会返回“真”或“假”，就像执行命令后的返回值为 0 表示真，非 0 表示假一样。

在 bash 编程里，条件测试常用的语法形式见表 6-1。

表 6-1 条件测试常用的语法

条件测试语法	说明
语法 1：test < 测试表达式 >	这是利用 test 命令进行条件测试表达式的方法。test 命令和“ < 测试表达式 >”之间至少有一个空格。
语法 2：[< 测试表达式 >]	这是通过 [] （单中括号）进行条件测试表达式的方法，和 test 命令的用法相同，这是老男孩推荐的方法。[] 的边界和内容之间至少有一个空格。
语法 3：[[< 测试表达式 >]]	这是通过 [[]] （双中括号）进行条件测试表达式的方法，是比 test 和 [] 更新的语法格式。[[]] 的边界和内容之间至少有一个空格。
语法 4：((< 测试表达式 >))	这是通过 (()) （双小括号）进行条件测试表达式的方法，一般用于 if 语句里。(()) （双小括号）两端不需要有空格。

针对表 6-1 有几个注意事项需要说明一下：

- 语法 1 中的 test 命令和语法 2 中的 [] 是等价的。语法 3 中的 [[]] 为扩展的 test 命令，语法 4 中的 (()) 常用于计算，**老男孩建议使用相对友好的语法 2，即中括号（[]）的语法格式。**
- 在 [[]]（双中括号）中可以使用通配符等进行模式匹配，这是其区别于其他几种语法格式的地方。
- &&、||、>、< 等操作符可以应用于 [[]] 中，但不能应用于 [] 中，在 [] 中一般用 -a、-o、-gt（用于整数）、-lt（用于整数）代替上述操作符。
- 对于整数的关系运算，也可以使用 Shell 的算术运算符 (())。

6.1.2　test 条件测试的简单语法及示例

test 条件测试的语法格式为：test < 测试表达式 >

对于如下语句：

```
test -f file && echo true || echo false
```

该语句表示如果 file 文件存在，则输出 true，否则（||）输出 false。这里的 && 是并且的意思。test 的 -f 参数用于测试文件是否为普通文件，test 命令若执行成功（为真），则执行 && 后面的命令，而 || 后面的命令是 test 命令执行失败之后（为假）所执行的命令。

test 命令测试表达式的逻辑也可以用上述表达形式的一半逻辑（即仅有一个 && 或 ||）来测试，示例如下。

```
test -f /tmp/oldboy.txt && echo 1 #<== 若表达式成功，则输出 1。
test -f /tmp/oldboy.txt || echo 0 #<== 若表达式不成功，则输出 0。
```

另外，逻辑操作符 && 和 || 的两端既可以有空格，也可以无空格，这主要看读者习惯。老男孩的习惯是，尽量减少输入空格，因此在逻辑操作符 && 和 || 的两端会尽量不输入空格，主要是考虑输入应简洁、快速，当然，带了空格看起来会更美观一些。在写本书时，老男孩也曾为究竟应使用哪种方式纠结了很久，最后终于决定使用带空格的语法。

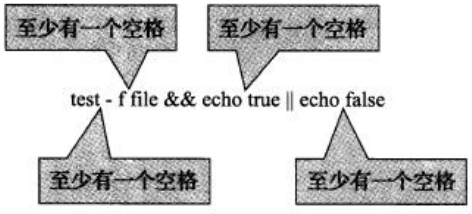


图 6-1　test 测试语法的格式说明

test 测试语法的格式说明如图 6-1 所示。

范例 6-1：在 test 命令中使用 -f 选项（文件存在且为普通文件则表达式成立）测试文件。

```
[root@oldboy ~]# test -f file && echo true || echo false
#<== 如果 file 文件存在并且是普通文件就为真，因为 file 文件不存在，所以输出了 false。
false
[root@oldboy ~]# touch file          #<== 现在创建不存在的普通文件 file。
```

```
[root@oldboy ~]# test -f file && echo true || echo false #<== 因为 file 文件存
在，所以输出了 true。
true
```

范例 6-2：在 test 命令中使用 -z 选项（如果测试字符串的长度为 0，则表达式成立）测试字符串。

```
[root@oldboy ~]# test -z "oldboy" && echo 1 || echo 0
#<== 如果测试字符串的长度为 0，则表达式成立，因为被测试的字符串为 oldboy，不为 0，表达式结
     果为假，因此返回 0。
0
[root@oldboy ~]# char="oldboy" #<== 将 oldboy 字符串赋值给变量 char。
[root@oldboy ~]# test -z "$char" && echo 1 || echo 0 #<== 对变量 char 进行测试，
                                                        注意要带 $。
0
[root@oldboy ~]# char=""        #<== 将空值赋值给变量 char。
[root@oldboy ~]# test -z "$char" && echo 1 || echo 0
#<== 如果测试字符串的长度为 0，则表达式成立，因为被测试的字符串为空，因此表达式的结果为真，
     返回 1。
1
```

提示：关于 test 测试表达式的更多知识可执行 man test 查看帮助，本书大部分场景都会使用 [] 的语法替代 test 命令的语法。

结论：test 命令测试的功能很强大，但是和 []、[[]] 的功能有所重合，因此，在实际工作中选择一种适合自己的语法就好了。对于其他的语法，能读懂别人写的脚本就可以了。

6.1.3 []（中括号）条件测试语法及示例

[] 条件测试的语法格式为：[< 测试表达式 >]

注意：中括号内部的两端要有空格，[] 和 test 等价，即 test 的所有判断选项都可以直接在 [] 里使用。

对于如下语句：

```
[ -f /tmp/oldboy.txt ] && echo 1 || echo 0
```

如果 /tmp/oldboy.txt 文件存在，则输出 1，否则（||）输出 0。这里的 && 表示并且。[] 的应用同 test 命令，若中括号里的命令执行成功（返回真），则执行 && 后面的命令，否则执行 || 后面的命令。

[] 测试表达式的逻辑也可以用如下的语法来判断逻辑的表达式写法（test 命令的用法也适合于此），即：

```
[ -f /tmp/oldboy.txt ] && echo 1 #<== 若表达式成功，则输出 1。
[ -f /tmp/oldboy.txt ] || echo 0 #<== 若表达式不成功，则输出 0。
```

另外，逻辑操作符 && 和 || 的两端可以有空格也可以无空格，本书同样使用带空格的语法形式。

[] 测试语法的格式说明如图 6-2 所示。

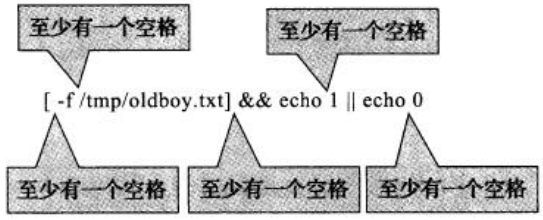


图 6-2　[] 测试语法的格式说明

范例 6-3：利用 [] 加 -f 选项（文件存在且为普通文件则表达式成立）测试文件。

```
[root@oldboy ~]# [ -f /tmp/oldboy.txt ] && echo 1 || echo 0
 #<== 如果 /tmp/oldboy.txt 文件存在并且是普通文件则为真，因为该文件不存在，所以输出了 0。
0
[root@oldboy ~]# touch /tmp/oldboy.txt #<== 创建文件。
[root@oldboy ~]# [ -f /tmp/oldboy.txt ] && echo 1 || echo 0
1 #<== 因为文件存在，所以输出 1。
[root@oldboy ~]# [ -f /tmp/oldboy.txt ] && echo 1
 #<== 可以只写前一半（即只有 && 部分），如果文件存在则输出 1。
1
[root@oldboy ~]# [ -f /tmp/oldboy.txt ] || echo 0
 #<== 可以只写后一半（即只有 || 部分），如果文件不存在则输出 0，因为文件存在，所以没有输出。
[root@oldboy ~]# [ -f /tmp/oldgirl.txt ] || echo 0
 #<== 如果文件不存在，则输出 0，因为 oldgirl.txt 不存在，所以输出了 0。
0
```

提示：[] 命令的选项和 test 命令的选项是通用的，因此，使用 [] 时的参数选项可以通过 man test 命令获得帮助。

6.1.4　[[]] 条件测试语法及示例

[[]] 条件测试的语法格式为：[[< 测试表达式 >]]。

注意：双中括号里的两端也要有空格。

对于如下语句：

```
[[ -f /tmp/oldboy.txt ]] && echo 1 || echo 0
```

如果 /tmp/oldboy.txt 文件存在就输出 1，否则（||）就输出 0。这里的 && 表示并且。[[]] 的应用属于 [] 和 test 命令的扩展命令，功能更丰富也更复杂。如果双中括号里的表达式成立（为真），则执行 && 后面的命令，否则执行 || 后面的命令。

[[]] 测试表达式的逻辑也可以使用如下的部分逻辑形式，即：

```
[[ -f /tmp/oldboy.txt ]] && echo 1    #<== 若表达式成功则输出 1。
[[ -f /tmp/oldboy.txt ]] || echo 0    #<== 若表达式不成功则输出 0。
```

另外，逻辑操作符 && 和 || 的两端可以有空格也可以无空格，本书使用的是带空格的语法形式。

双中括号内部的两端要有空格，[[]] 里的测试判断选项，也可以通过 man test 来获得，[[]] 表达式与 [] 和 test 用法的选项部分是相同的，其与 [] 和 test 测试表达式的区别在于，在 [[ ]] 中可以使用通配符等进行模式匹配；并且 &&、||、>、< 等操作符可以应用于 [[ ]] 中，但不能应用于 [ ] 中，在 [ ] 中一般使用 -a、-o、-gt（用于整数）、-lt（用于整数）等操作符代替上文提到的用于 [[]] 中的符号。除了使用通配符功能之外，建议放弃这个双中括号的写法，虽然它是较新的 test 命令的语法格式。

[[]] 测试语法的格式说明如图 6-3 所示。

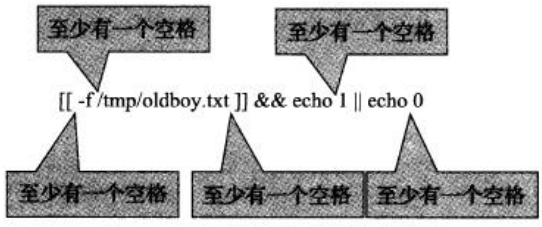


图 6-3　[[]] 测试语法的格式说明

范例 6-4：[[]] 的使用示例。

```
[root@oldboy ~]# [[ -f /tmp/oldgirl.txt ]] || echo 0
#<== 如果 /tmp/oldgirl.txt 文件存在并且是普通文件，则为真，因为文件不存在，所以输出了 0。
0
[root@oldboy ~]# touch /tmp/oldgirl.txt #<== 创建 oldgirl.txt，注意 /tmp 目录。
[root@oldboy ~]# [[ -f /tmp/oldgirl.txt ]] || echo 0 #<== 因为文件存在了，所以后
                                                        半部分没有输出。
[root@oldboy ~]# [[ -f /tmp/oldboy.txt ]] || echo 0 #<== 因为存在 /tmp/oldboy.txt，
                                                        所以后半部分没有输出。
[root@oldboy ~]# rm -f /tmp/oldboy.txt                  #<== 删除 oldboy.txt 文件。
[root@oldboy ~]# [[ -f /tmp/oldboy.txt ]] || echo 0 #<== 因为文件不存在了，所以
                                                        输出 0。
0
```

有关 test、[]、[[]] 这些操作符的用法，通过 help test 或 man test 查询即可得到帮助，完整的 []、[[]] 用法可通过 man bash 来获取。

6.2　文件测试表达式

6.2.1　文件测试表达式的用法

在讲解文件测试表达式之前，先举一个生活中的例子：如果你要找老男孩老师打台球，你一定不会先去台球厅，而是会先打电话联系，问他有没有时间一起打球。同理，如果在编程时需要处理一个对象，也应先对对象进行测试，只有在确定它符合要求时，才应进行操作处理，这样做的好处就是避免程序出错及无谓的系统资源消耗，这个需要测试的对象可以是文件、字符串、数字等。

在书写文件测试表达式时，通常可以使用表 6-2 中的文件测试操作符。

表 6-2　常用的文件测试操作符

常用文件测试操作符	说明
-d 文件，d 的全拼为 directory	文件存在且为目录则为真，即测试表达式成立
-f 文件，f 的全拼为 file	文件存在且为普通文件则为真，即测试表达式成立
-e 文件，e 的全拼为 exist	文件存在则为真，即测试表达式成立。注意区别于“-f”，-e 不辨别是目录还是文件
-r 文件，r 的全拼为 read	文件存在且可读则为真，即测试表达式成立
-s 文件，s 的全拼为 size	文件存在且文件大小不为 0 则为真，即测试表达式成立
-w 文件，w 的全拼为 write	文件存在且可写则为真，即测试表达式成立
-x 文件，x 的全拼为 executable	文件存在且可执行则为真，即测试表达式成立
-L 文件，L 的全拼为 link	文件存在且为链接文件则为真，即测试表达式成立
f1 -nt f2，nt 的全拼为 newer than	文件 f1 比文件 f2 新则为真，即测试表达式成立。根据文件的修改时间来计算
f1 -ot f2，ot 的全拼为 older than	文件 f1 比文件 f2 旧则为真，即测试表达式成立。根据文件的修改时间来计算

-eq 是等于

表 6-2 列出的是企业里比较常用的操作符，这些操作符号对于 [[]]、[]、test 的测试表达式几乎是通用的，更多的操作符请通过 man test 获得帮助。

6.2.2　文件测试表达式举例

1. 普通文件测试表达式示例

（1）普通文件（测试文件类型）

范例 6-5：普通文件条件表达式测试实践。

```
[root@oldboy ~]# touch oldboy  #<== 创建文件 oldboy。
[root@oldboy ~]# ls -l oldboy
-rw-r--r-- 1 root root 0 Aug  9 22:40 oldboy
[root@oldboy ~]# [ -f oldboy ] && echo 1 || echo 0 #<== 测试文件是否存在，如果存
                                                        在，则输出 1，否则输出 0
1  #<== 因为刚刚创建了 oldboy 文件，因此条件测试表达式成立，输出 1。
```

（2）目录文件（测试文件类型）

范例 6-6：目录文件条件表达式测试实践。

```
[root@oldboy ~]# mkdir oldgirl  #<== 创建目录 oldgirl。
[root@oldboy ~]# [ -f oldgirl ] && echo 1 || echo 0 #<== 测试 oldgirl 是否为普通文件。
0  #<== 输出为 0，证明 oldgirl 非普通文件，因为前面创建的是 oldgirl 目录，因此，输出 0 是对的。
[root@oldboy ~]# [ -e oldgirl ] && echo 1 || echo 0 #<== 测试 oldgirl 是否存在。
1  #<== 只要 oldgirl 存在就行，不管是目录还是普通文件，oldgirl 确实存在，因此输出 1。
[root@oldboy ~]# [ -d oldgirl ] && echo 1 || echo 0 #<== 测试 oldgirl 是否为目录。
1  #<== 输出 1，证明 oldgirl 是目录。
[root@oldboy ~]# [ -d oldboy ] && echo 1 || echo 0 #<== 测试 oldboy 是否为目录。
0  #<== 输出 0，证明 oldboy 非目录。
```

2. 测试文件属性示例

范例 6-7：文件属性条件表达式测试实践。

```
[root@oldboy ~]# ls -l oldboy
-rw-r--r-- 1 root root 0 Aug  9 22:40 oldboy
#<== 文件权限默认为 644，权限基础可参考老男孩基础类图书或视频讲解。
[root@oldboy ~]# [ -r oldboy ] && echo 1 || echo 0  #<== 测试 oldboy 是否可读。
1  #<== 输出为 1，因为用户权限位有 r，因此，可以读取 oldboy。
[root@oldboy ~]# [ -w oldboy ] && echo 1 || echo 0  #<== 测试 oldboy 是否可写。
1  #<== 输出为 1，因为用户权限位有 w，因此，可以写入 oldboy。
[root@oldboy ~]# [ -x oldboy ] && echo 1 || echo 0  #<== 测试 oldboy 是否可执行。
0  #<== 输出为 0，因为用户权限位没有 x，因此，不可以执行 oldboy。
[root@oldboy ~]# chmod 001 oldboy  #<== 修改 oldboy 的权限位 001。
[root@oldboy ~]# ls -l oldboy
---------x 1 root root 0 Aug  9 22:40 oldboy #<== 修改后的结果。
[root@oldboy ~]# [ -w oldboy ] && echo 1 || echo 0
1  #<== 用户权限位明明没有 w，为什么还是返回 1 呢?
[root@oldboy ~]# echo 'echo test' >oldboy  #<== 因为确实可以写啊，这是 root 用户
                                                比较特殊的地方。
[root@oldboy ~]# [ -r oldboy ] && echo 1 || echo 0
1  #<== 用户权限位明明没有 r，为什么还是返回 1 呢?
[root@oldboy ~]# cat oldboy
echo test  #<== 因为确实可以读啊，这是 root 用户比较特殊的地方。
[root@oldboy ~]# [ -x oldboy ] && echo 1 || echo 0
1
[root@oldboy ~]# ./oldboy   #<== 可执行。
test
```

提示：测试文件的读、写、执行等属性，不光是根据文件属性 rwx 的标识来判断，还要看当前执行测试的用户是否真的可以按照对应的权限操作该文件。

3. 测试 Shell 变量示例

首先定义 file1 和 file2 两个变量，并分别赋予这两个变量对应的系统文件路径及文件名的值，如下：

```
[root@oldboy ~]# file1=/etc/services;file2=/etc/rc.local  #<== 分号用于分隔两个命令。
[root@oldboy ~]# echo $file1 $file2
/etc/services /etc/rc.local
```

范例 6-8：对单个文件变量进行测试。

```
[root@oldboy ~]# [ -f "$file1" ] && echo 1 || echo 0    #<== 文件存在且为普通文件，
                                                            所以为真（1）。
1
[root@oldboy ~]# [ -d "$file1" ] && echo 1 || echo 0    #<== 是文件而不是目录，所以
                                                            为假（0）。
0
[root@oldboy ~]# [ -s "$file1" ] && echo 1 || echo 0    #<== 文件存在且大小不为 0，
                                                            所以为真（1）。
1
[root@oldboy ~]# [ -e "$file1" ] && echo 1 || echo 0    #<== 文件存在，所以为真（1）。
1
```

范例 6-9：对单个目录或文件进行测试。

```
[root@oldboy ~]# [ -e /etc ] && echo 1 || echo 0
1
[root@oldboy ~]# [ -w /etc/services ] && echo 1 || echo 0
1
[root@oldboy ~]# su - oldboy  #<== 切换到普通用户。
[oldboy@oldboy ~]$ [ -w /etc/services ] && echo 1 || echo 0 #<== 文件不可写，所
                                                               以返回 0。
0
```

范例 6-10：测试时变量的特殊写法及问题。

用 [] 测试变量时，如果被测试的变量不加双引号，那么测试结果可能会是不正确的，示例如下：

```
[root@oldboy ~]# echo $oldgirl    #<== 这是一个不存在的变量，如果读者已经定义，则可
                                      以执行 unset oldgirl 取消。
[root@oldboy ~]# [ -f $oldgirl ] && echo 1 || echo 0    #<== 不加引号测试变量。
1 #<== 明明 $oldgirl 变量不存在内容还返回 1，逻辑就不对了
[root@oldboy ~]# [ -f "$oldgirl" ] && echo 1 || echo 0 #<== 加引号测试变量。
0 #<== 加了双引号就返回 0，逻辑就对了。
```

如果是文件实体路径，那么加引号与不加引号的结果是一样的：

```
[root@oldboy ~]# [ -f "/etc/services" ] && echo 1 || echo 0 #<== 加引号测试文件。
1
[root@oldboy ~]# [ -f /etc/services ] && echo 1 || echo 0 #<== 不加引号测试文件。
1
```

范例 6-11：在生产环境下，系统 NFS 启动脚本的条件测试。

```
[root@oldboy ~]# more /etc/init.d/nfs
# Source networking configuration.
[ -f /etc/sysconfig/network ] && . /etc/sysconfig/network
#<== 如果 /etc/sysconfig/network 文件存在，则加载文件。
# Check for and source configuration file otherwise set defaults
[ -f /etc/sysconfig/nfs ] && . /etc/sysconfig/nfs
#<== 如果 /etc/sysconfig/nfs 文件存在，则加载文件。
```

特别提示：系统脚本是我们学习编程的第一标杆，新手要多参考脚本来学习，虽然有些脚本也不是特别规范。

范例 6-12：实现系统 bind 启动脚本 named（bind DNS 服务）。

```
[ -r /etc/sysconfig/network ] && . /etc/sysconfig/network
#<== 若文件存在且可读，则加载 /etc/sysconfig/network。
[ -x /usr/sbin/$named ] || exit 5 #<== 如果 /usr/sbin/$named 不可执行，则退出。
```

特别提示：前面所讲的都是 [-f /etc] && echo 1 || echo 0 的用法，bind 启动脚本 [-x /usr/sbin/$named] || exit 5 的用法更值得注意，这里只用了一部分判断，结果却更简洁。

范例 6-13：写出简单高效的测试文件。

在做测试判断时，不一定非要按照“前面的操作成功了如何，否则如何”的方法来进行。直接做部分判断，有时看起来更简洁。例如：

```
[root@oldboy ~]# [ -x oldboy ] && echo 1
#<== 如果 oldboy 可执行，则输出 1；如果不可执行，则不做任何输出。
1
[root@oldboy ~]# [ -f /etc ] || echo 0
#<== 如果 /etc 是文件这一点不成立，则输出 0；如果成立，则不做任何输出。
0
```

范例 6-14：实现系统脚本 /etc/init.d/nfs。

```
[root@oldboy ~]# sed -n '44,50p' /etc/init.d/nfs  #<== 查看 NFS 脚本的第 44 ~ 50 行。
```

```
# Check that networking is up.
[ "${NETWORKING}" != "yes" ] && exit 6 #<== 如果 ${NETWORKING} 的变量内
                                           容不等于 yes，则退出。
[ -x /usr/sbin/rpc.nfsd ] || exit 5
#<== 如果 /usr/sbin/rpc.nfsd 的脚本不可执行，则以返回值 5 退出脚本。如果可执
    行，则不做任何输出。
[ -x /usr/sbin/rpc.mountd ] || exit 5
[ -x /usr/sbin/exportfs ] || exit 5
```

6.2.3　特殊条件测试表达式案例

以下写法适用于所有的条件测试表达式，是工作中比较常用的替代 if 语句的方法。判断条件测试表达式的条件成立或不成立后，还需要继续执行多条命令语句的语法形式如下。

例如：当条件 1 成立时，同时执行命令 1、命令 2、命令 3。不用 if 测试表达式的格式如下：

```
[ 条件 1 ] &&{
    命令 1
    命令 2
    命令 3
}
[[ 条件 1 ]] &&{
    命令 1
    命令 2
    命令 3
}
test 条件 1 &&{
    命令 1
    命令 2
    命令 3
}
```

> **提示：本书的大部分普通实例都是以 [] 为例进行讲解的，读者可以自行练习 test 及 [[]] 的用法。**

上面的判断相当于下面 if 语句的效果。

```
if [ 条件 1 ]
  then
    命令 1
    命令 2
    命令 3
fi
```

范例 6-15：当条件不成立时，执行大括号里的多条命令，这里要使用逻辑操作符“||”。

```
[root@oldboy scripts]# cat 6_15.sh
[ -f /etc ] || {          #<== 如果 /etc/ 是普通文件不成立，则执行大括号里的命令集合，这显
                              然不成立啊！
    echo 1
    echo 2
    echo 3
}
[root@oldboy scripts]# sh 6_15.sh   #<==/etc/ 是普通文件显然是不成立的，因此会执行
                                        大括号里的命令集！
1
2
3
```

如果把上述脚本写在一行里面，那么里面的每个命令都需要用分号结尾，示例如下所示：

```
[root@oldboy ~]# [ -f /etc/services ] && { echo "I am oldboy"; echo "I am linuxer"; }
I am oldboy
I am linuxer
```

提示：本例的两种用法都很简洁，但是不如 if 条件语句容易理解，因此，请读者根据自身情况选择使用，更多帮助请通过 man test 查询。

6.3 字符串测试表达式

6.3.1 字符串测试操作符

字符串测试操作符的作用包括：比较两个字符串是否相同、测试字符串的长度是否为零、字符串是否为 NULL [⊖]等。

在书写测试表达式时，可以使用表 6-3 中的字符串测试操作符。

表 6-3 字符串测试操作符

常用字符串测试操作符	说明
-n " 字符串 "	若字符串的长度不为 0，则为真，即测试表达式成立，n 可以理解为 no zero
-z " 字符串 "	若字符串的长度为 0，则为真，即测试表达式成立，z 可以理解为 zero 的缩写
" 串 1 " = " 串 2 "	若字符串 1 等于字符串 2，则为真，即测试表达式成立，可使用 “==” 代替 “=”
" 串 1" != " 串 2 "	若字符串 1 不等于字符串 2，则为真，即测试表达式成立，但不能用 “!==” 代替 “!=”

以下是针对字符串测试操作符的提示：

- 对于字符串的测试，一定要将字符串加双引号之后再进行比较。如 [-n "$myvar"]，特别是使用 [] 的场景。

⊖ 通过 bash 区分零长度字符串和空字符串。

❑ 比较符号（例如 = 和 !=）的两端一定要有空格。

❑ “!=” 和 “=” 可用于比较两个字符串是否相同。

范例 6-16：字符串条件表达式测试实践。

```
[root@oldboy ~]# [ -n "abc" ] && echo 1 || echo 0  #<== 如果字符串长度不为 0，
                                                        则输出 1，否则输出 0。
1 #<== 因为字符串为 abc，长度不为 0，因此为真，输出 1。
[root@oldboy ~]# test -n "abc" && echo 1 || echo 0 #<==test 的用法同上述 [] 的用法。
1
[root@oldboy ~]# test -n "" && echo 1 || echo 0
0
[root@oldboy ~]# var="oldboy" #<== 给变量 var 赋值 oldboy 字符串。
[root@oldboy ~]# [ -n "$var" ] && echo 1 || echo 0    #<== 如果字符串长度不为 0，
                                                        则输出 1，否则输出 0。
1 #<== 因为变量 var 内容字符串为 oldboy，长度不为 0，因此为真，输出 1。
[root@oldboy ~]# [ -n $var ] && echo 1 || echo 0 #<== 去掉双引号在这里看起来也
                                                        是对的，不过还是加上为好。
1
[root@oldboy ~]# var="oldgirl"
[root@oldboy ~]# [ -z "$var" ] && echo 1 || echo 0  #<== 使用 -z，变量长度为 0，
                                                        则为真。
0    #<== 变量 var 的值为 oldgirl，长度不为 0，所以表达式不成立，输出 0。
[root@oldboy ~]# [ "abc" = "abc" ] && echo 1 || echo 0
#<== 字符串相等，输出 1，注意 “=” 两端要有空格。
1
[root@oldboy ~]# [ "abc" = "abd" ] && echo 1 || echo 0
#<== 字符串不相等，输出 0，注意 “=” 两端要有空格。
0
[root@oldboy ~]# [ "$var" = "oldgirl" ] && echo 1 || echo 0
#<== 变量值和字符串相等，输出 1。
1
[root@oldboy ~]# [ "$var" == "oldgirl" ] && echo 1 || echo 0
#<== 使用 “==” 代替 “=”，注意 “=” 两端要有空格。
1
[root@oldboy ~]# [  "$var" != "oldgirl" ] && echo 1 || echo 0
0
```

范例 6-17：进行字符串比较时，等号两端没有空格带来的问题。

```
[root@oldboy ~]# [ "abc"="1" ] && echo 1||echo 0  #<== 若等号两端不带空格，则会
                                                        出现明显的逻辑错误。
1  #<== 明明表达式不成立，却输出了 1。
[root@oldboy ~]# [ "abc" = "1" ] && echo 1||echo 0 #<== 带空格的就是准确的。
0  #<== 表达式不成立，输出 0。
```

结论：字符串比较时若等号两端没有空格，则会导致判断出现逻辑错误，即使语法没问题，但是结果依然可能不对。

范例 6-18：字符串不加引号可能带来的问题。

```
[root@oldboy ~]# var="" #<== 将变量内容置为空。
[root@oldboy ~]# [ -n "$var" ] && echo 1 || echo 0   #<== 有双引号。
0 #<== 给变量加双引号，返回 0，-n 不为空时为真，因为变量内容为空，因此输出 0 是对的。
[root@oldboy ~]# [ -n $var ] && echo 1 || echo 0     #<== 去掉双引号。
1 #<== 同样的表达式，不加引号和加双引号后测试的结果相反，可见加双引号的重要性。
[root@oldboy ~]# [ -z "$var" ] && echo 1 || echo 0 #<== 如果字符串长度为 0，则
                                                        输出 1，否则输出 0。
1
```

结论：字符串不加双引号，可能会导致判断上出现逻辑错误，即使语法没问题，但是结果依然可能不对。

6.3.2 字符串测试生产案例

范例 6-19：有关双引号和等号两端空格的生产系统标准。

```
[root@oldboy ~]# sed -n '30,31p' /etc/init.d/network  #<== 系统网卡启动脚本案例。
# Check that networking is up.
[ "${NETWORKING}" = "no" ] && exit 6    #<== 字符串变量和字符串都加了双引号，比较符
                                            号“=”两端也都有空格。
```

来看一个类似网友遇到的错误示例[⊖]：

```
[root@oldboy ~]# sed -n '18,22p' /etc/init.d/rpcbind #<==rpcbind 启动脚本。
if [ "$LANG" = "ja" -o "$LANG" = "ja_JP.eucJP" ]; then
#<== 字符串变量和字符串都加了双引号，比较符号“=”两端也都有空格。
    if [ "$TERM" = "linux" ] ; then
        LANG=C
    fi
fi
```

范例 6-20：系统脚本 /etc/init.d/nfs 字符串测试的应用示例。

```
[root@oldboy ~]# sed -n '65,80p' /etc/init.d/nfs
        [ -z "$MOUNTD_NFS_V2" ] && MOUNTD_NFS_V2=default
#<==-z 的应用，如果变量 MOUNTD_NFS_V2 的长度为 0 则赋值 default。
        [ -z "$MOUNTD_NFS_V3" ] && MOUNTD_NFS_V3=default
#<==-z 的应用，如果变量 MOUNTD_NFS_V2 的长度为 0 则赋值 default。
       # Number of servers to be started by default
        [ -z "$RPCNFSDCOUNT" ] && RPCNFSDCOUNT=8 #<==-z 的应用。

        # Start daemons.
        [ -x /usr/sbin/rpc.svcgssd ] && /sbin/service rpcsvcgssd start

        # Set the ports lockd should listen on
        if [ -n "$LOCKD_TCPPORT" -o -n "$LOCKD_UDPPORT" ]; then  #<==-n 的应用。
                [ -x /sbin/modprobe ] && /sbin/modprobe lockd $LOCKDARG
                [ -n "$LOCKD_TCPPORT" ] && \
```

⊖ 参见 http://oldboy.blog.51cto.com/2561410/1433688。

```
                    /sbin/sysctl -w fs.nfs.nlm_tcpport=$LOCKD_TCPPORT >/dev/
null 2>&1
                [ -n "$LOCKD_UDPPORT" ] && \
                    /sbin/sysctl -w fs.nfs.nlm_udpport=$LOCKD_UDPPORT >/
dev/null 2>&1
```

6.4　整数二元比较操作符

6.4.1　整数二元比较操作符介绍

在书写测试表达式时，可以使用表 6-4 中的整数二元比较操作符。

表 6-4　整数二元比较操作符使用参考

在 [] 以及 test 中使用的比较符号	在 (()) 和 [[]] 中使用的比较符号	说明
-eq	== 或 =	相等，全拼为 equal
-ne	!=	不相等，全拼为 not equal
-gt	>	大于，全拼为 greater than
-ge	>=	大于等于，全拼为 greater equal
-lt	<	小于，全拼为 less than
-le	<=	小于等于，全拼为 less equal

以下是针对上述符号的特别说明：

- “=”和“!=”也可在 [] 中做比较使用，但在 [] 中使用包含“>”和“<”的符号时，需要用反斜线转义，有时不转义虽然语法不会报错，但是结果可能会不对。
- 也可以在 [[]] 中使用包含“-gt”和“-lt”的符号，但是不建议这样使用。
- 比较符号两端也要有空格。

范例 6-21：二元数字在 [] 中使用“<”、“>”非标准符号的比较。

```
[root@oldboy ~]# [ 2 > 1 ] && echo 1 || echo 0
1
[root@oldboy ~]# [ 2 < 1 ] && echo 1 || echo 0
1  #<== 这里的结果逻辑不对，条件不成立，则应该返回 0，可见，“<”操作符在 [] 里使用时会带来问题。
[root@oldboy ~]# [ 2 \< 1 ] && echo 1 || echo 0
0 #<== 转义后这里是正确的。
[root@oldboy ~]# [ 2 = 1 ] && echo 1 || echo 0    #<== 比较相等符号是正确的。
0
[root@oldboy ~]# [ 2 = 2 ] && echo 1 || echo 0    #<== 比较相等符号是正确的。
1
[root@oldboy ~]# [ 2 != 2 ] && echo 1 || echo 0   #<== 比较不相等符号也是正确的。
0
```

对于比较符号的应用，建议读者尽可能地按照表 6-4 中标记的方法来使用，以避免出现逻辑错误。

范例 6-22：二元数字在 [] 中使用 -gt、-le 类符号的比较。

```
[root@oldboy ~]# [ 2 -gt 1 ] && echo 1 || echo 0
1  #<==2 大于 1 成立，输出 1。
[root@oldboy ~]# [ 2 -ge 1 ] && echo 1 || echo 0
1  #<==2 大于等于 1 成立，输出 1。
[root@oldboy ~]# [ 2 -le 1 ] && echo 1 || echo 0
0  #<==2 小于等于 1 不成立，输出 0。
[root@oldboy ~]# [ 2 -lt 1 ] && echo 1 || echo 0
0  #<==2 小于 1 不成立，输出 0。
```

范例 6-23：二元数字配合不同种类的操作符在 [[]] 中的比较。

```
[root@oldboy ~]# [[ 5 > 6 ]] && echo 1 || echo 0
0  #<==5 大于 6 不成立，输出 0。
[root@oldboy ~]# [[ 5 < 6 ]] && echo 1 || echo 0
1  #<==5 小于 6 成立，输出 1。
[root@oldboy ~]# [[ 5 != 6 ]] && echo 1 || echo 0
1  #<==5 不等于 6 成立，输出 1。
[root@oldboy ~]# [[ 5 = 6 ]] && echo 1 || echo 0
0  #<==5 等于 6 不成立，输出 0。
[root@oldboy ~]# [[ 5 -gt 6 ]] && echo 1 || echo 0
0  #<==5 大于 6 不成立，输出 0。
[root@oldboy ~]# [[ 5 -lt 6 ]] && echo 1 || echo 0
1  #<==5 小于 6 成立，输出 1。
[root@oldboy ~]# [[ 65 > 66 ]] && echo 1 || echo 0
0  #<==65 大于 66 不成立，输出 0。
[root@oldboy ~]# [[ 65 < 66 ]] && echo 1 || echo 0
1  #<==65 小于 66 成立，输出 1。
[root@oldboy ~]# [[ 65 = 66 ]] && echo 1 || echo 0
0  #<==65 等于 66 不成立，输出 0。
```

提示：[[]] 是扩展的 test 命令，其语法更丰富也更复杂。对于实际工作中的常规比较，不建议使用 [[]]，会给 Shell 学习带来很多麻烦，除非是特殊的正则匹配等，在 [] 无法使用的场景下才会考虑使用 [[]]。

范例 6-24：二元数字在 (()) 中的比较。

```
[root@oldboy ~]# ((3>2)) && echo 1 || echo 0
1  #<==3 大于 2 成立，输出 1。
[root@oldboy ~]# ((3<2)) && echo 1 || echo 0
0  #<==3 小于 2 不成立，输出 0。
[root@oldboy ~]# ((3==2)) && echo 1 || echo 0
0  #<==3 等于 2 不成立，输出 0。
[root@oldboy ~]# ((3!==2)) && echo 1 || echo 0 #<== “!==” 符号不可用，语法错误
-bash: ((: 3!==2: syntax error: operand expected (error token is "=2")
0
```

```
[root@oldboy ~]# ((3!=2))&& echo 1 || echo 0
1
```

有关 []、[[]]、(()) 用法的小结：

- 整数加双引号的比较是对的。
- [[]] 中用类似 -eq 等的写法是对的，[[]] 中用类似 >、< 的写法也可能不对，有可能会只比较第一位，逻辑结果不对。
- [] 中用类似 >、< 的写法在语法上虽然可能没错，但逻辑结果不对，可以使用 =、!= 正确比较。
- (()) 中不能使用类似 -eq 等的写法，可以使用类似 >、< 的写法。

提示：对于工作场景中的整数比较，推荐使用 [] (类似 -eq 的用法)，这是本书作者的习惯，当然使用 (()) 的写法也是可以的。

6.4.2　整数变量测试实践示例

范例 6-25：通过 [] 实现整数条件测试。

```
[root@oldboy ~]# a1=98;a2=99
[root@oldboy ~]# [ $a1 -eq $a2 ] && echo 1 || echo 0 #<== 测试 $a1 是否等于 $a2。
0
[root@oldboy ~]# [ $a1 -gt $a2 ] && echo 1 || echo 0 #<== 测试 $a1 是否大于 $a2。
0
[root@oldboy ~]# [ $a1 -lt $a2 ] && echo 1 || echo 0 #<== 测试 $a1 是否小于 $a2。
1
```


提示：有关整数（要确认是整数，否则会报错）大小的比较，推荐使用本例中的方法。

范例 6-26：利用 [[]] 和 (()) 实现直接通过常规数学运算符进行比较。

```
[root@oldboy ~]# [[ $a1 > $a2 ]] && echo 1 || echo 0   #<== 测试 $a1 是否大于 $a2，
                                                          尽量不用此写法。
0
[root@oldboy ~]# [[ $a1 < $a2 ]] && echo 1 || echo 0   #<== 测试 $a1 是否小于 $a2，
                                                          尽量不用此写法。
1
[root@oldboy ~]# (($a1>=$a2)) && echo 1 || echo 0      #<== 测试 $a1 是否大于等于 $a2，
                                                          此写法也可以。
0
[root@oldboy ~]# (($a1<=$a2)) && echo 1 || echo 0      #<== 测试 $a1 是否小于等于 $a2，
                                                          此写法也可以。
1
```

有关整数（要确认是整数，否则会报错）的大小比较，(()) 语法要优于 [[]]，但还是推荐优先使用 []，次选是 (())，不推荐使用 [[]]。示例如下：

```
[ $num1 -eq $num2 ]      #<== 注意比较符号两边的空格和比较符号的写法，必须要有空格。
(($num1>$num2))          #<== 比较符号两边无需空格（多空格也可），使用常规数学的比较符号即可。
```

范例 6-27：系统脚本中使用整数比较的案例。

```
[root@oldboy ~]# grep -w "\-eq" /etc/init.d/nfs #<== 过滤出相等（-eq）的例子。
        [ $RETVAL -eq 0 ] && RETVAL=$rval        #<== 使用 []，且两边都要有一个空格。
                [ $RETVAL -eq 0 ] && RETVAL=$rval #<== 使用 "-eq" 的比较操作符
                                                      的写法。
                [ $RETVAL -eq 0 ] && RETVAL=$rval
        [ $RETVAL -eq 0 ] && RETVAL=$rval
                [ $RETVAL -eq 0 ] && RETVAL=$rval
[root@oldboy ~]# grep -w "\-gt" /etc/init.d/nfs    #<== 过滤出大于（-gt）的例子。
        if [ $cnt -gt 0 ]; then
```

上述例子就是最好的学习规范，可效仿。

6.5 逻辑操作符

6.5.1 逻辑操作符介绍

在书写测试表达式时，可以使用表 6-5 中的逻辑操作符实现复杂的条件测试。

表 6-5 逻辑操作符

在 [ ] 和 test 中使用的操作符	在 [[ ]] 和 (()) 中使用的操作符	说明
-a	&&	and，与，两端都为真，则结果为真
-o	||	or，或，两端有一个为真，则结果为真
!	!	not，非，两端相反，则结果为真

对于上述操作符，有如下提示：

- 逻辑操作符前后的表达式是否成立，一般用真假来表示。
- “!”的中文意思是反，即与一个逻辑值相反的逻辑值。
- -a 的中文意思是“与”（and 或 &&），前后两个逻辑值都为“真”，综合返回值才为“真”，反之为“假”。
- -o 的中文意思是“或”（or 或 ||），前后两个逻辑值只要有一个为“真”，返回值就为“真”。
- 连接两含 [ ]、test 或 [[ ]] 的表达式可用 && 或 ||。

逻辑操作符运算规则

-a 和 && 的运算规则：只有逻辑操作符两端的表达式都成立时才为真；真（true）表示成立，对应的数字为 1；假（false）表示不成立，对应的数字为 0，这一点相当于如

下表达式：

```
[root@oldboy ~]# [ -f /etc/hosts -a -f /etc/services ] && echo 1 || echo 0
#<== 单中括号文件测试。
1
[root@oldboy ~]# [[ -f /etc/hosts && -f /etc/services ]] && echo 1 || echo 0
#<== 双中括号文件测试。
1
```

使用 -a 和 && 的综合表达式结果，相当于将两端表达式结果的对应数字（0 或 1）相乘。

当左边为真，右边为假的时候，相乘结果为 1*0=0，总结果为假（0）。

当左边为假，右边为真的时候，相乘结果为 0*1=0，总结果依然为假（0）。

当左边为真，右边也为真的时候，相乘结果为 1*1=1，总结果为真（1）。

当左边为假，右边也为假的时候，相乘结果为 0*0=0，总结果为假（0）。

简单表示为：

```
and 结果 1*0=0 假
and 结果 0*1=0 假
and 结果 1*1=1 真
and 结果 0*0=0 假
```

结论：and(&&) 也称为与，只有两端都是 1 时才为真，相当于取前后表达式的交集。

-o 或 || 两端都是 0 才为假，任何一端不为 0 就是真，这相当于将两边表达式结果的对应数字（0 或 1）相加，对应的表达式为：

```
[root@oldboy ~]# [ 5 -eq 6 -o 5 -gt 3 ] && echo 1 || echo 0
1
[root@oldboy ~]# ((5==6||5>3)) && echo 1 || echo 0
1
```

-o 或 || 的运算规则为：

当左边为真，右边为假的时候，相加结果为 1+0=1，总结果为真（1）。

当左边为假，右边为真的时候，相加结果为 0+1=1，总结果依然为真（1）。

当左边为真，右边也为真的时候，相加结果为 1+1=2，总结果为真（1），非 0 即真。

当左边为假，右边也为假的时候，相加结果为 0+0=0，总结果为假（0）。

简单表示为：

```
or 结果 1+0=1 真
or 结果 1+1=2 真（非 0 即为真）
or 结果 0+1=1 真
or 结果 0+0=0 假
```

结论：or（||）也称为或，它的两端表达式的结果都是 0 时才为假，不为 0 就是真。相当于对前后表达式结果取并集。

6.5.2 逻辑操作符实践示例

范例 6-28：[] 里的逻辑操作符配合文件测试表达式使用的示例。

```
[root@oldboy ~]# f1=/etc/rc.local;f2=/etc/services
#<== 定义 f1 和 f2 两个变量，分别赋值两个已知存在的文件路径。
[root@oldboy ~]# echo -ne "$f1 $f2\n"  #<== 测试输出 f1 和 f2 两个变量的值。
/etc/rc.local /etc/services
[root@oldboy ~]# [ -f "$f1" -a -f "$f2" ] && echo 1 || echo 0 #<==[] 里使用 -a(and)。
#<== 测试 f1 和 f2 两个变量值是否都为普通文件，如果成立，则输出 1，否则输出 0。
1
[root@oldboy ~]# [ -f "$f1" -o -f "$f222" ] && echo 1 || echo 0
#<== 测试 f1 和 f222 两个变量值是否都为普通文件，如果成立，则输出 1，否则输出 0；f222 变量是未定义的，但是 f1 变量是有的，并且是普通文件，-o 两边有一个表达式成立（即为真），因此输出 1。
1
[root@oldboy ~]# [ -f "$f111" -o -f "$f222" ] && echo 1 || echo 0
#<== 测试 f111 和 f222 两个变量值中是否有一个为普通文件，如果成立，则输出 1，否则输出 0。
0 #<==f111 和 f222 两个变量都是未定义的，因此输出 0。
[root@oldboy ~]# [ -f "$f1" && -f "$f2" ] && echo 1 || echo 0 #<== 这是错误语法，[] 中不能用 && 或 ||。
-bash: [: missing `]'
0
[root@oldboy ~]# [ -f "$f1" ] && [ -f "$f2" ] && echo 1 || echo 0
#<== 如果在 [] 中想使用 &&，则这样用。
1
```

范例 6-29：[[]] 里逻辑操作符配合字符串的条件表达式的测试示例。

```
[root@oldboy ~]# a="oldboy";b="oldgirl"  #<== 定义 a 和 b 两个变量，赋值两个已知的字符串。
[root@oldboy ~]# echo -ne "$a $b\n"  #<== 测试输出 a 和 b 两个变量的值。
oldboy oldgirl
[root@oldboy ~]# [[ ! -n "$a" && "$a" = "$b" ]] && echo 1 || echo 0
#<== [[]] 内部用 && 或 ||。
#<== 测试 $a 长度不为 0 是否成立，并且 $a 等于 $b 是否成立，两个表达式是否同时成立。
0 #<== 因为 $a 长度不为 0 成立，结果为 1，加上前面“!”取反，就是不成立，结果变为 0。$a 等于 $b 成立，结果为 1，两个表达式综合起来 0*1=0，因此不成立，输出 0。
[root@oldboy ~]# [[ -z "$a" || "$a" != "$b" ]] && echo 1 || echo 0
#<== [[]] 中 || 的使用。
1
[root@oldboy ~]# [[ -z "$a" -o "$a" != "$b" ]] && echo 1 || echo 0
#<== [[]] 内部用 -a 或 -o 会报错。
-bash: syntax error in conditional expression
-bash: syntax error near `-o'
```

范例 6-30：(()) 里逻辑操作符配合整数的条件表达式测试示例。

```
[root@oldboy ~]# m=21;n=38
[root@oldboy ~]# ((m>20&&n>30)) && echo 1 || echo 0 #<== (()) 内部用 && 或 ||。
1
```

```
[root@oldboy ~]# ((m<20||n>30)) && echo 1 || echo 0
1
[root@oldboy ~]# ((m<20||n<30)) && echo 1 || echo 0
0
[root@oldboy ~]# ((m<20 -a n<30)) && echo 1 || echo 0
#<== (()) 内部用 -a 或 -o 也会报语法错误。
-bash: ((: m<20 -a n<30: syntax error in expression (error token is
"n<30")
0
```

范例 6-31：使用多个 [] 号，并通过与或非进行混合测试。

```
[root@oldboy ~]# m=21;n=38
[root@oldboy ~]# [ $m -gt 20 -a $n -lt 30 ] && echo 1 || echo 0
0
[root@oldboy ~]# [ $m -gt 20 ]||[ $n -lt 30 ] && echo 1 || echo 0
#<== 多个 [] 号之间用 && 或 || 连接。
1
```

回顾一下前面已经讲解过的内容。

- “-a”和“-o”逻辑操作符号需要用于 [] 中。
- “&&”和“||”逻辑操作符号可用于 [[]] 或 (()) 中，也可以在外部连接多个 []。
- 注意，在 [] 和 [[]] 的两端及比较符号的两端，必须要有空格，但是对于 (()) 不需要。

范例 6-32：NFS 系统启动脚本中有关 [] 与或非判断的使用案例。

```
[root@oldboy ~]# egrep -wn "\-a|\-o" /etc/init.d/nfs
75:     if [ -n "$LOCKD_TCPPORT" -o -n "$LOCKD_UDPPORT" ]; then  #<==[]-o 的用法
87:     [ "$NFSD_MODULE" != "noload" -a -x /sbin/modprobe ] && { #<==[]-a 的用法
102:    if [ -n "$RQUOTAD" -a "$RQUOTAD" != "no" ]; then  #<==[]-a 的用法
170:    if [ -n "$RQUOTAD" -a "$RQUOTAD" != "no" ]; then  #<==[]-a 的用法
209:    if [ -n "$RQUOTAD" -a "$RQUOTAD" != "no" ]; then  #<==[]-a 的用法
229:    if [ $MOUNTD = 1 -o $NFSD = 1 ] ; then
```

提示：可见 [] 中使用 -a 或 -o 更常见，[[]] 中使用 && 或 || 不常见，使用 && 或 || 连接两个 [] 的多表达式判断也不常见。

范例 6-33：系统启动脚本中有关 [[]] 的用法和与或非判断的使用案例。

在操作系统中，[[]] 的用法不是很多，并且大多数情况都用于与通配符匹配的场景。

这里不得不通过大海捞针的方法（遍历 /etc/init.d/ 下的所有脚本）来帮助大家查找 [[]] 的用法：

```
[root@oldboy ~]# for n in `ls /etc/init.d/*`;do egrep -wn "\[\[ " $n&&echo
$n;done
119:  if [[ "$dst" == /dev/mapper* ]] \  #<==[[]] 通配符匹配的场景，此方法不支持 []。
```

```
/etc/init.d/halt
68:    if [[ $? = 0 ]]; then #<==[[]] 的常规场景，此方法支持 []。
/etc/init.d/httpd
561:   if [[ -n "$_target" ]]; then #<==[[]] 的常规场景，此方法支持 []。
576:   if [[ "$_rmnt" == "$_mnt" ]] || ! is_dump_target_configured; then
/etc/init.d/kdump
50:    if [[ $route == *" via "* ]] ; then #<==[[]] 通配符匹配的场景，此方法不支持 []。
71:    if ! [[ "$SYSLOGADDR" =~ $MATCH ]]; then #<==[[]] 通配符匹配的场景，此方法不支持 []。
/etc/init.d/netconsole
162:   if [[ "$rootfs" == nfs* || "$rootopts" =~ _r?netdev ]] ; then
#<==[[]] 通配符匹配的场景，此方法不支持 []。
/etc/init.d/network
```

由上述遍历可见，[[]] 的普通应用场景不多，但在 [[]] 通配符匹配的场景下，其他测试表达式无法替代，因此，如果需要通配符或正则匹配就用 [[]]。

其实，前面的内容已经提到了逻辑操作符的使用方法。6.5.1 节的表 6-5 中有相应的列表说明，忘记了的读者可以回头看看。

6.5.3 逻辑操作符企业案例

范例 6-34：输入或通过命令行传入一个字符或数字，如果传入的数字等于 1，就打印 1；如果等于 2，就打印 2；如果不等于 1 也不能于 2，就提示输入不对，然后退出程序。

参考答案 1：使用 read 读入内容方案。

```
[root@oldboy scripts]# cat 6_34_1.sh
#!/bin/sh
echo -n "pls input a char:" #<== 打印提示字符串，-n 表示不换行。
read var #<== 读取用户的输入并赋值给 var 变量。
[ "$var" == "1" ] &&{ #<== 条件表达式判断，看变量是否等于 1，注意普通字符比较多地用字
符串比较的语法，即加双引号比较，而不是使用整数比较的语法，整数比较容易出错，除非确定是整数。
    echo 1
    exit 0 #<== 每个逻辑正确，则以返回值 0 退出脚本，从而避免执行脚本后面无用的代码。
}
[ "$var" == "2" ] &&{ #<== 条件表达式判断，看变量是否等于 2。
    echo 2
    exit 0
}
[ "$var" != "2" -a "$var" != "1" ] &&{  #<== 条件表达式加逻辑操作符判断，看变量是
否不等于 2，并且不等于 1，如果都成立，则执行命令。
    echo error
    exit 0
}
```

执行结果如下：

```
[root@oldboy scripts]# sh 6_34_2.sh
pls input a char:1
```

```
1
[root@oldboy scripts]# sh 6_34_1.sh
pls input a char:2
2
[root@oldboy scripts]# sh 6_34_1.sh
pls input a char:4
error
```

参考答案 2：使用脚本命令行传参读入内容的解决方案。

```
[root@oldboy scripts]# cat 6_34_2.sh
#!/bin/sh
var=$1 #<== 将上文的 read 读入信息，改为脚本传参，接收脚本的第一个参数 $1 赋值给 var。
[ "$var" == "1" ] &&{
    echo 1
    exit 0
}
[ "$var" == "2" ] &&{
    echo 2
    exit 0
}
[ "$var" != "2" -a "$var" != "1" ] &&{
    echo error
    exit 0
}
```

提示：除了传参知识以外，其他代码和参考答案 1 一致，因此就不进行注释了。

执行结果如下：

```
[root@oldboy scripts]# sh 6_34_2.sh 1
1
[root@oldboy scripts]# sh 6_34_2.sh 2
2
[root@oldboy scripts]# sh 6_34_2.sh 4
error
```

自己写的
#!/bin/sh
a=$1
b=$2

范例 6-35：开发 Shell 脚本，分别实现以脚本传参和 read 读入的方式比较两个整数的大小。用条件表达式（禁止用 if）进行判断并以屏幕输出的方式提醒用户比较的结果。注意：一共是开发两个脚本。在用脚本传参和 read 读入的方式实现时，需要对变量是否为数字及传参个数是否正确给予提示。

参考答案 1：采用 read 方法。

```
[root@oldboy scripts]# cat 6_35_1.sh
#!/bin/sh
read -p "Pls input two num:" a b  #<== 请求用户输入两个参数，读入后分别赋值给变量 a 和 b。
```

```
#no1
[ -z "$a" ]||[ -z "$b" ] &&{   #<== 如果 $a 变量长度为 0 或 $b 变量长度为 0，即任何一个
                                    变量为空，则执行命令。
    echo "Pls input two num again." #<==a 或 b 没值，表示用户输入错误，给出提示。
    exit 1 #<== 以返回值 1 退出脚本。
}
#no2
expr $a + 10 &>/dev/null #<== 判断 $a 是否为整数，不输出任何信息。前文已讲解过了 expr
                              判断整数的方式。
RETVAL1=$? #<== 将返回值赋值给 RETVAL1，后面会用这个返回值做判断。
expr $b + 10 &>/dev/null #<== 判断 $b 是否为整数，不输出任何信息。
RETVAL2=$? #<== 将返回值赋值给 RETVAL2，后面会用这个返回值做判断。
test $RETVAL1 -eq 0 -a $RETVAL2 -eq 0||{ #<== 利用 test 进行返回值是否为 0 的判断。
如果有一个返回值不为 0，则说明有一个变量不为整数，不合要求，打印提示后退出。
    echo "Pls input two "num" again."
    exit 2
}
#no3
[ $a -lt $b ] &&{   #<== 整数比较，$a 是否小于 $b。
    echo "$a < $b"
    exit 0
}
#no4
[ $a -eq $b ] &&{   #<== 整数比较，$a 是否等于 $b。
    echo "$a = $b"
    exit 0
}
#no5
[ $a -gt $b ] &&{   #<== 整数比较，$a 是否大于 $b。
    echo "$a > $b"
}
```

执行结果如下：

```
[root@oldboy scripts]# sh 6_35_1.sh
Pls input two num:6 2     #<== 常规正确输入 6 和 2。
6 > 2
[root@oldboy scripts]# sh 6_35_1.sh
Pls input two num:6 6     #<== 常规正确输入 6 和 6。
6 = 6
[root@oldboy scripts]# sh 6_35_1.sh
Pls input two num:2 6     #<== 常规正确输入 2 和 6。
2 < 6
[root@oldboy scripts]# sh 6_35_1.sh
Pls input two num:        #<== 非常规错误输入，直接回车不输入。
Pls input two num again.  #<== 给出提示，退出脚本。
[root@oldboy scripts]# sh 6_35_1.sh
Pls input two num:dd ff   #<== 非常规错误输入，输入非数字。
Pls input two num again.  #<== 给出提示，退出脚本。
```

参考答案 2：通过命令行传参的方法实现。

```
[root@oldboy scripts]# cat 6_35_2.sh
#!/bin/sh
a=$1 #<== 将上文的 read 读入信息改为脚本传参，接收脚本的第一个参数 $1 赋值给 a。
b=$2 #<== 将上文的 read 读入信息改为脚本传参，接收脚本的第二个参数 $2 赋值给 b。
#no1
[ $# -ne 2 ] &&{ #<== 传参专用判断手法，判断传参个数是否为两个。
    echo "USAGE:$0 NUM1 NUM2" #<== 传参不合要求，给出用法提示。
    exit 1 #<== 以返回值 1 退出脚本。
}
#no2
expr $a + 10 &>/dev/null     #<== 整数判断，前文已讲。
RETVAL1=$?
expr $b + 10 &>/dev/null     #<== 整数判断，前文已讲。
RETVAL2=$?
test $RETVAL1 -eq 0 -a $RETVAL2 -eq 0||{
    echo "Pls input two "num" again."
    exit 2
}
#no3
[ $a -lt $b ] &&{
    echo "$a < $b"
    exit 0
}
#no4
[ $a -eq $b ] &&{
    echo "$a = $b"
    exit 0
}
#no5
[ $a -gt $b ] &&{
    echo "$a > $b"
}
```

执行结果如下：

```
[root@oldboy scripts]# sh 6_35_2.sh
USAGE:6_34_2.sh NUM1 NUM2
[root@oldboy scripts]# sh 6_35_2.sh 6 2
6 > 2
[root@oldboy scripts]# sh 6_35_2.sh 6 6
6 = 6
[root@oldboy scripts]# sh 6_35_2.sh 2 6
2 < 6
[root@oldboy scripts]# sh 6_35_2.sh oldboy oldgirl
Pls input two num again.
```

范例 6-36：打印选择菜单，按照选择项一键安装不同的 Web 服务。

示例菜单：

```
[root@oldboy scripts]# sh menu.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
```

要求：

1）当用户输入 1 时，输出“start installing lamp”提示，然后执行 /server/scripts/lamp.sh 输出“lamp is installed”，并退出脚本，此为工作中所用的 lamp 一键安装脚本。

2）当用户输入 2 时，输出“start installing lnmp”提示，然后执行 /server/scripts/lnmp.sh 输出“lnmp is installed”，并退出脚本，此为工作中所用的 lnmp 一键安装脚本。

3）当输入 3 时，退出当前菜单及脚本。

4）当输入任何其他字符时，给出提示“Input error”后退出脚本。

5）对执行的脚本进行相关的条件判断，例如：脚本文件是否存在，是否可执行等的判断，尽量使用前面讲解过的知识点。

先来看一个解答前的热身示例脚本。打印简单的所选菜单示例 1。

```
[root@oldboy scripts]# cat 6_36_1.sh
cat <<END   #<== 这里的 cat 用法就是官方所说的 here 文档用法，其实就是按指定格式打印多行文本。
    1.panxiaoting
    2.gongli
    3.fanbinbing
END   #<== 注意顶格写，开头不要带有空格，END 成对出现，可以用任意的成对字符来替代，不要和
          内容冲突即可。
read -p "Which do you like?Pls input the num:" a   #<== 读入用户的选择，赋值给变量 a。
[ "$a" = "1" ] &&{   #<== 条件表达式判断 a 的值是否为 1，注意，这里推荐用字符串的语法格式判断。
    echo "I guess,you like panxiaoting"   #<== 根据用户选择的结果，回应应用输出。
    exit 0 #<== 退出脚本，不再向下执行。
}
[ "$a" = "2" ] &&{ #<== 条件表达式判断 a 的值是否为 2。
    echo "I guess,you like gongli"
    exit 0   #<== 退出脚本，不再向下执行。
}

[ "$a" = "3" ] &&{ #<== 条件表达式判断 a 的值是否为 3。
    echo "I guess,you like fangbingbing"
    exit 0   #<== 退出脚本，不再向下执行。
}
[[ ! "$a" =~ [1-3] ]] &&{   #<== 这里就是用到了 [[]] 的通配符匹配的用法，即 a 是否为 1 或 2 或 3。
    echo "I guess,you are not man."
}
```

执行结果如下：

```
[root@oldboy scripts]# sh 6_36_1.sh
   1.panxiaoting
   2.gongli
   3.fanbinbing
Which do you like?Pls input the num:1 #<== 符合所选菜单的数字要求，给予选择结果的提示。
I guess,you like panxiaoting
[root@oldboy scripts]# sh 6_36_1.sh
   1.panxiaoting
   2.gongli
   3.fanbinbing
Which do you like?Pls input the num:2 #<== 符合所选菜单的数字要求，给予选择结果的提示。
I guess,you like gongli
[root@oldboy scripts]# sh 6_36_1.sh
   1.panxiaoting
   2.gongli
   3.fanbinbing
Which do you like?Pls input the num:3 #<== 符合所选菜单的数字要求，给予选择结果的提示。
I guess,you like fangbingbing
[root@oldboy scripts]# sh 6_36_1.sh
   1.panxiaoting
   2.gongli
   3.fanbinbing
Which do you like?Pls input the num:5 #<== 不符合所选菜单的数字要求，给予非法输入的提示。
I guess,you are not man.
[root@oldboy scripts]# sh 6_36_1.sh
   1.panxiaoting
   2.gongli
   3.fanbinbing
Which do you like?Pls input the num:99 #<== 不符合所选菜单的数字要求，给予非法输入的提示。
I guess,you are not man.
```

以下是第 2 个热身示例脚本。打印简单的所选菜单示例 2，本例是使用函数功能编写的脚本，有关函数知识，读者可以参考后面的第 8 章。

```
[root@oldboy 01]# cat menu.sh
menu(){
cat <<END
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
END
}
menu
[root@oldboy 01]# sh menu.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
```

以下是正式解答：

```
[root@oldboy scripts]# mkdir -p /server/scripts  #<== 建立脚本存放路径。
[root@oldboy scripts]# cd /server/scripts        #<== 切换到脚本下。
[root@oldboy scripts]# echo "echo lamp is installed" >lamp.sh
#<== 模拟 lamp 脚本输出。
[root@oldboy scripts]# echo "echo lnmp is installed" >lnmp.sh
#<== 模拟 lnmp 脚本输出。
[root@oldboy scripts]# chmod +x lnmp.sh lamp.sh  #<== 给予执行权限。
[root@oldboy scripts]# cat 6_36_2.sh             #<== 正式脚本如下。
#!/bin/sh
path=/server/scripts #<== 定义脚本路径
[ ! -d "$path" ] && mkdir $path -p #<== 条件表达式判断，如果目录不存在，则创建目录。
#menu
cat <<END #<== 利用 cat 命令打印选择菜单，这里也可以用 select 语句打印选择菜单。
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
END
read num #<== 接收用户选择的数字。
expr $num + 1 &>/dev/null      #<== 判断是否为整数。
[ $? -ne 0 ] &&{               #<== 根据返回值进行判断。
    echo "the num you input must be {1|2|3}"
    exit 1
}
[ $num -eq 1 ] &&{             #<== 如果用户选择 1，则执行 lamp 安装命令。
    echo "start installing lamp."
    sleep 2;
    [ -x "$path/lamp.sh" ]||{ #<== 判断脚本是否可执行，若不可执行则给予提示。
        echo "$path/lamp.sh does not exist or can not be exec."
        exit 1
    }
    $path/lamp.sh #<== 执行脚本安装脚本，工作中建议用 source $path/lamp.sh 替代，这
                      里的目的是练习 -x 的判断。
    #source $path/lamp.sh #<== 脚本中执行脚本，使用 source 比 sh 或不加解释器等更好一些。
    exit $?
}

[ $num -eq 2 ] &&{             #<== 如果用户选择 2，则执行 lnmp 安装命令。
    echo "start installing LNMP."
    sleep 2;
    [ -x "$path/lnmp.sh" ]||{ #<== 判断脚本是否可执行，若不可执行则给予提示。
        echo "$path/lnmp.sh does not exist or can not be exec."
        exit 1
    }
    $path/lnmp.sh #<== 执行脚本安装脚本，工作中建议用 source $path/lamp.sh 替代，这
                      里的目的是练习 -x 的判断。
```

```
    #source $path/lnmp.sh #<== 脚本中执行脚本，使用 source 比使用 sh 或不加解释器等更
                               好一些。
    exit $?
}
[ $num -eq 3 ] &&{ #<== 如果用户输入 3，则退出脚本。
    echo bye.
    exit 3
}
# 这里有三种用户的输入不等于 1、2 或 3 的综合用法。
[[ ! $num =~ [1-3] ]] &&{ #<==[[]] 的正则匹配方法。
    echo "the num you input must be {1|2|3}"
    echo "Input ERROR"
    exit 4
}
```

提示：这里关于判断用户的输入是否等于 1、2 或 3 的用法共给出了三种，请读者重视。

执行结果如下：

```
[root@oldboy scripts]# sh 6_36_2.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
1 #<== 选择 1，执行 lamp 脚本。
start installing lamp.
lamp is installed
[root@oldboy scripts]# sh 6_36_2.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
2 #<== 选择 2，执行 lnmp 脚本。
start installing LNMP.
lnmp is installed
[root@oldboy scripts]# sh 6_36_2.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
3 #<== 选择 3，退出脚本。
bye.
[root@oldboy scripts]# sh 6_36_2.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
```

```
    pls input the num you want:
5 #<== 其他数字，给予提示，并退出。
the num you input must be {1|2|3}
Input ERROR
```


提示：使用菜单时还可以用 select 语句，不过还是 cat 的方法更常用，也更简单。

6.6 测试表达式 test、[]、[[]]、(()) 的区别总结

测试表达式的语法比较复杂且容易混淆，对于初学者，一定要给自己设定个知识边界，表 6-6 列出了测试表达式 []、[[]]、(())、test 的区别。

表 6-6 不同符号测试表达式 []、[[]]、(())、test 的区别

测试表达式符号	[]	test	[[]]	(())
边界为是否需要空格	需要	需要	需要	不需要
逻辑操作符	!、-a、-o	!、-a、-o	!、&&、||	!、&&、||
整数比较操作符	-eq、-gt、-lt、-ge、-le	-eq、-gt、-lt、-ge、-le	-eq、-gt、-lt、-ge、-le 或 =、>、<、>=、<=	=、>、<、>=、<=
字符串比较操作符	=、==、!=	=、==、!=	=、==、!=	=、==、!=
是否支持通配符匹配	不支持	不支持	支持	不支持

普通的读者学习 Shell 编程主要是为了解决工作中的问题，因此无须掌握全部的语法，建议多用老男孩推荐的 [] 的用法，对其他语法了解即可，当有需要时，可以翻看本书或查阅 bash 文档（man bash），以及对应命令（man test）的帮助。

特别说明：可访问如下地址或手机扫二维码查看第 6 章的核心脚本代码。

http://oldboy.blog.51cto.com/2561410/1855641

Linux

第7章

if 条件语句的知识与实践

对于 if 条件语句，简单地说，其语义类似于汉语里的“如果…那么”。if 条件语句是 Linux 运维人员在实际生产工作中使用得最频繁也是最重要的语句，因此，请务必重视 if 条件语句的知识，并牢固掌握。

7.1 if 条件语句

7.1.1 if 条件语句的语法

1. 单分支结构

第一种语法：

```
if  < 条件表达式 >
  then
    指令
fi
```

第二种语法：

```
if  < 条件表达式 >; then
    指令
fi
```

上文的“< 条件表达式 >”部分可以是 test、[]、[[]]、(()) 等条件表达式，甚至可以直接使用命令作为条件表达式。每个 if 条件语句都以 if 开头，并带有 then，最后以 fi

结尾。

第二种语法中的分号相当于命令换行，上面的两种语法含义是相同的，读者可根据习惯自行选择。本书中主要使用第一种语法格式。

在所有编程语言里，if 条件语句几乎是最简单的语句格式，且用途最广。当 if 后面的 < 条件表达式 > 成立时（真），就会执行 then 后面的指令或语句；否则，就会忽略 then 后面的指令或语句，转而执行 fi 下面的程序。

if 单分支语句执行流程逻辑图如图 7-1 所示。

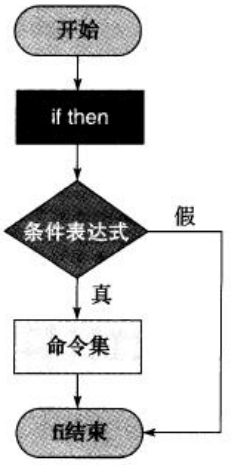


图 7-1　if 单分支语句执行流程逻辑图

条件语句还可以嵌套（即 if 条件语句里面还有 if 条件语句），注意每个 if 条件语句中都要有一个与之对应的 fi（if 反过来写），每个 if 和它下面最近的 fi 成对搭配，语法示例如下：

```
if  < 条件表达式 >
  then
    if  < 条件表达式 >
      then
        指令
    fi
fi
```

提示：通常在书写 Shell 条件语句编程时，要让成对的条件语句关键字的缩进相对应，以便于阅读浏览。

前文曾讲解过的文件条件表达式 **[-f "$file1"] && echo 1** 就等价于下面的 if 条件语句。

```
if [ -f "$file1" ];then
    echo 1
fi
```

为了便于大家记忆 if 单分支语句的语法，老男孩给出了形象的语法表述。if 条件语句单分支的中文编程就相当于一个女孩对你说：

```
如果 < 你有房 >
  那么
    我就嫁给你
果如
```

提示：如果真能用中文编程该有多好！

2. 双分支结构

if 条件语句的单分支结构主体就是“如果…，那么…”，而 if 条件语句的双分支结构主体则为“如果…，那么…，否则…”。

if 条件语句的双分支结构语法为：

```
if  < 条件表达式 >
  then
    指令集 1
else
    指令集 2
fi
```

前文的文件测试条件表达式 [-f "$file1"] && echo 1 || echo 0 就相当于下面的双分支的 if 条件语句。

```
if [-f "$file1" ]
  then
    echo 1
else
    echo 0
fi
```

此外，也可以把 then 和 if 放在一行用分号（;）隔开。

同样，老男孩也对此给出了形象的描述，if 条件语句双分支的中文编程就相当于一个女孩对你说：

```
如果 < 你有房 >
  那么
    我就嫁给你
否则
    我再考虑下
果如
```

提示：这个语句很形象地描述了社会的现实，加油吧！

if 双分支语句执行流程逻辑图如图 7-2 所示。

3. 多分支结构

if 条件语句多分支结构的主体为“如果…，那么…，否则如果…，那么，否则如果…，那么…，否则…”。

if 条件语句多分支语法为：

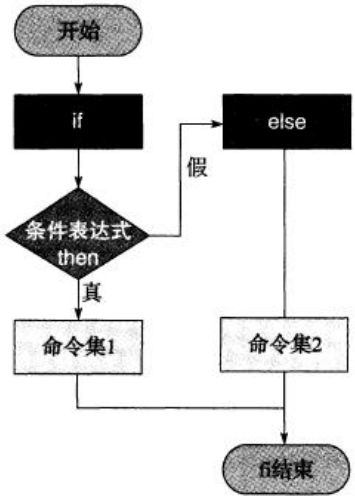


图 7-2　if 双分支语句执行流程逻辑图

```
if  <条件表达式 1>
  then
    指令 1
elif <条件表达式 2>
  then
    指令 2
else
    指令 3
fi
----------------------- 多个 elif----------------------------
if  <条件表达式 1>
  then
    指令
elif <条件表达式 2>
  then
    指令
elif <条件表达式 3>
  then
    指令
    ... ...
else
    指令
fi
```

提示：

1）注意多分支 elif 的写法，每个 elif 都要带有 then。

2）最后结尾的 else 后面没有 then。

多分支 if 条件语句的形象描述就相当于一个女孩对你说：

```
如果 <你有房>                          #<== 有钱。
  那么
    我就嫁给你
或者如果 <你爸有背景>                  #<== 有权。
  那么
    我也可以嫁给你
或者如果 <你很努力很吃苦>              #<== 有潜力，老男孩曾经就是在此条件里。
  那么
    我们可以先谈谈男女朋友
否则
    不理你                             #<== 没钱，没家庭背景，还不勤奋努力，必然遭淘汰。
果如
```

提示：老男孩曾经的写照是 8 个字——勤奋努力，善于总结。

if 多分支语句执行流程对应的逻辑图如图 7-3 所示。

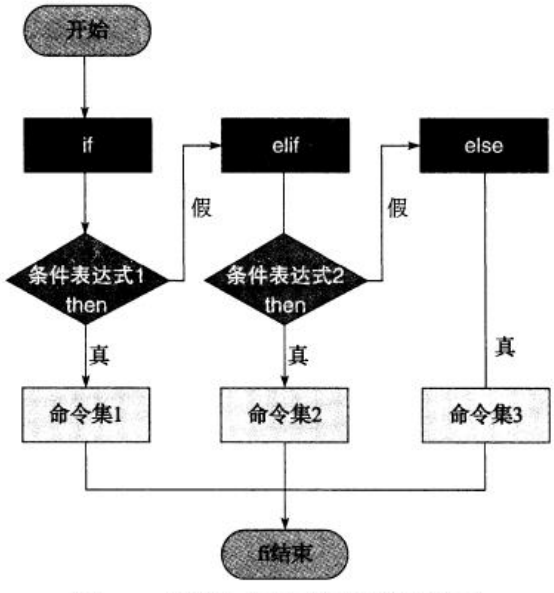


图 7-3　if 多分支语句执行流程逻辑图

7.1.2　if 条件语句多种条件表达式语法

前文已经说过，if 条件语句（包括双多分支 if）的“<条件表达式>”部分可以是 test、[]、[[]]、(()) 等条件表达式，甚至还可以直接使用命令作为条件表达式，具体的语

法如下。

（1）test 条件表达式

```
if test 表达式
  then
    指令
fi
```

（2）[] 条件表达式

```
if [ 字符串或算术表达式 ]
  then
    指令
fi
```

（3）[[]] 条件表达式

```
if [[ 字符串表达式 ]]
  then
    指令
fi
```

（4）(()) 条件表达式

```
if (( 算术表达式 ))
  then
    指令
fi
```

（5）命令表达式

```
if 命令
  then
    指令
fi
```

说明：以上表达式除了语法不同之外，具体的应用是一致的，实际工作场景中，读者只需选择一种适合自己习惯的表达式就好。

7.1.3 单分支 if 条件语句实践

下面列举几个使用 if 条件语句的例子。

范例 7-1：把下面测试文件中条件表达式的语句改成 if 条件语句。

```
[root@oldboy ~]# [ -f /etc/hosts ] && echo 1   #<==[]（单中括号）条件表达式语法。
1
```

```
[root@oldboy ~]# [[ -f /etc/hosts ]] && echo 1  #<==[[]](双中括号) 条件表达式语法。
1
[root@oldboy ~]# test -f /etc/hosts && echo 1  #<==test 条件表达式语法。
1
```

参考答案：

```
[root@oldboy scripts]# cat 7_1.sh
if [ -f /etc/hosts ]              #<==[] (单中括号) 条件表达式改为 if 单分支语句。
  then
    echo "[1]"
fi

if [[ -f /etc/hosts ]]            #<==[[]] (双中括号) 条件表达式改为 if 单分支语句。
  then
    echo "[[1]]"
fi

if test -f /etc/hosts             #<==test 条件表达式改为 if 单分支语句。
  then
    echo "test1"
fi
```

执行结果如下：

```
[root@oldboy scripts]# sh 7_1.sh
[1]
[[1]]
test1
```

范例 7-2：开发 Shell 脚本判断系统剩余内存的大小，如果低于 100MB，就邮件报警给系统管理员，并且将脚本加入系统定时任务，即每 3 分钟执行一次检查。

在解答示例之前，先介绍一些重要的内容：开发程序时，一定要重视分析问题及程序设计过程（例如：分模块实现第一关、第二关、第三关），确定需求并完成设计后，编码就是很容易的事情了。因此，请大家重视分析问题和程序设计的思想，而不要一上来就开始编码，这也是老男孩常说的，技术的提升仅是量的积累，思想的提升才是质的飞跃！

对于开发程序而言，一般来说应该遵循下面的 3 步法则。

（1）分析需求

明白开发需求，是完成程序的大前提，因此，分析需求至关重要，一切不以需求为主的程序开发，都是不倡导的！

（2）设计思路

设计思路就是根据需求，把需求进行拆解，分模块逐步实现，例如本题可以分为如下几步：

1）获取当前系统剩余内存的值（先在命令行实现）。

2）配置邮件报警（可采用第三方邮件服务器）。

3）判断取到的值是否小于 100MB，如果小于 100MB，就报警（采用 if 语句）。

4）编码实现 Shell 脚本。

5）加入 crond 定时任务，每三分钟检查一次。

（3）编码实现

编码实现就是具体的编码及调试过程，工作中很可能需要先在测试环境下调试，调试好了，再发布到生产环境中。

本例的最终实现过程如下。

1）获取内存大小，注意：内存大小是下面命令中对应 **buffers/cache 那一行**结尾的值（buffers）。

```
[root@oldboy scripts]# free -m
             total       used       free     shared    buffers     cached
Mem:           994        903         91          0        103        672
-/+ buffers/cache:        127        867
#<=== 这里的 867 是系统的剩余内存，因为 Linux 系统有缓冲和缓存的特性，即把系统多余的内存临时用于缓冲和缓存，以提升系统的性能和效率。
Swap:            0          0          0
[root@oldboy scripts]# free -m|awk 'NR==3 {print $NF}' #<=== 利用 awk 获取到 867，默认单位为 MB。
867
```

2）发邮件的客户端常见的有 mail 或 mutt；服务端有 sendmail 服务（CentOS 5 下默认的）、postfix 服务（CentOS 6 下默认的）。这里不使用本地的邮件服务，而是使用本地的 mail 客户端，以及第三方的邮件服务商，例如：163.com（需要提前注册用户），利用这个邮件账号来给接收报警的人发送报警邮件。

```
[root@oldboy scripts]# echo -e "set from=oldboy@163.com smtp=smtp.163.com\
nset smtp-auth-user=oldboy smtp-auth-password=oldboy123 smtp-auth=login" >>/
etc/mail.rc   #<== 配置 mail.rc，将 Linux 本地作为邮件客户端，使用注册的 163 账号及 smtp 地址发送邮件。
[root@oldboy scripts]# tail -2 /etc/mail.rc
set from=oldboy@163.com smtp=smtp.163.com
#<== from=oldboy@163.com 为邮件地址，smtp=smtp.163.com 为邮件服务器。
set smtp-auth-user=oldboy smtp-auth-password=oldboy123 smtp-auth=login
#<==oldboy 为用户名，oldboy123 为密码。
[root@oldboy scripts]# echo "oldboy"|mail -s "title" oldboy@163.com
#<== 测试邮件发送（服务器要能上网）。
[root@oldboy scripts]# echo oldboy >/tmp/test.txt #<== 将正文放入文件。
[root@oldboy scripts]# mail -s "title" oldboy@163.com </tmp/test.txt
#<== 读取文件内容并发送邮件。查看 163 邮件是否发送成功，本书是用自己的账号给自己发送邮件。
```

3）编写 Shell 脚本。

```
[root@oldboy scripts]# cat 7_2.sh
```

```
FreeMem=`free -m|awk 'NR==3 {print $NF}'`  #<== 获取系统当前的内存值，赋给变量 FreeMem。
CHARS="Current memory is  $FreeMem."       #<== 定义字符串 CHARS 变量，作为输出及供邮
                                               件正文使用。
if [ $FreeMem -lt 100 ]                    #<==[]（单中括号）条件表达式语法，这是老
                                               男孩推荐的用法，如果小于 100，则执行命令。
  then
    echo $CHARS|tee /tmp/messages.txt #<== 屏幕输出提示，并写入文件。
    mail -s "`date +%F-%T`$CHARS" test@oldboyedu.com </tmp/messages.txt
                                           #<== 发送邮件报警。
fi
```

提示：test@oldboyedu.com 是报警邮箱，工作中应尽量使用企业邮箱，以防止邮件被屏蔽。

执行测试：

```
[root@oldboy scripts]# /bin/sh /server/scripts/7_2.sh
Current memory is 866.
```

提示：

1）测试时，我们没办法把内存调小，但可以把阈值调大，例如将 100MB 改成 1000MB。

2）注意，如果使用第三方邮件服务，则服务器本地无需开启邮件服务。

接下来，把上述脚本加入 crond 定时任务中，每 3 分钟检查一次，达到阀值就发邮件报警：

```
[root@oldboy scripts]# crontab -l|tail -2
# monitor sys mem at 20160813 by oldboy
*/3 * * * * /bin/sh /server/scripts/7_2.sh &>/dev/null
```

监测内存并报警有多种 Shell 写法，如下：

```
if ((FreeMem<1000))                #<==(()) 条件表达式语法。
  then
    echo $CHARS|tee /tmp/messages.txt
    #mail -s "`date +%F-%T`$CHARS" 490004487@qq.com </tmp/messages.txt
fi

if [[ $FreeMem -lt 1000 ]]         #<==[[]] 条件表达式语法。
  then
    echo $CHARS|tee /tmp/messages.txt
    #mail -s "`date +%F-%T`$CHARS" 490004487@qq.com </tmp/messages.txt
fi
```

```
if test $FreeMem -lt 1000          #<==test 条件表达式语法。
  then
    echo $CHARS|tee /tmp/messages.txt
    #mail -s "`date +%F-%T`$CHARS" 490044487@qq.com </tmp/messages.txt
fi
```

读者可以进行实践扩展：监控本地磁盘、MySQL 服务和 Web 服务。

老男孩运维思想：

做事情若有多种选择，就会比较轻松，没有选择就会比较痛苦。

汇报领导交代的任务就要给领导多种选择，只有一种选择，老大没得选，不叫有能力。有两种选择，老大左右为难，不叫有能力。有三种或以上的选择，才叫有能力。

提交解决方案、面试提问、笔试等都是如此，甚至老男孩写书也会尽量给读者多种思路，供读者选择。

7.1.4 if 条件语句的深入实践

范例 7-3：分别使用 read 读入及脚本传参的方式比较两个整数的大小。

说明：这个例子前文用条件表达式已经实现过了，这里是希望大家改成用 if 条件语句来实现，另外，read 读入和命令行传参是输入内容的两种方法（对应两个脚本）。

参考答案 1：使用单分支 if 语句和 read 读入实现整数大小的比较。

比较两个整数的大小有三种情况，因此用一个单分支 if 是无法实现的，需要 3 个单分支的 if 判断才行，每一个单分支 if 条件语句相当于前文的一个条件表达式判断，最终的脚本如下：

```
[root@oldboy scripts]# cat 7_3_1.sh
#!/bin/sh
read -p "pls input two num:" a b    #<== 读入两个输入，分别赋值给变量 a 和 b。
if [ $a -lt $b ];then               #<== 如果 $a 小于 $b，则执行命令。
    echo "yes,$a less than $b"      #<== 打印输出，提醒用户。
    exit 0                          #<== 判断完毕，成功执行，以 0 值退出脚本，此处如果不退出，则会继续执行下面的 if 语句，而这是不必要的。
fi
if [ $a -eq $b ];then               #<== 如果 $a 等于 $b，则执行命令，同理，成功后以 0 值退出脚本。
    echo "yes,$a equal $b"
    exit 0
fi
if [ $a -gt $b ];then               #<== 如果 $a 大于 $b，则执行命令，同理，成功后以 0 值退出脚本。
```

```
    echo "yes,$a greater than $b"
    exit 0
fi
```

本例执行结果如下：

```
[root@oldboy scripts]# sh 7_3_1.sh
pls input two num:6 2
yes,6 greater than 2
[root@oldboy scripts]# sh 7_3_1.sh
pls input two num:2 6
yes,2 less than 6
[root@oldboy scripts]# sh 7_3_1.sh
pls input two num:6 6
yes,6 equal 6
```

提示：此题目并未要求对传参个数及参数是否为整数进行判断，读者可以自己试一试，如果有问题，可以参考前文及后文的讲解。

上述使用 if 单分支语句比较参数大小的脚本有几个问题：

- 没有对参数的个数及传入变量是不是整数做判断。
- if 条件语句多而杂，导致逻辑不够清晰。

那么有没有好一点的方法呢？当然有，见后文多分支 if 及 case 语句功能。

参考答案 2：使用多分支 if 语句和利用 read 读入变量实现整数大小的比较。

```
[root@oldboy scripts]# cat 7_3_2.sh
#!/bin/sh
read -p "pls input two num:" a b        #<== 读入两个输入，分别赋值给变量 a 和 b。
if [ $a -lt $b ];then                   #<== 使用 if 多分支语法进行比较。
    echo "yes,$a less than $b"          #<== 由于是一个 if 语句，因此就不需要退出脚本了。
elif [ $a -eq $b ];then
    echo "yes,$a equal $b"
else [ $a -gt $b ]
    echo "yes,$a greater than $b"
fi
```

提示：此题目并未要求对传参个数及参数是否为整数进行判断，读者可以自己试一试，如果有问题，可以参考前文及后文的讲解。

执行结果如下：

```
[root@oldboy scripts]# sh 7_3_2.sh
pls input two num:8 4
yes,8 greater than 4
```

```
[root@oldboy scripts]# sh 7_3_2.sh
pls input two num:4 8
yes,4 less than 8
[root@oldboy scripts]# sh 7_3_2.sh
pls input two num:8 8
yes,8 equal 8
```

参考答案 3：用脚本传参的方式比较整数大小。

对上文的 read 脚本进行简单修改即可实现传参的比较。单分支实现脚本如下：

```
[root@oldboy scripts]# cat 7_3_3.sh
#!/bin/sh
a=$1 #<== 将脚本命令行的第一个参数赋值给变量 a。
b=$2 #<== 将脚本命令行的第二个参数赋值给变量 b。
if [ $a -lt $b ];then
    echo "yes,$a less than $b"
    exit 0
fi
if [ $a -eq $b ];then
    echo "yes,$a equal $b"
    exit 0
fi
if [ $a -gt $b ];then
    echo "yes,$a greater than $b"
    exit 0
fi
```

多分支实现脚本如下：

```
[root@oldboy scripts]# cat 7_3_4.sh
#!/bin/sh
a=$1 #<== 将脚本命令行的第一个参数赋值给变量 a。
b=$2 #<== 将脚本命令行的第二个参数赋值给变量 b。
if [ $a -lt $b ];then
    echo "yes,$a less than $b"
elif [ $a -eq $b ];then
    echo "yes,$a equal $b"
else [ $a -gt $b ]
    echo "yes,$a greater than $b"
fi
```

执行结果略。

7.2 if 条件语句企业案例精讲

7.2.1 监控 Web 和数据库的企业案例

范例 7-4：用 if 条件语句针对 Nginx Web 服务或 MySQL 数据库服务是否正常进行

检测，如果服务未启动，则启动相应的服务。

这是企业级运维实战综合题，需要读者对 Nginx Web 服务或 MySQL 数据库服务很熟悉才行，本例同样适用于其他的 Web 服务和数据库服务。

大家还记得前面说过的开发程序前的三部曲吧？这里就采用这种方式来解答此题。

（1）分析问题

先想一想监控 Web 服务和 MySQL 数据库服务是否异常的方法有哪些，见表 7-1。

表 7-1　监控 Web 服务和 MySQL 数据库服务是否异常的常见方法

端口监控	1）在服务器本地监控服务端口的常见命令有 netstat、ss、lsof 2）从远端监控服务器本地端口的命令有 telnet、nmap、nc
监控服务进程或进程数	此方法适合本地服务器，注意，过滤的是进程的名字。命令为： ps -ef|grep nginx|wc -l ps -ef|grep mysql|wc -l
在客户端模拟用户访问	使用 wget 或 curl 命令进行测试（如果监测数据库，则需要转为通过 Web 服务器去访问数据库），并对测试结果做三种判断： 1）利用返回值（echo $?）进行判断 2）获取特殊字符串以进行判断（需要事先开发好程序） 3）根据 HTTP 响应 header 的情况进行判断
登录 MySQL 数据库判断	通过 MySQL 客户端连接数据库，根据返回值或返回内容判断。例如： mysql -uroot -poldboy123 -e "select version();" &>/dev/null; echo $?

注：查看远端端口是否通畅的 3 个简单实用的案例见：http://oldboy.blog.51cto.com/2561410/942530。

此外，对端口进程等进行判断时，尽量先通过 grep 过滤端口和进程特殊标记字符串，然后结合 wc 将过滤到的结果转成行数再比较，这样相对简单有效，且经过 wc -l 命令处理之后的结果一定是数字，这样再进行判断就会比较简便。如果单纯地根据具体的列取具体的值判断会很麻烦，如果确实想采用取值判断的方法，那就尽量用字符串比较的语法。

提示：掌握技术思想比解决问题本身更重要（具体见 http://oldboy.blog.51cto.com/2561410/1196298）。

（2）监测 MySQL 数据库异常

1）MySQL 数据库环境准备，如下：

```
[root@oldboy ~]# yum install mysql-server -y
[root@oldboy ~]# /etc/init.d/mysqld start
正在启动 mysqld:                                    [ 确定 ]
[root@oldboy ~]# netstat -lntup|grep mysql
tcp        0      0 0.0.0.0:3306        0.0.0.0:*         LISTEN      2275/mysqld
```

2）通过命令行检测数据库服务是否正常，只有先确定命令行是正确的，才能确保

将它放到脚本里也是正确的。

首先采用端口监控的方式。在服务器本地监控端口的命令有 netstat、ss、lsof，具体实现多种命令的方法如下：

```
[root@oldboy scripts]# netstat -lnt|grep 3306|awk -F "[ :]+" '{print $5}'
#<== 这个就是前面所描述的根据具体的列取值判断的方法，获取到值，然后看看其是否等于 3306，老男孩不推荐采用这种取值方法，因为第一取值麻烦；第二如果使用数字比较，当端口不存在时就会报错，而一旦使用了取值判断方法，即使看起来是数字，也要尽量使用字符串进行比较，否则你将掉进坑里，很久都不会爬出来。
3306
[root@oldboy scripts]# netstat -lntup|grep 3306|wc -l
#<== 过滤关键字端口，转成数字，此方法很好。
1
[root@oldboy scripts]# netstat -lntup|grep mysql|wc -l
#<== 过滤关键字进程，转成数字，此方法很好。
1
[root@oldboy scripts]# ss -lntup|grep mysql|wc -l
#<==ss 类似于 netstat 命令，参数选项可通用。
1
[root@oldboy scripts]# ss -lntup|grep 3306|wc -l
#<==ss 类似于 netstat 命令，参数选项可通用。
1
[root@oldboy scripts]# lsof -i tcp:3306|wc -l
#<== 利用 lsof 检查 tcp 协议的 3306 端口。
2
```

从远端监控服务器监控本地端口的命令有 telnet、nmap、nc，这三个命令有可能需要事先安装好才能使用，安装方法为：

```
[root@oldboy scripts]# yum install telnet nmap nc -y
[root@oldboy scripts]# nmap 127.0.0.1 -p 3306|grep open|wc -l
#<== 查看远端 3306 端口是否开通，过滤 open 关键字，结果返回 1，说明有 open 关键字，表示 3306 端口是通的。
1
[root@oldboy scripts]# echo -e "\n"|telnet 127.0.0.1 3306 2>/dev/null|grep Connected|wc -l
#<==telnet 是常用来检测远端服务器端口是否通畅的一个好用的命令，在非交互时需要采用特殊写法才行，过滤的关键字为 Connected，返回 1，说明有 Connected，表示 3306 端口是通的。
1
[root@oldboy scripts]# nc -w 2 127.0.0.1 3306 &>/dev/null
#<==nc 的命令很强大，这里用来检测端口。根据执行命令的返回值判断端口是否通畅，如果返回 0，则表示通畅，-w 为超时时间。
[root@oldboy scripts]# echo $?
0
```

本例为了统一 IP 地址，因此使用的都是同一个 IP，即 127.0.0.1，在实际工作中，应该用自己服务器的 IP 来替代。

下面对服务进程或进程数进行监控（适合本地服务器）：

```
[root@oldboy scripts]# ps -ef|grep mysql|grep -v grep|wc -l
3
```

以下是在客户端模拟用户访问的方式进行监控。

使用 wget 或 curl 命令访问 URL 地址来测试（如果要检测数据库是否异常，需要转为通过访问 Web 服务器去访问数据库）时，有三种判断思路。

第一种是根据执行命令的返回值判断成功与否，本例的 URL 使用了网上的地址，在实际工作中应使用开发人员提供给我们的访问数据库的程序地址。

```
[root@oldboy scripts]# wget --spider --timeout=10 --tries=2 www.baidu.com
&>/dev/null
#<== 在 wget 后面加 url 的检测方法，&>/dev/null 表示不输出，只看返回值。--spider 的意思是模拟爬取，--timeout=10 的意思是 10 秒超时，--tries=2 表示如果不成功，则重试 2 次。
[root@oldboy scripts]# echo $?  #<== 查看命令执行的返回值，0 为成功。
0
[root@oldboy scripts]# wget -T 10 -q --spider http://www.baidu.com >&/dev/
null  #<== 用法同第一个 wget，-q 表示安静的。
[root@oldboy scripts]# echo $?  #<== 查看命令执行的返回值，0 为成功。
0
[root@oldboy scripts]# curl -s -o /dev/null http://www.baidu.com
#<== 利用 curl 进行检测，-s 为沉默模式，-o /dev/null 表示将输出定向到空。
[root@oldboy scripts]# echo $?  #<== 查看命令执行的返回值，0 为成功。
0
```

第二种是根据执行命令后获取到的字符串进行判断。此方法需要有前端动态程序支持，即开发 PHP 或 Java 程序从数据库里取出指定的字符串，看它和期待的是否相等。下面以 PHP 服务为例（对于一个 PHP 程序访问数据库，如果访问成功，则给出固定的输出）。

```
[root@oldboy scripts]# /server/scripts/testmysql.php
<?php
/*
#this scripts is created by oldboy
#oldboy QQ:31333741
#site: http://www.etiantian.org
#blog: http://oldboy.blog.51cto.com
*/
    //$link_id=mysql_connect('主机名','用户','密码');
    $link_id=mysql_connect('localhost','root','matianhai') or mysql_error();
    //$link_id=mysql_connect('localhost','test','');
    if($link_id){
        echo "mysql successful by oldboy !";
    }else{
        echo mysql_error();
    }
?>
```

命令行执行：

```
[root@oldboy scripts]# php /server/scripts/testmysql.php #<== 注意 .php 需执行
(yuminstall php -y) 安装
mysql successful by oldboy !     #<== 可以通过 grep 过滤这里输出的关键字，进而判断访
                                     问数据库是否成功。
```

提示：需要事先安装 PHP 软件才行，或者将程序放到 Lnmp 服务器的站点目录，然后 curl 或 \wget 访问 http 地址。此方法是监控数据库是否异常的最佳的方法。

3）开发监控 MySQL 数据库的脚本。

这里将给出多种开发脚本供读者参考。

脚本 1：

```
#!/bin/sh
echo method1--------------------
if [ `netstat -lnt|grep 3306|awk -F "[ :]+" '{print $5}'` -eq 3306 ]
#<== 这里的判断思路不是很好，即①最好不要用整数进行比较，因为一旦端口不存在，取值就会为
空，进行整数比较会报错。②不要根据列取具体的值，而是要过滤关键字，通过 wc 转成行数判断。
  then
    echo "MySQL is Running."
else
    echo "MySQL is Stopped."
    /etc/init.d/mysqld start
fi
```

脚本 2：

```
echo method2--------------------
if [ "`netstat -lnt|grep 3306|awk -F "[ :]+" '{print $5}'`" = "3306" ]
#<== 用字符串的方式进行比较就好多了，避免了脚本 1 中整数比较的错误发生，但是取值麻烦了一些。
  then
    echo "MySQL is Running."
else
    echo "MySQL is Stopped."
    /etc/init.d/mysqld start
fi
```

脚本 3：

```
echo method3--------------------
if [ `netstat -lntup|grep mysqld|wc -l` -gt 0 ]  #<== 过滤进程名，转成数字，很
                                                     优秀的取值判断方法。
  then
    echo "MySQL is Running."
else
    echo "MySQL is Stopped."
```

```
    /etc/init.d/mysqld start
fi
```

脚本 4：

```
echo method4--------------------
if [ `lsof -i tcp:3306|wc -l` -gt 0 ]  #<== 过滤端口转成数字，很优秀的取值判断方法。
  then
    echo "MySQL is Running."
else
    echo "MySQL is Stopped."
    /etc/init.d/mysqld start
fi
```

脚本 5：

```
echo method5--------------------
[ `rpm -qa nmap|wc -l` -lt 1 ] && yum install nmap -y &>/dev/null
#<== 防止因 nmap 没有安装而导致的错误。
if [ `nmap 127.0.0.1 -p 3306 2>/dev/null|grep open|wc -l` -gt 0 ]
#<== 远端的端口检查，推荐使用，同理，这里也不要过滤出 open，然后再做字符比较，转成数字最佳。
  then
    echo "MySQL is Running."
else
    echo "MySQL is Stopped."
    /etc/init.d/mysqld start
fi
```

脚本 6：

```
echo method6--------------------
[ `rpm -qa nc|wc -l` -lt 1 ] && yum install nc -y &>/dev/null
#<== 防止因 nc 没有安装而导致的错误。
if [ `nc -w 2  127.0.0.1 3306 &>/dev/null&&echo ok|grep ok|wc -l` -gt 0 ]
#<== 这个判断有点特别，即若 nc 执行成功，则输出和之后的过滤都没问题，最后转成数字，思路决定出路。
  then
    echo "MySQL is Running."
else
    echo "MySQL is Stopped."
    /etc/init.d/mysqld start
fi
```

脚本 7：

```
echo method7--------------------
if [ `ps -ef|grep -v grep|grep mysql|wc -l` -gt 0 ]  #<== 传统的过滤进程的方法，grep
                                                          -v grep 是排除此命令自身。
  then
    echo "MySQL is Running."
else
```

```
        echo "MySQL is Stopped."
        /etc/init.d/mysqld start
    fi
```

（3）监控 Nginx Web 服务异常

监控 Nginx Web 服务异常的方法和监控 MySQL 数据库一样，也是使用端口、进程或通过 wget/curl 访问来进行检测。

1）Nginx web 服务环境准备，如下：

```
    [root@oldboy ~]# wget -O /etc/yum.repos.d/epel.repo http://mirrors.aliyun.
com/repo/epel-6.repo
    [root@oldboy ~]# yum install nginx -y
    [root@oldboy ~]# /etc/init.d/nginx start
    正在启动 Nginx:                                          [ 确定 ]
    [root@oldboy scripts]# echo oldboy >/usr/share/nginx/html/index.html
    #<== 建立测试网页，内容为 oldboy。
    [root@oldboy scripts]# curl http://127.0.0.1          #<== 返回 oldboy。
    oldboy
```

2）通过命令行检测 Nginx 服务是否正常，同样只有命令行是正确的，才能确保它放到脚本里也是正确的。

首先采用端口监控的方式。在监控 Nginx 和监控数据库端口时，除了端口不同，其他的命令方法都是一模一样的，想要了解详细注释，可参见前文监控数据库的代码注释。在服务器本地监控端口的命令有 netstat、ss、lsof，如下：

```
    [root@oldboy ~]# netstat -lnt|grep -w 80|awk -F "[ :]+" '{print $5}'
    80
    [root@oldboy ~]# netstat -lntup|grep -w 80|wc -l
    1
    [root@oldboy ~]# netstat -lntup|grep mysql|wc -l
    1
    [root@oldboy ~]# ss -lntup|grep mysql|wc -l
    1
    [root@oldboy ~]# ss -lntup|grep -w 80|wc -l
    1
    [root@oldboy ~]# lsof -i tcp:80|wc -l
    3
```

在远端监控服务器本地端口的命令有 telnet、nmap、nc，如下：

```
    [root@oldboy ~]# nmap 127.0.0.1 -p 80|grep open|wc -l
    1
    [root@oldboy ~]# echo -e "\n"|telnet 127.0.0.1 80 2>/dev/null|grep
Connected|wc -l
    1
    [root@oldboy ~]# nc -w 2 127.0.0.1 80 &>/dev/null
```

```
[root@oldboy ~]# echo $?
0
```

对服务进程或进程数进行监控的方式（适合本地服务器）如下：

```
[root@oldboy ~]#  ps -ef|grep nginx|grep -v grep|wc -l
2
[root@oldboy ~]# ps -C nginx --no-header
    3147 ?        00:00:00 nginx
    3150 ?        00:00:00 nginx
[root@oldboy ~]# ps -C nginx --no-header|wc -l
2
```

以下是在客户端模拟用户访问的监控方式。先通过 wget 或 curl 命令进行测试。执行 wget 或 curl 命令之后，再看返回值（$?），为 0，则成功。

```
[root@oldboy ~]# wget --spider --timeout=10 --tries=2 http://127.0.0.1 &>/
dev/null
[root@oldboy ~]# echo $?
0
[root@oldboy ~]# wget -T 10 -q --spider http://127.0.0.1 &>/dev/null
[root@oldboy ~]# echo $?
0
[root@oldboy ~]# curl -s -o /dev/null http://127.0.0.1
[root@oldboy ~]# echo $?
0
```

以下是获取字符串的方式（判断获取的字符串是否和事先设定的相等）。

```
[root@oldboy ~]# curl http://127.0.0.1  #<== 根据指定的返回值判断服务是否正常。
oldboy
```

以下是根据 HTTP 响应 header 的结果进行判断（200、301、302 都表示正常）。

```
[root@oldboy ~]# curl -I -s -w "%{http_code}\n" -o /dev/null http://127.0.0.1
200
```

3）开发监控 Nginx Web 服务的脚本[⊖]，这里同样给出多个参考脚本。

脚本 1：

```
[root@oldboy scripts]# cat 7_5.sh
#!/bin/sh
echo http method1--------------------
if [ `netstat -lnt|grep 80|awk -F "[ :]+" '{print $5}'` -eq 80 ]
#<== 取端口数字表 ，有 Bug，不建议使用，如果非要取端口，则请改为字符串比较的语法。
  then
    echo "Nginx is Running."
else
```

⊖　由监控数据库的脚本改进而来。

```
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 2：

```
echo http method2-------------------
if [ "`netstat -lnt|grep 80|awk -F "[ :]+" '{print $5}'`" = "80" ]
#<== 字符串比较语法，取值太麻烦，不如过滤后转为行数再比较的方法好。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 3：

```
echo http method3-------------------
if [ `netstat -lntup|grep nginx|wc -l` -gt 0 ]
#<== 通过检查端口的命令，过滤进程名（比过滤端口好）转成数字后再比较，这是推荐的用法。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 4：

```
echo http method4-------------------
if [ `lsof -i tcp:80|wc -l` -gt 0 ]
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 5：

```
echo http method5-------------------
[ `rpm -qa nmap|wc -l` -lt 1 ] && yum install nmap -y &>/dev/null
if [ `nmap 127.0.0.1 -p 80 2>/dev/null|grep open|wc -l` -gt 0 ]
#<==nmap 检测方法，优势是适合从远端进行检查，推荐使用。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 6：

```
echo http method6-------------------
[ `rpm -qa nc|wc -l` -lt 1 ] && yum install nc -y &>/dev/null
if [ `nc -w 2  127.0.0.1 80 &>/dev/null&&echo ok|grep ok|wc -l` -gt 0 ]
#<==nc 方法，前面已讲解过。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 7：

```
echo http method7-------------------
if [ `ps -ef|grep -v grep|grep nginx|wc -l` -ge 1 ]  #<== 过滤进程方式，排除自身。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 8：

```
echo http method8-------------------
if [[ `curl -I -s -o /dev/null -w "%{http_code}\n" http://127.0.0.1` =~
[23]0[012] ]]
#<== 获取状态码，然后做正则数字匹配（目标是 200、301、302，但实际上多余了几个 201、202、
     300），这是 [[]] 的特殊匹配用法之一，前文也讲过了。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 9：

```
echo http method9-------------------
if [ `curl -I http://127.0.0.1 2>/dev/null|head -1|egrep "200|302|301"|wc
-l` -eq 1  ]
#<== 此方法通过扩展的 egrep 过滤正确的状态码，然后利用 wc 转成数字，此方法比较优秀，推荐读者使用。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

脚本 10：

```
echo http method10-------------------
if [ "`curl -s http://127.0.0.1`" = "oldboy" ]
#<== 根据访问网站 URL，将返回的结果和期待的值进行比较，这个方法略微麻烦，但是结果最准确，
     适用于数据库及更深层次的对网站集群后端各个应用的检测。
  then
    echo "Nginx is Running."
else
    echo "Nginx is Stopped."
    /etc/init.d/nginx start
fi
```

7.2.2 比较大小的经典拓展案例

范例 7-5：使用 if 条件语句比较两个整数的大小。使用传参方法时，需要对传参个数及传入的参数是否为整数进行判断。

说明：此题的解答方法在前文条件表达式的部分已详细讲过，讲 if 单分支时也给过解法，这里是扩展的脚本。

参考答案：

```
[root@oldboy scripts]# cat 7_5_1.sh
#!/bin/bash
a=$1                                #<== 将脚本命令行的第一个参数赋值给变量 a。
b=$2                                #<== 将脚本命令行的第二个参数赋值给变量 b。
#no.1 judge arg nums.
if [ $# -ne 2 ];then                #<== 判断传参个数。
    echo "USAGE:$0 arg1 arg2"       #<== 若传参个数不符合要求，则打印提示后退出脚本。
    exit 2
fi

#no.2 judge if int
expr $a + 1 &>/dev/null             #<== 判断变量 a 是否为整数。
RETVAL1=$?                          #<== 获取 expr 命令的返回值并赋值给 RETVAL1。
expr $b + 1 &>/dev/null             #<== 判断变量 b 是否为整数。
RETVAL2=$?                          #<== 获取 expr 命令的返回值并赋值给 RETVAL2。
if [ $RETVAL1 -ne 0 -a $RETVAL2 -ne 0 ];then
#<== 执行判断，如果返回值不都为 0，说明至少有一个变量不是整数，则打印提示，退出脚本。
    echo "please input two int again"
    exit 3
fi
#no.3 compart two num.
#<== 经过参数个数判断及整数判断后，就可以放心地进行比较了。
if [ $a -lt $b ];then
    echo "$a<$b"
```

```
elif [ $a -eq $b ];then
    echo "$a=$b"
else
    echo "$a>$b"
fi
```

对于新手，老男孩不推荐使用 if 条件语句的复杂判断写法（逻辑不够清晰）。

```
#!/bin/sh
read -p "Pls input two num:" a b
#no2 udge a and b if is int
expr $a + 0 &>/dev/null
RETVAL1=$?
expr $b + 0 &>/dev/null
RETVAL2=$?
if [ -z "$a" ] || [ -z "$b" ]
  then
    echo "Pls input two num agagin."
    exit 1
elif test $RETVAL1 -ne 0 -o $RETVAL2 -ne 0
  then
    echo "Pls input two "num" again."
    exit 2
elif [ $a -lt $b ]
  then
    echo "$a < $b"
elif [ $a -eq $b ]
  then
    echo "$a = $b"
else
    echo "$a > $b"
fi
exit 0
```

7.2.3　判断字符串是否为数字的多种思路

范例 7-6：判断字符串是否为数字的多种思路。

参考答案 1：使用 sed 加正则表达式。

判断思路：删除一个字符串中的所有数字，看字符串的长度是否为 0，如果不为 0，则说明不是整数。

```
    [root@oldboy scripts]# [ -n "`echo oldboy123|sed 's/[0-9]//g'`" ] && echo
char || echo int
    char
    [root@oldboy scripts]# [ -n "`echo 123|sed 's/[0-9]//g'`" ] && echo char ||
echo int
    int
    [root@oldboy scripts]# [ -z "`echo 123|sed 's/[0-9]//g'`" ] && echo int ||
echo char
```

```
    int
    [root@oldboy scripts]# [ -z "`echo oldgirl123|sed 's/[0-9]//g'`" ] && echo
int || echo char
    char
```

除了使用 sed 的替换外，还可以使用变量的子串替换等方法，如下：

```
    [root@oldboy scripts]# num=oldboy521
    [root@oldboy scripts]# [ -z "`echo "${num//[0-9]/}"`" ] && echo int || echo char
    char
    [root@oldboy scripts]# num=521
    [root@oldboy scripts]# [ -z "`echo "${num//[0-9]/}"`" ] && echo int || echo char
    int
```

参考答案 2：变量的子串替换加正则表达式（特殊判断思路）。

判断思路：如果 num 的长度不为 0，并且把 num 中的非数字部分删除，然后再看结果是不是等于 num 本身，如果两者都成立，则 num 就是数字。

```
    -n "$num"                    #<== 如果 num 长度不为 0，则表达式成立。
    "$num" = "${num//[^0-9]/}" #<== 把 num 中的非数字部分删除（即把字母删除），然后看结果
                                    是不是等于 num 本身，如果两者都成立，就是数字。
    # 例：对于非数字 oldboy123，删除 oldboy，就是 123 了，肯定不等于 oldboy123。
    # 例：如果是数字 123，删除非数字的部分，结果还是 123，肯定等于 123。
```

完整表达式如下：

```
    [root@oldboy scripts]# num=521
    [root@oldboy scripts]# [ -n "$num" -a "$num" = "${num//[^0-9]/}" ] && echo
"it is num"
    it is num
    [root@oldboy scripts]# num=oldboy521
    [root@oldboy scripts]# [ -n "$num" -a "$num" = "${num//[^0-9]/}" ] && echo
"it is num"
```

参考答案 3：通过 expr 计算判断（前文已用过多次了）。

```
    [root@oldboy scripts]# expr oldboy + 1 &>/dev/null    #<== 把字符串或变量和整数相加
                                                              看执行是否成功。
    [root@oldboy scripts]# echo $?
    2
    [root@oldboy scripts]# expr 123 + 1 &>/dev/null
    [root@oldboy scripts]# echo $?
    0
    [root@oldboy scripts]# expr 0 + 0 &>/dev/null
    [root@oldboy scripts]# echo $?
    1
```

参考答案 4：利用“=~”符号判断

读者可以使用 man bash 查看相关帮助，它在 [[]] 里也有过介绍。

```
[root@oldboy scripts]# [[ oldboy123 =~ ^[0-9]+$ ]] && echo int || echo char
char
[root@oldboy scripts]# [[ 123 =~ ^[0-9]+$ ]] && echo int || echo char
int
```

参考答案 5：利用 bc 判断字符串是否为整数（仅为参考方法，不推荐，因为有 Bug）。

```
[root@oldboy scripts]# echo oldboy|bc
0
[root@oldboy scripts]# echo oldboy123|bc
0
[root@oldboy scripts]# echo 123|bc      #<== 返回结果为本身的就是整数。
123
[root@oldboy scripts]# echo 123oldboy|bc
(standard_in) 1: syntax error
```

7.2.4　判断字符串长度是否为 0 的多种思路

范例 7-7：实现判断字符串长度是不是为 0 的多种方法。

1）使用字符串条件表达式 -z 和 -n 的语法如下：

```
[root@oldboy scripts]# [ -z "oldboy" ] && echo 1 || echo 0
0
[root@oldboy scripts]# [ -n "oldboy" ] && echo 1 || echo 0
1
```

2）使用变量子串判断的语法如下：

```
[root@oldboy scripts]# char=oldboy
[root@oldboy scripts]# [ ${#char} -eq 0 ] && echo 1 || echo 0
```

3）使用 expr length 函数判断的语法如下：

```
[root@oldboy scripts]# [ `expr length "oldboy"` -eq 0 ] && echo 1 || echo 0
0
```

4）使用 wc 的 -L 参数统计判断的语法如下：

```
[root@oldboy scripts]# [ ` echo oldboy|wc -L ` -eq 0 ] && echo 1 || echo 0
0
```

5）使用 awk length 函数判断的语法如下：

```
[root@oldboy scripts]# [ `echo oldboy|awk '{print length}'` -eq 0 ] &&
echo 1 || echo 0
0
```

7.2.5　更多的生产场景实战案例

范例 7-8：监控 memcached 服务是否正常，模拟用户（Web 客户端）检测。

此题为企业笔试题，需要读者了解并可以搭建 memcached 服务（memcached 是运维场景中常用的内存缓存软件），且可以用 nc 或 telnet 命令访问 memcached 服务，加上用 set/get 来模拟检测。

参考答案：

```
#!/bin/sh
printf "del key\r\n"|nc 127.0.0.1 11211 &>/dev/null
#<== 使用 nc 连接 memcached，并执行 del key 清理 key 这个名称的键值。
printf "set key  0 0 10 \r\noldboy1234\r\n"|nc 127.0.0.1 11211 &>/dev/null
#<== 插入键值 key-->oldboy1234 到 memcached。
McValues=`printf "get key\r\n"|nc 127.0.0.1 11211|grep oldboy1234|wc -l`
#<== 查看键值并赋值给变量。
if [ $McValues -eq 1 ]   #<== 如果键值数为 1，则表示插入成功，否则表示失败。
  then
    echo "memcached status is ok"
else
    echo "memcached status is bad"
fi
```

请读者自行思考如何监控 MC 服务、命中（hit）率、响应时间这三个指标。

范例 7-9：开发 rsync 服务的启动脚本。

rsync 是运维场景中最常用的数据同步软件，本例就是要完成一个类似系统的启动 rsync 服务的方法，即使用 /etc/init.d/rsyncd {start|stop|restart} 即可启动和停止 rsync 服务，这里是用 if 条件语句来实现相应效果的，其实通过 case 语句来实现效果最佳，不过由于还没讲解到 case 语句，因此这里主要练习 if 语句。

1）分析问题。要想开发出 rsync 服务的启动脚本，就需要熟悉 rsync 服务的配置，以及 rsync 服务是如何启动，以及如何停止的。

2）要实现 /etc/init.d/rsyncd {start|stop|restart} 的操作语法，就需要用到脚本的传参。根据传入的参数，进行判断，然后执行对应的启动和停止命令。

实现过程如下。

第 1 步，rsync 服务的简单准备，代码如下：

```
[root@oldboy scripts]# rpm -qa rsync              #<== 查看软件包是否安装。
rsync-3.0.6-12.el6.x86_64
[root@oldboy scripts]# touch /etc/rsyncd.conf #<== 为了简便起见，这里创建了一个空文件，实际工作中是有配置内容的。
[root@oldboy scripts]# rsync --daemon             #<== 启动 rsync 服务命令。
[root@oldboy scripts]# netstat -lnt|grep 873      #<== 过滤 873 端口，即 rsync 服务端口。
tcp        0      0 0.0.0.0:873                 0.0.0.0:*                   LISTEN
tcp        0      0 :::873                      :::*                        LISTEN
```

第 2 步，实现 rsync 服务的启动和停止方法，以及端口检测方法。

停止方法：

```
[root@oldboy scripts]# pkill rsync
[root@oldboy scripts]# netstat -lntup|grep 873
```

启动方法：

```
[root@oldboy scripts]# rsync --daemon
[root@oldboy scripts]# netstat -lnt|grep 873
tcp        0      0 0.0.0.0:873                 0.0.0.0:*                   LISTEN
tcp        0      0 :::873                      :::*                        LISTEN
```

第 3 步，判断 rsync 服务是否启动的方法。

常规方法有检测端口以及进程是否存在，还可以当服务启动时，创建一个锁文件（/var/lock/subsys/），而当服务停止时，就删除这个锁文件，这样就可以通过判断这个文件有无，来确定服务是否是启动状态，这是一些系统脚本常用的手法。

第 4 步，开发 rsync 服务的启动脚本。

```
[root@oldboy scripts]# cat /etc/init.d/rsyncd
#!/bin/bash
if [ $# -ne 1 ]           #<== 如果传参的个数不等于 1，
  then                    #<== 则执行下面的命令打印语法，提示用户，并退出。
    echo $"usage:$0 {start|stop|restart}"
    exit 1
fi
if [ "$1" = "start" ]     #<== 如果参数的值为 start，则执行 then 下面的语句。
  then
    rsync --daemon        #<== 启动 rsync 服务。
    sleep 2               #<== 休息 2 秒，这点很重要，停止及启动服务后，建议先休息 1 ~ 2
                              秒，再判断。
    if [ `netstat -lntup|grep rsync|wc -l` -ge 1 ]
#<== 如果过滤 rsync 字符串的行数大于 1，则表示 rsync 服务启动了。
      then
        echo "rsyncd is started."
        exit 0
    fi
elif [ "$1" = "stop" ]    #<== 如果参数的值为 stop，则执行 then 下面的语句。
  then
    killall rsync &>/dev/null  #<== 停止服务，这里停止服务的方法有多种，读者可以自行选择。
    sleep 2               #<== 休息 2 秒，再判断，防止没有停止完服务，导致判断不准确。
    if [ `netstat -lntup|grep rsync|wc -l` -eq 0 ]
#<== 如果过滤 rsync 字符串的行数为 0，则表示 rsync 服务停止了。
      then
        echo "rsyncd is stopped."
        exit 0
    fi
elif [ "$1" = "restart" ]         #<== 如果参数的值为 restart，则执行 then 下面的语句。
  then
    killall rsync                 #<== 关闭服务。
    sleep 1
    killpro=`netstat -lntup|grep rsync|wc -l`
```

```
    rsync --daemon                                  #<== 启动服务。
    sleep 1
    startpro=`netstat -lntup|grep rsync|wc -l`
    if [ $killpro -eq 0 -a $startpro -ge 1 ]   #<== 判断，给出提示。
      then
        echo "rsyncd is restarted."
        exit 0
    fi
else                                    #<== 若没有按照帮助提示传参，则给出提示并退出脚本。
    echo $"usage:$0 {start|stop|restart}"
    exit 1
fi
```

执行结果如下：

```
[root@oldboy scripts]# /etc/init.d/rsyncd start
rsyncd is started.
[root@oldboy scripts]# netstat -lnt|grep 873
tcp        0      0 0.0.0.0:873                 0.0.0.0:*                   LISTEN
tcp        0      0 :::873                      :::*                        LISTEN
[root@oldboy scripts]# /etc/init.d/rsyncd stop
rsyncd is stopped.
[root@oldboy scripts]# netstat -lnt|grep 873
[root@oldboy scripts]# /etc/init.d/rsyncd start
rsyncd is started.
[root@oldboy scripts]# netstat -lntup|grep 873
tcp        0      0 0.0.0.0:873                 0.0.0.0:*       LISTEN      5050/rsync
tcp        0      0 :::873                      :::*            LISTEN      5050/rsync
[root@oldboy scripts]# /etc/init.d/rsyncd restart
[root@oldboy scripts]# netstat -lntup|grep 873
tcp        0      0 0.0.0.0:873                 0.0.0.0:*       LISTEN      5066/rsync
#<== 进程号发生变化，说明进程确实重启了。
tcp        0      0 :::873                      :::*            LISTEN      5066/rsync
```

此脚本可以实现基本的启动和停止功能，但是，离专业和规范还有一定的距离，在后面学习完更多知识后可再来改写。

范例 7-10：使用上述开发好的 rsync 服务启动脚本，实现通过 chkconfig 来管理开机自启动。

chkconfig 命令是系统用来管理脚本开机自启动的命令，但是需要脚本支持 chkconfig 管理才行，具体的方法是在脚本的开头解释器之后加入如下两行内容：

```
# chkconfig: 2345 20 80
# description: Saves and restores system entropy pool
```

说明：

- 两行内容的开头都要有 # 号。

- 第一行是说需要 chkconfig 管理，2345 是 Linux 运行级别，表示该脚本默认在 2345 级别为启动状态，20 是脚本的开始启动顺序，80 是脚本的停止顺序，这两个数字都是不超过 99 的数字，一般情况下，可以根据服务的启动需求来选择，应用服务一般要靠后启动为佳，越早停止越好。

改好后的 rsync 的启动脚本如下：

```
[root@oldboy scripts]# cat /etc/init.d/rsyncd
#!/bin/bash
# chkconfig: 2345 20 80
# description: Rsyncd Startup scripts by oldboy.
if [ $# -ne 1 ]
  then
    echo $"usage:$0 {start|stop|restart}"
    exit 1
fi
if [ "$1" = "start" ]
  then
    rsync --daemon
    sleep 2
    if [ `netstat -lntup|grep rsync|wc -l` -ge 1 ]
      then
        echo "rsyncd is started."
        exit 0
    fi
elif [ "$1" = "stop" ]
  then
    killall rsync &>/dev/null
    sleep 2
    if [ `netstat -lntup|grep rsync|wc -l` -eq 0 ]
      then
        echo "rsyncd is stopped."
        exit 0
    fi
elif [ "$1" = "restart" ]
  then
    killall rsync
    sleep 1
    killpro=`netstat -lntup|grep rsync|wc -l`
    rsync --daemon
    sleep 1
    startpro=`netstat -lntup|grep rsync|wc -l`
    if [ $killpro -eq 0 -a $startpro -ge 1 ]
      then
        echo "rsyncd is restarted."
        exit 0
    fi
else
```

```
    echo $"usage:$0 {start|stop|restart}"
    exit 1
fi
```

设置开机自启动的过程如下。

设置前检查 rsyncd 的开机自启动情况：

```
[root@oldboy scripts]# chkconfig --list rsyncd  #<== 因为没有设置，所以没有输出。
```

添加脚本开机自启动并检查：

```
[root@oldboy scripts]# chkconfig --add rsyncd  #<== 注意，rsyncd 脚本一定要放在
                                                    /etc/init.d 下。
[root@oldboy scripts]# chkconfig --list rsyncd #<== 检查自启动结果，2345 级别默
                                                    认为启动状态。
rsyncd          0：关闭  1：关闭  2：启用  3：启用  4：启用  5：启用  6：关闭
```

特别说明：可访问如下地址或手机扫二维码查看第 7 章的核心脚本代码

http://oldboy.blog.51cto.com/2561410/1855640

第8章

Shell 函数的知识与实践

8.1 Shell 函数的概念与作用介绍

在讲解 Shell 函数之前，先来回顾 Linux 系统中别名的作用。

```
[root@oldboy ~]# alias ssh='/etc/init.d/sshd'
[root@oldboy ~]# ssh restart #<== 执行 ssh 就相当于执行了 /etc/init.d/sshd，最直
                                  接的好处就是简化了输入。
停止 sshd:                                                    [ 确定 ]
正在启动 sshd:                                                [ 确定 ]
```

函数也有类似于别名的作用，例如可简化程序的代码量，让程序更易读、易改、易用。

简单地说，函数的作用就是将程序里多次被调用的相同代码组合起来（函数体），并为其取一个名字（即函数名），其他所有想重复调用这部分代码的地方都只需要调用**这个名字**就可以了。当需要修改这部分重复代码时，只需要改变函数体内的一份代码即可实现对所有调用的修改，也可以把函数独立地写到文件里，当需要调用函数时，再加载进来使用。

使用 Shell 函数的优势整理如下：

- 把相同的程序段定义成函数，可以减少整个程序的代码量，提升开发效率。
- 增加程序的可读性、易读性，提升管理效率。

❑ 可以实现程序功能模块化，使得程序具备通用性（可移植性）。

对于 Shell 来说，Linux 系统里的近 2000 个命令可以说都是 Shell 的函数，所以，Shell 的函数也是很多的，这一点需要读者注意。

8.2 Shell 函数的语法

下面是 Shell 函数的常见语法格式。

其标准写法为：

```
function 函数名 () {          #<== 作者推荐的书写函数的方法（带括号）
    指令 ...
    return n
}
```

简化写法 1：

```
function 函数名 {             #<== 不推荐读者使用此方法（无括号）
    指令 ...
    return n
}
```

在 Shell 函数的语法中，当有 function 时，函数名后面的小括号“( )”部分可以省略不写。

简化写法 2：

```
函数名 () {                   #<== 不用 function 的方法
    指令 ...
    return n
}
```

在 Shell 函数的语法中，function 表示声明一个函数，这部分可以省略不写。

8.3 Shell 函数的执行

Shell 的函数分为最基本的函数和可以传参的函数两种，其执行方式分别说明如下。

1）执行不带参数的函数时，直接输入函数名即可（注意不带小括号）。格式如下：

```
函数名
```

有关执行函数的重要说明：

❑ 执行 Shell 函数时，函数名前的 function 和函数后的小括号都不要带。

❑ 函数的定义必须在要执行的程序前面定义或加载。

- Shell 执行系统中各种程序的执行顺序为：系统别名→函数→系统命令→可执行文件。
- 函数执行时，会和调用它的脚本共用变量，也可以为函数设定局部变量及特殊位置参数。
- 在 Shell 函数里面，return 命令的功能与 exit 类似，return 的作用是退出函数，而 exit 是退出脚本文件。
- return 语句会返回一个退出值（即返回值）给调用函数的当前程序，而 exit 会返回一个退出值（即返回值）给执行程序的当前 Shell。
- 如果将函数存放在独立的文件中，被脚本加载使用时，需要使用 source 或“.”来加载。
- 在函数内一般使用 local 定义局部变量，这些变量离开函数后就会消失。

2）带参数的函数执行方法，格式如下：

```
函数名 参数1 参数2
```

函数后接参数的说明：

- Shell 的位置参数（$1、$2…、$#、$*、$? 及 $@）都可以作为函数的参数来使用。
- 此时父脚本的参数临时地被函数参数所掩盖或隐藏。
- $0 比较特殊，它仍然是父脚本的名称。
- 当函数执行完成时，原来的命令行脚本的参数即可恢复。
- 函数的参数变量是在函数体里面定义的。

8.4　Shell 函数的基础实践

范例 8-1：开发脚本建立两个简单函数并调用执行。

```
[root@oldboy scripts]# cat 8_1.sh
#!/bin/bash
#定义两个函数，名字为oldboy和oldgirl。
oldboy(){
    echo "I am oldboy."
}
function oldgirl(){
    echo "I am oldgirl."
}
oldboy              #<== 在一个脚本里执行函数名调用函数。
oldgirl             #<== 在一个脚本里执行函数名调用函数。
```

务必要先定义函数然后再执行函数，否则会报错，示例如下：

```
[root@oldboy scripts]# cat 8_2.sh
#!/bin/bash
oldgirl
function oldgirl(){
    echo "I am oldgirl."
}
oldgirl
[root@oldboy scripts]# sh 8_2.sh
8_2.sh: line 2: oldgirl: command not found      #<== 提示：第二行因为 oldgirl 函数不存在，所以就把 oldgirl 当作命令了，但是系统又没有 oldgirl 命令。
I am oldgirl.
```

范例 8-2：分离函数体和执行函数的脚本文件（更规范的方法）。

首先建立函数库脚本（默认不会执行函数）。

使用 cat 命令追加多行文本，以将函数代码追加到系统的函数文件中，即 /etc/init.d/functions，命令如下：

```
cat >>/etc/init.d/functions<<- EOF      #<== here 文档的另一种写法，此写法允许后面的 EOF 可以使用 Tab 键，而不顶格。

oldboy(){
    echo "I am oldboy."
}
EOF                                     #<== 就是这里的 EOF 可以使用 Tab 键，而不顶格，但不能使用空格。
```

追加的函数代码如下：

```
[root@oldboy scripts]# tail -3 /etc/init.d/functions
oldboy(){
    echo "I am oldboy."
}
```

然后开发执行脚本以调用上述函数。

```
[root@oldboy scripts]# cat 8_3.sh
#!/bin/bash
# 加载函数所在的文件（functions 是 Linux 系统内置的脚本函数库）
[ -f /etc/init.d/functions ] && . /etc/init.d/functions || exit 1
# 提示：可以用 source 或 "."（点号）来加载脚本 functions 中的命令或变量参数等。
# 调用函数
oldboy
```

执行结果如下：

```
[root@oldboy scripts]# sh 8_3.sh
I am oldboy.
```

范例 8-3：写出带参数的 Shell 函数示例。

这里通过建立函数 oldgirl 来做示例讲解，代码截图如图 8-1 所示。

范例 8-4：将函数的传参转换成脚本文件命令行传参。

参考答案见图 8-2。

```
[root@oldboy scripts]# tail -3 /etc/init.d/functions
oldgirl() {
    echo "I am oldgirl.you are $1"
}
[root@oldboy scripts]# cat 8_4.sh
#!/bin/bash
[ -f /etc/init.d/functions ] && . /etc/init.d/functions||exit 1
oldgirl xiaoting #<==将 xiaoting 作为参数传给 oldgirl 函数。
[root@oldboy scripts]# sh 8_4.sh
I am oldgirl.you are xiaoting #<==由/etc/init.d/functions 里的 oldgirl 函数将 xiaoting 传给$1，
                                因此，输出了 xiaoting。
```

图 8-1　建立函数 oldgirl 的代码段

```
[root@oldboy scripts]# tail -3 /etc/init.d/functions
oldgirl() {
    echo "I am oldgirl.you are $1" #<==这里的$1 为脚本的位置参数。
}
[root@oldboy scripts]# cat 8_4.sh
#!/bin/bash
[ -f /etc/init.d/functions ] && . /etc/init.d/functions||exit 1
oldgirl $1 #<==这个$1 本身是脚本的位置参数，但是$1 占据的位置为函数的参数位置，现在他们重合了。
[root@oldboy scripts]# sh 8_4.sh bingbing #<==传入 bingbing。
I am oldgirl.you are bingbing
```

图 8-2　将函数的传参转换成脚本文件命令行传参的示例图

8.5　利用 Shell 函数开发企业级 URL 检测脚本

范例 8-5：将函数的传参转换成脚本文件命令行传参，判断任意指定的 URL 是否存

在异常。

此题结合了前面章节的检查网站 URL 的案例，并涉及当下的函数传参及脚本传参。

程序设计思路及实现：

1）实现脚本传参，检查 Web URL 是否正常。

```
[root@oldboy scripts]# cat 8_5.sh
#!/bin/sh
# 判断传参个数是否为 1 个。
if [ $# -ne 1 ]
  then
    echo $"usage:$0 url"
    exit 1
fi
# 利用 wget 进行访问测试，wget 的相关参数在前面章节已经讲解过了。
wget --spider -q -o /dev/null --tries=1 -T 5 $1  #<==-T 指定超时时间，这里的 $1
                                                    为脚本的参数。
if [ $? -eq 0 ]
  then
    echo "$1 is yes."
else
    echo "$1 is no."
fi
```

2）将上述检测的功能写成函数，并将函数传参转换成脚本命令行传参，判断任意指定的 URL 是否存在异常。

```
[root@oldboy scripts]# cat 8_5_1.sh
#!/bin/sh
function usage(){      #<== 帮助函数。
    echo $"usage:$0 url"
    exit 1
}

function check_url(){ #<== 检测 URL 函数。
    wget --spider -q -o /dev/null --tries=1 -T 5 $1  #<== 这里的 $1 就是函数传参。
    if [ $? -eq 0 ]
      then
        echo "$1 is yes."
    else
        echo "$1 is no."
    fi
}
function main(){          #<== 主函数。
    if [ $# -ne 1 ]       #<== 如果传入的是多个参数，则打印帮助函数，提示用户。
      then
        usage
```

```
    fi
    check_url $1          #<== 接收函数的传参，即把下文 main 结尾的 $* 传到这里。
}
main $*                   #<== 这里的 $* 就是把命令行接收的所有参数作为函数参数传给函
                              数内部，是一种常用手法。
```

提示：学习了函数以后应尽量将脚本功能模块化，每个模块实现一个功能，并且让脚本可以通用。

执行结果如下：

```
[root@oldboy scripts]# sh 8_5_1.sh
usage:8_5_1.sh url
[root@oldboy scripts]# sh 8_5_1.sh  www.oldboyedu.com
www.oldboyedu.com is yes.
[root@oldboy scripts]# sh 8_5_1.sh  www.oldgirl123.com
www.oldgirl123.com is no.
```

范例 8-6：将函数的传参转换成脚本文件命令行传参，判断任意指定的 URL 是否存在异常，并以更专业的输出显示，效果如图 8-3 所示。

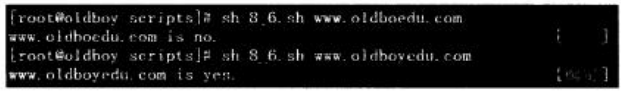


图 8-3　将函数的传参转换成脚本文件命令行传参专业显示效果图

参考答案：

```
[root@oldboy scripts]# cat 8_6.sh
#!/bin/sh
. /etc/init.d/functions  #<== 引入系统函数库。
function usage(){
    echo $"usage:$0 url"
    exit 1
}
function check_url(){
    wget --spider -q -o /dev/null --tries=1 -T 5 $1
    if [ $? -eq 0 ]
      then
        action "$1 is yes." /bin/true      #<== 这里的 action 就是在脚本开头引入系
                                               统函数库后调用的。
    else
        action "$1 is no." /bin/false
    fi
}
```

```
function main(){
    if [ $# -ne 1 ]
      then
        usage
    fi
    check_url $1
}
main $*
```

执行效果如下：

```
[root@oldboy scripts]# sh 8_6.sh www.oldboedu.com  #<== 输入错误地址。
www.oldboedu.com is no.                                    [失败]
[root@oldboy scripts]# sh 8_6.sh www.oldboyedu.com
www.oldboyedu.com is yes.                                  [确定]
```

8.6 利用 Shell 函数开发一键优化系统脚本

范例 8-7：编写 Shell 开发 Linux 系统一键优化脚本。

在开始编写 Shell 脚本之前，请大家回忆一下，如何优化 Linux 系统？

编程类似于拍电视剧，要想编写一个好的程序，必须要有剧本（解决什么需求），有了剧本，还需要有演员，然后要切换多场景分段拍戏，最后剪辑合成一部电视剧。下面就按此步骤来实现。

1）先寻找原始剧本，即思考如何优化 Linux 系统，并写出来。

这里仅给出一些基础的优化项，目的是为读者提供完成企业级一键优化系统脚本的思路、方法和实现。

- 安装系统时精简安装包（最小化安装）。
- 配置国内的高速 yum 源。
- 禁用开机不需要启动的服务。
- 优化系统内核参数 /etc/sysctl.conf。
- 增加系统文件描述符、堆栈等配置。
- 禁止 root 远程登录，修改 SSH 端口为特殊端口，禁止 DNS 及空密码。
- 有外网 IP 的机器要开启、配置防火墙，仅对外开启需要提供服务的端口、配置或关闭 SELinux。
- 清除无用的默认系统账户或组（非必须）(添加运维成员的用户)。
- 锁定敏感文件，如 /etc/passwd（非必须）。
- 配置服务器和互联网时间同步。
- 初始化用户，并配置 sudo 对普通用户权限的控制。
- 修改系统字符集。

❑ 补装系统软件及升级系统到最新。

更多优化可参见《跟老男孩学 Linux 运维：Web 集群实战》一书。

2）将剧本变成可以拍戏的信息。

```
    #0. 更改 yum 源
    mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.
backup &&\
    wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/
Centos-6.repo

    #1. 关闭 SELinux
    sed -i 's/SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config
    grep SELINUX=disabled /etc/selinux/config
    setenforce 0
    getenforce

    #2. 关闭 iptables
    /etc/init.d/iptables stop  #<== 加载 2 次，确保关闭。
    /etc/init.d/iptables stop
    chkconfig iptables off

    #3. 精简开机自启动服务
    chkconfig|awk '{print "chkconfig",$1,"off"}'|bash
    chkconfig|egrep "crond|sshd|network|rsyslog|sysstat"|awk '{print
"chkconfig",$1,"on"}'|bash
    export LANG=en
    chkconfig --list|grep 3:on

    #4. 提权 oldboy 可以 sudo
    useradd oldboy
    echo 123456|passwd --stdin oldboy
    \cp /etc/sudoers /etc/sudoers.ori
    echo "oldboy  ALL=(ALL) NOPASSWD: ALL " >>/etc/sudoers
    tail -1 /etc/sudoers
    visudo -c

    #5. 中文字符集
    cp /etc/sysconfig/i18n /etc/sysconfig/i18n.ori
    echo 'LANG="zh_CN.UTF-8"'  >/etc/sysconfig/i18n
    source /etc/sysconfig/i18n
    echo $LANG

    #6. 时间同步
    echo '#time sync by oldboy at 2010-2-1' >>/var/spool/cron/root
    echo '*/5 * * * * /usr/sbin/ntpdate time.nist.gov >/dev/null 2>&1' >>/var/
spool/cron/root
    crontab -l
```

```
#7.命令行安全(此行注释了，表示可不设置)
#echo 'export TMOUT=300' >>/etc/profile
#echo 'export HISTSIZE=5' >>/etc/profile
#echo 'export HISTFILESIZE=5' >>/etc/profile
#tail -3 /etc/profile
#. /etc/profile

#8.加大文件描述
echo '*               -       nofile          65535 ' >>/etc/security/limits.conf
tail -1 /etc/security/limits.conf

#9.内核优化
cat >>/etc/sysctl.conf<<EOF
net.ipv4.tcp_fin_timeout = 2
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_keepalive_time = 600
net.ipv4.ip_local_port_range = 4000    65000
net.ipv4.tcp_max_syn_backlog = 16384
net.ipv4.tcp_max_tw_buckets = 36000
net.ipv4.route.gc_timeout = 100
net.ipv4.tcp_syn_retries = 1
net.ipv4.tcp_synack_retries = 1
net.core.somaxconn = 16384
net.core.netdev_max_backlog = 16384
net.ipv4.tcp_max_orphans = 16384
#以下参数是对iptables防火墙的优化，防火墙不开，会有提示，可以忽略不理。
net.nf_conntrack_max = 25000000
net.netfilter.nf_conntrack_max = 25000000
net.netfilter.nf_conntrack_tcp_timeout_established = 180
net.netfilter.nf_conntrack_tcp_timeout_time_wait = 120
net.netfilter.nf_conntrack_tcp_timeout_close_wait = 60
net.netfilter.nf_conntrack_tcp_timeout_fin_wait = 120
EOF
Sysctl -p
yum update -y
yum install lrzsz nmap tree dos2unix nc -y
```

这些优化技巧是和脚本无关的，需要读者事先掌握，还记得第 1 章我们讲解的如何学好 Shell 编程么？如果只会 Shell 语法，不会网络服务和 Linux 的命令，就无法写出这么多剧本来，更谈不上编程了。

3）选演员分段拍戏，将每个优化模块写成函数。

```
[root@oldboy scripts]# cat sys_opt.sh
#!/bin/bash
# author:oldboy
```

```
    # qq:31333741
    #set env
    export PATH=$PATH:/bin:/sbin:/usr/sbin
    # Require root to run this script.
    if [ "$UID" != "0" ]; then
        echo "Please run this script by root."
        exit 1
    fi

    #define cmd var
    SERVICE=`which service`
    CHKCONFIG=`which chkconfig`

    function mod_yum(){
        #modify yum path
        if [ -e /etc/yum.repos.d/CentOS-Base.repo ]
          then
            mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.
repo.backup&&\
            wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/
Centos-6.repo
        fi
    }

    function close_selinux(){
        #1.close selinux
        sed -i 's/SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config
        #grep SELINUX=disabled /etc/selinux/config
        setenforce 0 &>/dev/null
        #getenforce
    }

    function close_iptables(){
        #2.close iptables
        /etc/init.d/iptables stop
        /etc/init.d/iptables stop
        chkconfig iptables off
    }

    function least_service(){
        #3.least service startup
        chkconfig|awk '{print "chkconfig",$1,"off"}'|bash
        chkconfig|egrep "crond|sshd|network|rsyslog|sysstat"|awk '{print
"chkconfig",$1,"on"}'|bash
        #export LANG=en
        #chkconfig --list|grep 3:on
    }
    function adduser(){
        #4.add oldboy and sudo
```

```
    if [ `grep -w oldboy /etc/passwd|wc -l` -lt 1 ]
      then
        useradd oldboy
        echo 123456|passwd --stdin oldboy
        \cp /etc/sudoers /etc/sudoers.ori
        echo "oldboy  ALL=(ALL) NOPASSWD: ALL " >>/etc/sudoers
        tail -1 /etc/sudoers
        visudo -c &>/dev/null
    fi
}

function charset(){
    #5.charset config
    cp /etc/sysconfig/i18n /etc/sysconfig/i18n.ori
    echo 'LANG="zh_CN.UTF-8"'  >/etc/sysconfig/i18n
    source /etc/sysconfig/i18n
    #echo $LANG
}

function time_sync(){
    #6.time sync.
    cron=/var/spool/cron/root
    if [ `grep -w "ntpdate" $cron|wc -l` -lt 1 ]
      then
        echo '#time sync by oldboy at 2010-2-1' >>$cron
        echo '*/5 * * * * /usr/sbin/ntpdate time.nist.gov >/dev/null 2>&1' >>$cron
        crontab -l
    fi
}

function com_line_set(){
    #7.command set.
    if [ `egrep "TMOUT|HISTSIZE|ISTFILESIZE" /etc/profile|wc -l` -lt 3  ]
      then
        echo 'export TMOUT=300' >>/etc/profile
        echo 'export HISTSIZE=5' >>/etc/profile
        echo 'export HISTFILESIZE=5' >>/etc/profile
        . /etc/profile
    fi
}

function open_file_set(){
    #8.increase open file.
    if [ `grep 65535 /etc/security/limits.conf|wc -l` -lt 1 ]
      then
        echo '*          -      nofile       65535 ' >>/etc/security/limits.conf
        tail -1 /etc/security/limits.conf
    fi
}
function set_kernel(){
```

```
        #9.kernel set.
        if [ `grep kernel_flag /etc/sysctl.conf|wc -l` -lt 1 ]
          then
            cat >>/etc/sysctl.conf<<EOF
            #kernel_flag
            net.ipv4.tcp_fin_timeout = 2
            net.ipv4.tcp_tw_reuse = 1
            net.ipv4.tcp_tw_recycle = 1
            net.ipv4.tcp_syncookies = 1
            net.ipv4.tcp_keepalive_time = 600
            net.ipv4.ip_local_port_range = 4000    65000
            net.ipv4.tcp_max_syn_backlog = 16384
            net.ipv4.tcp_max_tw_buckets = 36000
            net.ipv4.route.gc_timeout = 100
            net.ipv4.tcp_syn_retries = 1
            net.ipv4.tcp_synack_retries = 1
            net.core.somaxconn = 16384
            net.core.netdev_max_backlog = 16384
            net.ipv4.tcp_max_orphans = 16384
            net.nf_conntrack_max = 25000000
            net.netfilter.nf_conntrack_max = 25000000
            net.netfilter.nf_conntrack_tcp_timeout_established = 180
            net.netfilter.nf_conntrack_tcp_timeout_time_wait = 120
            net.netfilter.nf_conntrack_tcp_timeout_close_wait = 60
            net.netfilter.nf_conntrack_tcp_timeout_fin_wait = 120
    EOF
            sysctl -p
            fi
    }
    function init_ssh(){
        \cp /etc/ssh/sshd_config /etc/ssh/sshd_config.`date +"%Y-%m-%d_%H-%M-%S"`
        # sed -i 's%#Port 22%Port 52113%' /etc/ssh/sshd_config
        sed -i 's%#PermitRootLogin yes%PermitRootLogin no%' /etc/ssh/sshd_config
        sed -i 's%#PermitEmptyPasswords no%PermitEmptyPasswords no%' /etc/ssh/
sshd_config
        sed -i 's%#UseDNS yes%UseDNS no%' /etc/ssh/sshd_config
        /etc/init.d/sshd reload &>/dev/null
    }

    function update_linux(){
        #10.upgrade linux.
        if [ `rpm -qa lrzsz nmap tree dos2unix nc|wc -l` -le 3 ]
          then
            yum install lrzsz nmap tree dos2unix nc -y
            #yum update -y
        fi
    }
    main(){
```

```
    mod_yum
    close_selinux
    close_iptables
    least_service
    adduser
    charset
    time_sync
    com_line_set
    open_file_set
    set_kernel
    init_ssh
    update_linux
}
main
```

4）剪辑出成品，测试审查并上线。

```
此处无代码，仅仅是步骤之一。
```

5）开发脚本，对修改的内容做检查（检验优化结果）。

```
[root@oldboy scripts]# cat check_opt.sh
#!/bin/bash
#################################################
#this scripts is created by oldboy
#oldboy QQ:31333741
#blog:http://oldboy.blog.51cto.com
#################################################
#set env
export PATH=$PATH:/bin:/sbin:/usr/sbin
# Require root to run this script.
if [ "$UID" != "0" ]; then
    echo "Please run this script by root."
    exit 1
fi

# Source function library.
. /etc/init.d/functions

function check_yum(){
    Base=/etc/yum.repos.d/CentOS-Base.repo
    if [ `grep aliyun $Base|wc -l` -ge 1 ];then
        action "$Base config" /bin/true
    else
        action "$Base config" /bin/false
    fi
}

function check_selinux(){
    config=/etc/selinux/config
```

```
        if [ `grep "SELINUX=disabled" $config|wc -l ` -ge 1 ];then
            action "$config config" /bin/true
        else
            action "$config config" /bin/false
        fi
    }


    function check_service(){
        export LANG=en
         if [ `chkconfig|grep 3:on|egrep "crond|sshd|network|rsyslog|sysstat"|
wc -l` -eq 5 ]
          then
            action "sys service init" /bin/true
        else
            action "sys service init" /bin/false
        fi
    }
    function check_open_file(){
        limits=/etc/security/limits.conf
        if [ `grep 65535 $limits|wc -l` -eq 1 ]
          then
            action "$limits" /bin/true
        else
            action "$limits" /bin/false
        fi
    }
    main(){
        check_yum
        check_selinux
        check_service
        check_open_file
    }
    main
```

执行结果如下：

```
[root@oldboy scripts]# sh check_opt.sh
/etc/yum.repos.d/CentOS-Base.repo config                   [  OK  ]
/etc/selinux/config config                                 [  OK  ]
sys service init                                           [  OK  ]
/etc/security/limits.conf                                  [  OK  ]
```

图 8-4 是执行结果截图。

图 8-4　检验优化结果的效果图

8.7 利用 Shell 函数开发 rsync 服务启动脚本

范例 8-8：开发启动 rsync 服务的系统服务脚本。

本例在第 7 章讲解 if 条件语句时就已经讲解过了，这里主要是将更加专业的脚本展示给读者。

源脚本为：

```
[root@oldboy scripts]# cat /etc/init.d/rsyncd
#!/bin/bash
# chkconfig: 2345 20 80
# description: Rsyncd Startup scripts by oldboy.
if [ $# -ne 1 ]
  then
    echo $"usage:$0 {start|stop|restart}"
    exit 1
fi
if [ "$1" = "start" ]
  then
    rsync --daemon
    sleep 2
    if [ `netstat -lntup|grep rsync|wc -l` -ge 1 ]
      then
        echo "rsyncd is started."
        exit 0
    fi
elif [ "$1" = "stop" ]
  then
    killall rsync &>/dev/null
    sleep 2
    if [ `netstat -lntup|grep rsync|wc -l` -eq 0 ]
      then
        echo "rsyncd is stopped."
        exit 0
    fi
elif [ "$1" = "restart" ]
  then
    killall rsync
    sleep 1
    killpro=`netstat -lntup|grep rsync|wc -l`
    rsync --daemon
    sleep 1
    startpro=`netstat -lntup|grep rsync|wc -l`
    if [ $killpro -eq 0 -a $startpro -ge 1 ]
      then
        echo "rsyncd is restarted."
        exit 0
    fi
else
    echo $"usage:$0 {start|stop|restart}"
```

```
    exit 1
fi
```

使用 start、stop 函数将代码模块化，使用系统函数 action 优化显示的脚本如下：

```
[root@oldboy ~]# cat /etc/init.d/rsyncd1
#!/bin/bash
# chkconfig: 2345 20 80
# description: Rsyncd Startup scripts by oldboy.
. /etc/init.d/functions

function usage(){
    echo $"usage:$0 {start|stop|restart}"
    exit 1
}
function start(){
    rsync --daemon
    sleep 1
    if [ `netstat -lntup|grep rsync|wc -l` -ge 1 ]
      then
        action "rsyncd is started." /bin/true
    else
        action "rsyncd is started." /bin/false
    fi
}

function stop(){
    killall rsync &>/dev/null
    sleep 2
    if [ `netstat -lntup|grep rsync|wc -l` -eq 0 ]
      then
        action "rsyncd is stopped." /bin/true
    else
        action "rsyncd is started." /bin/false
    fi
}
function main(){
    if [ $# -ne 1 ]
      then
        usage
    fi
    if [ "$1" = "start" ]
      then
        start
    elif [ "$1" = "stop" ]
      then
        stop
    elif [ "$1" = "restart" ]
      then
        stop
```

```
        sleep 1
        start
    else
        usage
    fi
}
main $*
```

本脚本实现了高度模块化，即用函数 1、函数 2、…、main 函数、main $* 传参，并能对其调用执行，这样的脚本不但专业规范，而且看上去也很高大上，值得读者花功夫去研究、学习和掌握。

执行效果如下：

```
[root@oldboy ~]# /etc/init.d/rsyncd1 start
rsyncd is started.                                         [  OK  ]
[root@oldboy ~]# /etc/init.d/rsyncd1 stop
rsyncd is stopped.                                         [  OK  ]
[root@oldboy ~]# /etc/init.d/rsyncd1 start
rsyncd is started.                                         [  OK  ]
[root@oldboy ~]# /etc/init.d/rsyncd1 restart
rsyncd is stopped.                                         [  OK  ]
rsyncd is started.                                         [  OK  ]
```

图 8-5 为执行结果截图。

图 8-5　rsync 服务启动脚本执行效果

是不是觉得该脚本已经比较专业了？其实，上述脚本还可以更专业，例如，采用后文将要讲解的 case 语句来实现，用 exit 处理返回值，且通过 PID 来判断服务启动的情况。

特别说明：可访问如下地址或手机扫二维码查看第 8 章的核心脚本代码

http://oldboy.blog.51cto.com/2561410/1855639

第9章

case 条件语句的应用实践

case 条件语句相当于多分支的 if/elif/else 条件语句，但是它比这些条件语句看起来更规范更工整，常被应用于实现系统服务启动脚本等企业应用场景中。

在 case 语句中，程序会将 case 获取的变量的值与表达式部分的值 1、值 2、值 3 等逐个进行比较，如果获取的变量值和某个值（例如值 1）相匹配，就会执行值（例如值 1）后面对应的指令（例如指令 1，其可能是一组指令），直到执行到双分号（;;）才停止，然后再跳出 case 语句主体，执行 case 语句（即 esac 字符）后面的其他命令。

如果没有找到匹配变量的任何值，则执行“ *)”后面的指令（通常是给使用者的使用提示），直到遇到双分号（;;）（此处的双分号可以省略）或 esac 结束，这部分相当于 if 多分支语句中最后的 else 语句部分。另外，case 语句中表达式对应值的部分，还可以使用管道等更多功能来匹配。

9.1 case 条件语句的语法

case 条件语句的语法格式为：

```
case "变量" in
    值1)
        指令1...
        ;;
    值2)
        指令2...
```

```
        ;;
    *)
        指令 3...
esac
```

说明：当变量的值等于值 1 时，执行指令 1；等于值 2 时执行指令 2，以此类推；如果都不符合，则执行“*)”后面的指令，即指令 3。此外，注意不同行内容的缩进距离。

为了便于大家记忆，下面是某女生写的 case 条件语句的中文形象描述：

```
case “找老公条件” in
    家里有房子）
        嫁给你 ...          #<== 钱
        ;;
    家庭有背景）
        嫁给你 ...          #<== 权
        ;;
    很努力吃苦）
        先谈谈男女朋友 ...  #<== 潜力股（曾经的老男孩）
        ;;
    *)
        good bye！！ ...    #<== 淘汰
esac
```

case 条件语句的执行流程逻辑图如图 9-1 所示。

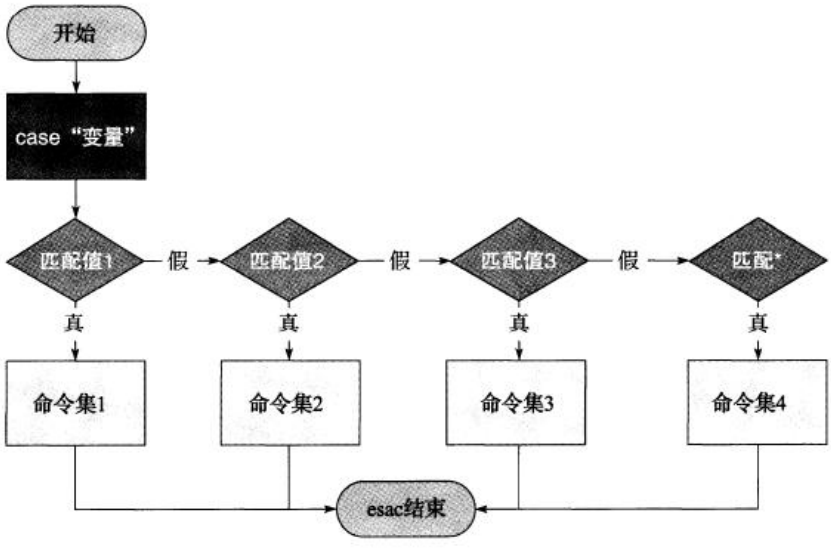


图 9-1　case 条件语句的执行流程逻辑图

9.2　case 条件语句实践

范例 9-1：根据用户的输入判断用户输入的是哪个数字。

如果用户输入的是 1 ~ 9 的任意数字，则输出对应输入的数字；如果是其他数字及字符，则返回输入不正确的提示并退出程序。

参考答案 1：使用 case 语句实现。

```
[root@oldboy scripts]# cat 9_1.sh
#!/bin/bash
# this script is created by oldboy.
# oldboy@oldboyedu.com
#<== 前面是版权及作者信息。
read -p "Please input a number:" ans  #<== 打印信息提示用户输入，输入信息赋值给 ans 变量。
case "$ans" in                        #<==case 语句获取 ans 变量的值，进入程序匹配比较。
    1)                                #<== 如果用户输入的信息为 1，则执行下面的 echo 命
                                          令输出。
        echo  "The num you input is 1"
        ;;                            #<== 匹配每个值后，执行值后面的命令，直到双分号
                                          的位置，双分号为终止符。
    2)                                #<== 如果用户输入的信息为 2，则执行下面的 echo 命
                                          令输出。
        echo "The num you input is 2"
        ;;
    [3-9])  #<== 如果用户输入的信息为 3-9 中的任意数字，注意范围匹配的正则写法，则执行下面
                 的 echo 命令输出。
        echo "The num you input is $ans"
        ;;
    *)      #<== 如果不匹配上面任何一个值，则执行下面的 echo 命令输出。
        echo "Please input [0-9] int"
        exit;
        #<== esac 语句结束前的最后一个值匹配，可以省略双分号。
esac
```

参考答案 2：使用 if 语句实现。

```
[root@oldboy scripts]# cat 9_2.sh
#!/bin/bash
# this script is created by oldboy.
# oldboy@oldboyedu.com
read -p "please input a number:" ans
if [ $ans -eq 1 ];then
    echo "the num you input is 1"
elif [ $ans -eq 2 ];then
    echo "the num you input is 2"
elif [ $ans -ge 3 -a $ans -le 9 ];then
    echo "the num you input is $ans"
else
    echo "the num you input must be [1-9]"
```

```
        exit
    fi
```

对比 case 语句和 if 语句，会发现 case 语句更简洁更规范，if 语句看起来则要复杂一些。

范例 9-2：执行 shell 脚本，打印一个如下的水果菜单：

（1）apple

（2）pear

（3）banana

（4）cherry

当用户输入对应的数字选择水果的时候，告诉他选择的水果是什么，并给水果单词加上一种颜色（随意），要求用 case 语句来实现。

在解题之前，先来学点预备知识。

Linux 命令行中给字体加颜色的命令为：

```
[root@oldboy scripts]# echo -e "\E[1;31m 红色字 oldboy\E[0m"
红色字 oldboy
[root@oldboy scripts]# echo -e "\033[31m 红色字 oldboy \033[0m"
红色字 oldboy
```

在上述命令中：

- echo -e 可以识别转义字符，这里将识别特殊字符的含义，并输出。
- \E 可以使用 \033 替代。
- “[1”数字 1 表示加粗显示（可以加不同的数字，以代表不同的意思，详细信息可用 man console_codes 获得）。
- 31m 表示为红色字体，可以换成不同的数字，以代表不同的意思。
- “红色字 oldboy”表示待设置的内容。
- “[0m”表示关闭所有属性，可以换成不同的数字，以代表不同的意思。

有关 ANSI 控制码的说明如下。

- \33[0m 表示关闭所有属性。
- \33[1m 表示设置高亮度。
- \33[4m 表示下划线。
- \33[5m 表示闪烁。
- \33[7m 表示反显。
- \33[8m 表示消隐。
- **\33[30m -- \33[37m 表示设置前景色。**
- **\33[40m -- \33[47m 表示设置背景色。**

console codes 的更多知识可以参考 man console_codes，普通读者了解即可。

下面先来一个给内容加颜色的热身示例：

```
[root@oldboy scripts]# cat plush_color.sh
#!/bin/sh
RED_COLOR='\E[1;31m'                          #<== 把颜色定义为变量，以方便使用。
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
RES='\E[0m'
echo -e "$RED_COLOR oldboy $RES"              #<== 变量中间就是待加颜色的字符串。
echo -e "$YELLOW_COLOR oldgirl $RES"
[root@oldboy scripts]# sh plush_color.sh
      红色
oldgirl 黄色
```

下面正式解答范例 9-2。

参考答案 1：

```
[root@oldboy scripts]# cat 9_2.sh
#!/bin/sh
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
RES='\E[0m'
echo ' #<== 使用 echo 打印菜单，带大家练习下 echo 打印菜单的方法，不过还是使用 cat 命令更好。
  ======================
  1.apple
  2.pear
  3.banana
  4.cherry
  ======================
'
read -p "pls select a num:" num #<== 打印信息提示用户输入，输入信息并赋值给 num 变量。
case "$num" in                  #<== case 语句获取 num 变量的值，进入程序匹配比较。
    1)              #<== 如果用户输入的信息为 1，则执行下面的 echo 命令输出。
        echo -e "${RED_COLOR}apple${RES}"
        ;;
    2)              #<== 如果用户输入的信息为 2，则执行下面的 echo 命令输出，下同。
        echo -e "${GREEN_COLOR}pear${RES}"
       ;;
    3)
       echo -e "${YELLOW_COLOR}banana${RES}"
       ;;
    4)
       echo -e "${BLUE_COLOR}cherry${RES}"
       ;;
    *)              #<== 如果不匹配上面任何一个值，则执行下面的 echo 命令输出。
       echo "muse be {1|2|3|4}"
                    #<== esac 语句结束前的最后一个值匹配，则可以省略双分号。
esac
```

执行结果如图 9-2 所示。

```
[root@oldboy scripts]# sh 9_2.sh
  ====================
  1.apple
  2.pear
  3.banana
  4.cherry
  ====================

pls select a num:1
apple
[root@oldboy scripts]# sh 9_2.sh
  ====================
  1.apple
  2.pear
  3.banana
  4.cherry
  ====================

pls select a num:2
pear
```

图 9-2　菜单功能测试图

参考答案 2：

```
[root@oldboy scripts]# cat 9_2_2.sh
#!/bin/sh
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
RES='\E[0m'
menu(){
cat <<END          #<== 采用 cat 命令方式打印菜单，推荐此方式，当然了，还可用 select 循环
                       打印菜单，不过不常用。
1.apple
2.pear
3.banana
END
}
menu
read -p "pls input your choice:" fruit
case "$fruit" in
    1)
        echo -e "${RED_COLOR}apple${RES}"
        ;;
    2)
        echo -e "${GREEN_COLOR}pear${RES}"
        ;;
    3)
        echo -e "${YELLOW_COLOR}banana${RES}"
        ;;
    *)
        echo -e "no fruit you choose."
esac
```

参考答案 3：

```
[root@oldboy scripts]# cat 9_2_3.sh
#!/bin/sh
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
RES='\E[0m'
function usage(){          #<== 将使用帮助写成 usage 函数，方便重复使用。
    echo "USAGE: $0 {1|2|3|4}"
    exit 1
}
function menu(){           #<== 将菜单内容写成函数，方便重复使用，看起来也更专业。
    cat <<END
    1.apple
    2.pear
    3.banana
END
}
function chose(){          #<== 将输入变量判断内容写成函数，方便重复使用，看起来也更专业。
read -p "pls input your choice:" fruit
case "$fruit" in
    1)
        echo -e "${RED_COLOR}apple${RES}"
        ;;
    2)
        echo -e "${GREEN_COLOR}pear${RES}"
        ;;
    3)
        echo -e "${YELLOW_COLOR}banana${RES}"
        ;;
    *)
        usage
esac
}
function main(){           #<== 主函数，执行定义的所有函数，这是程序的入口，模拟 C 语言
                               的编程方式，看上去更高大上。
    menu
    chose
}
main                       #<== 执行主函数，进而执行 Shell 脚本。
```

说明：这是一个比较专业、规范的脚本，即美观又实用，在同质化竞争激烈的今天，我们就要处处比别人做得好。不是好一点点，而是好很多！那么从写脚本开始做起吧，伙伴们。

执行结果如下：

```
[root@oldboy scripts]# sh 9_2_3.sh
1.apple
2.pear
3.banana
pls input ycur choice:1
apple                          #<== 这是红色字
[root@oldboy scripts]# sh 9_2_3.sh
1.apple
2.pear
3.banana
pls input your choice:2
pear                           #<== 这是绿色字
[root@oldboy scripts]# sh 9_2_3.sh
1.apple
2.pear
3.banana
pls input your choice:3
banana                         #<== 这是黄色字
[root@oldboy scripts]# sh 9_2_3.sh
1.apple
2.pear
3.banana
pls input your choice:4
USAGE: 9_2_3.sh {1|2|3|4}
```

9.3 实践：给输出的字符串加颜色

9.3.1 给输出的字符串加颜色的基础知识

在 Linux 脚本中，可以通过 echo 的 -e 参数，结合特殊的数字给不同的字符加上颜色并显示。

范例 9-3：给内容加上不同的颜色。

内容的颜色可用数字表示，范围为 30 ~ 37，每个数字代表一种颜色。代码如下：

```
echo -e "\033[30m 黑色字 oldboy trainning \033[0m"  #<==30m 表示黑色字。
echo -e "\033[31m 红色字 oldboy trainning \033[0m"  #<==31m 表示红色字。
echo -e "\033[32m 绿色字 oldboy trainning \033[0m"  #<==32m 表示绿色字。
echo -e "\033[33m 棕色字 oldboy trainning \033[0m"  #<==33m 表示棕色字 (brown),
                                                       和黄色字相近。
echo -e "\033[34m 蓝色字 oldboy trainning \033[0m"  #<==34m 表示蓝色字。
echo -e "\033[35m 洋红字 oldboy trainning \033[0m"  #<==35m 表示洋红色字 (magenta),
                                                       和紫色字相近。
echo -e "\033[36m 蓝绿色 oldboy trainning \033[0m"  #<==36m 表示蓝绿色字 (cyan),
                                                       和浅蓝色字相近。
echo -e "\033[37m 白色字 oldboy trainning \033[0m"  #<==37m 表示白色字。
```

说明：不同的数字对应不同的字体颜色，详情请参见系统帮助（来自使用 man console_codes 命令的结果）。

执行结果见图 9-3。

图 9-3　不同的数字对应的字体颜色执行结果图

范例 9-4：通过定义变量的方式给字体加颜色（推荐使用的方式）。

```
[root@oldboy scripts]# cat 9_4.sh
#!/bin/bash
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
PINK='\E[1;35m'
RES='\E[0m'
echo -e  "${RED_COLOR}======red color======${RES}"
echo -e  "${YELLOW_COLOR}======yellow color======${RES}"
echo -e  "${BLUE_COLOR}======blue color======${RES}"
echo -e  "${GREEN_COLOR}======green color======${RES}"
echo -e  "${PINK}======pink color======${RES}"
```

执行结果见图 9-4。

图 9-4　执行结果

9.3.2　结合 case 语句给输出的字符串加颜色

范例 9-5：请开发一个给指定内容加指定颜色的脚本。

要求：使用 case 语句及通过脚本传入指定内容和指定颜色，在脚本命令行传 2 个

参数，给指定的内容（第一个参数）加指定的颜色（第二个参数），演示效果如图 9-5 所示。

```
[root@oldboy scripts]# sh 9_5_1.sh
Usage 9_5_1.sh content {red|yellow|blue|green}
[root@oldboy scripts]# sh 9_5_1.sh oldboy red
oldboy
[root@oldboy scripts]# sh 9_5_1.sh oldgirl green
oldgirl
[root@oldboy scripts]# sh 9_5_1.sh cherry BLUE
cherry
```

图 9-5 演示效果

实现脚本如下：

```
[root@oldboy scripts]# cat 9_5_1.sh
#!/bin/bash
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
PINK='\E[1;35m'
RES='\E[0m'
#<== 以上部分将颜色字符定义为变量，方便后续使用。
if [ $# -ne 2 ];then      #<== 如果脚本传入的参数个数不等于 2，则打印帮助提示，退出脚本。
    echo "Usage $0 content {red|yellow|blue|green|pink}"
    exit
fi
case "$2" in      #<==$2 为脚本传入的第二个参数，即颜色字符串。
    red|RED)      #<== 使用竖线 | 可以匹配大写和小写等字符，也就是可以用竖线来匹配多个值。
        echo -e  "${RED_COLOR}$1${RES}"          #<== 给 $1（第一个参数）加颜色。
        ;;
    yellow|YELLOW)
        echo -e  "${YELLOW_COLOR}$1${RES}"
        ;;
    green|GREEN)
        echo -e  "${GREEN_COLOR}$1${RES}"
        ;;
    blue|BLUE)
        echo -e  "${BLUE_COLOR}$1${RES}"
        ;;
    pink|PINK)
        echo -e  "${PINK_COLOR}$1${RES}"
        ;;
    *)
        echo "Usage $0 content {red|yellow|blue|green|pink}"
        exit
esac
```

如果脚本中要加颜色的内容很多，还可以专门写一个给内容加颜色的函数，如下：

```
[root@oldboy scripts]# cat 9_5_2.sh
#!/bin/bash
plus_color(){  #<== 定义一个给指定字符内容加颜色的函数，其实就是将前面的脚本 9_5_1.sh
                    整个包起来定义为函数，那么原来的传参就变成了函数的参数了。
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
PINK='\E[1;35m'
RES='\E[0m'
if [ $# -ne 2 ];then
    echo "Usage $0 content {red|yellow|blue|green|pink}"
    exit
fi
case "$2" in
    red|RED)
        echo -e  "${RED_COLOR}$1${RES}"
        ;;
    yellow|YELLOW)
        echo -e  "${YELLOW_COLOR}$1${RES}"
        ;;
    green|GREEN)
        echo -e  "${GREEN_COLOR}$1${RES}"
        ;;
    blue|BLUE)
        echo -e  "${BLUE_COLOR}$1${RES}"
        ;;
    pink|PINK)
        echo -e  "${PINK_COLOR}$1${RES}"
        ;;
    *)
        echo "Usage $0 content {red|yellow|blue|green|pink}"
        exit
esac
}
plus_color "I" red          #<== 传入 I 字符和红色单词给 plus_color 函数。
plus_color "am" green       #<== 传入 am 字符和绿色单词给 plus_color 函数。
plus_color "oldboy" blue    #<== 传入 oldboy 字符和蓝色单词给 plus_color 函数。
```

执行结果如图 9-6 所示。

```
[root@oldboy scripts]# sh 9_5_2.sh
I
am
oldboy
```

图 9-6　演示效果图

本题更专业更规范的实现脚本如下，此脚本为全函数模块化、标准化、专业化的实现：

```
[root@oldboy scripts]# cat 9_5_3.sh
#!/bin/bash
function AddColor(){ #<== 定义加颜色函数 AddColor。
RED_COLOR='\E[1;31m'
GREEN_COLOR='\E[1;32m'
YELLOW_COLOR='\E[1;33m'
BLUE_COLOR='\E[1;34m'
PINK='\E[1;35m'
RES='\E[0m'
if [ $# -ne 2 ];then
    echo "Usage $0 content {red|yellow|blue|green}"
    exit
fi
case "$2" in
    red|RED)
        echo -e  "${RED_COLOR}$1${RES}"
        ;;
    yellow|YELLOW)
        echo -e  "${YELLOW_COLOR}$1${RES}"
        ;;
    green|GREEN)
        echo -e  "${GREEN_COLOR}$1${RES}"
        ;;
    blue|BLUE)
        echo -e  "${BLUE_COLOR}$1${RES}"
        ;;
    pink|PINK)
        echo -e  "${PINK_COLOR}$1${RES}"
        ;;
    *)
        echo "Usage $0 content {red|yellow|blue|green}"
        exit
esac
}
function main(){          #<== 定义主函数 main
    AddColor $1 $2        #<== 令颜色函数后面接 $1 和 $2，即将函数参数转换为获取脚本的
参数，注意，这里的 $1 和 $2 不能用引号括起来，否则脚本不传参也会被 AddColor 认为已传入参数。在此
特别感谢看老男孩 VIP 教程视频的某位同学提出了此问题。
}
main $*           #<== 执行 main 函数，利用 $* 接收命令行的所有参数，并传入到 main 函数。
```

9.3.3 给输出的字符串加背景颜色

范例 9-6：给输出的字符串加不同的背景颜色。

字的背景颜色对应的数字范围为 40 ~ 47，代码如下。

```
echo -e "\033[40;37m 黑底白字 oldboy\033[0m"   #<==40m 表示黑色背景。
echo -e "\033[41;37m 红底白字 oldboy\033[0m"   #<==41m 表示红色背景。
```

```
echo -e "\033[42;37m 绿底白字 oldboy\033[0m"      #<==42m 表示绿色背景。
echo -e "\033[43;37m 棕底白字 oldboy\033[0m"      #<==43m 表示棕色背景（brown），和
                                                    黄色背景相近。
echo -e "\033[44;37m 蓝底白字 oldboy\033[0m"      #<==44m 表示蓝色背景。
echo -e "\033[45;37m 洋红底白字 oldboy\033[0m"    #<==45m 表示洋红色背景（magenta），
                                                    和紫色背景相近。
echo -e "\033[46;37m 蓝绿底白字 oldboy\033[0m"    #<==46m 表示蓝绿色背景（cyan），和
                                                    浅蓝色背景相近。
echo -e "\033[47;30m 白底黑字 oldboy\033[0m"      #<==47m 表示白色背景。
```

说明：不同的数字所对应的背景字体颜色见系统帮助（来源于 man console_codes 命令的执行结果）。

执行结果如图 9-7 所示。

图 9-7　加背景颜色效果图

9.4　case 语句企业级生产案例

范例 9-7：实现通过传参的方式往 /etc/openvpn_authfile.conf 里添加用户，具体要求如下。

1）命令用法为：

```
USAGE: sh adduser {-add|-del|-search} username
```

2）传参要求为：

参数为 -add，表示添加后面接的用户名。

参数为 -del，表示删除后面接的用户名。

参数为 -search，表示查找后面接的用户名。

3）如果有同名的用户，则不能添加，如果没有对应的用户，则无需删除，查找到用户或没有用户时应给出明确提示。

4）/etc/openvpn_authfile.conf 不能被所有外部用户直接删除及修改。

参考答案：

```
[root@oldboy scripts]# cat add-openvpn-user
#!/bin/sh
#create by oldboy
#time :19:14 2012-3-21
#Source function library.
. /etc/init.d/functions
#config file path
FILE_PATH=/etc/openvpn_authfile.conf    #<== 这是 openvpn 的登录授权文件路径。
[ ! -f $FILE_PATH ] && touch $FILE_PATH   #<== 如果变量对应的文件不存在，则创建文件。
usage(){                                  #<== 帮助函数。
    cat <<EOF                             #<== 这是一个可以替代 echo 的输出菜单等内
                                              容的方法。
    USAGE: `basename $0` {-add|-del|-search} username
EOF
}

#judge run user
if [ $UID -ne 0 ] ;then  #<== 必须是 root 用户，才能执行本脚本。
    echo "Youare not supper user,please call root!"
    exit 1;
fi

#judge arg numbers.
if [ $# -ne 2 ] ;then    #<== 传入的参数必须为两个。
    usage
    exit 2
fi
# 满足条件后进入 case 语句判断。
case "$1" in             #<== 获取命令行第一个参数的值。
    -a|-add)             #<== 如果匹配 -a 或 -add，则执行下面的命令语句。
        shift            #<== 将 $1 清除，将 $2 替换为 $1，位置参数左移。
        if grep "^$1$" ${FILE_PATH} >/dev/null 2>&1   #<== 过滤命令行第一个参
                                                          数的值，如果有
          then           #<== 则执行下面的指令。
            action $"vpnuser,$1 is exist" /bin/false
            exit
         else            #<== 如果文件中不存在命令行传参的一个值，则执行下面的指令。
            chattr -i ${FILE_PATH}      #<== 解锁文件。
            /bin/cp ${FILE_PATH} ${FILE_PATH}.$(date +%F%T)
#<== 备份文件（尾部加时间）。
            echo "$1" >> ${FILE_PATH}     #<== 将第一个参数（即用户名）加入到文件。
            [ $? -eq 0 ] && action $"Add $1" /bin/true      #<== 如果返回值为 0，提
                                                               示成功。
            chattr +i ${FILE_PATH}      #<== 给文件加锁。
        fi
        ;;
```

```
    -d|-del)        #<== 如果命令行的第一个参数匹配 -d 或 -del，则执行下面的命令语句。

        shift
        if [ `grep "\b$1\b" ${FILE_PATH}|wc -l` -lt 1 ]   #<== 过滤第一个参数值，
                                                              并看文件中是否存在。

          then    #<== 如果不存在，则执行下面的指令。
            action $"vpnuser,$1 is not exist." /bin/false
            exit
        else      #<== 否则执行下面的指令，存在才删除，不存在就提示不存在，不需要删除。
            chattr -i ${FILE_PATH}              #<== 给文件解锁，准备处理文件的内容。
            /bin/cp ${FILE_PATH} ${FILE_PATH}.$(date +%F%T)
#<== 备份文件（尾部加时间）。
            sed -i "/^${1}$/d" ${FILE_PATH}       #<== 删除文件中包含命令行传参的用户。
            [ $? -eq 0 ] && action $"Del $1" /bin/true
#<== 如果返回值为 0，提示成功。
            chattr +i ${FILE_PATH}              #<== 给文件加锁。
            exit
        fi
        ;;
    -s|-search)   #<== 如果命令行的第一个参数匹配 -s 或 -search，就执行下面的命令语句。
        shift
        if [ `grep -w "$1" ${FILE_PATH}|wc -l` -lt 1 ]
#<== 过滤第一个参数值，并看文件中是否存在。
          then
            echo $"vpnuser,$1 is not exist.";exit
        else
            echo $"vpnuser,$1 is exist.";exit
        fi
        ;;
    *)
        usage
        exit
        ;;
esac
```

执行结果如下：

```
[root@oldboy scripts]# sh add-openvpn-user
    USAGE: add-openvpn-user {-add|-del|-search} username
[root@oldboy scripts]# sh add-openvpn-user -add oldboy
Add oldboy                                                    [ 确定 ]
[root@oldboy scripts]# rm -f /etc/openvpn_authfile.conf
rm: 无法删除 "/etc/openvpn_authfile.conf": 不允许的操作    #<== 因为使用 chattr 锁
                                                              定了（为了安全）。
[root@oldboy scripts]# sh add-openvpn-user -search oldboy
vpnuser,oldboy is exist.
[root@oldboy scripts]# sh add-openvpn-user -search oldgirl
vpnuser,oldgirl is not exist.
```

```
[root@oldboy scripts]# sh add-openvpn-user -add oldgirl
Add oldgirl                                                [ 确定 ]
[root@oldboy scripts]# sh add-openvpn-user -search oldgirl
vpnuser,oldgirl is exist.
[root@oldboy scripts]# sh add-openvpn-user -del oldgirl
Del oldgirl                                                [ 确定 ]
[root@oldboy scripts]# sh add-openvpn-user -search oldgirl
vpnuser,oldgirl is not exist.
```

本题除了 case 语句的应用之外，还有几个重要应用，那就是 grep 精确过滤单词的三种方法：

```
[root@oldboy scripts]# grep -w "oldboy" /etc/openvpn_authfile.conf
oldboy
[root@oldboy scripts]# grep "\boldboy\b" /etc/openvpn_authfile.conf
oldboy
[root@oldboy scripts]# grep "^oldboy$" /etc/openvpn_authfile.conf
oldboy
```

本例为老男孩在企业场景下管理 openvpn 的授权文件，只有这个文件里已存在的用户才能连接 vpn 服务器，相关内容具体见：http://oldboy.blog.51cto.com/2561410/986933。

范例 9-8：已知 Nginx Web 服务的管理命令如下，

启动服务命令为 /application/nginx/sbin/nginx

停止服务命令为 /application/nginx/sbin/nginx -s stop

请用 case 语句开发脚本，以实现 Nginx 服务启动及关闭的功能，具体脚本命令为 /etc/init.d/nginxd {start|stop|restart}，并实现通过 chkconfig 进行开机自启动的管理。

环境准备提示：

如果读者对 Nginx 环境还不是很熟悉，那么请参考《跟老男孩学 Linux 运维：Web 集群实战》第 5 章的内容。

解题思路：

1）先判断 Nginx 的 PID 文件是否存在（Nginx 服务正常启动后 PID 文件就会存在），如果不存在，即表示 Nginx 没有运行，则运行 Nginx 服务的启动命令（可以把此部分写成 start 函数）。待要停止时，如果 PID 存在，就运行 Nginx 服务停止命令，否则就不运行停止命令（可以把此部分写成 stop 函数）。

2）通过脚本传入参数 start 或 stop 等，通过 case 语句获取参数进行判断。

3）为了看起来更专业，这里采用前文讲解的系统函数库 functions 中的 action 函数。

4）对函数及命令运行的返回值进行处理，使脚本看起来更专业、规范。

5）通过 chkconfig 来管理 Nginx 脚本，实现开机自启动。

最后实现的脚本如下：

```
[root@oldboy scripts]# chmod +x /etc/init.d/nginxd
[root@oldboy scripts]# cat /etc/init.d/nginxd
#!/bin/sh
# chkconfig: 2345 40 98                    #<== 设定 2345 级别，开机第 40 位启动脚本，
                                                关机第 98 位关闭脚本。
# description: Start/Stop Nginx server     #<== 描述信息。
path=/application/nginx/sbin               #<== 设定 Nginx 启动命令路径。
pid=/application/nginx/logs/nginx.pid      #<== 设定 Nginx PID 文件路径。
RETVAL=0                                   #<== 设定 RETVAL 为 0，作为返回值变量。
. /etc/init.d/functions                    #<== 加载系统函数库，目的是便于后面使用
                                                action 等重要函数。
start(){                                   #<== 定义 start 启动函数。
    if [ ! -f $pid ];then                  #<== 如果 PID 文件不存在，则执行命令。
    #if [ `netstat -lntup|grep nginx|wc -l` -eq 0 ];then #<== 也可以根据端口进行判断。
        $path/nginx                                  #<== 启动 Nignx 命令。
        RETVAL=$?                          #<== 获取启动 Nignx 命令后的状态返回值。
        if [ $RETVAL -eq 0 ];then          #<== 如果返回值为 0，则执行下面的指令。
            action "nginx is started" /bin/true  #<== 打印专业的启动提示。
            return $RETVAL                 #<==retrun 将返回值，返回给命令脚本。
        else
            action "nginx is started" /bin/false #<== 如果返回值不为 0，则打印
                                                      启动失败的专业提示。
            return $RETVAL                 #<==retrun 将返回值，返回给命令脚本。
        fi                                 #<== 状态返回值判断 if 语句结束。
    else
        echo "nginx is running"            #<== 如果存在 Nginx PID 文件，则输出 Nginx
                                                正在运行的提示。
        return 0                           #<== retrun 将返回值，返回给命令脚本。
    fi
}
stop(){      #<== 定义 start 启动函数，这部分内容和 start 函数几乎一样，因此不再进行详细
                 注释，读者可参考 start 部分，看能否自行注释。
    if [ -f $pid ];then
    #if [ `netstat -lntup|grep nginx|wc -l` -eq 0 ];then
        $path/nginx -s stop
        RETVAL=$?
        if [ $RETVAL -eq 0 ];then
            action "nginx is stopped" /bin/true
            return $RETVAL
        else
            action "nginx is stopped" /bin/false
            return $RETVAL
        fi
    else
        echo "nginx is no running"
        return $RETVAL
```

```
    fi
}

case "$1" in                    #<== 通过特殊参数 $1 接收脚本传参的字符串 (start|stop|restart)。
    start)                      #<== 如果 $1 接收的脚本传参的值为 start，则执行 start 函数。
        start                   #<== 执行 start 函数。
        RETVAL=$?               #<== 获取 start 函数执行后的返回值。
        ;;
    stop)
        stop
        RETVAL=$?               #<== 获取 stop 函数执行后的返回值。
        ;;
    restart)
        stop
        sleep 1
        start
        RETVAL=$?               #<== 获取函数执行后的返回值。
        ;;
    *)
        echo $"Usage: $0 {start|stop|restart}"
        exit 1
esac
exit $RETVAL                    #<== 将脚本的返回值返回到执行脚本的当前 Shell。
```

执行结果如图 9-8 所示。

```
[root@oldboy scripts]# /etc/init.d/nginxd
Usage: /etc/init.d/nginxd {start|stop|restart|reload}
[root@oldboy scripts]# /etc/init.d/nginxd start
nginx is running
[root@oldboy scripts]# /etc/init.d/nginxd stop
nginx is stopped                                           [确定]
[root@oldboy scripts]# /etc/init.d/nginxd start
nginx is started                                           [确定]
[root@oldboy scripts]# /etc/init.d/nginxd restart
nginx is stopped                                           [确定]
nginx is started                                           [确定]
```

图 9-8　Nginx 启动脚本执行效果图

加入开机自启动，命令如下：

```
[root@oldboy scripts]# chkconfig --add nginxd
[root@oldboy scripts]# chkconfig --list nginxd
nginxd          0:关闭  1:关闭  2:启用  3:启用  4:启用  5:启用  6:关闭
```

范例 9-9：开发 MySQL 多实例中 3306 实例的启动停止脚本。

启动命令为：mysqld_safe --defaults-file=/data/3306/my.cnf &

启动过程为：

```
[root@oldboy 3306]# /data/3306/mysql start
```

```
Starting MySQL...
[root@oldboy 3306]# netstat -lnt|grep 3306
tcp        0      0 0.0.0.0:3306        0.0.0.0:*           LISTEN
```

停止命令为：mysqladmin -u root -poldboy123 -S /data/3306/mysql.sock shutdown
停止过程为：

```
[root@oldboy 3306]# /data/3306/mysql stop
Stoping MySQL...
[root@oldboy 3306]# netstat -lnt|grep 3306
```

请读者自行完成本 MySQL 多实例启动脚本的编写（脚本命令为 /data/3306/mysql {start|stop|restart}），可利用函数、if 语句、case 语句等综合实现。（15 分钟）

MySQL 服务多实例企业级实战的环境准备可参考《跟老男孩学 Linux 运维：Web 集群实战》第 9 章。

9.5　case 条件语句的 Linux 系统脚本范例

范例 9-10：使用 yum 命令安装 Nginx 后，对 Nginx 自带的启动服务脚本进行全文注释。

说明：执行 yum installl nginx –y 正确安装 Nginx 服务后才会有此脚本存在。

```
[root@oldboy scripts]# cat /etc/init.d/nginx -n
     1  #!/bin/sh
     2  #
     3  # nginx - this script starts and stops the nginx daemon   #<== 功能注释。
     4  #
     5  # chkconfig:   - 85 15  #<== 开机自启动设置。
     6  # description:  Nginx is an HTTP(S) server, HTTP(S) reverse \
                                #<== 对脚本的描述。
     7  #               proxy and IMAP/POP3 proxy server
     8  # processname: nginx
     9  # config:      /etc/nginx/nginx.conf
    10  # config:      /etc/sysconfig/nginx
    11  # pidfile:     /var/run/nginx.pid
    12
    13  # Source function library.
    14  . /etc/rc.d/init.d/functions           #<== 加载系统函数库 functions。
    15
    16  # Source networking configuration.
    17  . /etc/sysconfig/network               #<== 加载网络配置。
    18
    19  # Check that networking is up.
```

```
20 [ "$NETWORKING" = "no" ] && exit 0          #<== 检查网络服务是否启动，如果为
                                                    no，则退出脚本。
21
22 nginx="/usr/sbin/nginx"                     #<== 将启动命令赋值给变量 nginx。
23 prog=$(basename $nginx)                     #<== 取变量中的名称部分。
24
25 sysconfig="/etc/sysconfig/$prog"            #<== 将路径赋值给 sysconfig 变量。
26 lockfile="/var/lock/subsys/nginx"           #<== 定义锁文件路径。
27 pidfile="/var/run/${prog}.pid"              #<== 定义 PID 文件路径及名字。
28
29 NGINX_CONF_FILE="/etc/nginx/nginx.conf" #<== 定义 Nginx 配置文件路径。
30
31 [ -f $sysconfig ] && . $sysconfig          #<== 如果存在 sysconfig，就加载。
32
33
34 start() {                                  #<== 定义 start 函数。
35     [ -x $nginx ] || exit 5                 #<== 如果 $nginx 可执行不成立，则
                                                    退出脚本。
36     [ -f $NGINX_CONF_FILE ] || exit 6  #<== 如果配置文件不存在，则退出脚本。
37     echo -n $"Starting $prog: "             #<== 打印开始启动提示。
38     daemon $nginx -c $NGINX_CONF_FILE  #<== 指定配置文件启动 Nginx 服务。
39     retval=$?    #<== 将启动命令的返回值赋值给 retval，后面会进行判断。
40     echo         #<== 打印空行。
41     [ $retval -eq 0 ] && touch $lockfile      #<== 如果返回值为 0，则创
建锁文件，这个锁可理解为服务成功标识。
42     return $retval                          #<== 将返回值返回脚本。
43 }
44
45 stop() {                                   #<==stop 函数。
46     echo -n $"Stopping $prog: "             #<== 打印停止提示。
47     killproc -p $pidfile $prog              #<== 使用 killproc 函数指定参数停
                                                    止 Nginx 服务。
48     retval=$?    #<== 将启动命令的返回值赋值给 retval，后面会进行判断。
49     echo         #<== 打印空行。
50     [ $retval -eq 0 ] && rm -f $lockfile #<== 如果返回值为 0，则删除锁文件，
这个锁可理解为服务是否成功的标识，因为关闭服务成功了，所以删除锁文件。
51     return $retval                          #<== 将返回值返回脚本。
52 }
53
54 restart() {                                #<== 重启服务函数。
55     configtest_q || return 6                #<== 如果检查语法不成功，则退出函数
56     stop                                    #<== 执行停止函数。
57     start                                   #<== 执行启动函数。
58 }
59
60 reload() {                                 #<== 重新加载配置函数。
61     configtest_q || return 6                #<== 如果检查语法不成功，则退出函数
62     echo -n $"Reloading $prog: "
```

```
        killproc -p $pidfile $prog -HUP    #<== 使用 killproc 函数指定参数停
止 Nginx 服务，注意 -HUP 为优雅停止参数，即不影响用户体验。
        echo                               #<== 打印空行。
}

configtest() {                             #<== 检查语法的函数。
    $nginx -t -c $NGINX_CONF_FILE          #<==-t 为检查语法，-c 为指定配置文件。
}

configtest_q() {                           #<== 安静的检查语法函数。
    $nginx -t -q -c $NGINX_CONF_FILE       #<== -q 表示如果没有错误则不输出信息。
}

rh_status() {                              #<== 状态函数。
    status $prog                           #<== 打印 Nginx 服务状态。
}

rh_status_q() {                            #<== 安静的状态检查函数。
    rh_status >/dev/null 2>&1              #<== 输出和错误都定向到空。
}

# Upgrade the binary with no downtime.
upgrade() {                                #<== 升级函数，这个一般用不到，请
                                               读者忽略。
    local oldbin_pidfile="${pidfile}.oldbin"

    configtest_q || return 6
    echo -n $"Upgrading $prog: "
    killproc -p $pidfile $prog -USR2
    retval=$?
    sleep 1
    if [[ -f ${oldbin_pidfile} && -f ${pidfile} ]];  then
        killproc -p $oldbin_pidfile $prog -QUIT
        success $"$prog online upgrade"
        echo
        return 0
    else
        failure $"$prog online upgrade"
        echo
        return 1
    fi
}

# Tell nginx to reopen logs
reopen_logs() {                            #<== 打开 log 函数，这个一般用不
                                               到，请读者忽略。
    configtest_q || return 6
    echo -n $"Reopening $prog logs: "
    killproc -p $pidfile $prog -USR1
```

```
109     retval=$?
110     echo
111     return $retval
112 }
113
114 case "$1" in                          #<== 关键内容开始，获取传参值。
115     start)
116         rh_status_q && exit 0         #<== 如果状态检查是成功的，则退出脚本，即
                                              不需要启动。
117         $1                            #<== 获取 $1 值，执行 start 函数。
118         ;;
119     stop)
120         rh_status_q || exit 0         #<== 如果状态检查成功不成立，则退出脚本，
                                              即不需要停止。
121         $1                            #<== 获取 $1 值，执行 stop 函数。
122         ;;
123     restart|configtest|reopen_logs)
124         $1                            #<== 获取 $1 值，执行 restart 等函数。
125         ;;
126     force-reload|upgrade)
127         rh_status_q || exit 7
128         upgrade
129         ;;
130     reload)
131         rh_status_q || exit 7
132         $1                            #<== 获取 $1 值，执行 reload 函数。
133         ;;
134     status|status_q)
135         rh_$1     #<== 获取 $1 值，执行 rh_$1 函数，即 rh_status 或 rh_status_q。
136         ;;
137     condrestart|try-restart)
138         rh_status_q || exit 7
139         restart
140             ;;
141     *)            #<== 若不匹配上述值的内容，则打印使用帮助提示，并退出脚本。
142           echo $"Usage: $0 {start|stop|reload|configtest|status|for
ce-reload|upgrade|restart|reopen_logs}"
143         exit 2
144 esac
```

Linux 系统内部及前人的标杆脚本很值得我们去参考和学习，读者如果有精力可以阅读并对下面的脚本进行注释：

```
/etc/init.d/rpcbind #<== 执行 yum installl rpcbind -y 正确安装 rpcbind 服务后才会
                         有此脚本存在。
/etc/init.d/functions
此脚本的部分注释可参考 http://www.cnblogs.com/image-eye/archive/2011/10/26/2220405.html
/etc/rc.d/rc.sysinit
```

9.6　本章小结

（1）case 语句和 if 条件语句的适用性

case 语句比较适合变量值较少且为固定的数字或字符串集合的情况，如果变量的值是已知固定的 start/stop/restart 等元素，那么采用 case 语句来实现就比较适合。

（2）case 语句和 if 条件语句的常见应用场景

- case 主要是写服务的启动脚本，一般情况下，传参不同且具有少量的字符串，其适用范围较窄。
- if 就是取值判断、比较，应用面比 case 更广。几乎所有的 case 语句都可以用 if 条件语句来实现。

（3）case 语句的特点及优势

case 语句就相当于多分支的 if/elif/else 语句，但是 case 语句的优势是更规范、易读。

特别说明：可访问如下地址或手机扫二维码查看第 9 章的核心脚本代码

http://oldboy.blog.51cto.com/2561410/1855461

第10章

while 循环和 until 循环的应用实践

循环语句命令常用于重复执行一条指令或一组指令，直到条件不再满足时停止，Shell 脚本语言的循环语句常见的有 while、until、for 及 select 循环语句。

while 循环语句主要用来重复执行一组命令或语句，在企业实际应用中，常用于守护进程或持续运行的程序，除此以外，大多数循环都会用后文即将讲解的 for 循环语句。

10.1 当型和直到型循环语法

10.1.1 while 循环语句

while 循环语句的基本语法为：

```
while < 条件表达式 >
do
    指令 ...
done
```

 提示：注意代码缩进。

while 循环语句会对紧跟在 while 命令后的条件表达式进行判断，如果该条件表达式成立，则执行 while 循环体里的命令或语句（即语法中 do 和 done 之间的指令），每一次执行到 done 时就会重新判断 while 条件表达式是否成立，直到条件表达式不成立时才会

跳出 while 循环体。如果一开始条件表达式就不成立，那么程序就不会进入循环体（即语法中 do 和 done 之间的部分）中执行命令了。

可以用为手机充值（费用就是条件），然后发短信打电话的案例来形象地说明 while 循环语句的执行，如下。

```
while <手机话费是否充足>    #<== 当话费余额不足时，就不能发短信了，也就是不能进入循环了。
do
    发短信                  #<== 发短信后话费会减少。
done
```

while 循环执行流程对应的逻辑图如图 10-1 所示。

图 10-1　while 循环执行流程对应的逻辑图

10.1.2　until 循环语句

until 循环语句的语法为：

```
until <条件表达式>
do
    指令 ...
done
```

until 循环语句的用法与 while 循环语句的用法类似，区别是 until 会在条件表达式不成立时，进入循环执行指令；条件表达式成立时，终止循环。until 的应用场景很罕见，

读者了解下即可，本书不会深入探讨此循环语句。

10.2 当型和直到型循环的基本范例

首先来了解一下 Shell 中的两个休息命令：sleep 1 表示休息 1 秒，usleep 1000000 也表示休息 1 秒。

下面是与 while 和 until 循环语句相关的示例。

范例 10-1：每隔 2 秒输出一次系统负载（负载是系统性能的基础重要指标）情况。

参考答案 1：每隔 2 秒在屏幕上输出一次负载值。

```
[root@oldboy scripts]# cat 10_1_1.sh
#!/bin/sh
while true          #<== 提示：while true 表示条件永远为真，因此会一直运行，像死循环一
                        样，我们称之为守护进程。
do
    uptime
    sleep 2         #<== 休息 2 秒后继续循环，在 while true 循环中最好增加类似 sleep 的
                        命令，目的是控制循环的频率，否则会消耗大量系统资源，那样就真成
                        了名副其实的死循环了。
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 10_1_1.sh
11:04:22 up 14:52,  1 user,  load average: 0.00, 0.00, 0.00
11:04:24 up 14:52,  1 user,  load average: 0.00, 0.00, 0.00
11:04:26 up 14:52,  1 user,  load average: 0.00, 0.00, 0.00
11:04:28 up 14:52,  1 user,  load average: 0.00, 0.00, 0.00
```

参考答案 2：将负载值追加到 log 里，使用微秒单位。

```
[root@oldboy scripts]# cat 10_1_2.sh
#!/bin/sh
while [ 1 ]                 #==> 这里的条件和上面的写法有区别，注意 [] 里面两端要有空
                                格，true 和 1 都表示条件永远成立。
do
    uptime >>/tmp/uptime.log  #==> 将负载值输入到 log 文件里，注意，此处最好写绝对路径。
    usleep 2000000            #==> 这里的单位是微秒，其实结果也是两秒。
done
```

通过在脚本的结尾使用 & 符号来在后台运行脚本：

```
[root@oldboy scripts]# sh 10_1_2.sh &
[1] 14318
[root@oldboy scripts]# tail -f /tmp/uptime.log     #==> 使用 tail 命令实时观察输出结果。
11:08:35 up 14:57,  1 user,  load average: 0.00, 0.00, 0.00
```

```
11:08:37 up 14:57,  1 user,  load average: 0.00, 0.00, 0.00
11:08:39 up 14:57,  1 user,  load average: 0.00, 0.00, 0.00
11:08:41 up 14:57,  1 user,  load average: 0.00, 0.00, 0.00
```

在实际工作中，一般会通过客户端 SSH 连接服务器，因此可能就会有在脚本或命令执行期间不能中断的需求，若中断，则会前功尽弃，更要命的是会破坏系统数据。下面是防止脚本执行中断的几个可行方法：

1）使用 sh /server/scripts/while_01.sh & 命令，即使用 & 在后台运行脚本。

2）使用 nohup /server/scripts/uptime.sh & 命令，即使用 nohup 加 & 在后台运行脚本。

3）利用 screen 保持会话，然后再执行命令或脚本，即使用 screen 保持当前会话状态。

此外，让进程在后台可靠运行的几种方法的参考资料如下：

http://www.ibm.com/developerworks/cn/linux/l-cn-nohup/

10.3　让 Shell 脚本在后台运行的知识

有关脚本运行的相关用法和说明见表 10-1。

表 10-1　脚本运行的相关用法和说明

用法	说明
sh while1.sh &	把脚本 while1.sh 放到**后台执行（在后台运行脚本时常用的方法）**
ctl+c	**停止执行**当前脚本或任务
ctl+z	**暂停执行**当前脚本或任务
bg	把当前脚本或任务放到**后台执行**，bg 可以理解为 background
fg	把当前脚本或任务放到**前台执行**，如果有多个任务，可以使用 fg 加任务编号调出对应的脚本任务，如 fg 2，是指调出第二个脚本任务，fg 可以理解为 frontground
jobs	查看当前执行的脚本或任务
kill	**关闭执行**的脚本任务，即以“kill % 任务编号”的形式关闭脚本，这个任务编号，可以通过 jobs 来获得

下面针对表 10-1 中的知识进行实践演示，如下：

```
[root@oldboy scripts]# sh 10_1_2.sh &   #==> 结尾使用 & 符号表示在后台运行脚本。
[1] 14318
[root@oldboy scripts]# fg          #==> 执行 fg 命令将脚本放到前台执行，如果有多个脚本任务，则可以使用 fg 加 jobs 输出中的任务编号调出对应编号的脚本到前台。
sh 10_1_2.sh
^Z                                 #==> 执行 Ctrl+z 快捷键出现如下结果，临时暂停执行脚本。
[1]+  Stopped                 sh 10_1_2.sh
[root@oldboy scripts]# bg          #==> 将当前执行的脚本放到后台运行。
[1]+ sh 10_1_2.sh &
```

```
[root@oldboy scripts]# jobs      #==> 查看当前 Shell 下运行的脚本任务。
[1]+  Running                 sh 10_1_2.sh &
[root@oldboy scripts]# fg 1      #==> 可以使用 fg 加 jobs 输出中的任务编号调出对应编号
                                      的脚本到前台来运行。
sh 10_1_2.sh
^C                               #==> 当脚本在前台运行时，可以执行 Ctrl+c 快捷键停止
                                      脚本运行。
[1]+  Stopped                 sh while-1-1.sh
```

以下是使用 kill 命令关闭 jobs 任务脚本的示例。

```
[root@oldboy ~]# jobs
[1]-  Running                 sh while1.sh &
[2]+  Running                 sh while1.sh &
[root@oldboy ~]# kill %2                        #<== 注意任务编号的写法。
[root@oldboy ~]# jobs
[1]-  Running                 sh while1.sh &
[2]+  Terminated              sh while1.sh      #<== 脚本已关闭。
```

更多有关进程管理的 Linux 相关命令如下。

- kill、killall、pkill：杀掉进程。
- ps：查看进程。
- pstree：显示进程状态树。
- top：显示进程。
- renice：改变优先权。
- nohup：用户退出系统之后继续工作。
- pgrep：查找匹配条件的进程。
- strace：跟踪一个进程的系统调用情况。
- ltrace：跟踪进程调用库函数的情况。

范例 10-2：使用 while 循环竖向打印 54321。

参考答案 1：

```
[root@oldboy scripts]# cat 10_2_1.sh
#!/bin/sh
i=5                              #<== 因为是从大到小打印，所以初始化 i 的值为 5。
while ((i>0))                    #<== 双小括号条件表达式的用法，当 i 大于 0 不成立时就跳出循环。
do
    echo  "$i"                   #<== 打印变量 i 的值。
    ((i--))                      #<== i 的值自减，每次自减 1。
done
```

提示：当 i=1 的时候，((i--)) 的值就是 0，那么，此时就不满足 i 大于 0 了，条件不成立时就会退出循环，因此，上面的代码不会把 0 打印出来。

执行结果如下：

```
[root@oldboy scripts]# sh 10_2_1.sh
5
4
3
2
1
```

参考答案 2：使用双中括号条件表达式。

```
#!/bin/sh
i=5
while [[ $i > 0 ]]                          #<== 双中括号条件表达式的用法。
do
    echo $i
    ((i--))
done
```

参考答案 3：使用脚本传参需要打印的数字。

```
i="$1"                                      #<== 使用 i 接收脚本传参。
while [ $i -gt 0 ]                          #<== 单中括号条件表达式的用法。
do
    echo $i
    ((i--))
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 10_2_3.sh 5   #<== 也可以传入其他数字
5
4
3
2
1
```

参考答案 4：使用 until 命令。

```
[root@oldboy scripts]# cat 10_2_4.sh
#!/bin/sh
i=5
until [[ $i < 1 ]]                          #<== 当条件表达式不成立时，进入循环执行命令，
                                                因为 5<1 不成立，因此进入循环执行命令。
do
    echo $i
    ((i--))
```

```
done
```

提示：当 i 自减到 0 时，条件表达式（0<1）成立，因此跳出循环，当条件不成立时，则进入循环。

执行结果如下：

```
[root@oldboy scripts]# sh 10_2_4.sh
5
4
3
2
1
```

思考：如果不用 while、until 及 for 循环，还有什么命令或方法可以打印上述数字序列？

范例 10-3：计算从 1 加到 100 之和（请用 1+2+3+…+100 的方法）。

参考答案：

```
[root@oldboy scripts]# cat 10_3_1.sh
#!/bin/bash
# this script is created by oldboy.
i=1                #<==i 为自增的变量，从 1 到 100，初始值为 1。
sum=0              #<== 总和变量初始值为 0。
while ((i<=100))   #<== 条件是 i 小于等于 100，也就是从 1 加到 100。
do
    ((sum=sum+i)) #<== 把 i 的值和当前 sum 总和的值做加法并将结果重新赋值给 sum。
    ((i++))        #<==i 每次自增 1。
done
[ "$sum" -ne 0 ] && printf "totalsum is: $sum\n" #<== 打印总和。
```

执行结果如下：

```
[root@oldboy scripts]# sh 10_3_1.sh
totalsum is: 5050
```

下面是通过数学公式计算的结果：

```
[root@oldboy scripts]# cat 10_3_2.sh
#!/bin/sh
i=100
((sum=i*(i+1)/2))          #<== 利用数学公式进行计算，效率很高。
echo $sum
[root@oldboy scripts]# sh 10_3_2.sh
5050
```

提示：使用求和公式，代码简单而且运算高效。当 i 达到 10 万的时候就能明显感觉到公式计算和循环计算的差别[⊖]。

范例 10-4：猜数字游戏。首先让系统随机生成一个数字，给这个数字设定一个范围（1 ~ 60），让用户输入所猜的数字。游戏规则是：对输入进行判断，如果不符合要求，就给予高或低的提示，猜对后则给出猜对所用的次数，请用 while 语句实现。

以下解答来自老男孩高薪运维班 23 期学员阿彪的课堂练习。

```
[root@oldboy scripts]# cat 10_4_1.sh
#!/bin/bash
total=0                              #<== 初始化猜的次数为 0。
export LANG="zh_CN.UTF-8"            #<== 指定中文字符集，防止乱码。
NUM=$((RANDOM%61))                   #<== 随机数除以 61 取余数，最大值不超过 60，每执行一
次脚本就会生成一个处于 1 ~ 60 之间的随机数字，供用户猜。
echo " 当前苹果的价格是每斤 $NUM 元 "
echo "========================"
usleep 1000000
clear
echo ' 这苹果多少钱一斤啊?
      请猜 0 ~ 60 的数字 '
apple(){                             #<== 对输入检测的函数以 apple 命名。
    read -p " 请输入你的价格: " PRICE
    expr $PRICE + 1 &>/dev/null #<== 判断输入的价格是否为整数。
    if [ $? -ne 0 ]                  #<== 如果返回值不为 0，即表示非整数，则给出提示。
      then
        echo " 别逗我了，快猜数字 "
        apple                        #<== 重新加载猜价函数。
    fi
}

guess(){                             #<== 猜价格函数。
    ((total++))                      #<== 次数加 1。
    if [ $PRICE -eq $NUM ]           #<== 如果用户输入价格等于脚本执行时随机生成的价格，
                                         则执行 then 后面的命令。
      then
        echo " 猜对了，就是 $NUM 元 "
        if [ $total -le 3 ];then                    #<== 如果猜的次数小于 3 次，则执行
                                                        then 下面的命令。
            echo " 一共猜了 $total 次，太牛了。"    #<== 打印赞赏提示。
        elif [ $total -gt 3 -a $total -le 6 ];then  #<== 如果猜的次数有 3 ~ 6 次
            echo " 一共猜了 $total 次，次数有点多，加油啊。"  #<== 打印鼓励提示。
        elif [ $total -gt 6 ];then                  #<== 如果猜的次数多于 6 次，则执行
                                                        then 下面的命令。
            echo " 一共猜了 $total 次，行不行，猜了这么多次 "  #<== 打印打击提示。
```

⊖ Linux 系统计算从 1 加到 100 之和的 15 种思路见 http://oldboy.blog.51cto.com/2561410/767862。

```
        fi
        exit 0                                  #<== 猜对后退出脚本。
    elif [ $PRICE -gt $NUM ];then               #<== 如果用户输入的价格大于随机生成的价格，
                                                    则告诉用户猜高了。
        echo "嘿嘿，要不你用这个价买？ "        #<== 用户猜高了的幽默提示。
        echo "再给你一次机会，请继续猜："
        apple                                   #<== 没猜对，继续输入数字猜。
    elif [ $PRICE -lt $NUM ];then               #<== 如果用户输入的价格小于随机生成的价格，
                                                    则告诉用户猜低了。
        echo "太低太低"                         #<== 用户猜低了的幽默提示。
        echo "再给你一次机会，请继续猜："
        apple                                   #<== 没猜对，继续输入数字猜。
    fi
}
main(){                                         #<== 主函数 main。
    apple
    while true          #<== 对 guess 函数进行循环，猜对后退出脚本，若没猜对，则在 guess
                            里调用 apple 函数提示用户输入数字继续猜。
    do
        guess
    done
}
main
```

提示：注意调整 Linux 系统字符集及 SSH 客户端连接的字符集为 UTF-8。

执行结果如下：

```
[root@oldboy scripts]# sh 10_4_1.sh
当前苹果的价格是每斤 27 元 #<== 每执行一次脚本就会生成一个范围在 1 ~ 60 的随机数字，供用户猜。
========================
这苹果多少钱一斤啊?
      请猜 0 ~ 60 的数字
请输入你的价格：30
嘿嘿，要不你用这个价买?
再给你一次机会，请继续猜：
请输入你的价格：26
太低太低
再给你一次机会，请继续猜：
请输入你的价格：27
猜对了，就是 27 元
一共猜了 3 次，太牛了。
```

范例 10-5：手机充值 10 元，每发一次短信（输出当前余额）花费 1 角 5 分钱，当余额低于 1 角 5 分钱时就不能再发短信了，提示“余额不足，请充值”（允许用户充值后继续发短信），请用 while 语句实现。

在解答之前，先进行单位换算，统一单位，让数字变成整数，即：
10 元 =1000 分，1 角 5 分 =15 分
参考答案 1（简单版）：

```
sum=1000                              #<== 手机总费用。
i=15                                  #<== 每条短信的费用。
while ((sum>=i))                      #<== 当总费用大于单条短信费用时，进入循环。
do
    ((sum=sum-i))                     #<== 发一条短信就减掉一次单条短信费用。
    [ $sum -lt $i ] && break          #<== 如果总费用小于单条短信费用，则跳出循环。
    echo "send message,left $sum"     #<== 打印发信息提示。
done
echo "money is not enough:$sum"       #<== 提示 money 不够用。
```

参考答案 2（简单版）：

```
sum=1000
i=15
while ((sum>=i))
do
    ((sum=sum-i))
    [ $sum -lt $i ] &&{
        echo "send message,left $sum money is not enough"
        break
    }
    echo "send message,left $sum"
done
```

参考答案 3（专业脚本，来自学员阿续）：

```
[root@oldboy scripts]# cat 10_5_1.sh
#!/bin/sh
export LANG="zh_CN.UTF-8"             #<== 指定中文字符集以防止乱码。
sum=15                                #<== 初始总费用为 15。
msg_fee=15                            #<== 发一条短信需要 15。
msg_count=0                           #<== 初始发送次数为 0。
menu(){                               #<== 定义和用户交互的菜单。
    cat <<END
当前余额为 ${sum} 分，每条短信需要 ${msg_fee} 分
==========================
      1. 充值
      2. 发消息
      3. 退出
==========================
END
}
recharge(){                                  #<== 付费函数
    read -p "请输入金额充值 :" money         #<== 提示用户输入金额。
```

```
    expr $money + 1 &>/dev/null #<== 判断金额是否为整数。
    if [ $? -ne 0 ]             #<== 如果返回值不为 0，即不是整数，则打印提示输入整数。
      then
        echo "then money your input is error,must be int."
    else                        #<== 如果返回值为 0，即表示金额为整数，则进行充值操作。
        sum=$(($sum+$money))    #<== 将金额加到总费用里。
        echo " 当前余额为 :$sum"  #<== 打印充值后的金额。
    fi
}
sendInfo(){                                 #<== 定义发送短信的函数。
    if [ ${sum} -lt $msg_fee ]              #<== 如果总费用小于 15（单条短信的费用），则
                                                打印 then 后面的提示。
      then
        printf " 余额不足：$sum ，请充值。\n"
    else                    #<== 如果总费用大于 15（单条短信的费用），则进入 while 循环。
        while true          #<== 注意这里的条件为真。
        do
            read -p " 请输入短信内容（不能有空格）:" msg
                            #<== 提示用户输入要发送的信息。
            sum=$(($sum-$msg-fee))              #<== 减掉短信的费用。
            printf "Send "$msg" successfully!\n"  #<== 打印成功发送短信的提示。
            printf " 当前余额为：$sum\n"          #<== 打印当前余额。
            if [ $sum -lt $msg_fee ]            #<== 如果总费用小于单条费用。
              then
                printf " 余额不足，剩余 $sum 分 \n" #<== 则打印余额不足。
                return  1                       #<== 跳出循环和函数。
            fi
        done
    fi
}
main(){                     #<== 定义主函数。
    while true              #<== 开始执行持续循环，条件始终为真，注意当条件始终为真时，
                                就要想办法在循环里加入跳出循环的命令。
    do
        menu                #<== 执行菜单函数。
        read -p " 请输入数字选择：" men             #<== 接收用户输入。
        case "$men" in
            1)              #<== 当用户的输入为 1，表示想充值了，则加载 recharge 充值函数。
                recharge
                ;;
            2)              #<== 当用户的输入为 2，表示想发消息了，则加载 sendInfo 函数。
                sendInfo
                ;;
            3)              #<== 当用户的输入为 3，则表示想退出了。
                exit 1
                ;;
            *)              #<== 当用户的输入为其他值，给出正确的输入提示。
                printf " 选择错误，必须是 {1|2|3}\n"
        esac
```

```
    done
}
main                                    #<== 最后执行主函数 main。
```

执行结果如下：

```
[root@oldboy scripts]# sh 10_5_1.sh
当前余额为 15 分，每条短信需要 15 分    #<== 执行脚本后，打印菜单，供用户选择。
==========================
      1. 充值
      2. 发消息
      3. 退出
==========================
请输入数字选择：2                       #<== 选择 2。
请输入短信内容（不能有空格）:oldboy     #<== 短信内容为 oldboy。
Send oldboy successfully!
当前余额为：0
余额不足，剩余 0 分
当前余额为 0 分，每条短信需要 15 分
==========================
      1. 充值
      2. 发消息
      3. 退出
==========================
请输入数字选择：1                       #<== 选择 1。
请输入金额充值 :16                      #<== 输入充值金额。
当前余额为 :16
当前余额为 16 分，每条短信需要 15 分
==========================
      1. 充值
      2. 发消息
      3. 退出
==========================
请输入数字选择：2
请输入短信内容（不能有空格）:oldgirl
Send oldgirl successfully!
当前余额为：1
余额不足，剩余 1 分
当前余额为 1 分，每条短信需要 15 分
==========================
      1. 充值
      2. 发消息
      3. 退出
==========================
请输入数字选择：3
```

参考答案 4（专业脚本，来自学员二麻）：

```
[root@oldboy scripts]# cat 10_5_2.sh
#!/bin/sh
TOTAL=500                               #<== 定义总的手机费用，为了测试方便初始值为 500。
```

```
    MSG_FEE=499       #<== 定义一条短信费用，为了测试方便初始值为 499，即发一条就得充值。
    . /etc/init.d/functions          #<== 加载系统函数库。
    function IS_NUM(){               #<== 定义判断是否为数字的函数。
        expr $1 + 1 &>/dev/null      #<== 将传参加数字，看返回值。
        if [ $? -ne 0 -a "$1" != "-1" ];then    #<== 返回值不为 0，并且传参不等于 -1，
这个是考虑到排除 expr 加和为 0 的情况，加和为 0 属于特殊情况。
            return 1  #<== 以返回值 1 退出函数，这个返回值后面将会用到。
        fi
            return 0  #<== 输入为数字，以返回值 0 退出函数，这个返回值后面将会用到。
    }

    function color(){ #<== 定时给内容加颜色函数，整个脚本的内容在前面已经讲过了。
        RED_COLOR='\E[1;31m'
        YELLOW_COLOR='\E[1;33m'
        BLUE_COLOR='\E[1;34m'
        PINK='\E[1;35m'
        RES='\E[0m'
        if [ $# -ne 2 ];then
            echo "Usage $0 content {red|yellow|blue|pink}"
            exit
        fi
        case "$2" in
            red|RED)
                echo -e  "${RED_COLOR}$1${RES}"
                ;;
            yellow|YELLOW)
                echo -e  "${YELLOW_COLOR}$1${RES}"
                ;;
            blue|BLUE)
                echo -e  "${BLUE_COLOR}$1${RES}"
                ;;
            pink|PINK)
                echo -e  "${PINK_COLOR}$1${RES}"
                ;;
            *)
                echo "Usage $0 content {red|yellow|blue|pink}"
                exit
        esac
    }
    function consum(){                          #<== 定义发送短信的函数。
        color "You have left $TOTAL money,Send a msg need to charge $MSG_FEE
money" yellow                                   #<== 给双引号内容加黄色显示。
        if [ $TOTAL -lt $MSG_FEE ];then         #<== 如果总费用小于单条短信费用，
            charge                              #<== 则加载充值函数，提示用户充值。
        fi
        read -p "Pls input your msg:" TXT   #<== 提示用户发送信息。
        read -p "Are you to send?[y|n]" OPTION      #<== 给出是否发送确认。
        case "$OPTION" in
```

```
        [yY]|[yY][eE][sS])                          #<== 满足 yes 任意组合，就发送信息。
            color "Send "$TXT" successfully!" yellow
                                                    #<== 打印信息并给双引号内容加黄色显示。
            echo $TXT >>/tmp/consum.log #<== 将发送的信息记录到日志。
            ((TOTAL=TOTAL-MSG_FEE))     #<== 扣费。
            color "Your have $TOTAL left!" yellow
                                                    #<== 打印信息并给双引号内容加黄色显示。
            ;;
        [nN]|[nN][oO])   #<== 满足 no 任意组合，就取消发送信息。
            echo "Canceled"
            ;;
        *)
            echo "Invalid Input,this msg doesnt send out"
            ;;
    esac
}
function charge(){        #<== 定义充值函数。
    if [ $TOTAL -lt $MSG_FEE ];then     #<== 如果总费用低于一条短信费用时，就提示
                                            用户充值。
        color "Money is not enough,Are U want to charge?[y|n]" red
        #<== 以红色内容提示用户充值。
        read  OPT2                      #<== 提示输入 y|n。
        case "$OPT2" in
            y|Y) #<== 如果用户的输入匹配了 y 或 Y，则执行下面的 while 循环。
                while true
                do
                    read -p "How much are you want to charge?[INT]" CHARGE
                                        #<== 提示充值多少?
                    IS_NUM $CHARGE&&break||{
                        #<== 对充值是否为数字进行判断，如果不为数字，则给出提示；如
                            果为数字，则以返回值 0 跳出循环及函数，见 IS_NUM 函数体部分。
                        echo "INVALID INPUT"    #<== 提示错误输入。
                        exit 100                #<== 以 100 为返回值退出脚本。
                    }
                done
                ((TOTAL+=CHARGE)) && echo "you have $TOTAL money."
                                    #<== 进行充值，并计算充值后的金额打印。
                if [ $TOTAL -lt $MSG_FEE ];then
                                    #<== 如果充值后总费用仍小于单条短信的费用
                    charge          #<== 则加载充值函数 charge，提示用户继续充值。
                fi
                ;;
            n|N) #<== 如果用户的输入匹配了 n 或 N，即表示不充值，则执行下面的命令提示。
                color "You have left $TOTAL money,can not send a msg,bye" red
                ;;
            *)   #<== 输入其他值，进行充值提示。
                charge
                ;;
```

```
        esac
    fi
}

main(){             #<== 定义主函数
    while [ $TOTAL -ge $MSG_FEE ]  #<== 当总费用大于单条短信费用时，进入循环发信息。
    do
        consum      #<== 调用发送信息消费的函数。
        charge      #<== 调用充值函数。
    done
}
main                #<== 执行总的函数 main。
```

执行结果如下：

```
[root@oldboy scripts]# sh 10_5_2.sh
You have left 500 money,Send a msg need to charge 499 money
Pls input your msg:oldboy
Are you to send?[y|n]y
Send oldboy successfully!
Your have 1 left!
Money is not enough,Are U want to charge?[y|n]
y
How much are you want to charge?[INT]500
you have 501 money.
You have left 501 money,Send a msg need to charge 499 money
Pls input your msg:oldgirl
Are you to send?[y|n]n
Canceled
You have left 501 money,Send a msg need to charge 499 money
Pls input your msg:oldgirl
Are you to send?[y|n]y
Send oldgirl successfully!
Your have 2 left!
Money is not enough,Are U want to charge?[y|n]
n
You have left 2 money,can not send a msg,bye
```

10.4 企业生产实战：while 循环语句实践

范例 10-6：使用 while 守护进程的方式监控网站，每隔 10 秒确定一次网站是否正常。

参考答案：

```
[root@oldboy scripts]# cat 10_6_1.sh
#!/bin/sh
if [ $# -ne 1 ];then            #<== 判断，若传参的个数不为 1，
```

```
        echo $"usage $0 url"              #<== 则打印正确使用提示。
        exit 1                            #<== 以返回值 1 退出脚本。
    fi
    while true                            #<== 永远为真，进入 while 循环。
    do
        if [ `curl -o /dev/null --connect-timeout 5 -s -w "%{http_code}"  $1|
egrep -w "200|301|302"|wc -l` -ne 1 ]
        #<== 对传入的 URL 参数获取状态码，过滤 200、301、302 任意之一转为数字，如果不等于 1，
则表示状态信息不对。
          then
            echo "$1 is error."         #<== 提示 URL 访问错误。
            #echo "$1 is error."|mail -s "$1 is error." 31333741--@qq.com
                                          #<== 发送邮件报警。
        else
            echo "$1 is ok"             #<== 否则，提示 URL 访问 OK。
        fi
        sleep 10  #<== 休息 10 秒继续执行 while 循环，注意，当 while 后面有 true 等永远为真
的条件时，一般在循环里要有退出循环的条件或类似 sleep 休息的命令，否则会大量消耗系统资源。
    done
```

执行结果如下：

```
    [root@oldboy scripts]# sh 10_6_1.sh http://www.oldboyedu.com
    http://www.oldboyedu.com is ok    #<== 每隔 10 秒检查一次，提示正常。
    http://www.oldboyedu.com is ok    #<== 每隔 10 秒检查一次，提示正常。
    [root@oldboy scripts]# sh 10_6_1.sh http://www.abcdefg.com #<== 换个不存在的地址。
    http://www.abcdefg.com is error.  #<== 每隔 10 秒检查一次，提示异常。
    http://www.abcdefg.com is error.  #<== 每隔 10 秒检查一次，提示异常。
```

范例 10-7：使用 while 守护进程的方式监控网站，每隔 10 秒确定一次网站是否正常。

参考答案 1：引入函数库并且采用模拟用户访问的方式。

```
    [root@oldboy scripts]# cat 10_7_1.sh
    #!/bin/sh
    . /etc/init.d/functions           #<== 这里和上一个脚本不同，引入了函数库。
    if [ $# -ne 1 ];then
        echo $"usage $0 url"
        exit 1
    fi
    while true
    do
        if [ `curl -o /dev/null --connect-timeout 5 -s -w "%{http_code}"  $1|
egrep -w "200|301|302"|wc -l` -ne 1 ]
          then
            action "$1 is error." /bin/false  #<== 这里和上一个脚本不同，输出显示更专业了。
            #echo "$1 is error."|mail -s "$1 is error." 31333741--@qq.com
        else
```

```
        action "$1 is ok" /bin/true     #<== 这里和上一个脚本不同，输出显示更专业了。
    fi
    sleep 10
done
```

执行结果见图 10-2。

```
[root@oldboy scripts]# sh 10_7_1.sh http://www.oldboyedu.com
http://www.oldboyedu.com is ok                              [确定]
http://www.oldboyedu.com is ok                              [确定]
^C
[root@oldboy scripts]# sh 10_7_1.sh http://www.abcdefghi.com
http://www.abcdefghi.com is error.                          [失败]
http://www.abcdefghi.com is error.                          [失败]
^C
```

图 10-2　检测 URL 专业显示的效果图

参考答案 2：采用 Shell 数组（可参考本书第 13 章）的方法，同时检测多个 URL 是否正常，并给出专业的展示效果，这是实际工作中所用的脚本。

```
[root@oldboy scripts]# cat 10_7_2.sh
#!/bin/bash
# this script is created by oldboy.
# e_mail:31333741@qq.com
# function:case example
# version:1.3
. /etc/init.d/functions
check_count=0
url_list=(                  #<== 定义检测的 URL 数组，包含多个 URL 地址。
http://blog.oldboyedu.com
http://blog.etiantian.org
http://oldboy.blog.51cto.com
http://10.0.0.7
)
function wait()             #<== 定义 3,2,1 倒计时函数。
{
    echo -n '3 秒后，执行检查 URL 操作 .';
    for ((i=0;i<3;i++))
    do
        echo -n ".";sleep 1
    done
    echo
}
function check_url()        #<== 定义检测 URL 的函数。
{
    wait                    #<== 执行倒计时函数。
    for ((i=0; i<`echo ${#url_list[*]}`; i++)) #<== 循环数组元素。
    do
        wget -o /dev/null -T 3 --tries=1 --spider ${url_list[$i]} >/dev/null 2>&1
                                                #<== 检测是否可以访问数组元素的地址。
        if [ $? -eq 0 ]                         #<== 如果返回值为 0，则表示访问成功。
```

```
        then
          action "${url_list[$i]}" /bin/true #<== 优雅地显示成功结果。
        else
          action "${url_list[$i]}" /bin/false #<== 优雅地显示失败结果。
        fi
    done
    ((check_count++))                          #<== 检测次数加 1。
}
main(){                                        #<== 定义主函数。
    while true                                 #<== 开启一个持续循环。
    do
        check_url                              #<== 加载检测 url 的函数。
        echo "-------check count:${check_count}----------"
        sleep 10                               #<== 间歇 10 秒。
    done
}
main                                           #<== 调用主函数运行程序。
```

执行结果见图 10-3。

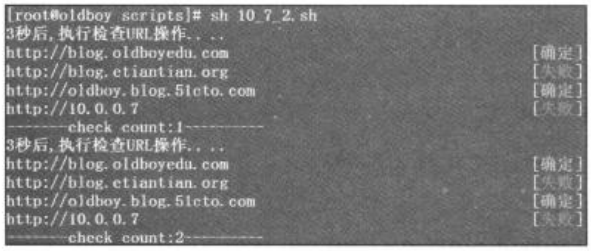


图 10-3　更专业的展示输出的效果

提示：实际使用时，一些基础的函数脚本（例如：加颜色的函数）是放在函数文件里的，例如放在 /etc/init.d/functions 里，与执行的内容部分相分离，这看起来更清爽，大型的语言程序都是这样开发的。

范例 10-8：分析 Apache 访问日志，把日志中每行的访问字节数对应的字段数字相加，计算出总的访问量。给出实现程序，请用 while 循环实现。（3 分钟）

本题要讲解的知识点是利用 while 循环读取文件操作的方法。根据题意可知，在 Web 日志里有一列记录了访问资源的大小，把这些资源的大小相加即为本题的答案。

参考答案：这里采用 while 循环与 bash exec 内置命令功能配合完成示例。

```
[root@oldboy scripts]# cat 10_8_1.sh
#!/bin/bash
sum=0                  #<== 初始化资源大小总和为 0。
```

```
    exec <$1            #<== 将传参 $1 输入重定向给 exec。
    while read line     #<== 按行读取传参的文件内容。
    do
        size=`echo $line|awk '{print $10}'` #<== 获取每行的第 10 列，即资源访问字节列。
        expr $size + 1 &>/dev/null          #<== 数字判断。
        if [ $? -ne 0 ];then                #<== 如果非数字，
            continue                        #<== 则执行 continue 终止本次循环，即不是
                                                数字的列不会进行加法操作。
        fi
        ((sum=sum+$size))                   #<== 令所获取的字节做加法，并赋值给 sum。
    done
    echo "${1}:total:${sum}bytes =`echo $((${sum}/1024))`KB"   #<== 循环完毕后，打
印结果。
```

> 提示：此题的 Shell 实现不是最佳的方法，通过 awk 可以更快地实现。awk 的实现方法就留给大家思考了。

执行结果如下：

```
    [root@oldboy scripts]# sh 10_8_1.sh access_2010-12-8.log   #<== 这个日志文件可
以是任意的 Apache 或 Nginx 等访问的日志文件。
    access_2010-12-8.log:total:226905bytes =221KB
```

下面补充非 while 循环的其他两种方法供读者参考：

```
    [root@oldboy scripts]# awk '{print $10}' access_2010-12-8.log|grep -v
"-"|awk '{sum+=$1}END{print sum}'
    226905
```

10.5 while 循环按行读文件的方式总结

解答完范例 10-8 后，相信大家对 while 循环按行读文件的方式已经有了基本的了解，下面就来总结 while 循环按行读文件的几种常见方式。

方式 1：采用 exec 读取文件，然后进入 while 循环处理。

```
    exec <FILE
    sum=0
    while read line
    do
        cmd
    done
```

方式 2：使用 cat 读取文件内容，然后通过管道进入 while 循环处理。

```
    cat FILE_PATH|while read line
```

```
do
    cmd
done
```

方式 3：在 while 循环结尾 done 处通过输入重定向指定读取的文件。

```
while read line
do
    cmd
done<FILE
```

范例 10-9：开发一个 Shell 脚本实现 Linux 系统命令 cat 读文件的基本功能。

参考答案：

```
[root@oldboy scripts]# cat 10_9_1.sh
while read line
do
    echo $line
done<$1
```

执行结果如下：

```
[root@oldboy scripts]# sh 10_9_1.sh /etc/hosts
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.
localdomain4
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6
```

10.6　企业级生产高级实战案例

范例 10-10：写一个 Shell 脚本解决类 DDoS 攻击的生产案例。请根据 Web 日志或系统网络连接数，监控某个 IP 的并发连接数，若短时内 PV 达到 100，即调用防火墙命令封掉对应的 IP。防火墙命令为：“iptables -I INPUT -s IP 地址 -j DROP”。

参考答案 1：

先分析 Web 日志，可以每分钟或每小时分析一次，这里给出按小时处理的方法。可以将日志按小时进行分割，分成不同的文件，根据分析结果把 PV 数高的单 IP 封掉。例如，每小时单 IP 的 PV 数超过 500，则即刻封掉，这里简单地把日志的每一行近似看作一个 PV，实际工作中需要计算实际页面的数量，而不是请求页面元素的数量，另外，很多公司都是以 NAT 形式上网的，因此每小时单 IP 的 PV 数超过多少就会被封掉，还要根据具体的情况具体分析，本题仅给出一个实现的案例，读者使用时需要考虑自身网站的业务去使用。

解答如下：

```
[root@oldboy scripts]# cat 10_10_1.sh
file=$1                    #<== 定义一个变量接收命令行传参，参数为日志文件类型。
```

```
while true
do
    awk '{print $1}' $1|grep -v "^$"|sort|uniq -c >/tmp/tmp.log
    #<== 分析传入的日志文件，并在排序去重后追加到一个临时文件里。
    exec </tmp/tmp.log   #<== 读取上述临时文件。
    while read line      #<== 进入 while 循环处理。
    do
        ip=`echo $line|awk '{print $2}'`        #<== 获取文件中的每一行的第二列。
        count=`echo $line|awk '{print $1}'`     #<== 获取文件中的每一行的第一列。
        if [ $count -gt 500 ] && [ `iptables -L -n|grep "$ip"|wc -l` -lt 1 ]
        #<== 如果 PV 数大于 500，并且防火墙里没有封过此 IP。
          then
            iptables -I INPUT -s $ip -j DROP     #<== 则封掉 PV 数大于 500 的 IP。
            echo "$line is dropped" >>/tmp/droplist_$(date +%F).log
                                                 #<== 记录处理日志。
        fi
    done
    sleep 3600  #<== 读者可以按分钟进行分析，不过日志的分割或过滤也得按分钟才行。
done
```

说明：为了方便测试，可以将超过就封的 PV 数调低，检测频率也可调快一些。

执行结果如下：

```
[root@oldboy scripts]# sh 10_10_1.sh access.log
```

单独打开窗口查看 iptables 的情况，结果如下：

```
[root@oldboy ~]# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  59.33.26.105         0.0.0.0/0     #<== 这就是脚本封掉的 IP。
DROP       all  --  124.115.4.18         0.0.0.0/0     #<== 这就是脚本封掉的 IP。

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

参考答案 2：

分析 Linux 系统的网络连接数，而不是分析 Web 日志。

设计思路：

首先要分析单 IP 占网络连接数的情况，即取当前网络连接状态为 ESTABLISHED 的行数，然后分析对应客户端列不同 IP 连接数量的排序，对排序比较高的 IP 进行封堵。

以下是待用的模拟数据：

```
[root@oldboy scripts]# sed -n '20,30p' netstat.log
tcp    0    0    115.29.49.213:80      120.237.97.10:54195     ESTABLISHED
tcp    0    0    115.29.49.213:80      49.80.146.230:13453     FIN_WAIT2
tcp    0    0    115.29.49.213:80      113.104.25.50:56714     FIN_WAIT2
tcp    0    0    115.29.49.213:80      101.226.89.193:41639    ESTABLISHED
tcp    0    0    115.29.49.213:80      119.147.225.185:58321   TIME_WAIT
tcp    0    0    115.29.49.213:80      54.183.177.237:64129    TIME_WAIT
tcp    0    0    115.29.49.213:80      120.198.202.48:41960    ESTABLISHED
tcp    0    1    115.29.49.213:80      119.127.188.242:38843   FIN_WAIT1
tcp    0    0    115.29.49.213:34081   223.4.9.70:80           TIME_WAIT
tcp    0    0    115.29.49.213:80      58.223.4.14:46716       FIN_WAIT2
tcp    0    0    115.29.49.213:80      122.90.74.255:12177     FIN_WAIT2
```

实现对应客户端列不同的 IP 连接数量排序的方法有：

```
[root@oldboy scripts]# grep "ESTABLISHED" netstat.log|awk -F "[ :]+"
'{print $(NF-3)}'|sort|uniq -c|sort -rn -k1|head -5
      4 118.242.18.177
      3 123.6.8.223
      3 114.250.252.127
      2 123.244.104.42
      2 121.204.108.160
[root@oldboy scripts]# grep "ESTABLISHED" netstat.log|awk -F "[ :]+" '{
++S[$(NF-3)]}END {for(key in S) print S[key], key}'|sort -rn -k1|head -5
      4 118.242.18.177
      3 123.6.8.223
      3 114.250.252.127
      2 123.244.104.42
      2 121.204.108.160
```

在命令行中采用如下命令封掉 IP：

```
iptables -I INPUT -s 121.204.108.160 -j DROP
```

让程序每 3 分钟执行一次，这里使用 while 守护进程的方式来实现。

参考答案 1：

```
[root@oldboy scripts]# cat 10_10_2.sh
#!/bin/sh
file=$1              #<== 定义一个变量以接收命令行传参，参数为日志文件类型。实际工作中可以
先使用 netstat 命令将网络状态生成临时文件，然后再传参给 $1。
if expr "$file" : ".*\.log" &>/dev/null #<==expr 的用法，判断扩展名是否以 .log 结尾。
  then
    :                                   #<== 这个冒号表示什么都不做。
else
    echo $"usage:$0 xxx.log"            #<== 对不符合扩展名类型的给出正确提示。
    exit 1                              #<== 退出脚本。
```

```
    fi

    while true
    do
        grep "ESTABLISHED" $1|awk -F "[ :]+" '{ ++S[$(NF-3)]}END {for(key in S)
print S[key], key}'|sort -rn -k1|head -5 >/tmp/tmp.log  #<== 分析传入的网络状态日志，
获取到客户端 IP 列的信息，并去重排序。
        while read line                              #<== 读取去重后的 /tmp/tmp.log 文件。
        do
            ip=`echo $line|awk '{print $2}'`     #<== 获取文件中的每一行的第二列。
            count=`echo $line|awk '{print $1}'` #<== 获取文件中的每一行的第一列。
            if [ $count -gt 500 ] && [ `iptables -L -n|grep "$ip"|wc -l` -lt 1 ]
            #<== 如果 PV 数大于 500，并且防火墙里没有封过此 IP，可通过设置小的值测试。
              then
                iptables -I INPUT -s $ip -j DROP  #<== 则封掉 PV 数大于 500 的 IP。
                echo "$line is dropped" >>/tmp/droplist_$(date +%F).log
            #<== 记录处理日志。
            fi
        done</tmp/tmp.log    #<== 读入临时日志文件，即使是放在结尾，也是在进入循环的时候
就读入了。
        sleep 180
    done
```

> 提示：对于分钟级别的频率任务，也可以不用 while，而用定时任务来实现。

参考答案 2：更专业的脚本（分模块实现）

```
    [root@oldboy scripts]# cat 10_10_3.sh
    #!/bin/sh
    file=$1
    JudgeExt(){                  #<== 定义判断扩展名的函数，可以传入日志文件。
        if expr "$1" : ".*\.log" &>/dev/null
          then
            :
        else
            echo $"usage:$0 xxx.log"
            exit 1
        fi
    }
    IpCount(){                   #<== 分析日志，对访问的 IP 去重排序。
    grep "ESTABLISHED" $1|awk -F "[ :]+" '{ ++S[$(NF-3)]}END {for(key in S)
print S[key], key}'|sort -rn -k1|head -5 >/tmp/tmp.log
    }
    ipt(){                       #<== 防火墙封堵函数。
        local ip=$1
        if [ `iptables -L -n|grep "$ip"|wc -l` -lt 1 ]
          then
```

```
        iptables -I INPUT -s $ip -j DROP
        echo "$line is dropped" >>/tmp/droplist_$(date +%F).log
    fi
}
main(){                              #<== 主函数 main。
    JudgeExt $file                   #<== 传入日志判断扩展名。
    while true
    do
        IpCount $file                #<== 对传入日志文件排序去重。
        while read line
        do
            ip=`echo $line|awk '{print $2}'`
            count=`echo $line|awk '{print $1}'`
            if [ $count -gt 3 ]  #<== 去重后检测，若某 IP 超过 3 次，则封堵，实际工作中
                                     可以调整成你需要的阈值。
              then
                ipt $ip          #<== 封掉对应的 IP。
            fi
        done</tmp/tmp.log
        sleep 180
    done
}
main
```

其他实战题目见“天津项目实践抓阄题目”：
http://oldboy.blog.51cto.com/2561410/1308647。

10.7　本章小结

（1）While 循环结构及相关语句综合实践小结

- while 循环的特长是执行守护进程，以及实现我们希望循环持续执行不退出的应用，适合用于频率小于 1 分钟的循环处理，其他的 while 循环几乎都可以被后面即将要讲到的 for 循环及定时任务 crond 功能所替代。
- case 语句可以用 if 语句来替换，而在系统启动脚本时传入少量固定规则字符串的情况下，多用 case 语句，其他普通判断多用 if 语句。
- 一句话场景下，if 语句、for 语句最常用，其次是 while（守护进程）、case（服务启动脚本）。

（2）Shell 脚本中各个语句的使用场景

- 条件表达式，用于简短的条件判断及输出（文件是否存在，字符串是否为空等）
- if 取值判断，多用于不同值数量较少的情况。
- for 最常用于正常的循环处理中。

- while 多用于守护进程、无限循环（要加 sleep 和 usleep 控制频率）场景。
- case 多用于服务启动脚本中，打印菜单可用 select 语句，不过很少见，一般用 cat 的 here 文档方法来替代。

函数的作用主要是使编码逻辑清晰，减少重复语句开发。

特别说明：可访问如下地址或手机扫二维码查看第 10 章的核心脚本代码。

http://oldboy.blog.51cto.com/2561410/1855442

Linux

第11章

for 和 select 循环语句的应用实践

for 循环语句和 while 循环语句类似，但 for 循环语句主要用于执行次数有限的循环，而不是用于守护进程及无限循环。for 循环语句常见的语法有两种，下面将在不同的语法中对 for 循环语句进行详尽的讲解。

11.1 for 循环语法结构

第一种 for 循环语句为变量取值型，语法结构如下：

```
for 变量名 in 变量取值列表
do
    指令 ...
done
```

提示：在此结构中“ in 变量取值列表”可以省略，省略时相当于 in“ $@”，也就是使用 for i 就相当于使用 for i in“ $@”。

在这种 for 循环语句语法中，for 关键字后面会有一个“变量名”，变量名依次获取 in 关键字后面的变量取值列表内容（以空格分隔），每次仅取一个，然后进入循环（do 和 done 之间的部分）执行循环内的所有指令，当执行到 done 时结束本次循环。之后，“变量名”再继续获取变量列表里的下一个变量值，继续执行循环内的所有指令，当执行

到 done 时结束返回，以此类推，直到取完变量列表里的最后一个值并进入循环执行到 done 结束为止。

下面给出变量取值型 for 循环语句的形象记忆方法。

```
for 男人 in 世界上所有男人
do
    if [ 有房 ] && [ 有车 ] && [ 存款 ] && [ 会做家务 ] && [ 帅气 ] && [ 体贴 ]
&& [ 逛街买东西 ];then
        echo "女孩喜欢这个男人"
    else
        rm -f $男人（不符合条件的）
    fi
done
```

这里 for 关键字之后的“男人”就是变量，“世界上所有男人”是“男人”这个变量的取值列表范围，就是把每个男人作为变量值，分别进入 for 循环运行一遍，符合条件（if 配合 for）的女孩就喜欢，否则就删除他。

第二种 for 循环语句称为 C 语言型 for 循环语句，其语法结构如下：

```
for((exp1; exp2; exp3))
do
    指令 ...
done
```

说明：此种循环语句和 while 循环语句类似，但语法结构比 while 循环更规范、工整。

for 关键字后的双括号内是三个表达式，第一个是变量初始化（例如：i=0），第二个为变量的范围（例如：i<100），第三个为变量自增或自减（例如：i++）。当第一个表达式的初始化值符合第二个变量的范围时，就进入循环执行；当条件不满足时就退出循环。

for 循环结构执行流程对应的逻辑图如图 11-1 所示。

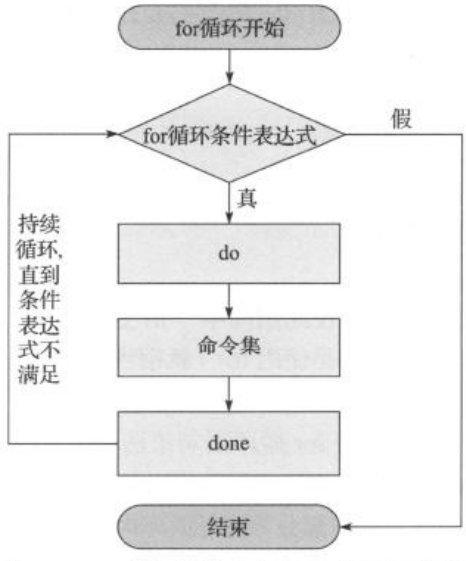


图 11-1 for 循环结构执行流程对应的逻辑图

范例 11-1：for 和 while 循环的对比。

先来看 for 循环语句：

```
[root@oldboy scripts]# cat 11_1_1.sh
for ((i=1;i<=3;i++))
do
    echo $i
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_1_1.sh
1
2
3
```

以上 for 循环语句可以改为前面第 10 章介绍的 while 循环语句，如下：

```
[root@oldboy scripts]# cat 11_1_2.sh
i=1
while ((i<=3))
do
    echo $i
    ((i++))
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_1_2.sh
1
2
3
```

特别说明：

1）如果希望程序持续运行，则多用 while，包括守护进程。

2）如果是有限次循环，则多用 for，实际工作中使用 for 的机会更多。

11.2　for 循环语句的基础实践

下面是几个 for 循环语句的示例。

范例 11-2：竖向打印 5、4、3、2、1 这 5 个数字。

参考答案 1：直接列出元素的方法。

```
[root@oldboy scripts]# cat 11_2_1.sh
#直接列出变量列表的所有元素，打印 5、4、3、2、1
for num in 5 4 3 2 1  #==>提示：5 4 3 2 1 需要用空格隔开。
do
    echo $num
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_2_1.sh
5
4
3
```

```
2
1
```

参考答案 2：利用大括号 {} 生成数字序列的方法。

```
[root@oldboy scripts]# echo {5..1}
5 4 3 2 1
[root@oldboy scripts]# cat 11_2_2.sh
for n in {5..1}        #<== 实质上也相当于列表。
do
    echo $n
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_2_2.sh
5
4
3
2
1
```

参考答案 3：采用 seq 生成数字序列的用法（这里先简略介绍，后文有细讲）。

```
[root@oldboy scripts]# cat 11_2_3.sh
for n in `seq 5 -1 1` #<==5 是起始数字，-1 是步长，即每次减一，1 是结束数字。
do
    echo $n
done
```

执行结果省略。

范例 11-3：获取当前目录下的目录或文件名，并将其作为变量列表打印输出。

模拟数据如下：

```
[root@oldboy scripts]# mkdir -p /test/{test.txt,oldboy.txt,oldgirl.txt}
[root@oldboy scripts]# ls -l /test
总用量 12
drwxr-xr-x 2 root root 4096 9月   5 09:46 oldboy.txt
drwxr-xr-x 2 root root 4096 9月   5 09:46 oldgirl.txt
drwxr-xr-x 2 root root 4096 9月   5 09:46 test.txt
```

实现代码如下：

```
[root@oldboy scripts]# cat 11_3_1.sh
cd /test
for filename in `ls`  #<== 列表当前的所有文件，注意命令应用反引号括起来。
do
    echo $filename
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_3_1.sh
oldboy.txt
oldgirl.txt
test.txt
```

范例 11-4：用 for 循环批量修改文件扩展名（把 txt 改成 jpg）。

测试数据如下：

```
[root@oldboy scripts]# mkdir -p /test
[root@oldboy scripts]# touch /test/{test.txt,oldboy.txt,oldgirl.txt}
[root@oldboy scripts]# ls -l /test
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:31 oldboy.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 oldgirl.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 test.txt
```

做此类题要有程序设计思维，可先在命令行实现通过变量的方式对一个文件进行改名，然后在脚本中批量处理就容易了。注意，要采用通用的方法，而不仅仅是命令。下面的命令可实现对文件进行改名：

```
[root@oldboy scripts]# cd /test
[root@oldboy test]# ls -l
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:31 oldboy.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 oldgirl.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 test.txt
[root@oldboy test]# filename=oldboy.txt          #<== 将一个文件名赋值给 filename。
[root@oldboy test]# echo $filename
oldboy.txt
[root@oldboy test]# echo $filename|cut -d . -f1 #<== 取出文件名部分（排除扩展名）。
oldboy
[root@oldboy test]# echo "`echo $filename|cut -d . -f1`.gif"
#<== 将最终需要更改的文件名和扩展名拼接起来。
oldboy.gif
[root@oldboy test]# mv $filename `echo $filename|cut -d . -f1`.gif
#<== 通过变量的方式实现改名，这里的方法就是通用的方法，即不针对任何一个文件，后面的批量改名可以直接拿到循环里来处理。
[root@oldboy test]# ls -lrt
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:31 test.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 oldgirl.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 oldboy.gif  #<== 成功修改。
```

然后使用 for 循环脚本批量处理，代码如下：

```
[root@oldboy scripts]# ls -lrt /test
```

```
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:31 test.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 oldgirl.txt
-rw-r--r-- 1 root root 0 9月   5 10:31 oldboy.gif
```

脚本实现如下：

```
[root@oldboy scripts]# cat 11_4_1.sh
#!/bin/sh
cd /test
for filename in `ls|grep "txt$"`  #<== 获取当下目录中的所有文件名，作为取值列表。
do
    mv $filename `echo $filename|cut -d . -f1`.gif  #<== 这个就是上面命令行的
                                                        命令（未做任何修改的）。
done
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_4_1.sh
[root@oldboy scripts]# ls -lrt /test
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:31 test.gif
-rw-r--r-- 1 root root 0 9月   5 10:31 oldgirl.gif
-rw-r--r-- 1 root root 0 9月   5 10:31 oldboy.gif
```

实际上，本题还有更简单的实现方法，即通过 rename 命令来直接实现，如下：

```
[root@oldboy scripts]# cd /test
[root@oldboy test]# ls -l
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:31 oldboy.gif
-rw-r--r-- 1 root root 0 9月   5 10:31 oldgirl.gif
-rw-r--r-- 1 root root 0 9月   5 10:31 test.gif
[root@oldboy test]# rename "gif" "txt" *.gif
#<== rename 是专业的改名工具，在老男孩的命令类图书里会讲解此命令。
[root@oldboy test]# ll
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:36 oldboy.gif
-rw-r--r-- 1 root root 0 9月   5 10:36 oldgirl.gif
-rw-r--r-- 1 root root 0 9月   5 10:31 test.gif
```

11.3 for 循环语句的企业级案例

范例 11-5：在 Linux 下批量修改文件名，将图 11-2 所示命令中的“_finished”去掉。

提示：通过此题的解答可以学习到 sed、awk、rename、mv 等命令的实战应用。

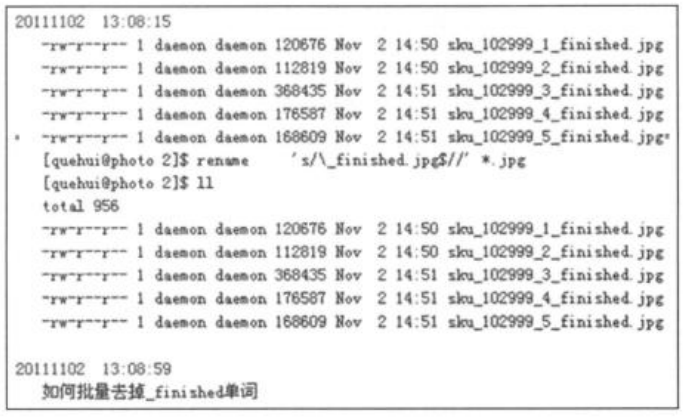


图 11-2　2011 年来自网友的问题

本题的基本解题思路和范例 11-4 类似，先进行单个文件的改名，然后再用循环实现批量改名，这也是最常规的做法，当然，还可以用专业的改名工具 rename 来处理（本节主要是学习 for 循环知识）。

准备测试数据，如下：

```
[root@oldboy test]# mkdir /oldboy
[root@oldboy test]# cd /oldboy
[root@oldboy oldboy]# touch stu_102999_1_finished.jpg stu_102999_2_
finished.jpg stu_102999_3_finished.jpg
[root@oldboy oldboy]# touch stu_102999_4_finished.jpg stu_102999_5_
finished.jpg
[root@oldboy oldboy]# ls -l
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_1_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_2_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_3_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_4_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_5_finished.jpg
```

以下是脚本实现方法。

参考答案 1：采用 Shell 脚本、for 循环加 sed 的方法。

以下命令可用于检查数据并对单个文件实现改名。

```
[root@oldboy oldboy]# ls -l
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_1_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_2_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_3_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_4_finished.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_5_finished.jpg
[root@oldboy oldboy]# file=stu_102999_1_finished.jpg
```

```
[root@oldboy oldboy]# echo $file
stu_102999_1_finished.jpg
[root@oldboy oldboy]# echo $file|sed 's/_finished//g'
stu_102999_1.jpg #<== 也可以用变量子串替换的方法，例如：echo "${file%_finished*}.jpg。
[root@oldboy oldboy]# mv $file `echo $file|sed 's/_finished//g'`
[root@oldboy oldboy]# ls -l stu_102999_1.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_1.jpg
```

最终的开发脚本如下：

```
[root@oldboy scripts]# cat 11_5_1.sh
#!/bin/sh
cd /oldboy
for file in `ls *.jpg`
do
    mv $file `echo $file|sed 's/_finished//g'` #<== 使用 mv 命令更改文件，拼接新的
                                                    文件名字符串是本题的重点。
done
```

执行结果如下：

```
[root@oldboy scripts]# ll /oldboy
总用量 0
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_1.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_2.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_3.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_4.jpg
-rw-r--r-- 1 root root 0 9月   5 10:43 stu_102999_5.jpg
```

参考答案 2：使用 ls 结合 awk 实现，这个方法中没有 for 循环，但它可以在很多场景中替换 for 循环。

这种解法将留给读者自行尝试，不过在此之前，为了测试方便，可以先利用 ls 结合 awk 的方式对上面的文件进行数据恢复，即都加上“_finished”，如下：

```
[root@oldboy oldboy]# ls
stu_102999_1.jpg   stu_102999_2.jpg   stu_102999_3.jpg   stu_102999_4.jpg
stu_102999_5.jpg
[root@oldboy oldboy]# ls|awk -F "." '{print $0,$1,$2}'
stu_102999_1.jpg stu_102999_1 jpg
stu_102999_2.jpg stu_102999_2 jpg
stu_102999_3.jpg stu_102999_3 jpg
stu_102999_4.jpg stu_102999_4 jpg
stu_102999_5.jpg stu_102999_5 jpg
[root@oldboy oldboy]# ls|awk -F "." '{print $0,$1"_finished."$2}'
stu_102999_1.jpg stu_102999_1_finished.jpg
stu_102999_2.jpg stu_102999_2_finished.jpg
stu_102999_3.jpg stu_102999_3_finished.jpg
stu_102999_4.jpg stu_102999_4_finished.jpg
```

```
    stu_102999_5.jpg stu_102999_5_finished.jpg
    [root@oldboy oldboy]# ls|awk -F "." '{print "mv",$0,$1"_finished."$2}'
    #==> 拼成了 mv 修改命令字符串。
    mv stu_102999_1.jpg stu_102999_1_finished.jpg
    mv stu_102999_2.jpg stu_102999_2_finished.jpg
    mv stu_102999_3.jpg stu_102999_3_finished.jpg
    mv stu_102999_4.jpg stu_102999_4_finished.jpg
    mv stu_102999_5.jpg stu_102999_5_finished.jpg
    [root@oldboy oldboy]# ls|awk -F "." '{print "mv",$0,$1"_finished."$2}'|bash
    #==> 拼成了 mv 修改命令字符串后，交给 bash 执行。
    [root@oldboy oldboy]# ls
    stu_102999_1_finished.jpg   stu_102999_3_finished.jpg   stu_102999_5_
finished.jpg
    stu_102999_2_finished.jpg  stu_102999_4_finished.jpg
```

提示：注意是对文件名的修改，而不是字符串的修改。

参考答案 3：通过专业的改名命令 rename 来实现。

```
    [root@oldboy oldboy]# ls|awk -F "." '{print "mv",$0,$1"_finished."$2}'|bash
    #<== 执行具体修改。
    [root@oldboy oldboy]# ls
    stu_102999_1_finished.jpg   stu_102999_3_finished.jpg   stu_102999_5_
finished.jpg
    stu_102999_2_finished.jpg  stu_102999_4_finished.jpg
    [root@oldboy oldboy]# rename "_finished" "" *.jpg #<== 注意不要落了双引号。
    [root@oldboy oldboy]# ls
    stu_102999_1.jpg   stu_102999_2.jpg   stu_102999_3.jpg   stu_102999_4.jpg
stu_102999_5.jpg
```

范例 11-6：在生产环境下，批量去掉测试数据所用的 bd 字符（此为老男孩在生产环境中碰到的案例）。

当时的数据如下：

```
    [root@bigBD001 errorfiles]# ll
    total 16
    -rw-r--r-- 1 root root 1426 Nov 29 11:05 bd502.html
    -rw-r--r-- 1 root root 1426 Nov 29 11:05 bd503.html
    -rw-r--r-- 1 root root 1426 Nov 29 11:05 bd504.html
```

实现命令如下：

```
    [root@bigBD001 errorfiles]# rename "bd" "" *.html
    [root@bigBD001 errorfiles]# ll
    total 16
    -rw-r--r-- 1 root root 1426 Nov 29 11:05 502.html
```

```
-rw-r--r-- 1 root root 1426 Nov 29 11:05 503.html
-rw-r--r-- 1 root root 1426 Nov 29 11:05 504.html
```

提示：对于这个案例，完全可以用 mv 命令逐个去改，但是为了秉承“在使用中记忆”的思想，所以还是用了批量修改的方法。

范例 11-7：通过脚本实现仅 sshd、rsyslog、crond、network、**sysstat 服务**在开机时自启动。

在设置前，先来查看默认情况下开机时 Linux 系统开启的服务有哪些，由于通常工作在文本模式 3 级别，因此只需要查找 3 级别上开启的服务即可。查看命令如下：

```
[root@oldboy ~]# LANG=en  #<== 先调整成英文字符集，以方便在下面的命令中过滤中文字符串。
[root@oldboy ~]# chkconfig --list|grep 3:on
... 省略 ...
crond       0:off  1:off  2:on   3:on   4:on   5:on   6:off #<== 这是要保留的。
haldaemon   0:off  1:off  2:off  3:on   4:on   5:on   6:off
... 省略 ...
netfs       0:off  1:off  2:off  3:on   4:on   5:on   6:off
network     0:off  1:off  2:on   3:on   4:on   5:on   6:off #<== 这是要保留的。
postfix     0:off  1:off  2:on   3:on   4:on   5:on   6:off
rsyslog     0:off  1:off  2:on   3:on   4:on   5:on   6:off #<== 这是要保留的。
sshd        0:off  1:off  2:on   3:on   4:on   5:on   6:off #<== 这是要保留的。
sysstat     0:off  1:on   2:on   3:on   4:on   5:on   6:off #<== 这是要保留的。
udev-post   0:off  1:on   2:on   3:on   4:on   5:on   6:off
```

提示：可以看到，默认情况下开启了很多服务，我们需要保留开启的所有服务也包含在其中。这里只需要关注 3 级别上的设置是否为 on 即可（on 为开启状态），有关运行级别的知识请查阅相关资料或老男孩的其他文章。

了解了系统在 3 级别上开启的服务之后，就可以通过命令快速实现配置了，下面就来正式介绍几种通过命令或脚本设置开机自启动的方法。

参考答案 1：先将 3 级别文本模式下默认开启的服务都关闭，然后开启需要开启的服务。

操作命令如下：

```
LANG=en
for oldboy in `chkconfig --list|grep 3:on|awk '{print $1}'`;do chkconfig
--level 3 $oldboy off;done
for oldboy in crond network rsyslog sshd sysstat ;do chkconfig --level 3
$oldboy on;done
chkconfig --list|grep 3:on
```

操作过程如下：

```
    [root@oldboy ~]# LANG=en#<== 临时调整字符集为英文。
    [root@oldboy ~]# for oldboy in `chkconfig --list|grep 3:on|awk '{print
$1}'`;do chkconfig --level 3 $oldboy off;done      #<== 关掉所有开启的服务。
    [root@oldboy ~]# for oldboy in crond network rsyslog sshd sysstat; do
chkconfig --level 3 $oldboy on;done
    <== 开启需要开启的服务。
    [root@oldboy ~]# chkconfig --list|grep 3:on    #<== 查看设置结果。
    crond           0:off   1:off   2:on    3:on    4:on    5:on    6:off
    network         0:off   1:off   2:on    3:on    4:on    5:on    6:off
    rsyslog         0:off   1:off   2:on    3:on    4:on    5:on    6:off
    sshd            0:off   1:off   2:on    3:on    4:on    5:on    6:off
    sysstat         0:off   1:off   2:off   3:on    4:off   5:off   6:off
```

参考答案 2：通过 Shell 循环实现。

默认情况下开机需要保留的服务都已经是开启状态了，因此，只需要把 3 级别文本模式下已开启但不需要开启的服务都关掉就好了。

操作过程如下：

```
    [root@oldboy ~]# for oldboy in `chkconfig --list|grep "3:on"|awk '{print
$1}'|grep -vE "crond|network|sshd|rsyslog|sysstat"`;do chkconfig $oldboy
off;done
    [root@oldboy ~]# chkconfig --list|grep 3:on
    crond           0:off   1:off   2:on    3:on    4:on    5:on    6:off
    network         0:off   1:off   2:on    3:on    4:on    5:on    6:off
    rsyslog         0:off   1:off   2:on    3:on    4:on    5:on    6:off
    sshd            0:off   1:off   2:on    3:on    4:on    5:on    6:off
    sysstat         0:off   1:off   2:off   3:on    4:off   5:off   6:off
```

参考答案 3：不用 Shell 循环语句，就用一条命令实现。

默认情况下开机需要保留的服务都已经是开启状态了，因此，只需要把 3 级别文本模式下已开启但不需要开启的服务都关掉，这里将不用循环结构而是利用命令拼出所有要处理的命令字符串，然后通过 bash 将其当作命令执行即可。

操作命令如下：

```
    chkconfig --list|grep 3:on|grep -vE "crond|sshd|network|rsyslog|sysstat"
|awk '{print "chkconfig " $1 " off"}'|bash
```

操作过程为先拼接所有要操作的命令字符串：

```
    [root@oldboy ~]# chkconfig --list|grep 3:on|grep -vE "crond|sshd|network|r
syslog|sysstat" |awk '{print "chkconfig " $1 " off"}'
    chkconfig abrt-ccpp off
    chkconfig abrtd off
    chkconfig acpid off
    chkconfig atd off
```

```
chkconfig auditd off
chkconfig blk-availability off
chkconfig cpuspeed off
chkconfig haldaemon off
chkconfig htcacheclean off
```

然后将拼接得到的所有要操作的命令字符串通过 bash 运行，如下：

```
[root@oldboy ~]# chkconfig --list|grep 3:on|grep -vE "crond|sshd|network|r
syslog|sysstat" |awk '{print "chkconfig " $1 " off"}'|bash
```

上述方法可以简化为下面的方法：

```
[root@oldboy ~]# chkconfig|egrep -v "crond|sshd|network|rsyslog|sysstat"|a
wk '{print "chkconfig",$1,"off"}'|bash
```

或用下面的命令：

```
[root@oldboy ~]# chkconfig --list|grep 3:on|grep -vE "crond|sshd|network|r
syslog|sysstat" |awk '{print $1}'|sed -r 's#(.*)#chkconfig \1 off#g'|bash
```

范例 11-8：打印九九乘法表，实现图形如图 11-3 所示。

图 11-3　九九乘法表效果图

这是一个 for 循环嵌套的使用案例，实现代码如下：

```
[root@oldboy scripts]# cat 11_8_1.sh
#!/bin/bash
COLOR='\E[47;30m'          #<== 定义一个和题意要求相似的背景颜色。
RES='\E[0m'

for num1 in `seq 9`        #<== 外层 for 循环，乘法的第一个乘数范围为 1 ~ 9。
do
    for num2 in `seq 9` #<== 内层 for 循环，乘法的第二个乘数范围为 1 ~ 9。
    do
```

```
        if [ $num1 -ge $num2 ]    #<== 如果第一个乘数大于等于第二个乘数。
          then
            if (((num1*num2)>9)) #<== 如果两个数相乘大于 9，这是控制输出格式的。
              then
                echo -en "${COLOR}${num1}x${num2}=$((num1*num2))$RES "
                                  #<== 除了输出外，结尾还多了一个空格。
            else
                echo -en "${COLOR}${num1}x${num2}=$((num1*num2))$RES  "
                                  #<== 除了输出外，结尾多了两个空格。
            fi
        fi
    done
echo " "
done
```

执行结果如图 11-4 所示。

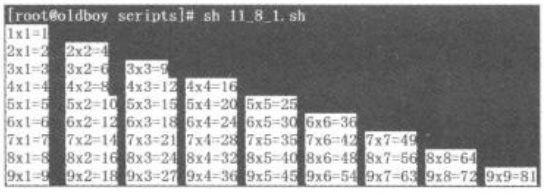


图 11-4　执行脚本后的九九乘法表效果图

范例 11-9：计算从 1 加到 100 之和（用 C 语言型的 for 循环结构实现）。

参考答案 1（用 for 循环实现）：

```
for((i=1;i<=100;i++))
do
    ((sum=sum+i))
done
echo $sum
```

参考答案 2（以上 for 循环语句的功能，用下面的 while 脚本同样可以实现）：

```
i=0
while ((i<=100))
do
    ((j=j+i))
    ((i++))
done
```

提示：一般的 for 循环和 while 循环可以互相转换，可以实现同样的功能。

这个方法比较直接但效率不佳，此类计算用数学公式会更快，尤其在数字很大的情况下，如下：

```
[root@oldboy scripts]# echo $(( (1+100) * 100/2 ))
5050
```

范例 11-10：每隔两秒访问一次 http://www.baidu.com，一共访问 5 次。

这里用 curl 实现对上述地址的访问。

```
for((i=0; i<5; i++))
do
    curl http://www.baidu.com
done
```

11.4 for 循环语句的企业高级实战案例

范例 11-11：实现 MySQL 分库备份的脚本。

基本的批量建库脚本如下，这里使用 for 循环在数据库服务器里批量创建数据库。

```
[root@oldboy scripts]# cat 11_11_1.sh
#!/bin/sh
PATH="/application/mysql/bin:$PATH"                    #<== 定义 mysql 命令所在路径。
MYUSER=root                                            #<== 定义数据库用户名。
MYPASS=oldboy123                                       #<== 定义数据库用户密码。
SOCKET=/data/3306/mysql.sock                           #<== 定义数据库 sock 文件。
MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"           #<== 定义登录数据库的命令。
for dbname in oldboy oldgirl xiaoting bingbing         #<== 要创建的数据库列表。
do
    $MYCMD -e "create database $dbname"                #<== 创建数据库的命令。
done
```

说明：不登录数据库创建数据库的命令为 mysql -uroot -poldboy123 -S /data/3306/mysql.sock -e "create database oldboy;"。

分库备份数据库（即每个库一个文件）的命令如下：

```
[root@oldboy scripts]# cat 11_11_2.sh
#!/bin/sh
PATH="/application/mysql/bin:$PATH"                    #<== 定义 mysql 命令所在路径。
DBPATH=/server/backup
MYUSER=root                                            #<== 定义数据库用户名。
MYPASS=oldboy123                                       #<== 定义数据库用户密码。
SOCKET=/data/3306/mysql.sock                           #<== 定义数据库 sock 文件。
MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"           #<== 定义登录数据库的命令。
MYDUMP="mysqldump -u$MYUSER -p$MYPASS -S $SOCKET"      #<== 备份数据库的核心命令部分。
[ ! -d "$DBPATH" ] && mkdir $DBPATH                    #<== 创建备份路径。
```

```
    for dbname in `$MYCMD -e "show databases;"|sed '1,2d'|egrep -v
"mysql|schema"`     #<== 登录数据库获取数据库里的所有数据库名。
    do
        $MYDUMP $dbname|gzip >$DBPATH/${dbname}_$(date +%F).sql.gz
                    #<== 对获取的数据库名循环备份。
    done
```

说明：备份数据库的命令为 mysqldump -uroot -poldboy123 -S /data/3306/mysql.sock oldboy |gzip>/server/backup/oldboy_$(date +%F).sql.gz。

执行结果如下：

```
    [root@oldboy scripts]# sh 11_11_2.sh
    [root@oldboy scripts]# ll /server/backup/
    total 16
    -rw-r--r-- 1 root root 451 Sep  5 12:08 bingbing_2016-09-05.sql.gz
    #<== 备份结果。
    -rw-r--r-- 1 root root 450 Sep  5 12:08 oldboy_2016-09-05.sql.gz
    -rw-r--r-- 1 root root 451 Sep  5 12:08 oldgirl_2016-09-05.sql.gz
    -rw-r--r-- 1 root root 455 Sep  5 12:08 xiaoting_2016-09-05.sql.gz
```

范例 11-12：实现 MySQL 分库分表备份的脚本。

准备测试数据：通过写脚本批量建表并插入数据。

```
    [root@oldboy scripts]# cat 11_12_1.sh
    #!/bin/sh
    PATH="/application/mysql/bin:$PATH"                    #<== 定义 mysql 命令所在路径。
    MYUSER=root                                            #<== 定义数据库用户名。
    MYPASS=oldboy123                                       #<== 定义数据库用户密码。
    SOCKET=/data/3306/mysql.sock                           #<== 定义数据库 sock 文件。
    MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"           #<== 定义登录数据库的命令。
    for dbname in oldboy oldgirl xiaoting bingbing
    do
        $MYCMD -e "use $dbname;create table test(id int,name varchar(16)); insert
into test values(1,'testdata');"                           #<== 批量建表及插入数据。
    done
```

使用如下脚本查看测试数据结果：

```
    [root@oldboy scripts]# cat 11_12_2.sh
    #!/bin/sh
    PATH="/application/mysql/bin:$PATH"                    #<== 定义 mysql 命令所在路径。
    MYUSER=root                                            #<== 定义数据库用户名。
    MYPASS=oldboy123                                       #<== 定义数据库用户密码。
    SOCKET=/data/3306/mysql.sock                           #<== 定义数据库 sock 文件。
    MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"           #<== 定义登录数据库的命令。
    for dbname in oldboy oldgirl xiaoting bingbing
```

```
    do
        echo ===============${dbname}.test====================
        $MYCMD -e "use $dbname;select * from ${dbname}.test;" #<== 批量查看数据。
    done
```

查看到的结果如下：

```
    [root@oldboy scripts]# sh 11_12_2.sh
    ===============oldboy.test====================
    +------+----------+
    | id   | name     |
    +------+----------+
    |    1 | testdata |
    +------+----------+
    ===============oldgirl.test====================
    +------+----------+
    | id   | name     |
    +------+----------+
    |    1 | testdata |
    +------+----------+
    ===============xiaoting.test====================
    +------+----------+
    | id   | name     |
    +------+----------+
    |    1 | testdata |
    +------+----------+
    ===============bingbing.test====================
    +------+----------+
    | id   | name     |
    +------+----------+
    |    1 | testdata |
    +------+----------+
```

以下是本题真正的解答方案，实现的脚本代码如下：

```
    [root@oldboy scripts]# cat 11_12_3.sh
    #!/bin/sh
    PATH="/application/mysql/bin:$PATH"                    #<== 定义 mysql 命令所在路径。
    DBPATH=/server/backup
    MYUSER=root                                            #<== 定义数据库用户名。
    MYPASS=oldboy123                                       #<== 定义数据库用户密码。
    SOCKET=/data/3306/mysql.sock                           #<== 定义数据库 sock 文件。
    MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"           #<== 定义登录数据库的命令。
    MYDUMP="mysqldump -u$MYUSER -p$MYPASS -S $SOCKET"
    [ ! -d "$DBPATH" ] && mkdir $DBPATH                    #<== 创建备份路径。
    for dbname in `$MYCMD -e "show databases;"|sed '1,2d'|egrep -v
"mysql|schema"`    #<== 登录数据库获取数据库里的所有数据库名。
    do
        mkdir $DBPATH/${dbname}_$(date +%F) -p             #<== 创建对应目录。
```

```
        for table in `$MYCMD -e "show tables from $dbname;"|sed '1d'`
        #<== 内层循环，获取每个库里的所有表，然后进入循环。
        do
        $MYDUMP $dbname $table|gzip >$DBPATH/${dbname}_$(date +%F)/${dbname}_
${table}.sql.gz
        #<== 备份指定的库内的表到指定目录下，并以库表名字命名备份的名字。
        done
    done
```

执行结果如下：

```
[root@oldboy scripts]# rm -f /server/backup/*
[root@oldboy scripts]# sh 11_12_3.sh
[root@oldboy scripts]# LANG=en
[root@oldboy scripts]# tree /server/backup/
/server/backup/
|-- bingbing_2016-09-05
|   `-- bingbing_test.sql.gz
|-- oldboy_2016-09-05
|   `-- oldboy_test.sql.gz
|-- oldgirl_2016-09-05
|   `-- oldgirl_test.sql.gz
`-- xiaoting_2016-09-05
    `-- xiaoting_test.sql.gz
4 directories, 4 files
```

范例 11-13：在生产环境下批量检查 Web 服务是否正常，并且发送相关邮件或手机报警信息。

```
[root@oldboy scripts]# cat 11_13_1.sh
#!/bin/bash
path=/server/scripts  #<== 定义脚本存放路径，大家需要注意这个规范。
MAIL_GROUP="1111@qq.com 2222@qq.com"    #<== 邮件列表，以空格隔开。
PAGER_GROUP="18600338340 18911718229"   #<== 手机列表，以空格隔开。
LOG_FILE="/tmp/web_check.log"           #<== 日志路径。

[ ! -d "$path" ] && mkdir -p $path      #<== 创建目录。
function UrlList(){                     #<== URL 列表函数。
    cat >$path/domain.list<<EOF #<== 由于还没有学习数组，这里先将所有 URL 地址放入文件里。
    http://blog.oldboyedu.com
    http://oldboy.blog.51cto.com
    http://10.0.0.7
    http://www.baidu.com
EOF
}
function CheckUrl(){                    #<== 检测 URL 的函数。
    FAILCOUNT=0                         #<== 初始化失败的次数为 0 次。
    for ((i=1;$i<=3;i++))               #<== 检测 3 次。
    do
```

```
        wget -T 5 --tries=1 --spider $1 >/dev/null 2>&1
                                        #<== 具体的访问 URL 的命令，不输出信息。
        if [ $? -ne 0 ]                 #<== 返回值如果不为 0，则表示访问 URL 失败了。
          then
            let FAILCOUNT+=1;           #<== 将失败的次数加 1。
        else
            break #<== 如果返回值为 0，则表示访问 URL 成功了，跳出 for 循环，不做 3 次检测了。
        fi
    done
    return $FAILCOUNT       #<== 将失败次数作为返回值，返回函数外的脚本中。
}

function MAIL(){            #<== 定义邮件函数。
    local SUBJECT_CONTENT=$1  #<== 将函数的第一个传参赋值给主题变量。
    for MAIL_USER  in `echo $MAIL_GROUP`                    #<== 遍历邮件列表。
    do
        mail -s "$SUBJECT_CONTENT " $MAIL_USER <$LOG_FILE #<== 发邮件。
    done
}
function PAGER(){           #<== 定义手机函数。
    for PAGER_USER  in `echo $PAGER_GROUP`                  #<== 遍历手机列表。
    do
        TITLE=$1                    #<== 函数的第一个传参赋值给主题变量。
        CONTACT=$PAGER_USER         #<== 手机号赋值给 CONTACT 变量。
        HTTPGW=http://oldboy.sms.cn/smsproxy/sendsms.action
        #<== 发短信地址，这个地址需要用户付费购买，如果想要免费就得用 139 或微信替代了。
        #send_message method1
        curl -d  cdkey=5ADF-EFA -d password=OLDBOY -d phone=$CONTACT -d message=
"$TITLE[$2]" $HTTPGW
        #<== 发送短信报警的命令。cdkey 是购买短信网关时，由售卖者提供的，password 是密码，
            也是由售卖者提供的。
    done
}
function SendMsg(){         #<== 定义发送消息的函数。
    if [ $1 -ge 3 ]         #<== 如果失败的次数大于等于 3，那么这里的 $1 是函数的传参，
                                接收访问 URL 失败的次数。
      then
        RETVAL=1
        NOW_TIME=`date +"%Y-%m-%d %H:%M:%S"`            #<== 报警时间。
        SUBJECT_CONTENT="http://$2 is error,${NOW_TIME}."#<== 报警主题。
        echo -e "$SUBJECT_CONTENT"|tee $LOG_FILE #<== 输出信息，并记录到日志。
        MAIL $SUBJECT_CONTENT #<== 发邮件报警，$SUBJECT_CONTENT 将作为函数参数传给
                                  MAIL 函数体的 $1。
        PAGER $SUBJECT_CONTENT $NOW_TIME
                              #<== 发短信报警，$SUBJECT_CONTENT 将作为函数参数传给
                                  MAIL 函数体的 $1，$NOW_TIME 作为函数体传给 $2。
    else #<== 如果失败的次数不大于 3，则认为 URL 是好的。
        echo "http://$2 is ok" #<== 打印 ok。
        RETVAL=0 #<== 以 0 作为返回值。
```

```
    fi
    return $RETVAL
}
function main(){          #<== 定义主函数。
    UrlList               #<== 加载 URL 列表。
    for url in `cat $path/domain.list` #<== 读取 URL 列表文件。
    do
        CheckUrl $url     #<== 传入 URL 给检测 URL 的函数进行检查。
        SendMsg $? $url #<== 传入第一个参数 "$?", 即 CheckUrl 里的返回值 (用检测失败的
                              次数作为返回值), 传入的第二个参数为检测的 URL。
    done
}
main                      #<== 整个脚本的执行导火索。
```

范例 11-14：批量创建 10 个系统账号（oldboy01-oldboy10），并设置密码（密码为随机数，要求是字符和数字的混合）[⊖]。

参考答案 1：先根据题意要求，理清开发思路。

1）创建 10 个系统账号，即 oldboy01~oldboy10。

对于给一个数字加 0 有多种实现方法，这里给出两种，其他方法见老男孩的博客。

方法 1：

```
[root@oldboy scripts]# seq -w 10
01
02
03
04
05
06
07
08
09
10
```

方法 2：

```
[root@oldboy scripts]# echo {01..10}
01 02 03 04 05 06 07 08 09 10
```

2）要想通过脚本创建账号，必须知道如何实现无交互设置密码，如下：

```
[root@oldboy scripts]# useradd oldgirl
[root@oldboy scripts]# echo 123456|passwd --stdin oldgirl
Changing password for user oldgirl.
passwd: all authentication tokens updated successfully
```

3）密码为随机数，并且是 8 位字符串，这是一个难点。

⊖ 不用 for 循环的实现思路可参见 http://user.qzone.qq.com/49000448/blog/1422183723。

实现随机数的方法也很多，这里先给出常见的 RANDOM 方法，如下：

```
[root@oldboy scripts]# echo $RANDOM
29671
```

但是，这样得到的随机数不符合题意，因此，可以采用 md5sum 进行加密的方式再取 8 位，如下：

```
[root@oldboy scripts]# echo $RANDOM|md5sum
28d8fd390daf7fef596da82774bc14f3  -
[root@oldboy scripts]# echo $RANDOM|md5sum|cut -c 5-12
85b623a2
```

最后是实现相应的脚本。由于是批量创建 10 个账号并设置密码，因此需要使用 for 循环。

先来看一个带陷阱的错误解答方法：

```
#!/bin/sh
rm -f /tmp/user.log
for i in `seq -w 10`
do
    useradd oldboy$i && \
    echo "echo $RANDOM|md5sum|cut -c 1-8"|passwd --stdin oldboy$i
    echo -e "user:oldboy$i \t pass:`echo $RANDOM|md5sum|cut -c 1-8`" >>/tmp/
user.log
done
```

上述脚本中用了两次随机数，如果不将其定义成变量，就会出现执行看起来相同的命令两次但是结果却不同的情况。

下面是一个输出不够美观但结果正确的解题方法：

```
[root@oldboy scripts]# cat 11_14_1.sh
#!/bin/sh
#author:oldboy
#blog:http://oldboy.blog.51cto.com
user="oldboy"
passfile="/tmp/user.log"
for num in `seq -w 10`
do
    useradd $user$num #<== 创建用户。
    pass="`echo "test$RANDOM"|md5sum|cut -c3-11`"    #<== 若多次用到随机数，
                                                        就要将其定义成变量。
    echo "$pass"|passwd --stdin $user$num #<== 设置密码。
    echo  -e "user:$user$num\tpasswd:$pass">>$passfile #<== 记录设置的账号和
                                                        密码信息。
done
echo ------this is oldboy trainning class contents----------------
cat $passfile
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_14_1.sh
Changing password for user oldboy01.
passwd: all authentication tokens updated successfully.
Changing password for user oldboy02.
passwd: all authentication tokens updated successfully.
Changing password for user oldboy03.
passwd: all authentication tokens updated successfully.
... 省略若干 ...
Changing password for user oldboy10.
passwd: all authentication tokens updated successfully.
------this is oldboy trainning class contents----------------
user:oldboy01   passwd:46c81f181
user:oldboy02   passwd:45d57f377
... 省略若干 ...
user:oldboy10   passwd:7ca06a695
```

通过下面的命令可测试账号的可用性：

```
[root@oldboy scripts]# su - oldboy01
[oldboy01@oldboy ~]$ su - oldboy02
密码：     #<== 敲入密码。
[oldboy02@oldboy ~]$ logout
```

下面的解题方法会调用系统函数库让输出更美观，同时增强逻辑性：

```
[root@oldboy scripts]# cat 11_14_2.sh
#!/bin/sh
#author:oldboy
#blog:http://oldboy.blog.51cto.com
. /etc/init.d/functions
user="oldboy"
passfile="/tmp/user.log"
for num in `seq -w 11 15`
do
    pass="`echo "test$RANDOM"|md5sum|cut -c3-11`"
    useradd $user$num &>/dev/null &&\
    echo "$pass"|passwd --stdin $user$num &>/dev/null &&\ #<== 注意这里的“&&”符。
    echo  -e "user:$user$num\tpasswd:$pass">>$passfile
    if [ $? -eq 0 ] #<== 根据返回值判断用户和密码是否添加成功。
      then
        action "$user$num is ok" /bin/true              #<== 优雅地显示。
    else
        action "$user$num is fail" /bin/false           #<== 优雅地显示。
    fi
done
echo ----------------------
cat $passfile && >$passfile
```

执行结果如下：

```
[root@oldboy scripts]# sh 11_14_2.sh
oldboy11 is ok                                             [  OK  ]
oldboy12 is ok                                             [  OK  ]
oldboy13 is ok                                             [  OK  ]
oldboy14 is ok                                             [  OK  ]
oldboy15 is ok                                             [  OK  ]
----------------------
user:oldboy11   passwd:4afe6e5e9
user:oldboy12   passwd:26ed64dc8
user:oldboy13   passwd:ae20945bb
user:oldboy14   passwd:d94812b76
user:oldboy15   passwd:b7a8359d5
```

提示：注意随机数的字符串要定义成变量，否则，每次执行结果都会不相同。

参考答案 2：

1）按照参考答案 1 的思路正常创建账号。

2）要批量创建密码，可使用 chpasswd 来实现，chpasswd 是一个批量更新用户口令的工具。

chpasswd 的使用示例如下：

```
[root@oldboy scripts]# useradd oldgirl01
[root@oldboy scripts]# echo "oldgirl01:123456"|chpasswd
[root@oldboy scripts]# su - oldboy
[oldboy@oldboy ~]$ su - oldgirl01
密码：
[oldgirl01@oldboy ~]$ whoami
oldgirl01
```

给多个用户设置密码的命令为：

```
chpasswd < 密码文件
```

但密码文件的内容必须以下面的格式来书写，并且不能有空行；

```
用户名 1：口令 1
用户名 2：口令 2
```

说明：用户必须存在。

最后实现的脚本如下：

```
[root@oldboy scripts]# cat 11_14_3.sh
#!/bin/sh
```

```
#author:oldboy
#blog:http://oldboy.blog.51cto.com
. /etc/init.d/functions
user="xiaoting"
passfile="/tmp/user.log"
for num in `seq -w 10`
do
    pass="`echo "test$RANDOM"|md5sum|cut -c3-11`"
    useradd $user$num &>/dev/null &&\
    echo  -e "$user${num}:$pass">>$passfile #<== 生成密码到文件，但并没有设置密码。
    if [ $? -eq 0 ]
      then
        action "$user$num is ok" /bin/true
    else
        action "$user$num is fail" /bin/false
    fi
done
echo ----------------------
chpasswd < $passfile #<== 这里是关键，作用是读取密码文件进行密码设置。
cat $passfile && >$passfile
```

11.5　Linux 系统产生随机数的 6 种方法

下面介绍 Linux 系统产生随机数的 6 种方法。

方法 1：通过系统环境变量（$RANDOM）实现，示例代码如下。

```
[root@oldboy scripts]# echo $RANDOM
30492
[root@oldboy scripts]# echo $RANDOM
4021
```

RANDOM 的随机数范围为 0 ~ 32767，因此，加密性不是很好，可以通过在输出的随机数后增加加密字符串（就是和密码生成有关的一个字符串）的方式解决，最后再一起执行 md5sum 操作并截取结果的后 n 位，这样一来，就无法根据随机数范围 0 ~ 32767 来猜出具体结果了。

示例：

```
[root@oldboy scripts]# echo "oldboy$RANDOM"|md5sum|cut -c 8-15
#<== 这里的 oldboy 就是上文提到的加密字符串，虽然 RANDOM 可以破解，但是只要无人知道你增加的
oldboy 字符串，就无法破解下面的字符串，破解 RANDOM 并进行 md5sum 操作的例子在后文有详细讲解。
91be8254
```

方法 2：通过 openssl 产生随机数，示例代码如下。

```
[root@oldboy scripts]# openssl rand -base64 8
F0hRoLu9o8c=
[root@oldboy scripts]# openssl rand -base64 80
```

```
Q6EzRQfqdvTBIF6W+1ARi8auIZOEp73NOBo38phak5syEsNKUGAzNrUKQvMJjiFq
RFcvd7ExfofD1ho844iX3XGlesgdnDTP2kbUUIHID30=
```

令数字与大小写字符相结合，并且带上特殊字符，可以达到很长的位数，这样的随机数很安全。

方法 3：通过时间（date）获得随机数，示例代码如下。

```
[root@oldboy scripts]# date +%s%N
1473061480765110440
[root@oldboy scripts]# date +%s%N
1473061481595654564
```

方法 4：通过 /dev/urandom 配合 chksum 生成随机数。

示例代码如下：

```
[root@oldboy scripts]# head /dev/urandom|cksum
1595867971 3433
[root@oldboy scripts]# head /dev/urandom|cksum
2594498471 1700
```

/dev/random 设备存储着系统当前运行环境的实时数据。它可以看作系统在某个时候的唯一值，因此可以用作随机数元数据。我们可以通过文件读取的方式，读到里面的数据。/dev/urandom 这个设备的数据与 random 里的一样。只是，它是非阻塞的随机数发生器，读取操作不会产生阻塞。

方法 5：通过 UUID 生成随机数。

示例代码如下：

```
[root@oldboy scripts]# cat /proc/sys/kernel/random/uuid
54b63594-98f3-4f41-b50f-3c152dce170e
[root@oldboy scripts]# cat /proc/sys/kernel/random/uuid
3cf5e2fe-32dd-4378-af09-cf668a7acd38
```

UUID 码全称是通用唯一识别码（Universally Unique Identifier，UUID），它是一个软件建构的标准，亦为自由软件基金会（Open Software Foundation，OSF）的组织在分布式计算环境（Distributed Computing Environment，DCE）领域的一部分。

UUID 的目的是让分布式系统中的所有元素都能有唯一的辨识信息，而不需要通过中央控制端来做辨识信息的指定。如此一来，每个人都可以创建不与其他人发生冲突的 UUID。在这样的情况下，就不需要考虑数据库创建时的名称重复问题了。它会让网络中任何一台计算机所生成的 UUID 码都是互联网整个服务器网络中唯一的编码。它的原信息会加入硬件、时间、机器当前运行信息等。

方法 6：使用 expect 附带的 mkpasswd 生成随机数。

mkpasswd 命令依赖于数据包 expect，因此必须通过“ yum install expect –y”命令先安装该数据包：

```
[root@oldboy scripts]# mkpasswd -l 9 -d 2 -c 3 -C 3 -s 1
pHKtjK(53
[root@oldboy scripts]# mkpasswd -l 9 -d 2 -c 3 -C 3 -s 1
1TrQwrP0:
[root@oldboy scripts]# mkpasswd -l 9 -d 2 -c 3 -C 3 -s 1
53fMTh~Gp
```

相关参数说明如下：

```
-l #    (length of password, default = 9)        #<== 指定密码长度。
-d #    (min # of digits, default = 2)           #<== 指定密码中数字的数量。
-c #    (min # of lowercase chars, default = 2) #<== 指定密码中小写字母的数量。
-C #    (min # of uppercase chars, default = 2) #<== 指定密码中大写字母的数量。
-s #    (min # of special chars, default = 1)    #<== 指定密码中特殊字符的数量。
```

上面的随机数长短不一，如何统一格式化呢？解答：使用 md5sum 命令。

示例如下：

```
[root@oldboy scripts]# mkpasswd -l 9 -d 2 -c 3 -C 3 -s 1|md5sum|cut -c 2-10
d81978b70
[root@oldboy scripts]# cat /proc/sys/kernel/random/uuid|md5sum|cut -c 2-10
292127444
[root@oldboy scripts]# head /dev/urandom|cksum|md5sum|cut -c 2-10
1834f4da9
[root@oldboy scripts]# date +%s%N|md5sum|cut -c 2-10
552008eba
[root@oldboy scripts]# openssl rand -base64 80|md5sum|cut -c 2-10
8a7eff744
[root@oldboy scripts]# echo "test$RANDOM"|md5sum|cut -c 2-10
5ca8306f2
```

11.6　select 循环语句介绍及语法

在第 6 章范例 6-36 中通过菜单选择实现了企业业务自动化部署，当时采用的生成菜单的方法就是 cat 方法（被称为 here 文档），这里给大家介绍另外一种实现菜单的方法，即通过 select 循环语句实现。

select 循环语句的主要作用可能就是创建菜单，在执行带 select 循环语句的脚本时，输出会按照数字顺序的列表显示一个菜单项，并显示提示符（默认是 #？），同时等待用户输入数字进行选择，下面就来带大家看看生成菜单项的语法及具体案例实践。

第一种 for 循环语句为变量取值型，语法结构如下：

```
select 变量名 [ in 菜单取值列表 ]
do
    指令...
done
```

提示：bash 帮助语法显示：select name [in word]; do list; done
在此结构中“ in 变量取值列表”可省略，省略时相当于使用 in “ $@”，使用 for i 就相当于使用 for i in “$@”。

在这种 select 循环语句的语法中，在执行脚本后，select 关键字后面会有一个“变量名”，变量名依次获取 in 关键字后面的变量取值列表内容（以空格分隔），每次仅取一个，然后进入循环（do 和 done 之间），执行循环内的所有指令，当执行到 done 时结束返回，之后，“变量名”再继续取变量列表里的下一个变量值，继续执行循环内的所有指令（do 和 done 之间的指令），当执行到 done 时结束返回，以此类推，直到取完最后一个变量列表里的值并进入循环执行到 done 结束为止。与 for 循环不同的是，select 循环执行后会出现菜单项等待用户选择（不会自动循环所有变量列表），而用户输入的只能是菜单项前面的数字序号，每输入一次对应的序号就会执行一次循环，直到变量后面对应列表取完为止。

select 循环结构执行流程对应的逻辑图如图 11-5 所示。

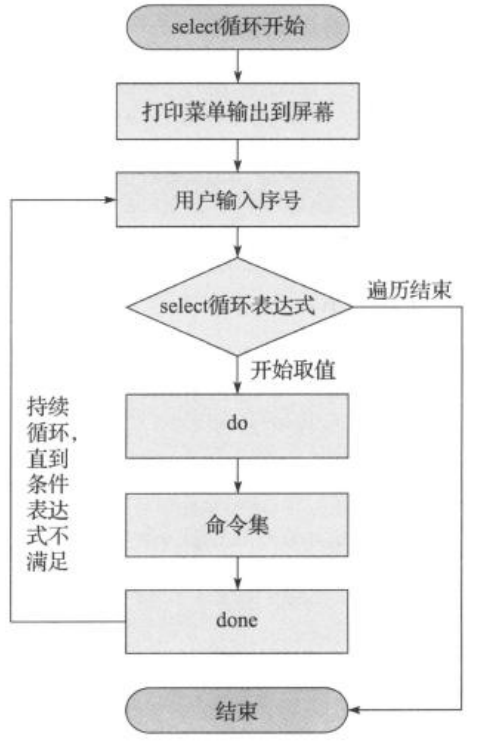


图 11-5 select 循环结构执行流程对应的逻辑图

11.7 select 循环语句案例

范例 11-15：用 select 循环打印简单菜单项的多种实现方法

方法 1：直接使用列表字符串。

```
[root@oldboy scripts]# cat 11_15_1.sh
#!/bin/bash
#Author:oldboy training
select name in oldboy oldgirl tingting #<==name 变量将遍历后面的以空格分隔的字符串。
do
    echo $name #<== 当选择对应菜单项前面的数字时，即打印对应的菜单项内容。
done
```

执行结果如下：

执行脚本后打印带数字序列（数字加右小括号）的菜单项，内容就是变量列表的内容。

```
[root@oldboy scripts]# sh 11_15_1.sh
1) oldboy
2) oldgirl
3) tingting
#? 1        #<== 这里必须是输入序号，不能是变量列表内容，例如 oldboy。
oldboy      #<== 输入对应序号，返回对应菜单项内容。
#? 2        #<== 这里必须是输入序号，不能是变量列表内容，例如 oldboy。
oldgirl     #<== 输入对应序号，返回对应菜单项内容。
#? 3        #<== 这里必须是输入序号，不能是变量列表内容，例如 oldboy。
tingting    #<== 输入对应序号，返回对应菜单项内容。
#? test     #<== 输入错误，则返回空。

#?          #<== 默认的提示符为 $? 号，不够优雅，后面的例子将换掉它。
```

方法 2：采用数组做变量列表。

```
[root@oldboy scripts]# cat 11_15_2.sh
#!/bin/bash
array=(oldboy oldgirl tingting)
select name in "${array[@]}"
do
    echo $name
done
```

方法三：把命令结果作为变量列表（菜单项）。

（1）数据准备

```
[root@oldboy scripts]# mkdir -p /tmp/test
[root@oldboy scripts]# mkdir -p /tmp/test/{oldboy,oldgirl,tingting}
[root@oldboy scripts]# ls -l /tmp/test/
总用量 12
drwxr-xr-x 2 root root 4096 11月  1 11:35 oldboy
drwxr-xr-x 2 root root 4096 11月  1 11:35 oldgirl
drwxr-xr-x 2 root root 4096 11月  1 11:35 tingting
```

（2）脚本开发

```
[root@oldboy scripts]# cat 11_15_3.sh
#!/bin/bash
#Author:oldboy training
select name in `ls /tmp/test`
do
    echo $name
done
```

（3）执行结果

```
[root@oldboy scripts]# sh 11_15_3.sh
1) oldboy
2) oldgirl
3) tingting
#?
```

提示：细心的读者可以看到变量列表部分和 for 循环是一样的。

通过上一个范例我们了解到，select 循环菜单项的默认提示很不友好，并且输入的是数字，打印的变量值却是数字对应的菜单项内容。那能不能针对默认提示符以及打印输入内容进行调整呢？当然可以，且看下面的案例。

范例 11-16：调整 select 循环菜单项的默认提示符及利用 select 变量打印数字序号。

开发脚本如下：

```
[root@oldboy scripts]# cat 11_15_4.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
PS3="please select a num from menu:"      #<== PS3 就是控制 select 循环的提示符，
                                              这可是新知识呦！
select name in oldboy oldgirl tingting
do
    echo -e "I guess you selected the menu is:\n $REPLY) $name"
                                          #<==REPLY 变量就是菜单项对应的数字。
done
```

本范例重点讲解了 select 循环的两个特殊变量，其中 PS3 系统环境变量用于控制 select 循环的提示符，REPLY 变量用于获取菜单项对应的数字，也就是用户输入的数字。

以下为执行演示。

```
[root@oldboy scripts]# sh 11_15_4.sh
1) oldboy
2) oldgirl
3) tingting
please select a num from menu:1
I guess you selected the menu is:
 1) oldboy
please select a num from menu:2
I guess you selected the menu is:
 2) oldgirl
please select a num from menu:3
I guess you selected the menu is:
 3) tingting
please select a num from menu:^C
```

范例 11-17：打印选择菜单，按照选择一键安装不同的 Web 服务。

示例菜单：

```
[root@oldboy scripts]# sh menu.sh
    1.[install lamp]
    2.[install lnmp]
    3.[exit]
    pls input the num you want:
```

要求：

1）当用户输入 1 时，输出“start installing lamp.”提示，然后执行 /server/scripts/lamp.sh，输出 "lamp is installed" 后退出脚本，这就是实际工作中所用的 lamp 一键安装脚本；

2）当用户输入 2 时，输出“start installing lnmp.”提示，然后执行 /server/scripts/lnmp.sh，输出 "lnmp is installed" 后退出脚本，这就是实际工作中所用的 lnmp 一键安装脚本；

3）当输入 3 时，退出当前菜单及脚本；

4）当输入任何其他字符时，给出提示“Input error”后退出脚本；

5）要对执行的脚本进行相关的条件判断，例如：脚本文件是否存在，是否可执行等判断，尽量用上前面讲解的知识点。

解答：本范例和范例 6-36 是一道题，但是采用的实现方法却完全不同，本例将采用 select 循环结构实现范例 6-36 脚本的升级版，读者可看看能否先不看答案自己实现本例。

参考解答脚本 1：

```
[root@oldgirl scripts]# cat 11_15_5.sh
#!/bin/sh
RETVAR=0
path=/server/scripts                          #<== 定义脚本路径。
[ ! -d "$path" ]  && mkdir $path -p           #<== 如果路径不存在，就创建。
function Usage(){                             #<== 定义帮助函数。
    echo "Usage:$0 argv"
    return 1
}
function InstallService(){                    #<== 定义安装服务函数。
    if [ $# -ne 1 ];then                      #<== 参数不等于 1，就打印帮助函数。
        Usage
    fi
    local RETVAR=0                            #<== 初始化返回值。
    echo "start installing ${1}."             #<== 打印开始安装服务，传参 $1，$1 是
                                                  函数的参数，本例即 lamp 或 lnmp。
    sleep 2;
    if [ ! -x "$path/${1}.sh" ];then #<== 如果安装服务脚本不可执行，则给出提示后退出。
```

```
        echo "$path/${1}.sh does not exist or can not be exec."
        retrun 1
    else
        $path/${1}.sh                    #<== 执行脚本。
        return $RETVAR                   #<== 返回值返回函数体外。
    fi
}
function main(){                         #<== 主函数。
    PS3="`echo pls input the num you want:`" #<== 菜单提示。
    select var in "Install lamp" "Install lnmp" "exit"
    #<==select 循环，菜单内容列表，列表中有空格就要加引号。
    do
        case "$var" in                   #<== 接收变量值进行匹配。
            "Install lamp")              #<== 如果变量值为 "Install lamp",
                InstallService lamp      #<== 调用安装服务函数，安装 lamp 服务。
                RETVAR=$?                #<== 将脚本执行结果返回函数体外。
                ;;
            "Install lnmp")              #<== 如果变量值为 "Install lnmp",
                InstallService lnmp      #<== 调用安装服务函数，安装 lnmp 服务。
                RETVAR=$?                #<== 将脚本执行结果返回函数体外。
                ;;
            exit)                        #<== 如果变量值为 exit,
                echo bye.                #<== 打印 bye.
                return 3                 #<== 携带返回值 3 返回函数体外。
                ;;
            *)                           #<== 如果变量值为其他字符，打印如下提示。
                echo "the num you input must be {1|2|3}"
                echo "Input ERROR"       #<== 打印提示。
        esac
    done
exit $ RETVAR
}
main                                     #<== 调用 main 函数，执行总的程序。
```

参考解答脚本 2：这个脚本实现更简单，脚本的差异已标出，其他相同部分不再进行注释。

```
[root@oldboy scripts]# cat 11_15_6.sh
#!/bin/sh
RETVAR=0
path=/server/scripts
[ ! -d "$path" ]  && mkdir $path -p
function Usage(){
    echo "Usage:$0 argv"
    return 1
}
function InstallService(){
    if [ $# -ne 1 ];then
        Usage
```

```
    fi
    local RETVAR=0
    echo "start installing ${1}."
    sleep 2;
    if [ ! -x "$path/${1}.sh" ];then
       echo "$path/${1}.sh does not exist or can not be exec."
        retrun 1
    else
        $path/${1}.sh
        return $RETVAR
    fi
}
function main(){
    PS3="`echo pls input the num you want:`"
    select var in "Install lamp" "Install lnmp" "exit"
    do
        case "$REPLY" in #<== 使用获取 select 循环对应的数字序列的环境变量。
            1)          #<== 如果匹配 1，则执行下面指令，到双分号结束。
                InstallService lamp
                RETVAR=$?
                ;;
            2)          #<== 如果匹配 2，则执行下面指令，到双分号结束。
                InstallService lnmp
                RETVAR=$?
                ;;
            3)          #<== 如果匹配 3，则执行下面指令，到双分号结束。
                echo bye.
                return 3
                ;;
            *)
                echo "the num you input must be {1|2|3}"
                echo "Input ERROR"
        esac
    done
exit $ RETVAR
}
main
```

执行结果如下：

```
[root@oldgirl scripts]# sh 11_15_5.sh
1) Install lamp
2) Install lnmp
3) exit
pls input the num you want:1
start installing lamp.
install lamp
pls input the num you want:2
```

```
start installing lnmp.
install lnmp
pls input the num you want:3
bye.
```

特别说明：可访问如下地址或手机扫二维码查看第 11 章的核心脚本代码

http://oldboy.blog.51cto.com/2561410/1855427

Linux

第12章

循环控制及状态返回值的应用实践

本章将带领大家学习以下几个特殊的命令：break（循环控制）、continue（循环控制）、exit（退出脚本）、return（退出函数）。

12.1 break、continue、exit、return 的区别和对比

在上述命令中，break、continue 在条件语句及循环语句（for、while、if 等）中用于控制程序的走向；而 exit 则用于终止所有语句并退出当前脚本，除此之外，exit 还可以返回上一次程序或命令的执行状态值给当前 Shell；return 类似于 exit，只不过 return 仅用于在函数内部返回函数执行的状态值。关于这几个命令的基本说明如表 12-1 所示。

表 12-1 条件与循环控制及程序返回值命令知识表

命　令	说　明
break n	如果省略 n，则表示跳出整个循环，n 表示跳出循环的层数
continue n	如果省略 n，则表示跳过本次循环，忽略本次循环的剩余代码，进入循环的下一次循环。n 表示退到第 n 层继续循环
exit n	退出当前 Shell 程序，n 为上一次程序执行的状态返回值。n 也可以省略，在下一个 Shell 里可通过“$?”接收 exit n 的 n 值
return n	用于在函数里作为函数的返回值，以判断函数执行是否正确。在下一个 Shell 里可通过“$?”接收 exit n 的 n 值

12.2 break、continue、exit 功能执行流程图

为了让读者更清晰地了解上述命令的区别，下面特别画了逻辑图，方便大家理解。

这里以 while 循环和 for 循环为例来说明。

在循环中 break 功能的执行流程逻辑图如图 12-1 所示。

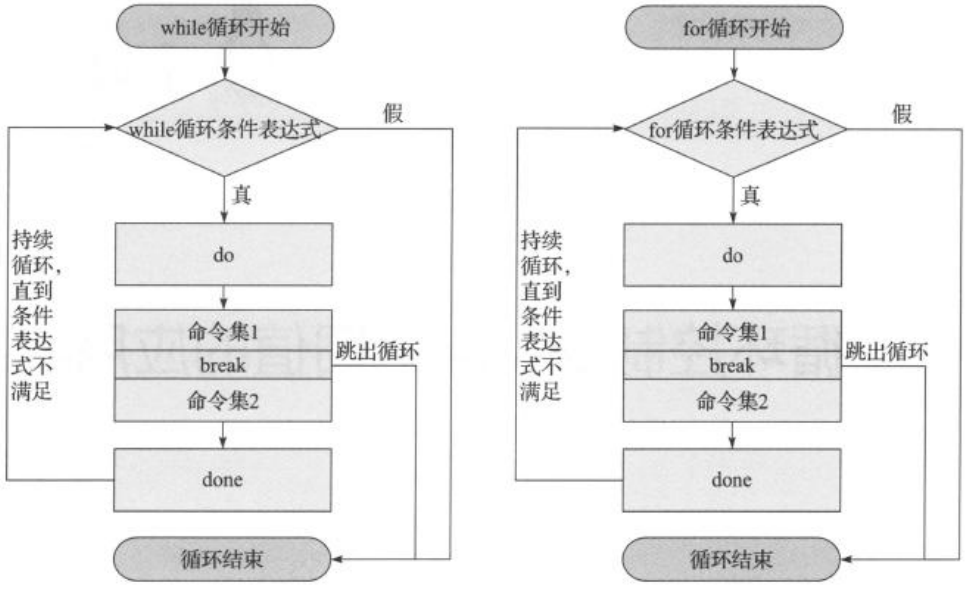


图 12-1　while 循环和 for 循环中 break 的功能执行流程逻辑

在循环中 continue 功能的执行流程逻辑图如图 12-2 所示。

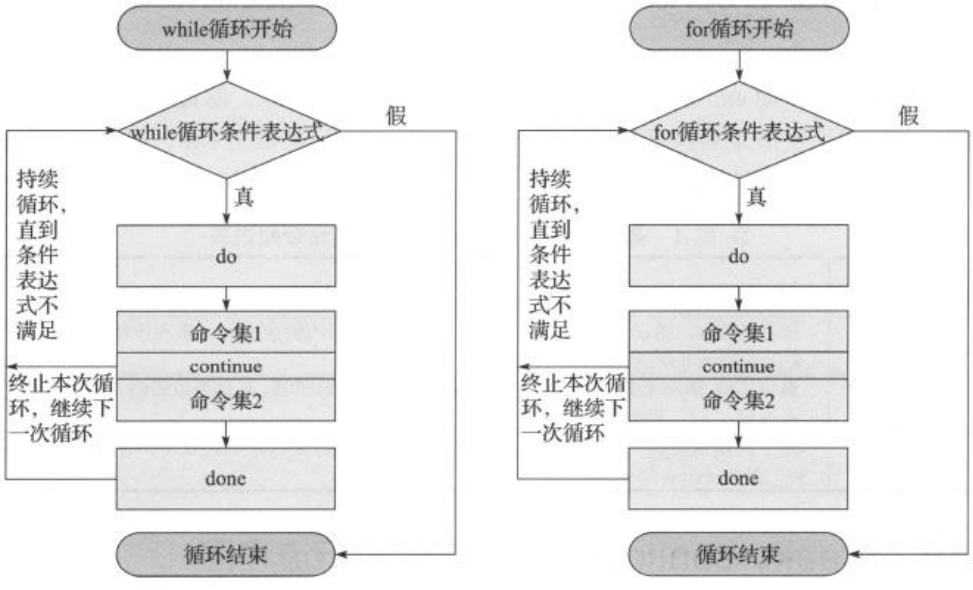


图 12-2　while 循环和 for 循环中 continue 的功能执行流程逻辑

在循环中 exit 功能的执行流程逻辑图如图 12-3 所示。

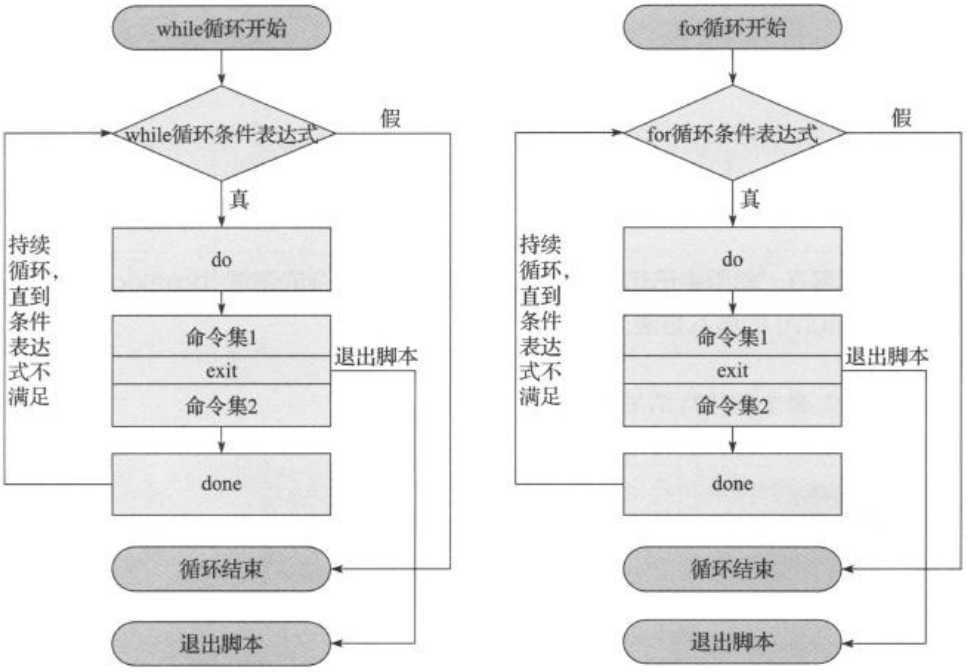


图 12-3　while 循环和 for 循环中 exit 的功能执行流程逻辑

12.3　break、continue、exit、return 命令的基础示例

下面是与 break、continue、exit、return 相关的示例。

范例 12-1：通过 break 命令跳出整个循环，执行循环下面的其他程序。

```
[root@oldboy scripts]# cat 12_1_1.sh
#!/bin/bash
if [ $# -ne 1 ];then      #<== 如果传参个数不为 1，则打印下面的使用提示给用户。
    echo $"usage:$0 {break|continue|exit|return}"  #<== 分别传入 4 个命令作为参数。
    exit 1                #<== 退出脚本。
fi
test(){                   #<== 定义测试函数。
    for((i=0; i<=5; i++))
    do
        if [ $i -eq 3 ] ;then
            $*;           #<== 这个地方的“$*”就是接收函数外的参数，将来就是
                              {break|continue|exit|return} 中的一个。
        fi
        echo $i
    done
```

```
    echo "I am in func."                          #<== 循环外的输出提示。
}
test $*                                           #<== 这里的“$*”为函数的传参。
func_ret=$?                                       #<== 接收并测试函数返回值。
if [ `echo $*|grep return|wc -l` -eq 1 ]          #<== 如果传参有 return。
  then
    echo "return's exit status:$func_ret"         #<== 则提示 return 退出状态。
fi
echo "ok"                                         #<== 函数外的输出提示。
```

说明：本着在“使用中记忆”的原则，本例采用了复杂的测试{break|continue|exit|return}的脚本方法。

传入 break 命令的执行结果为：

```
[root@oldboy scripts]# sh 12_1_1.sh
usage:12_1_1.sh {break|continue|exit|return}
[root@oldboy scripts]# sh 12_1_1.sh break
0
1
2
I am in func. #<== 循环外的输出提示。
ok            #<== 函数外的输出提示。
```

根据结果可以看到，i 等于 3 及以后的循环没有被执行，但循环外的 echo 执行了，执行到 break 时跳出了 if 及外层的 for 循环语句，然后执行 for 循环外部 done 后面的打印 ok 的语句。

传入 continue 命令的执行结果为：

```
[root@oldboy scripts]# sh 12_1_1.sh continue
0
1
2   #<== 没有 3。
4
5
I am in func. #<== 循环外的输出提示。
ok            #<== 函数外的输出提示。
```

可以看到，只有 i 等于 3 这层循环没有被执行，其他循环全部执行了，循环外的 echo 也执行了，说明执行到 continue 时，终止了本次循环，而继续下一次的循环，直到循环正常结束，接着继续执行了循环外面的所有语句。

传入 exit 119 命令的执行结果为：

```
[root@oldboy scripts]# sh 12_1_1.sh "exit 119"
0
```

```
1
2 #<== 只打印了 0,1,2。
[root@oldboy scripts]# echo $?
119 #<== 返回了 119，即传入的值。
```

根据执行结果可以看到，当进入循环里的 if 语句后遇到 "exit 119" 时，立刻退出程序，不但循环体 3 后面的数字没有输出，而且 for 循环体 done 外的 echo 和函数外的 ok 也没有输出，就直接退出了程序。另外，因为程序退出时指定了 119，所以执行脚本后获取 "$?" 的返回值时就返回了 "exit 119" 后面的 119 这个数字到当前的 Shell。

传入 "return 119" 命令的执行结果为：

```
[root@oldboy scripts]# sh 12_1_1.sh "return 119"
0
1
2
return's exit status:119          #<== 确实将 119 返回到了函数的外部脚本。
ok
[root@oldboy scripts]# echo $?   #<== 执行脚本后的返回值还是 0。
0
```

根据执行结果可以看到，当进入循环里的 if 语句后遇到 return 119，就没有打印 3 以下的数字，说明 return 跳出了循环体，程序也没有执行 for 循环体 done 外的 echo 命令，而是直接执行了函数 test 外的 if 语句及打印 ok 的命令，可见 return 的作用是退出当前函数。同时，return 将数字 119 作为函数的执行状态值返还给函数体外，执行脚本后打印返回值是 0，因为程序的最后一行是打印 ok 的命令，执行是成功的。

12.4　循环控制及状态返回值的企业级案例

范例 12-2：开发 Shell 脚本实现为服务器临时配置多个 IP，并且可以随时撤销配置的所有 IP。IP 的地址范围为：10.0.2.1~10.0.2.16，其中 10.0.2.10 不能配置。

本题主要用于考察 continue、return、exit 的综合应用，请读者细细品味。

首先，给网卡配置额外的 IP。以下介绍两种配置 IP 的命令（ifconfig/ip）。

使用 ifconfig 配置别名 IP 的方法：

```
ifconfig eth0:0 10.0.2.10/24 up     #<== 添加 IP。
ifconfig eth0:0 10.0.2.10/24 down   #<== 删除 IP。
```

使用 IP 配置辅助 IP 的方法：

```
ip addr add 10.0.2.11/24 dev eth0 label eth0:0  #<== 添加 IP。
ip addr del 10.0.2.11/24 dev eth0 label eth0:0  #<== 删除 IP。
```

然后批量配置 IP。要求 IP 地址的取值范围为：10.0.2.1~10.0.2.16，其中 10.0.2.10 不能配置。

```
for ip in {1..16}
do
    if [ $ip -eq 10 ]  #<== 如果变量为 10，即 IP 为 10.0.2.10，则调用 contiune
                           终止本次循环。
      then
        continue       #<== 终止本次循环。
    fi
    ip addr add 10.0.2.$ip/24 dev eth0 label eth0:$ip  #<== 配置 IP。
done
```

下面给出完整的脚本实现。

参考答案 1：本答案看似很好，但是实现中有不少冗余（相同）的代码。

```
[root@oldboy scripts]# cat 12_2_1.sh
#!/bin/sh
[ -f /etc/init.d/functions ] && . /etc/init.d/functions #<== 加载 functions 函数。
RETVAL=0
add(){                              #<== 配置 IP 函数。
    for ip in {1..16}               #<== 列表范围为 1 ~ 16。
    do
        if [ $ip -eq 10 ]           #<== 如果变量为 10，即 IP 为 10.0.2.10，
                                        则调用 contiune 终止本次循环。
          then
            continue                #<== 终止本次循环。
        fi
        ip addr add 10.0.2.$ip/24 dev eth0 label eth0:$ip &>/dev/null
                                    #<== 采用辅助 IP 形式配置 IP。
        RETVAL=$?                   #<== 获取配置 IP 命令的返回值。
        if [ $RETVAL -eq 0 ]        #<== 如果返回值为 0，则执行 then 的成功提示。
          then
            action "add $ip" /bin/true    #<== 优雅地提示成功。
        else
            action "add $ip" /bin/false   #<== 优雅地提示失败。
        fi
    done
    return $RETVAL                  #<== 获取返回值，并返回给函数外的命令。
}
del(){                              #<== 清除 IP 函数。
    for ip in {16..1} #<== 列表范围为 16 ~ 1，这里是为了删除需要，不一定必须这样。
    do
        if [ $ip -eq 10 ]           #<== 如果变量为 10，即 IP 为 10.0.2.10，则调用 contiune
                                        终止本次循环。
          then
            continue                #<== 终止本次循环。
        fi
        #ip addr del 10.0.2.$ip/24 dev eth0 &>/dev/null
```

```
        ifconfig eth0:$ip down &>/dev/null  #<== 采用别名形式配置 IP。
        RETVAL=$?                           #<== 获取配置 IP 命令的返回值。
        if [ $RETVAL -eq 0 ]                #<== 如果返回值为 0，则执行 then 的成功提示。
          then
            action "del $ip" /bin/true  #<== 优雅地提示成功。
        else
            action "del $ip" /bin/false #<== 优雅地提示失败。
        fi
    done
}
case "$1" in        #<== 获取命令行的传参。
    start)          #<== 如果匹配 start。
        add         #<== 则加载 add 函数。
        RETVAL=$?   #<== 获取函数 add 执行后的返回值。
        ;;
    stop)           #<== 如果匹配 stop。
        del         #<== 则加载 del 函数。
        RETVAL=$?   #<== 获取函数 del 执行后的返回值。
        ;;
    restart)        #<== 如果匹配 restart。
        del         #<== 则先加载 del 函数。
        sleep 2     #<== 休息 2 秒。
        add         #<== 然后加载 add 函数。
        RETVAL=$?   #<== 获取函数 add 执行的返回值。
        ;;
        *)          #<== 如果匹配其他任何值。
        printf "USAGE:$0 {start|stop|restart}\n" #<== 则给出正确使用的提示，这里
                                                     使用了 printf 命令。
esac
exit $RETVAL #<== 携带返回值退出脚本并将该值返给当前 Shell，这个不是必须的，但是专业规范的表现。
```

参考答案 2：将上述冗余（相同）的代码部分写成函数并使用，以减少代码量。

```
[root@oldboy scripts]# cat 12_2_2.sh
#!/bin/sh
[ -f /etc/init.d/functions ] && . /etc/init.d/functions
RETVAL=0
op(){           #<== 将上述 add 函数和 del 函数的内容整合为一个函数实现。
    if [ "$1" == "del"  ]
      then
        list=`echo {16..1}`
    else
        list=`echo {1..16}`
    fi
    for ip in $list
    do
        if [ $ip -eq 10 ]
          then
            continue
```

```
        fi
        ip addr $1 10.0.2.$ip/24 dev eth0 label eth0:$ip &>/dev/null
        RETVAL=$?
        if [ $RETVAL -eq 0 ]
          then
            action "$1 $ip" /bin/true     #<== 此处的提示用通用的 $1，传参来控制。
        else
            action "$1 $ip" /bin/false    #<== 此处的提示用通用的 $1，传参来控制。
        fi
    done
    return $RETVAL
}
case "$1" in
    start)
        op add                            #<== 启动时，就传参 add 给 op 函数。
        RETVAL=$?
        ;;
    stop)
        op del                            #<== 停止时，就传参 del 给 op 函数。
        RETVAL=$?
        ;;
    restart)
        op del
        sleep 2
        op add
        RETVAL=$?
        ;;
    *)
        printf "USAGE:$0 {start|stop|restart}\n"
esac
exit $RETVAL
```

范例 12-3：分析 Apache 访问日志，把日志中每行的访问字节数所对应的字段数字相加，计算出总的访问量。给出实现程序，请用 while 循环结构实现。（3 分钟）

本题在第 10 章中已经讲解过，当时用的是 while 循环，本例要讲解的知识点是：利用 continue 终止循环。这和 while 循环会有所不同，请细看。

参考答案 1：while 循环（采用 bash exec 内置命令和 expr 判断整数）。

```
[root@oldboy scripts]# cat 12_3_1.sh
#!/bin/bash
sum=0                                     #<== 初始化资源大小总和为 0。
exec <$1                                  #<== 将传参 $1 输入重定向给 exec。
while read line                           #<== 按行读取传参的文件内容。
do
    size=`echo $line|awk '{print $10}'`   #<== 获取每行的第 10 列，即资源访问字节列。
    expr $size + 1 &>/dev/null            #<== 数字判断。
    if [ $? -ne 0 ];then                  #<== 如果非数字，
        continue                          #<== 则执行 continue 终止本次循环，
```

```
                                                    即不是数字的列不对其进行加法。
        fi
        ((sum=sum+size))                          #<== 将获取到的字节做加法，并赋值给 sum。
    done
    echo "${1}:total:${sum}bytes =`echo $((${sum}/1024))`KB" #<== 循环完毕后，打印结果。
```

参考答案 2：while 循环（采用 bash exec 内置命令 + 变量子串替换特殊方法来判断整数）。

```
    exec <$1
    sum=0
    while read line
    do
        num=`echo $line|awk '{print $10}'`
        [ -n "$num" -a "$num" = "${num//[^0-9]/}" ] || continue
        #<== 若 num 为数字不成立，则执行 continue。
        ((sum=sum+num))
    done
    echo "${1}:${sum} bytes =`echo $((${sum}/1024))`KB"
```

参考答案 3：

```
    exec <access_2010-12-8.log
    sum=0
    while read line
    do
        [ -z "`echo $line|awk '{print $10}'|sed 's#[0-9]##g'`" ]||continue
        #<== 当双引号里的内容长度为 0 不成立（实际上还是判断要加和的字节列是否为数字），则执行
continue。
        ((sum=sum+`echo $line|awk '{print $10}'`))
    done
    echo $sum
```

范例 12-4：已知下面的字符串是通过将 RANDOM 随机数采用 md5sum 加密后任意取出连续 10 位的结果，请破解这些字符串对应的 md5sum 前的数字？

```
    4fe8bf20ed
```

解题思路：本题原本是想考察 break 的用法，但是还考察了 RANDOM 随机数的范围，该范围是 0 ~ 32767，请务必记住。

要想解决本题，首先要将 0 ~ 32767 范围内的所有数字通过 md5sum 加密，并把加密后的字符串和加密前的数字对应地写到日志里。

然后将题中给出的加密后的字符串 4fe8bf20ed 和指纹库里的所有使用 md5sum 加密后的字符串进行比对（grep 最佳），如果匹配，则把对应的行及对应的数字输出。

解答过程如下。

首先将 0 ~ 32767 范围内的所有数字通过 md5sum 加密，并把加密后的字符串和加

密前的数字对应地写到日志里，实现脚本如下：

```
[root@oldboy scripts]# cat 12_4_1.sh
#!/bin/bash
for n in {0..32767}
do
    echo "`echo $n|md5sum` $n" >>/tmp/zhiwen.log  #<== 注意加密前和加密后的对应关系。
done
```

说明：此脚本其实也可以和下面的实现脚本合并成一个脚本，这里分开写的目的是使阐述更清晰。

查看执行后的结果：

```
[root@oldboy scripts]# head /tmp/zhiwen1.log
897316929176464ebc9ad085f31e7284  - 0
b026324c6904b2a9cb4b88d6d61c81d1  - 1
26ab0db90d72e28ad0ba1e22ee510510  - 2
6d7fce9fee471194aa8b5b6e47267f03  - 3
48a24b70a0b376535542b996af517398  - 4
1dcca23355272056f04fe8bf20edfce0  - 5
9ae0ea9e3c9c6e1b9b6252c8395efdc1  - 6
84bc3da1b3e33a18e8d5e1bdd7a18d7a  - 7
c30f7472766d25af1dc80b3ffc9a58c7  - 8
7c5aba41f53293b712fd86d08ed5b36e  - 9
...省略 3 万多行 ..
```

然后将题中给出的加密后的字符串 4fe8bf20ed 和指纹库里所有使用 md5sum 加密后的字符串进行比对（grep 最佳），如果匹配，则输出对应的行及对应的数字。

```
[root@oldboy scripts]# cat 12_4_2.sh
#!/bin/bash
#>/tmp/zhiwen.log
#for n in {0..32767}
#do
# echo "`echo $n|md5sum` $n" >>/tmp/zhiwen.log
#done
#<== 上述内容被注释掉了，读者也可以打开注释查看。
md5char="4fe8bf20ed"    #<== 定义待破解的字符串。
while read line         #<== 进入循环。
do
    if [ `echo $line|grep "$md5char"|wc -l` -eq 1 ]
                        #<== 循环日志中的每一行都通过 grep 进行过滤，如果符合要求，则
                             wc 后的值会等于 1，表示查找到了。
      then
        echo $line      #<== 打印查找到的行。
        break
```

```
    fi
done </tmp/zhiwen.log   #<== 读取加密串的日志。
```

执行结果如：

```
[root@oldboy scripts]# sh 12_4_2.sh
1dcca23355272056f04fe8bf20edfce0 - 5
```

该 4fe8bf20ed 对应的 md5sum 加密前的随机数为数字 5。

特别说明：可访问如下地址或手机扫二维码查看第 12 章的核心脚本代码。

http://oldboy.blog.51cto.com/2561410/1855261

Linux

第13章

Shell 数组的应用实践

13.1 Shell 数组介绍

13.1.1 为什么会产生 Shell 数组

通常在开发 Shell 脚本时，定义变量采用的形式为“a=1;b=2;c=3”，可如果有多个变量呢？这时再逐个地定义就会很费劲，并且要是有多个不确定的变量内容，也会难以进行变量定义，此外，快速读取不同变量的值也是一件很痛苦的事情，于是数组就诞生了，它就是为了解决上述问题而出现的[⊖]。

13.1.2 什么是 Shell 数组

如果读者有过其他语言的编程经历，那么想必会熟悉数组的概念。简单地说，Shell 的数组就是一个元素集合，它把有限个元素（变量或字符内容）用一个名字来命名，然后用编号对它们进行区分。这个名字就称为数组名，用于区分不同内容的编号就称为数组下标。组成数组的各个元素（变量）称为数组的元素，有时也称为下标变量。

有了 Shell 数组之后，就可以用相同名字来引用一系列变量及变量值了，并通过数字（索引）来识别使用它们。在很多场合中，使用数组可以缩短和简化程序开发。

⊖ 上述描述是为了便于读者理解数组的作用。

13.2　Shell 数组的定义与增删改查

13.2.1　Shell 数组的定义

Shell 数组的定义有多种方法，列举如下。

方法 1：用小括号将变量值括起来赋值给数组变量，每个变量值之间要用空格进行分隔。

语法如下：

```
array=(value1 value2 value3 ... )
```

此为常用定义方法，需要重点掌握。

示例如下：

```
[root@oldboy ~]# array=(1 2 3)     #<== 用小括号将数组内容赋值给数组变量，
                                        数组元素用“空格”分隔开。
[root@oldboy ~]# echo ${array[*]} #<== 输出上面定义的数组的所有元素值，注意语法。
1 2 3
```

方法 2：用小括号将变量值括起来，同时采用键值对的形式赋值。

语法如下：

```
array=([1]=one [2]=two [3]=three)
```

此种方法为 key-value 键值对的形式，小括号里对应的数字为数组下标，等号后面的内容为下标对应的数组变量的值，此方法比较复杂，不推荐使用。

示例如下：

```
[root@oldboy scripts]# array=([1]=one [2]=two [3]=three)
[root@oldboy scripts]# echo ${array[*]} #<== 输出上面定义的数组的所有元素值。
one two three
[root@oldboy scripts]# echo ${array[1]} #<== 输出上面定义的数组的第一个元素值。
one
[root@oldboy scripts]# echo ${array[2]} #<== 输出上面定义的数组的第二个元素值。
two
[root@oldboy scripts]# echo ${array[3]} #<== 输出上面定义的数组的第三个元素值。
three
```

方法 3：通过分别定义数组变量的方法来定义。

语法如下：

```
array[0]=a;array[1]=b;array[2]=c
```

此种定义方法比较麻烦，不推荐使用。

示例如下：

```
[root@oldboy scripts]# array[0]=a
[root@oldboy scripts]# array[1]=b
[root@oldboy scripts]# array[2]=c
[root@oldboy scripts]# echo ${array[0]}
a
```

方法 4：动态地定义数组变量，并使用命令的输出结果作为数组的内容。

语法为：

```
array=($( 命令 ))
```

或：

```
array=(` 命令 `)
```

示例如下：

```
[root@oldboy scripts]# mkdir /array/ -p
[root@oldboy scripts]# touch /array/{1..3}.txt
[root@oldboy scripts]# ls -l /array/
总用量 0
-rw-r--r-- 1 root root 0 9月   6 09:38 1.txt
-rw-r--r-- 1 root root 0 9月   6 09:38 2.txt
-rw-r--r-- 1 root root 0 9月   6 09:38 3.txt
[root@oldboy scripts]# array=($(ls /array))
[root@oldboy scripts]# echo ${array[*]}
1.txt 2.txt 3.txt
```

说明：还可以使用 declare -a array 来定义数组类型，但是比较少这样用。

13.2.2 Shell 数组的打印及输出

1. 打印数组元素

此为常用知识点，需要重点掌握。示例如下：

```
[root@oldboy scripts]# array=(one two three)
[root@oldboy scripts]# echo ${array[0]}
#<== 打印单个数组元素用 ${ 数组名 [ 下标 ]}，当未指定数组下标时，数组的下标将从 0 开始。
one
[root@oldboy scripts]# echo ${array[1]}
two
[root@oldboy scripts]# echo ${array[2]}
three
[root@oldboy scripts]# echo ${array[*]} #<== 使用 * 或 @ 可以得到整个数组的内容。
one two three
[root@oldboy scripts]# echo ${array[@]} #<== 使用 * 或 @ 可以得到整个数组的内容。
one two three
```

2. 打印数组元素的个数

此为常用知识点，需要重点掌握。示例如下：

```
[root@oldboy scripts]# echo ${array[*]}  #<== 使用 * 或 @ 可以得到整个数组内容。
one two three
[root@oldboy scripts]# echo ${#array[*]} #<== 用 ${# 数组名 [@ 或 *]} 可以得到数组的长度，这和前文讲解的变量子串知识是一样的，因为数组也是变量，只不过是特殊的变量，因此变量的子串替换等知识也适合于数组。
3
[root@oldboy scripts]# echo ${array[@]}  #<== 使用 * 或 @ 可以得到整个数组内容。
one two three
[root@oldboy scripts]# echo ${#array[@]} #<== 用 ${# 数组名 [@ 或 *]} 可以得到数组的长度，这和前文讲解的变量子串知识是一样的，因为数组也是变量，只不过是特殊的变量，因此变量的子串替换等知识也适合于数组。
3
```

3. 数组赋值

此知识不常用，了解即可。可直接通过“数组名 [下标]”对数组进行引用赋值，如果下标不存在，则自动添加一个新的数组元素，如果下标存在，则覆盖原来的值。

示例如下：

```
[root@oldboy scripts]# array=(one two three)
[root@oldboy scripts]# echo ${array[*]}
one two three
[root@oldboy scripts]# array[3]=four      #<== 增加下标为 3 的数组元素。
[root@oldboy scripts]# echo ${array[*]}
one two three four
[root@oldboy scripts]# array[0]=oldboy
[root@oldboy scripts]# echo ${array[*]}
oldboy two three four
[root@oldboy scripts]# array[0]=oldboy    #<== 修改数组元素。
[root@oldboy scripts]# echo ${array[@]}
oldboy 2 3 4
```

4. 数组的删除

因为数组本质上还是变量，因此可通过“ unset 数组 [下标]”清除相应的数组元素，如果不带下标，则表示清除整个数组的所有数据。

示例如下：

```
[root@oldboy scripts]# echo ${array[*]}
oldboy two three four
[root@oldboy scripts]# unset array[1]     #<== 取消下标为 1 的数组元素。
[root@oldboy scripts]# echo ${array[*]}   #<== 打印输出后发现数组元素“two”不见了。
oldboy three four
[root@oldboy scripts]# unset array        #<== 删除整个数组。
[root@oldboy scripts]# echo ${array[*]}
#<== 没有任何内容了。
```

5. 数组内容的截取和替换

这里和前文变量子串的替换是一样的，因为数组是特殊的变量。数组元素部分的内容截取的示例如下：

```
[root@oldboy scripts]# array=(1 2 3 4 5)
[root@oldboy scripts]# echo ${array[@]:1:3}     #<== 截取 1 号到 3 号数组元素。
2 3 4
[root@oldboy scripts]# array=($(echo {a..z}))   #<== 将变量的结果赋值给数组变量。
[root@oldboy scripts]# echo ${array[@]}
a b c d e f g h i j k l m n o p q r s t u v w x y z
[root@oldboy scripts]# echo ${array[@]:1:3}     #<== 截取下标为 1 到 3 的数组元素。
b c d
[root@oldboy scripts]# echo ${array[@]:0:2}     #<== 截取下标为 0 到 2 的数组元素。
a b
```

替换数组元素部分内容的代码如下：

```
[root@oldboy scripts]# array=(1 2 3 1 1)
[root@oldboy scripts]# echo ${array[@]/1/b}     #<== 把数组中的 1 替换成 b，原数组
                                                    未被修改，和 sed 很像。
b 2 3 b b
```

提示：调用方法为 ${ 数组名 [@ 或 *]/ 查找字符 / 替换字符 }，该操作不会改变原先数组的内容，如果需要修改，可以参考上面的例子，重新定义数组。

删除数组元素部分内容的代码如下：

```
[root@oldboy scripts]# array=(one two three four five)
[root@oldboy scripts]# echo ${array[@]}
one two three four five
[root@oldboy scripts]# echo ${array[@]#o*}      #<== 从左边开始匹配最短的数组元素，
                                                    并删除。
one two three four five
[root@oldboy scripts]# echo ${array1[@]##o*}    #<== 从左边开始匹配最长的数组元素，
                                                    并删除。
two three four five
[root@oldboy scripts]# echo ${array[@]%f*}      #<== 从右边开始匹配最短的数组元素，
                                                    并删除。
one two three
[root@oldboy scripts]# echo ${array[@]%%f*}     #<== 从右边开始匹配最长的数组元素，
                                                    并删除。
one two three
```

提示：数组也是变量，因此也适合于前面讲解过的变量的子串处理的功能应用。数组的其他相关知识可通过 man bash 命令然后搜“Arrays”来了解。

13.3　Shell 数组脚本开发实践

范例 13-1：使用循环批量输出数组的元素。

方法 1：通过 C 语言型的 for 循环语句打印数组元素。

```
[root@oldboy scripts]# cat 13_1_1.sh
#!/bin/sh
array=(1 2 3 4 5)
for((i=0;i<${#array[*]};i++))   #<== 从数组的第一个下标 0 开始，循环数组的所有下标。
do
    echo ${array[i]}            #<== 打印数组元素。
done
```

输出结果如下：

```
[root@oldboy scripts]# sh 13_1_1.sh
1
2
3
4
5
```

方法 2：通过普通 for 循环语句打印数组元素。

```
[root@oldboy scripts]# cat 13_1_2.sh
#!/bin/sh
array=(1 2 3 4 5)
for n in ${array[*]} #<==${array[*]} 表示输出数组的所有元素，相当于列表数组元素。
do
    echo $n          #<== 这里就不是直接去数组里取元素了，而是取变量 n 的值。
done
```

输出结果同方法 1，此处略过。

方法 3：使用 while 循环语句打印数组元素。

```
[root@oldboy scripts]# cat 13_1_3.sh
#!/bin/sh
array=(1 2 3 4 5)
i=0
while ((i<${#array[*]}))
do
    echo ${array[i]}
    ((i++))
done
```

输出结果同方法 1，此处略过。

范例 13-2：通过竖向列举法定义数组元素并批量打印。

```
[root@oldboy scripts]# cat 13_2_1.sh
#!/bin/sh
array=(    #<== 对于元素特别长的情况，例如 URL 地址，将其竖向列出来看起来会更舒服和规范。
oldboy
oldgirl
xiaoting
bingbing
)
for ((i=0; i<${#array[*]}; i++))
do
    echo "This is num $i,then content is ${array[$i]}"
done
echo ----------------------
echo "array len:${#array[*]}"
```

输出结果如下：

```
[root@oldboy scripts]# sh 13_2_1.sh
This is num 0,then content is oldboy
This is num 1,then content is oldgirl
This is num 2,then content is xiaoting
This is num 3,then content is bingbing
----------------------
array len:4
```

范例 13-3：将命令结果作为数组元素定义并打印。

准备数据：

```
[root@oldboy scripts]# mkdir -p /array/
[root@oldboy scripts]# touch /array/{1..3}.txt
[root@oldboy scripts]# ls /array/
1.txt  2.txt  3.txt
```

以下为开发脚本：

```
[root@oldboy scripts]# cat 13_3_1.sh
#!/bin/bash
dir=($(ls /array))                  #<== 把 ls /array 命令结果放进数组里。
for ((i=0; i<${#dir[*]}; i++))      #<==${#dir[*]} 为数组的长度。
do
    echo "This is NO.$i,filename is ${dir[$i]}"
done
```

输出结果如下：

```
[root@oldboy scripts]# sh 13_3_1.sh
This is NO.0,filename is 1.txt
This is NO.1,filename is 2.txt
This is NO.2,filename is 3.txt
```

13.4　Shell 数组的重要命令

（1）定义命令

静态数组：

```
array=(1 2 3)
```

动态数组：

```
array=($(ls))
```

为数组赋值：

```
array[3]=4
```

（2）打印命令

打印所有元素：

```
${array[@]} 或 ${array[*]}
```

打印数组长度：

```
${#array[@]} 或 ${#array[*]}
```

打印单个元素：

```
${array[i]} #<==i 是数组下标。
```

（3）循环打印的常用基本语法

```
#!/bin/sh
arr=(
    10.0.0.11
    10.0.0.22
    10.0.0.33
)
#<==C 语言 for 循环语法
for ((i=0;i<${#arr[*]};i++))
do
    echo "${arr[$i]}"
done
echo ---------------
#<== 普通 for 循环语法
for n in ${arr[*]}
do
    echo "$n"
done
```

13.5 Shell 数组相关面试题及高级实战案例

范例 13-4：利用 bash for 循环打印下面这句话中字母数不大于 6 的单词（某企业面试真题）。

```
I am oldboy teacher welcome to oldboy training class
```

解答思路具体如下。

1）先把所有的单词放到数组里，然后依次进行判断。命令如下：

```
array=(I am oldboy teacher welcome to oldboy training class)
```

2）计算变量内容的长度，这在前文已经讲解过了。常见方法有 4 种：

```
[root@oldboy scripts]# char=oldboy
[root@oldboy scripts]# echo $char|wc -L
6
[root@oldboy scripts]# echo ${#char}
6
[root@oldboy scripts]# expr length $char
6
[root@oldboy scripts]# echo $char|awk '{print length($0)}'
6
```

方法 1：通过数组方法来实现。

```
arr=(I am oldboy teacher welcome to oldboy training class)
for ((i=0;i<${#arr[*]};i++))
do
    if [ ${#arr[$i]} -lt 6 ]
      then
        echo "${arr[$i]}"
    fi
done
echo ----------------------
for word in ${arr[*]}
do
    if [ `expr length $word` -lt 6 ];then
        echo $word
    fi
done
```

说明：本例给出了用两种 for 循环打印数组元素的方法。

方法 2：使用 for 循环列举取值列表法。

```
for word in I am oldboy teacher welcome to oldboy training class
```

```
#<== 看起来有点初级吧。
do
    if [ `echo $word|wc -L` -lt 6 ];then
        echo $word
    fi
done

chars="I am oldboy teacher welcome to oldboy training class"
#<== 定义字符串可以。
for word in $chars
do
    if [ `echo $word|wc -L` -lt 6 ];then
        echo $word
    fi
done
```

方法 3：通过 awk 循环实现。

```
    [root@oldboy scripts]# chars="I am oldboy teacher welcome to oldboy training
class"
    [root@oldboy scripts]# echo $chars|awk '{for(i=1;i<=NF;i++) if(length($i)<=6)
print $i}'
```

几种方法的输出结果统一为：

```
I
am
oldboy
to
oldboy
class
```

范例 13-5：批量检查多个网站地址是否正常。

要求：

1）使用 Shell 数组的方法实现，检测策略尽量模拟用户访问。

2）每 10 秒进行一次全部检测，无法访问的输出报警。

3）待检测的地址如下。

http://blog.oldboyedu.com

http://blog.etiantian.org

http://oldboy.blog.51cto.com

http://10.0.0.7

解题思路：

1）把 URL 定义成数组，形成函数。

2）编写 URL 检查脚本函数，传入数组的元素，即 URL。

3）组合实现整个案例，编写 main 主函数（即执行函数），每隔 10 秒检查一次。

下面的参考答案采用了 Shell 数组的方法，同时检测多个 URL 是否正常，并给出专业的展示效果（同范例 10-7，即给出了数组的用法，有需要的读者可以回看）：

```
[root@oldboy scripts]# cat 10_7_2.sh
#!/bin/bash
# this script is created by oldboy.
# e_mail:31333741@qq.com
# function:case example
# version:1.3
. /etc/init.d/functions

check_count=0
url_list=(                          #<== 定义检测的 URL 数组，包含多个 URL 地址。
http://blog.oldboyedu.com
http://blog.etiantian.org
http://oldboy.blog.51cto.com
http://10.0.0.7
)

function wait()                     #<== 定义 3,2,1 倒计时函数。
{
    echo -n '3 秒后，执行检查 URL 操作 .';
    for ((i=0;i<3;i++))
    do
        echo -n ".";sleep 1
    done
    echo
}
function check_url()                #<== 定义检测 URL 的函数。
{
    wait                            #<== 执行倒计时函数。
    for ((i=0; i<`echo ${#url_list[*]}`; i++)) #<== 循环数组元素。
    do
        wget -o /dev/null -T 3 --tries=1 --spider ${url_list[$i]} >/dev/null 2>&1
                                    #<== 检测是否可以访问数组元素的地址。
        if [ $? -eq 0 ]             #<== 如果返回值为 0，则表示访问成功。
          then
            action "${url_list[$i]}" /bin/true #<== 优雅地显示成功结果。
        else
            action "${url_list[$i]}" /bin/false #<== 优雅地显示失败结果。
        fi
    done
    ((check_count++))                                   #<== 检测次数加 1。
}
main(){                                                 #<== 定义主函数。
    while true                                          #<== 开启一个持续循环。
    do
        check_url                                       #<== 加载检测 url 的函数。
        echo "-------check count:${check_count}---------"
        sleep 10                                        #<== 间歇 10 秒。
```

```
    done
}
main                                                    #<== 调用主函数运行程序。
```

执行结果如图 13-1 所示。

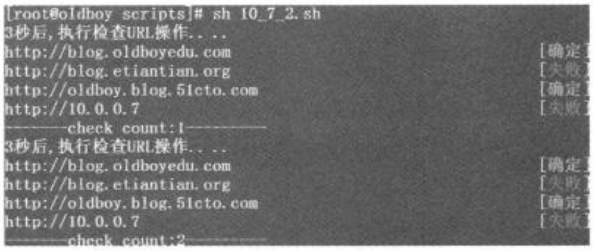


图 13-1　检测数组内 URL 输出的专业效果图

提示：实际使用时，一些基础的函数脚本（例如，加颜色的函数）是放在函数文件里的（例如，放在 /etc/init.d/functions 里），与执行的脚本内容部分分离，这样看起来会更清爽，大型的语言程序都是这样开发的。

范例 13-6：开发一个守护进程脚本，每 30 秒监控一次 MySQL 主从复制是否异常（包括不同步及延迟），如果有异常，则发送短信报警，并发送邮件给管理员存档（此为生产实战案例）。

提示：如果没主从复制的环境，可以把下面的文本放到文件里读取来模拟主从复制的状态。

```
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 10.0.0.51
                  Master_User: rep
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000013
          Read_Master_Log_Pos: 502547
               Relay_Log_File: relay-bin.000013
                Relay_Log_Pos: 251
        Relay_Master_Log_File: mysql-bin.000013
             Slave_IO_Running: Yes   #<==IO 线程状态必须为 Yes。
            Slave_SQL_Running: Yes   #<==SQL 线程状态必须为 Yes。
              Replicate_Do_DB:
          Replicate_Ignore_DB: mysql
           Replicate_Do_Table:
```

```
          Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 502547
              Relay_Log_Space: 502986
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0    #<== 和主库比较同步延迟的秒数，这个参数很重要。
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
```

解题思路：

1）判断主从复制是否异常，主要是检测如下参数对应的值。

```
             Slave_IO_Running: Yes   #<==IO 线程状态必须为 Yes。
            Slave_SQL_Running: Yes   #<==SQL 线程状态必须为 Yes。
          Seconds_Behind_Master: 0     #<== 和主库比较同步延迟的秒数，这个参数很重要。
```

2）读取状态数据或状态文件，然后取出对应的值，和正确的值进行比对，如果不符合，则表示存在故障，即调用报警脚本报警。

3）如果想要更专业，还可以在主从不同步时，查看相应的错误号，判断对应的错误号以自动修复主从复制故障（也可以通过在配置文件里的配置参数来实现自动忽略故障）。

以下为参考答案。

首先给出模拟数据（注意，使用时要去掉中文注释）。

```
[root@oldboy scripts]# cat slave.log
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 10.0.0.51
                  Master_User: rep
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000013
          Read_Master_Log_Pos: 502547
```

```
                Relay_Log_File: relay-bin.000013
                 Relay_Log_Pos: 251
         Relay_Master_Log_File: mysql-bin.000013
              Slave_IO_Running: Yes
             Slave_SQL_Running: Yes
               Replicate_Do_DB:
           Replicate_Ignore_DB: mysql
            Replicate_Do_Table:
        Replicate_Ignore_Table:
       Replicate_Wild_Do_Table:
   Replicate_Wild_Ignore_Table:
                    Last_Errno: 0
                    Last_Error:
                  Skip_Counter: 0
           Exec_Master_Log_Pos: 502547
               Relay_Log_Space: 502986
               Until_Condition: None
                Until_Log_File:
                 Until_Log_Pos: 0
            Master_SSL_Allowed: No
            Master_SSL_CA_File:
            Master_SSL_CA_Path:
               Master_SSL_Cert:
             Master_SSL_Cipher:
                Master_SSL_Key:
         Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                 Last_IO_Errno: 0
                 Last_IO_Error:
                Last_SQL_Errno: 0
                Last_SQL_Error:
```

然后开发脚本，有多种方法，下面分别给出各个参考方法。

方法 1：

```
[root@oldboy scripts]# awk -F ':' '/_Running|_Behind/{print $NF}' slave.log
#<== 获取所有复制相关的状态值，脚本里使用 slave.log 时注意完全路径。
Yes
Yes
0
[root@oldboy scripts]# cat 13_6_1.sh
count=0
status=($(awk -F ':' '/_Running|_Behind/{print $NF}' slave.log))
                                #<== 获取所有复制相关的状态值赋并值给数组 status。
for((i=0;i<${#status[*]};i++)) #<== 循环数组元素。
do
    if [ "${status[${i}]}" != "Yes" -a "${status[${i}]}" != "0" ]
    #<== 如果数组元素值不为 Yes 或 0 中的任意一个，那就表示复制出故障了。
      then
        let count+=1            #<== 错误数加 1。
```

```
        fi
    done
    if [ $count -ne 0 ];then #<== 只要错误数不等于 0，就表示状态值肯定是有问题的。
        echo "mysql replcation is failed"   #<== 提示复制出现问题。
    else
        echo "mysql replcation is sucess"   #<== 否则提示复制正常。
    fi
```

说明：本答案是为了引导读者学习，因此没有加每 30 秒的条件。

测试结果如下：

```
    [root@oldboy scripts]# sh 13_6_1.sh
    mysql replcation is sucess
    [root@oldboy scripts]# sed -i 's#Slave_IO_Running: Yes#Slave_IO_Running:
No#g' slave.log                                    #<== 模拟 IO 线程故障。
    [root@oldboy scripts]# sh 13_6_1.sh
    mysql replcation is failed
```

方法 2：本方法和方法 1 实现的功能差不多，但是开发手法更高大上一些。

```
    [root@oldboy scripts]# cat 13_6_2.sh
    #!/bin/bash
    CheckDb(){
        status=($(awk -F ':' '/_Running|_Behind/{print $NF}' slave.log))
        for((i=0;i<${#status[*]};i++))
        do
            count=0
            if [ "${status[${i}]}" != "Yes" -a "${status[${i}]}" != "0" ]
              then
                let count+=1
            fi
        done
        if [ $count -ne 0 ];then
            echo "mysql replcation is failed"
            return 1
        else
            echo "mysql replcation is sucess"
            return 0
        fi
    }
    main(){
        while true
        do
            CheckDb
            sleep 30
        done
    }
    main
```

测试结果如下：

```
    [root@oldboy scripts]# sed -i 's#Slave_IO_Running: No#Slave_IO_Running:
Yes#g' slave.log  #<== 模拟 IO 线程恢复正常。
    [root@oldboy scripts]# sh 13_6_2.sh
    mysql replcation is sucess
    mysql replcation is sucess
    mysql replcation is sucess
    ^C
    [root@oldboy scripts]# sed -i 's#Slave_IO_Running: Yes#Slave_IO_Running:
No#g' slave.log   #<== 提示复制出现问题。
    [root@oldboy scripts]# sh 13_6_2.sh
    mysql replcation is failed
    mysql replcation is failed
    ^C
```

说明：本答案还是没有完全满足题意，例如，报警短信和邮件的功能还没有开发。

方法 3（此为企业生产的正式检查脚本）：

```
    [root@oldboy scripts]# cat 13_6_3.sh
    #!/bin/bash
    ############################################
    # this script function is :
    # check_mysql_slave_replication_status
    # USER        YYYY-MM-DD - ACTION
    # oldboy      2009-02-16 - Created
    ############################################
    path=/server/scripts                    #<== 定义脚本存放路径，请大家注意这个规范。
    MAIL_GROUP="1111@qq.com 2222@qq.com"    #<== 邮件列表，以空格隔开。
    PAGER_GROUP="18600338340 18911718229"   #<== 手机列表，以空格隔开。
    LOG_FILE="/tmp/web_check.log"           #<== 日志路径。
    USER=root                               #<== 数据库用户。
    PASSWORD=oldboy123                      #<== 用户密码。
    PORT=3307 #<== 端口。
    MYSQLCMD="mysql -u$USER -p$PASSWORD -S /data/$PORT/mysql.sock"
                                            #<== 登录数据库命令。
    error=(1008 1007 1062)                  #<== 可以忽略的主从复制错误号。
    RETVAL=0
    [ ! -d "$path" ] && mkdir -p $path
    function JudgeError(){                  #<== 定义判断主从复制错误的函数。
        for((i=0;i<${#error[*]};i++))
        do
            if [ "$1" == "${error[$i]}" ]   #<== 如果传入的错误号和数组里的元素相匹配，
                                                则执行 then 后面的命令。
              then
                echo "MySQL slave errorno is $1,auto repairing it."
```

```
                $MYSQLCMD -e "stop slave;set global sql_slave_skip_counter=1;start slave;"
                                                    #<== 自动修复。
            fi
        done
        return $1
    }
    function CheckDb(){                             #<== 定义检查数据库主从复制状态的函数。
        status=($(awk -F ':' '/_Running|Last_Errno|_Behind/{print $NF}' slave.log))
        expr ${status[3]} + 1 &>/dev/null #<== 这个是延迟状态值，用于进行是否为数字的判断。
        if [ $? -ne 0 ];then                        #<== 如果不为数字。
            status[3]=300                           #<== 赋值 300，当数据库出现复制故障时，延迟
                                                        这个状态值有可能是 NULL，即非数字。
        fi
        if [ "${status[0]}" == "Yes" -a "${status[1]}" == "Yes" -a ${status[3]} -lt 120 ]
        #<== 两个线程都为 Yes，并且延迟小于 120 秒，即认为复制状态是正常的。
          then
            #echo "Mysql slave status is ok"
            return 0                        #<== 返回 0。
        else
            #echo "mysql replcation is failed"
            JudgeError ${status[2]}  #<== 否则，将错误号 ${status[2]} 传入 JudgeError 函数，
                                            判断错误号是否可以自动修复。
        fi
    }
    function MAIL(){                            #<== 定义邮件函数，在范例 11-13 中讲过此函数。
        local SUBJECT_CONTENT=$1                #<== 将函数的第一个传参赋值给主题变量。
        for MAIL_USER  in `echo $MAIL_GROUP`  #<== 遍历邮件列表。
        do
            mail -s "$SUBJECT_CONTENT " $MAIL_USER <$LOG_FILE #<== 发邮件。
        done
    }
    function PAGER(){#<== 定义手机函数，在范例 11-13 中讲过此函数。
        for PAGER_USER  in `echo $PAGER_GROUP` #<== 遍历手机列表。
        do
            TITLE=$1                            #<== 将函数的第一个传参赋值给主题变量。
            CONTACT=$PAGER_USER                 #<== 将手机号赋值给 CONTACT 变量。
            HTTPGW=http://oldboy.sms.cn/smsproxy/sendsms.action
            #<== 发送短信地址，这个地址需要用户付费购买，如果想要免费的，就得用微信替代了。
            #send_message method1
            curl -d  cdkey=5ADF-EFA -d password=OLDBOY -d phone=$CONTACT -d message=
"$TITLE[$2]" $HTTPGW
            #<== 发送短信报警的命令。cdkey 是购买短信网关时，由售卖者提供的，password 是密码，
                也是由售卖者提供的。
        done
    }
    function SendMsg(){
        if [ $1 -ne 0 ] #<== 传入 $1，如果不为 0，则表示复制有问题，这里的 $1 即为 CheckDb 里
的返回值（用检测失败的次数作为返回值），在后文执行主函数 main 时是通过调用 SendMsg 传参传进来的值。
          then
```

```
            RETVAL=1
            NOW_TIME=`date +"%Y-%m-%d %H:%M:%S"`      #<== 报警时间。
            SUBJECT_CONTENT="mysql slave is error,errorno is $2,${NOW_TIME}."
                                                      #<== 报警主题。
            echo -e "$SUBJECT_CONTENT"|tee $LOG_FILE #<== 输出信息，并记录到日志。
            MAIL $SUBJECT_CONTENT #<== 发邮件报警，$SUBJECT_CONTENT 作为函数参数
                                      传给 MAIL 函数体的 $1。
            PAGER $SUBJECT_CONTENT $NOW_TIME #<== 发短信报警，$SUBJECT_CONTENT 作为
函数参数传给 MAIL 函数体的 $1，$NOW_TIME 作为函数体传给 $2。
        else
            echo "Mysql slave status is ok"
            RETVAL=0 #<== 以 0 作为返回值。
        fi
        return $RETVAL
    }
    function main(){
        while true
        do
            CheckDb
            SendMsg $? #<== 传入第一个参数 "$?"，即 CheckDb 里的返回值 (用检测失败的次数作为返回值)。
            sleep 30
        done
    }
    main
```

13.6　合格运维人员必会的脚本列表

下面列举的知识点是老男孩要求所有学生必会的内容，这些内容不仅涉及了脚本知识，还有涉及了系统命令、大量网络服务的知识，这些都需要运维人员了解和掌握，Shell 编程仅仅是其中的一部分内容。

作为一个合格的运维人员，需要掌握的脚本知识列表如下：

1）系统及各类服务的监控脚本，例如：文件、内存、磁盘、端口，URL 监控报警等。

2）监控网站目录下的文件是否被篡改，以及当站点目录被批量篡改后如何批量恢复它们的脚本。

3）各类服务 Rsync、Nginx、MySQL 等的启动及停止专业脚本（使用 chkconfig 管理）。

4）MySQL 主从复制监控报警，以及自动处理不复制故障的脚本。

5）一键配置 MySQL 多实例、一键配置 MySQL 主从部署的脚本。

6）监控 HTTP、MySQL、Rsync、NFS、Memcached 等服务是否异常的生产脚本。

7）一键软件安装及优化的脚本，比如 LANMP、Linux 一键优化，一键数据库安装、优化等。

8）MySQL 多实例启动脚本，分库、分表自动备份脚本。

9）根据网络连接数及 Web 日志 PV 数封 IP 的脚本。

10）监控网站的 PV 及流量，并且对流量信息进行统计的脚本。

11）检查 Web 服务器多个 URL 地址是否异常的脚本，要是可以批量处理且通用的脚本。

12）对系统的基础配置一键优化的脚本。

13）TCP 连接状态及 IP 统计报警的脚本。

14）批量创建用户并设置随机 8 位密码的脚本。

提示：对于这些脚本，大部分都可以直接从本书中找到相关案例或类似的开发方法，建议读者在学习完本书后，自行练习，看是否可以搞定这些问题。

特别说明：可访问如下地址或手机扫二维码查看第 13 章的核心脚本代码。

http://oldboy.blog.51cto.com/2561410/1855316

Linux

第14章

Shell 脚本开发规范

Shell 脚本开发规范及习惯非常重要，有了好的规范和习惯，才能大大提升开发效率，降低后期的脚本维护成本，特别是在多人协作开发时，有一个互相遵守的规范显得特别重要。即使是自己一个人独自开发，也要采取一套科学的、固定的规范，这样脚本才更易读，易于后期维护。总之，就是要让自己养成一个一出手就是专业和规范的习惯。下面我们就来看看具体都有哪些规范和习惯。

14.1 Shell 脚本基本规范

在 Shell 脚本里，第一行通常用于指定脚本解释器，该行内容为：

```
#!/bin/bash
```

或：

```
#!/bin/sh
```

说明：此项在 Linux 系统场景下可能不是必须的，属于优秀规范和习惯。

而在 Shell 脚本的开头处解释器代码后，最好加上版本版权等信息，如下：

```
#Date:    16:29 2012-3-30
```

```
#Author:  Created by oldboy
#Mail:     31333741@qq.com
#Function: This scripts function is.....
#Version: 1.1
```

说明：可在修改 ~/.vimrc 配置文件时自动加上以上信息的功能。此项在 Linux 系统场景下不是必须的，属于优秀规范和习惯。

此外，Shell 脚本中尽量不要用中文注释，应用英文注释，以防止本机或切换系统环境后出现中文乱码的困扰。如果非要加中文，请根据自身的客户端对系统进行字符集调整，如：export LANG="zh_CN.UTF-8"，并在脚本中重新定义字符集，使其和系统一致。

Shell 脚本命名应以“.sh”为扩展名。例如：script-name.sh。

Shell 脚本应存放在固定的路径下，例如：/server/scripts。

以下则是代码书写技巧。

- 成对的符号应尽量一次写出来，然后退格在符号里增加内容，以防止遗漏。这些成对的符号包括：

```
{}、[]、''、``、""
```

- 中括号（[]）两端至少要有 1 个空格，因此，键入中括号时即可留出空格 [　　]，然后再退格键入中间的内容，并确保两端都至少有一个空格。即：先键入一对中括号，然后退一个格，输入两个空格，再退一个格，双中括号（[[]]）的写法也是如此。
- 对于流程控制语句应一次将格式写完，再添加内容。

比如，if 语句的格式一次完成应为：

```
if 条件内容
  then
    内容
fi
```

for 循环语句的格式一次完成应为：

```
for
do
    内容
done
```

提示：while 和 until、case 等语句也是一样。

❑ 通过缩进让代码更易读，比如：

```
if 条件内容
  then
    内容
fi
```

❑ 字符串赋值给变量时应加双引号，并且等号前后不能有空格。例如：

```
OLDBOY_FILE="test.txt"
```

❑ 脚本中的单引号、双引号及反引号，必须为英文状态下的符号，其实所有的 Linux 标准字符及符号都应该是英文状态下的符号，这一点需要特别注意。

14.2　Shell 脚本变量命名及引用变量规范

1. 全局变量定义

全局变量也称环境变量，它的定义应全部大写，如 APACHE_ERR 或 APACHEERR，名字对应的语义要尽量清晰，能够正确表达变量内容的含义，对于过长的英文单词可用前几个字符代替。多个单词间可用“_”号连接，全局变量的定义一般放在系统的全局路径中，并且最好采用 export 来定义，全局变量一般可以在任意子 Shell 中直接使用（特殊情况除外，例如：定时任务执行 Shell 时就最好在 Shell 里重新定义这些全局变量，否则可能会出现问题）。

范例 14-1：全局变量的定义示例。

```
[root@oldboy scripts]# tail -1 /etc/profile
export APACHEERR="hello"
[root@oldboy scripts]# source /etc/profile
[root@oldboy scripts]# echo $APACHEERR
hello
```

2. 局部变量定义

局部变量也称为普通变量，在常规脚本中，普通变量的命名也要尽可能统一，可以使用驼峰语法，即第二个单词的首字母大写，如 oldboyTraining，或者每个单词首字母大写，如 CheckUrl，当然也有网友喜欢采用全部大写或全部小写的方式，例如：CHECK、check，选一种适合你的即可，或者跟着本书的规范走。

Shell 函数中的变量可以使用 local 方式进行定义，使之只在本函数作用域内有效，防止函数中的变量名称与外部程序中的变量相同，从而造成程序异常。下面是在函数中定义变量的例子。

范例 14-2：实现函数内的变量定义。

```
function TestFunc(){
```

```
    local i
    for((i=0;i<n;i++))
    do
        echo 'do something'
    done
}
```

3. 变量的引用规范

在引用变量时，若变量前后都有字符，则需要使用 ${APACHE_ERR}（加大括号的方式）引用变量，以防止产生歧义；当变量内容为字符串时，需要使用 "${APACHE_ERR}"（外面加双引号的方式）引用变量；当变量内容为整数时，则最好直接使用 $APACHE_ERR 来引用变量。全局变量、局部变量、函数变量、数组变量等都是如此。

> 说明：对于需要环境变量的 Java 程序脚本等，在写脚本之前，最好通过 export 重新声明环境变量，以免在定时任务等场合的使用中出现问题。

14.3 Shell 函数的命名及函数定义规范

Shell 函数的命名可采用单词首字母大写的形式，如 CreateFile()，并且语义要清晰，比如，使用 CreateFile() 代替 CFile()，也可以使用小写形式，如 createfile()。

可以加前后缀，如后缀为 Max 则为最大值，为 Min 则表示最小值，前缀 Is 为判断型函数，Get 为取值函数，Do 则为处理函数，这也有益于对函数功能的理解，使函数名更直观、更清晰。

范例 14-3：对操作系统函数库脚本的函数名进行定义。

```
#/etc/init.d/functions
# Check whether file $1 is a backup or rpm-generated file and should be ignored
is_ignored_file() {   #<== 这里系统的函数名并没有大写，除了大小写之外，还是比较规范的。
    case "$1" in
        *~ | *.bak | *.orig | *.rpmnew | *.rpmorig | *.rpmsave)
            return 0
            ;;
    esac
    return 1
}
```

如果需要区别一些常规的字符串，可在函数名前加上 function 关键字，例如：

```
function CreateFile(){
}
```

显示函数返回值时，可在函数的结尾内容中包含 return 语句，并跟上返回值。即使

是不关心返回值的函数，也可能在后续调用时无意识地去判断它的返回值并进行一系列动作，使用 return 语句不会带来多少负担，但确实能让函数的逻辑变得更加清晰和严谨。

范例 14-4：为操作系统函数库脚本函数定义 return 返回值。

```
# Log a warning
warning() {
    local rc=$?
    #if [ -z "${IN_INITLOG:-}" ]; then
    #   initlog $INITLOG_ARGS -n $0 -s "$1" -e 1
    #fi
    [ "$BOOTUP" != "verbose" -a -z "${LSB:-}" ] && echo_warning
    return $rc
}
```

14.4　Shell 脚本（模块）高级命名规范

1）常规 Shell 脚本使用统一的后缀：.sh，例如 oldboy.sh。

2）模块的启动和停止脚本统一命名为 start_ 模块名 .sh 和 stop_ 模块名 .sh。

3）监控脚本通常以 *_mon.sh 为后缀。

4）控制脚本一般以 *_ctl.sh 为后缀。

14.5　Shell 脚本的代码风格

14.5.1　代码框架

易变的信息（如报警的收件人、机器名、端口、用户名密码、URL 等）最好都定义为变量或使用特殊位置的参数，这会使开发的脚本更具通用性。

把 Shell 的通用变量以配置文件的形式单独存放，以“功能 .cfg”来命名，例如 nginx.conf，并放入 conf 目录下；引用时通过在脚本开头使用 source conf/nginx.conf 的形式来加载。

将程序的功能分段、分模块采用函数等来实现，并存放到单独的函数文件里，如果是通用的公共函数可以存放于 /etc/init.d/functions 下，调用时采用 source 文件全路径即可。

把脚本中的功能和配置明确分开，主脚本只用于实现程序主干，加载配置及加载函数等功能实现应尽量封装在子函数中。

规范代码树如下。

```
[root@oldboy scripts]# tree
.
|-- bin
```

```
|   `-- ipsecctl
|-- conf
|   `-- ipsec.cfg
`-- func
    `-- functions
3 directories, 3 files
```

14.5.2 缩进规范

在使用条件语句时，每进行一层循环或是循环内部的操作时，就使用一个缩进，缩进一般用 TAB 键或加空格，本书推荐采用 4 个空格缩进。

范例 14-5：写出脚本缩进规范。

```
if [ -d oldboy_dir ]
  then
    cd oldboy_dir
    if [ -f oldboy_file ]
      then
        echo "DoSth"
    fi
    cd ..
fi
```

范例 14-6：给出系统脚本函数的缩进示例。

```
# Confirm whether we really want to run this service
confirm() {
  [ -x /usr/bin/rhgb-client ] && /usr/bin/rhgb-client --details=yes
  while : ; do
      echo -n $"Start service $1 (Y)es/(N)o/(C)ontinue? [Y] "
      read answer
      if strstr $"yY" "$answer" || [ "$answer" = "" ] ; then
         return 0
      elif strstr $"cC" "$answer" ; then
         rm -f /var/run/confirm
         [ -x /usr/bin/rhgb-client ] && /usr/bin/rhgb-client --details=no
         return 2
      elif strstr $"nN" "$answer" ; then
         return 1
      fi
  done
}
```

提示：可调整 vim 实现自动缩进，建议缩进 4 个空格。

14.6　Shell 脚本的变量及文件检查规范

脚本中要检查配置项是否为空、是否可执行等，尤其是对于一些重要的、会影响下面脚本正常运行的配置项，必须要进行是否为空等的检查，避免配置文件中出现遗漏等问题。

范例 14-7：针对字符串变量进行判断。

```
if [ -n "${FILE_PATH}" ]
  then
    echo "Do something"
fi
```

范例 14-8：给出 HTTP 脚本变量的定义方式。

```
httpd=${HTTPD-/usr/sbin/httpd}
prog=httpd
pidfile=${PIDFILE-/var/run/httpd.pid}
lockfile=${LOCKFILE-/var/lock/subsys/httpd}
```

提示：这样的定义可以防止变量出现空值，这是前面第 4 章讲解的变量子串的特殊知识。

第15章

Shell 脚本的调试

本章为大家讲解 Shell 脚本的调试知识，掌握了 Shell 脚本的调试技巧，可以让我们在开发大型脚本时做到事半功倍。虽然掌握 Shell 脚本的调试技巧很重要，但是如果能掌握并养成前面各个章节提到的 Shell 脚本开发的规范和习惯，就可以从源头上降低开发脚本的错误率，从而降低脚本调试的难度和时间，达到未雨绸缪的效果，这也是老男孩常说的平时应多学习好的习惯、规范和制度。不过，在讲解 Shell 脚本的调试之前，我们还是先来看几个常见的错误范例。

15.1 常见 Shell 脚本错误范例

15.1.1 if 条件语句缺少结尾关键字

范例 15-1：if 条件语句缺少结尾关键字引起的错误。

```
[root@oldboy scripts]# cat 15_1.sh
#!/bin/sh
if [ 10 -lt 12 ]
  then
    echo "Yes,10 is less than 12"
```

执行结果如下：

```
[root@oldboy scripts]# sh 15_1.sh
15_1.sh: line 5: syntax error: unexpected end of file
```

结果给出了提示，第 5 行存在语法错误：这不是所期待的（意外的）文件结尾。根据这个提示，我们知道脚本的尾部有问题，仔细观察发现，原来是缺少了 fi 结尾。

说明：在 Shell 脚本开发中，脚本缺少 fi 关键字是很常见的问题。另外，当执行脚本时提示输出错误后，不要只看那些提示的错误行，而是要观察整个相关的代码段。

Shell 脚本解释器一般不会对脚本错误进行精确的定位，而是在试图结束一个语句时进行错误统计，因此，掌握语法并养成良好的规范和习惯就显得很重要。

15.1.2　循环语句缺少关键字

for、while、until 和 case 语句中的错误是指实际语句段不正确，也许是漏写或拼错了固定结构中的一个保留字。

范例 15-2：循环结构语句中缺少关键字引起的错误。

```
[root@oldboy scripts]# cat -n 15_2.sh
   1  #!/bin/sh
   2  while true
   3  do
   4      status=`curl -I -s --connect-timeout 10 $1|head -1| awk '{print $2}'`
   5      ok=`curl -I -s --connect-timeout 10 $1|head -1|cut -d " " -f 3`
   6      if [ "$status" = "200" ] && [ "$ok" = "OK" ];the #<== 这里缺了个 n，应该为 then。
   7          echo "this url is good"
   8      else
   9          echo " this url is bad"
  10      fi
  11      sleep 3
  12  done
```

执行结果如下：

```
[root@oldboy scripts]# sh 15_2.sh www.baidu.com
15_2.sh: line 8: syntax error near unexpected token `else'
15_2.sh: line 8: `else'
```

通过提示可知，是第 8 行的语法错误，在“else”附近。经过前后观察可以发现，“the”少加了个“n”，应为“then”。

如果报错的是循环或条件独立的语句，对上下关联部分语句也都要看一下。

15.1.3　成对的符号落了单

成对的符号有 []、()、{}、""、''、`` 等，如果它们落了单，也会导致一些错误。

范例 15-3：成对的符号（双引号）落了单导致的错误。

```
[root@oldboy scripts]# cat -n 15_3.sh
     1  #!/bin/sh
     2  DBPATH=/server/backup
     3  MYUSER=root
     4  MYPASS=oldboy123
     5  SOCKET="/data/3306/mysql.sock          #==>这里故意设置成缺引号。
     6  MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"
     7  MYDUMP="mysqldump -u$MYUSER -p$MYPASS -S $SOCKET"
     8  [ ! -d "$DBPATH" ] && mkdir $DBPATH
     9  for dbname in `$MYCMD -e "show databases;"|sed '1,2d'|egrep -v "mysql|schema"`
    10  do
    11      $MYDUMP $dbname|gzip >$DBPATH/${dbname}_$(date +%F).sql.gz
    12  done
```

执行结果如下：

```
[root@oldboy scripts]# sh 15_3.sh
15_3.sh: line 7: unexpected EOF while looking for matching `"'
15_3.sh: line 13: syntax error: unexpected end of file
```

我们明明看到是第 5 行少了个双引号，但是报错的是第 7 行。原因是第 7 行和第 13 行调用了第 5 行的 sock 路径。

范例 15-4：成对的符号（中括号）落了单导致的错误。

```
[root@oldboy scripts]# cat -n 15_4.sh
     1  #!/bin/sh
     2  DBPATH=/server/backup
     3  MYUSER=root
     4  MYPASS=oldboy123
     5  SOCKET="/data/3306/mysql.sock"
     6  MYCMD="mysql -u$MYUSER -p$MYPASS -S $SOCKET"
     7  MYDUMP="mysqldump -u$MYUSER -p$MYPASS -S $SOCKET"
     8  [ ! -d "$DBPATH"  && mkdir $DBPATH     #==>这里缺了个后半个中括号“]”。
     9  for dbname in `$MYCMD -e "show databases;"|sed '1,2d'|egrep -v "mysql|schema"`
    10  do
    11      $MYDUMP $dbname|gzip >$DBPATH/${dbname}_$(date +%F).sql.gz
    12  done
```

执行结果如下：

```
[root@oldboy scripts]# sh 15_4.sh
15_4.sh: line 8: [: missing `]'                #==>这个错误提示很准。
```

15.1.4 中括号两端没空格

范例 15-5：中括号两端没空格导致的错误。

```
[root@oldboy scripts]# cat -n 15_5.sh
     1  #!/bin/bash
     2  #created by oldboy
     3  a=3
     4  b=1
     5  if [$a -lt $b]   #==> 中括号表达式两端无空格。
     6    then
     7      echo "Yes,$a < $b"
     8  else
     9      echo "Yes,$a >= $b"
    10  fi
```

执行结果如下：

```
[root@oldboy scripts]# sh 15_5.sh
15_5.sh: line 5: [3: command not found
Yes,3 >= 1
```

15.1.5　Shell 语法调试小结

Shell 的语法调试并不是很智能，报错也不是很精准，因此就需要我们在开发规范和书写脚本上多下工夫，企业里的 Shell 脚本大多都是比较短的，因此，开发起来也相对轻松。

如果能在开发过程中，重视书写习惯、开发规范和开发制度，那么就会减少脚本调试的难度和次数，提升开发效率。此外，要对 Shell 的基本语法十分熟练，这样才能更好地利用脚本调试。

此外，写脚本的思路要清晰，否则将给调试带来困难。可采用的思路如下：

首先思考开发框架，尽量模块化开发，复杂的脚本要简单化、分段实现。并采用打游戏的思想（第一关、第二关、第三关）去完善框架结构。

然后利用函数分模块开发，语法结构如下：

```
函数 1()
函数 2()
main()
main $*                    #<== 执行主函数。
```

需要注意的是，不要强制模块化，分块要合理。

15.2　Shell 脚本调试技巧

15.2.1　使用 dos2unix 命令处理在 Windows 下开发的脚本

范例 15-6：将 Windows 下编辑的脚本放置到 Linux 下执行的案例。

将 Windows 下编辑的脚本放置到 Linux 下执行的情况如下：

```
[root@oldboy scripts]# cat -v while.sh
#!/bin/bash
# this script is created by oldboy.
#!/bin/sh^M
i=1^M
sum=0^M
while ((i <=100 ))^M
do^M
        ((sum=sum+i))^M
        ((i++))^M
done^M
printf "totalsum is :$sum\n"^M
[root@oldboy scripts]# sh while.sh
'while.sh: line 10: syntax error near unexpected token `
'while.sh: line 10: `while ((i <=100 ))
```

你可能会发现，对于在 Windows 下开发的脚本，明明经检查没有发现问题，但就是在执行时会出现莫名其妙的语法错误。这时，最好执行 dos2unix 格式化一下。执行 dos2unix 格式化是一个很好的习惯。

```
[root@oldboy scripts]# dos2unix while.sh  #==>使用 dos2unix 格式化后的结果。
dos2unix: converting file while.sh to UNIX format ...
[root@oldboy scripts]# cat -v while.sh
#!/bin/bash
# this script is created by oldboy.
# function:while-3 example
# version:1.1
#!/bin/sh
i=1
sum=0
while ((i <=100 ))
do
    ((sum=sum+i))
    ((i++))
done
printf "totalsum is :$sum\n"
提示：正常了。^M 消失了。Windows 下代码的换行符和 Linux 下的不一样，导致了本例的问题。
[root@oldboy scripts]# sh while.sh
totalsum is :5050
```

如果没有安装 dos2unix，则用下面的命令进行安装：

```
yum install dos2unix -y
```

15.2.2 使用 echo 命令调试

echo 命令是最有用的调试脚本的工具之一。一般应在可能出现问题的脚本的重要部分加入 echo 命令，例如在变量读取或修改操作的前后加入 echo 命令，并紧挨着退出命令 exit。

范例 15-7：利用 echo 调试一个简单的判断脚本。

```
[root@oldboy scripts]# cat -n if-judgenum4-debug.sh
     1  #!/bin/bash
     2  #created by oldboy
     3  #date:20100918
     4  #function:int compare
     5  read -p "pls input two num:" a b
     6  echo $a $b  #<== 增加打印输出，确认变量值是否符合要求。
     7  exit        #<== 退出脚本，目的是不执行后面的代码。
     8  ############
     9  #if [ $a -lt $b ]
    10  if (($a < $b ))
    11    then
    12      echo "$a<$b"
    13  elif [ $a -eq $b ]
    14    then
    15      echo "$a=$b"
    16  else
    17      echo "$a>$b"
    18  fi
```

执行结果如下：

```
[root@oldboy scripts]# sh if-judgenum4-debug.sh
pls input two num:99 88
99>88
```

这个调试方法不是 Shell 的专利，PHP、ASP、Perl、Python 等语言都可以使用这样简单又好用的调试方法。

15.2.3　使用 bash 命令参数调试

除了上述的方法之外，本节将讲解最重量级的方法 bash 或 sh 的参数调试方法，例如：

```
[root@oldboy ~]# sh [-nvx] scripts.sh
```

参数说明如下。

- -n：不会执行该脚本，仅查询脚本语法是否有问题，并给出错误提示。
- -v：在执行脚本时，先将脚本的内容输出到屏幕上，然后执行脚本，如果有错误，也会给出错误提示。
- -x：将执行的脚本内容及输出显示到屏幕上，这是对调试很有用的参数。

说明：这些参数同样适用于 bash。

1. sh 参数 -n 的测试

范例 15-8：通过 -n 参数对脚本进行语法检查。

```
[root@oldboy ~]# cat script.sh
#!/bin/bash
echo "Hello $USER,"
echo "Today is $(date +%F)"
[root@oldboy ~]# sh -n script.sh
#<==-n 不会执行该脚本，仅用于查询脚本语法是否有问题，并给出错误提示。此处没有语法问题，因此不显示任何信息。
```

下面让脚本语法存在错误，然后再测试，这里把 echo "Hello $USER" 的后半双引号去掉了。

```
[root@oldboy ~]# cat -n script.sh
     1  #!/bin/bash
     2  echo "Hello $USER  #<== 结尾少了个双引号。
     3  echo "Today is $(date +%F)"
[root@oldboy ~]# sh -n script.sh
script.sh: line 3: unexpected EOF while looking for matching `"'
script.sh: line 4: syntax error: unexpected end of file
  #<== 上面两行报语法错误，提示的内容正是第 3 行结尾没有双引号。
```

2. sh 参数 -v 的测试

范例 15-9：对参数 -v 的测试。

普通的错误脚本的执行结果如下：

```
[root@oldboy ~]# sh -v script.sh
#!/bin/bash
echo "Hello $USER,
echo "Today is $(date +%F)" #<== 以上几行是完成的脚本内容，下面两行同 -n 的功能，
                                用于检查脚本语法。
script.sh: line 3: unexpected EOF while looking for matching `"'
script.sh: line 4: syntax error: unexpected end of file
```

带函数的错误脚本的执行结果如下：

```
[root@oldboy ~]# sh -v c.sh
. ./function
#!/bin/bash
oldboy(){
    echo " I am oldboy! "
}
oldgirl(){
    echo " I am oldgirl "
}
oldboy1
c.sh: line 2: oldboy1: command not found
```

```
oldgirl
I am oldgirl
```

3. sh 参数 -x 的测试

范例 15-10：跟踪 script.sh 脚本的执行过程。

```
[root@oldboy ~]# cat script.sh
#!/bin/bash
echo "Hello $USER,"
echo "Today is $(date +%F)"
```

执行结果如下：

```
[root@oldboy ~]# sh -x script.sh
+ echo 'Hello root,'
Hello root,
++ date +%F
+ echo 'Today is 2011-08-05'
Today is 2011-08-05
#<== 使用 -x 追踪脚本是一种非常好的方法，它可以在执行前列出所执行的所有程序段，
#<== 如果是程序段落，则在输出时，最前面会加上 + 符号，表示它是程序代码，
#<== 一般情况下如果是调试逻辑错误的脚本，用 -x 的效果更佳。
# 缺点：加载系统函数库等很多我们不想查看其整个过程的脚本时，会有太多输出，导致很难查看所需要的内容。
```

利用“sh -x 脚本名”调试的缺点可以用 set -x 命令来弥补，它可以缩小调试的作用域。

范例 15-11：在 script.sh 脚本的执行过程中输出脚本的行号，以便于跟踪。

在一个比较长的脚本中，你会看到很多的执行跟踪的输出，有时候阅读起来非常费劲，此时，可以在每一行的前面加上内容的行号，这会非常有用。要做到这样，只需要设置下面的环境变量：

```
[root@oldboy ~]# set|grep PS[1-5]
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
# 提示：PS4 变量在默认情况下表示加号。
[root@oldboy ~]# export PS4='+${LINENO}' #<== 此命令即可实现在跟踪过程中显示行号，
                                              也可以放到脚本中。
[root@oldboy ~]# sh -x script.sh
+2 echo 'Hello root,'
Hello root,
++3 date +%F
+3 echo 'Today is 2011-08-05'
Today is 2011-08-05
# 提示：+ 号后面的数字就表示行号。
```

这时可以看到，bash 在运行前打印出了每一行的命令。而且每行前面的 + 号均表

明了嵌套。这样的输出可以让你看到命令执行的顺序，并且可以让你知道整个脚本的行为。

特别说明：

参数 -x 是一个不可多得的参数，在生产环境中，老男孩就经常通过参数 -x 来实现调试的目的。一般情况下，如果执行脚本发生问题（非语法问题时），利用 -x 参数，就可以知道问题出在哪一行。

15.2.4 使用 set 命令调试部分脚本内容

set 命令也可以用于辅助脚本调试。以下是 set 命令常用的调试选项。

- set -n：读命令但并不执行。
- set -v：显示读取的所有行。
- set -x：显示所有命令及其参数。

提示：通过 set -x 命令开启调试功能，而通过 set +x 关闭调试功能。

set 命令的最大优点是，和 bash -x 相比，set -x 可以缩小调试的作用域。

范例 15-12：调试范例 11-8 开发的打印九九乘法表的简版脚本。

```
[root@oldboy scripts]# cat -n for-7-9x9-2-debug.sh
     1  #!/bin/bash
     2  set -x  #<== 表示开启脚本调试。
     3  for a in `seq 9`
     4  do
     5      for b in `seq 9`
     6      do
     7          [ $a -ge $b ] && echo -en "$a x $b = $(expr $a \* $b)  "
     8      done
     9  set +x  #<== 到这里结束脚本调试，即只针对 sed -x 和 set +x 之间的脚本进行调试。
    10  echo " "
    11  done
```

执行脚本查看调试输出结果：

```
[root@oldboy scripts]# sh for-7-9x9-2-debug.sh
++ seq 9
+ for a in ``seq 9``
++ seq 9
+ for b in ``seq 9``
+ '[' 1 -ge 1 ']'
++ expr 1 '*' 1
```

```
    + echo -en '1 x 1 = 1  '
    1 x 1 = 1  + for b in ``seq 9``
    + '[' 1 -ge 2 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 3 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 4 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 5 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 6 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 7 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 8 ']'
    + for b in ``seq 9``
    + '[' 1 -ge 9 ']'
    + set +x
    #<== 可以看出跟踪脚本运行的过程，但遇到 set +x 就结束了。
    2 x 1 = 2  2 x 2 = 4
    3 x 1 = 3  3 x 2 = 6  3 x 3 = 9
    4 x 1 = 4  4 x 2 = 8  4 x 3 = 12  4 x 4 = 16
    5 x 1 = 5  5 x 2 = 10  5 x 3 = 15  5 x 4 = 20  5 x 5 = 25
    6 x 1 = 6  6 x 2 = 12  6 x 3 = 18  6 x 4 = 24  6 x 5 = 30  6 x 6 = 36
    7 x 1 = 7  7 x 2 = 14  7 x 3 = 21  7 x 4 = 28  7 x 5 = 35  7 x 6 = 42  7 x 7 = 49
    8 x 1 = 8  8 x 2 = 16  8 x 3 = 24  8 x 4 = 32  8 x 5 = 40  8 x 6 = 48  8 x 7 = 56
8 x 8 = 64
    9 x 1 = 9  9 x 2 = 18  9 x 3 = 27  9 x 4 = 36  9 x 5 = 45  9 x 6 = 54  9 x 7 = 63
9 x 8 = 72  9 x 9 = 81
```

提示：加了 set -x，在运行脚本的时候，就不需要使用 sh -x 了。

熟悉 sh 及 set 的用法，可以让我们能够得心应手地管理 Linux 下 Shell 脚本执行的过程。在 Shell 脚本的学习上，需要“多看、多模仿，并将已有脚本修改成自己需要的样式”，这是最快的学习手段。网络上有很多实用的脚本，作为初学者，可以先将这些脚本拿过来，将其改写为适合自己服务器的脚本，然后再慢慢熟悉，慢慢尝试开发新的脚本。这种先模仿后开发的学习方法，会让你的学习事半功倍，而且学习起来也会很有兴趣和成就感。

Linux 系统上本来就有很多适合阅读的规范脚本，如果想要更加深入地掌握 Shell 脚本编程，最好的方法就是阅读系统的这些脚本，然后仔细研究每一行都是什么作用，为什么这样写，久而久之，你的 Shell 开发水平就会得到大幅度提高。

15.2.5 其他调试 Shell 脚本的工具

下面再引荐两款 Shell 的调试工具，由于老男孩已经习惯了前文讲解的调试方式，因此没有过多地研究过这些调试工具，如果读者有兴趣可以作为扩展知识来研究一下。

（1）Shell 调试工具：bashdb

Shell 调试器 bashdb 是一个类似于 GDB 的调试工具，可以完成对 Shell 脚本的断点设置、单步执行、变量观察等许多功能，但是老男孩很少使用它。

（2）Shell 调试工具：shellcheck

shellcheck 是一个可检查 sh/bash 脚本和命令语法的小工具，地址为：http://www.shellcheck.net/，老男孩也很少使用它。

15.3 本章小结

本章主要介绍了 Shell 的调试技巧，包括：

1）要记得首先用 dos2unix 对脚本（从其他地方拿来用的）进行格式化。

2）执行脚本根据报错来调试时，要知道有时所报错误会不准确，应多关联上下文查看。

3）可通过 sh -x 命令调试整个脚本，且显示执行过程。

4）set -x 和 set +x 命令用于调试部分脚本的执行过程（可在脚本中设置）。

5）可通过 echo 命令输出脚本中要确认的变量及相关内容，然后紧跟着使用 exit 退出，不执行后面程序，这种方式便于一步步跟踪脚本，对于逻辑错误的调试比较好用。写法即“echo $var;exit”

6）最关键的还是要语法熟练，养成良好的编码习惯，提高编程思想，将错误扼杀在萌芽状态之中，从而降低错误率，减轻调试的负担，提高开发效率。

Linux

第16章

Shell 脚本开发环境的配置和优化实践

本章以 Linux 下的编辑器 vim 为例来介绍 Shell 脚本开发的基本配置和优化，如果读者习惯用 Windows 下的编辑器来编辑 Shell 脚本也是可以的，不过有可能会因为编码格式等问题而导致开发的脚本拿到 Linux 下执行时产生错误（前面第 15 章调试章节已说明），因此，老男孩还是建议在 Linux 下开发 Shell 脚本。vim 编辑器非常强大，绝大多数的 Shell 开发场景，vim 都能够胜任。

16.1 使用 vim 而不是 vi 编辑器

vi 编辑器的功能类似于 Windows 下的记事本，比较适合编辑普通文本，但是用于编写脚本代码就不太适合了，例如缺少高亮显示代码、自动缩进等重要的功能；而 vim 编辑器则相当于 Windows 下的高级编辑器，类似 emeditor、editplus、notepad++ 等，为了提高开发效率，需要使用 vim 而不是 vi。当然了，vi 也是可以用来编写代码的，只不过效率不高而已。

因此，首先要做如下调整，以便只使用 vim 作为开发脚本的工具：

```
[root@oldboy scripts]# echo 'alias vi=vim' >>/etc/profile
[root@oldboy scripts]# tail -1 /etc/profile
alias vi=vim
[root@oldboy scripts]# source /etc/profile
```

经过上述调整后，当用 vi 命令时，会自动被 vim 替换。

16.2 配置文件 .vimrc 的重要参数介绍

Linux 环境下的 vim 编辑器默认功能不够强大，如果要进行 Shell 脚本的开发，还需要进行适当的设置，从而达到高效开发的目的。vim 编辑器有一个可以用来调整配置的配置文件，默认放置在用户家目录下，全路径及名字组合为：~/.vimrc（全局路径为 /etc/vimrc），这是一个隐藏文件，下面是老男孩在企业里开发 Shell 脚本时，对 .vimrc 进行的一个常用设置，供大家参考，具体参数及内容说明如下：

```
" ~/.vimrc
" vim config file
" date 2008-09-05
" Created by oldboy
" blog:http://oldboy.blog.51cto.com
"""""""""""""""""""""""
" => 全局配置
"""""""""""""""""""""""
"关闭兼容模式
set nocompatible

"设置历史记录步数
set history=100

"开启相关插件
filetype on
filetype plugin on
filetype indent on

"当文件在外部被修改时，自动更新该文件
set autoread

"激活鼠标的使用
set mouse=a

"""""""""""""""""""""""
" => 字体和颜色
"""""""""""""""""""""""
"开启语法
syntax enable

"设置字体
"set guifont=dejaVu\ Sans\ MONO\ 10
"
"" 设置配色
"colorscheme desert

"高亮显示当前行
set cursorline
```

```
hi cursorline guibg=#00ff00
hi CursorColumn guibg=#00ff00

"""""""""""""""""""""""""
" => 代码折叠功能 by oldboy
"""""""""""""""""""""""""
"激活折叠功能
set foldenable

"设置按照语法方式折叠(可简写 set fdm=XX)
"有 6 种折叠方法:
"manual 手工定义折叠
"indent 更多的缩进表示更高级别的折叠
"expr   用表达式来定义折叠
"syntax 用语法高亮来定义折叠
"diff   对没有更改的文本进行折叠
"marker 对文中的标志进行折叠
set foldmethod=manual

"设置折叠区域的宽度
"如果不为 0,则在屏幕左侧显示一个折叠标识列
"分别用"-"和"+"来表示打开和关闭的折叠。
set foldcolumn=0

"设置折叠层数为 3
setlocal foldlevel=3

"设置为自动关闭折叠
set foldclose=all

"用空格键来代替 zo 和 zc 快捷键实现开关折叠
"zo  O-pen a fold   (打开折叠)
"zc  C-lose a fold  (关闭折叠)
"zf  F-old creation (创建折叠)
nnoremap <space> @=((foldclosed(line('.')) < 0) ? 'zc' : 'zo')<CR>
"""""""""""""""""""""""""
" => 文字处理 by oldboy
"""""""""""""""""""""""""
"使用空格来替换 Tab
set expandtab

"设置所有的 Tab 和缩进为 4 个空格
set tabstop=4

"设定 << 和 >> 命令移动时的宽度为 4
set shiftwidth=4

"使得按退格键时可以一次删掉 4 个空格
```

```
set softtabstop=4

set smarttab

" 缩进，自动缩进（继承前一行的缩进）
"set autoindent 命令关闭自动缩进，是下面配置的缩写。
" 可使用 autoindent 命令的简写，即 ":set ai" 和 ":set noai"。
" 还可以使用 " :set ai sw=4" 在一个命令中打开缩进并设置缩进级别。
set ai

" 智能缩进
set si

" 自动换行
set wrap

" 设置软宽度
set sw=4
"""""""""""""""""""""
" => Vim 界面 by oldboy
"""""""""""""""""""""
"Turn on WiLd menu
set wildmenu

" 显示标尺
set ruler

" 设置命令行的高度
set cmdheight=1

" 显示行数
"set nu

"Do not redraw, when running macros.. lazyredraw
set lz

" 设置退格
set backspace=eol,start,indent

"Bbackspace and cursor keys wrap to
set whichwrap+=<,>,h,l

"Set magic on(设置魔术)
set magic

" 关闭遇到错误时的声音提示
" 关闭错误信息响铃
set noerrorbells

" 关闭使用可视响铃代替呼叫
```

```
set novisualbell

" 显示匹配的括号 ([{ 和 }])
set showmatch

"How many tenths of a second to blink
set mat=2

" 搜索时高亮显示搜索到的内容
set hlsearch

" 搜索时不区分大小写
" 还可以使用简写 (":set ic" 和 ":set noic")
set ignorecase

"""""""""""""""""""""""
" => 编码设置
"""""""""""""""""""""""
" 设置编码
set encoding=utf-8
" 设置文件编码
set fileencodings=utf-8

" 设置终端编码
set termencoding=utf-8
"""""""""""""""""""""""
" => 其他设置 by oldboy 2010
"""""""""""""""""""""""
" 开启新行时使用智能自动缩进
set smartindent
set cin
set showmatch

" 隐藏工具栏
set guioptions-=T

" 隐藏菜单栏
set guioptions-=m

" 置空错误铃声的终端代码
set vb t_vb=

" 显示状态栏 (默认值为 1, 表示无法显示状态栏)
set laststatus=2

" 粘贴不换行问题的解决方法
set pastetoggle=<F9>

" 设置背景色
```

```
set background=dark

" 设置高亮相关
highlight Search ctermbg=black  ctermfg=white guifg=white guibg=black
```

说明：读者只需简单了解这些参数即可，实际使用时只需把老男孩给的配置文件放到用户的家目录下，然后退出重新登录即可使用 vim。

在 Shell 脚本的开头自动增加解释器及作者等版权信息

```
autocmd BufNewFile *.py,*.cc,*.sh,*.java exec ":call SetTitle()"
func SetTitle()
    if expand("%:e") == 'sh'
        call setline(1, "#!/bin/bash")
        call setline(2, "#Author:oldboy")
        call setline(3, "#Blog:http://oldboy.blog.51cto.com")
        call setline(4, "#Time:".strftime("%F %T"))
        call setline(5, "#Name:".expand("%"))
        call setline(6, "#Version:V1.0")
        call setline(7, "#Description:This is a test script.")
    endif
endfunc
```

去掉注释后的配置（推荐使用此配置）如下：

```
[root@oldboy ~]# cat .vimrc-nozhushi
set nocompatible
set history=100
filetype on
filetype plugin on
filetype indent on
set autoread
set mouse=a
syntax enable
set cursorline
hi cursorline guibg=#00ff00
hi CursorColumn guibg=#00ff00
set nofen
set fdl=0
set expandtab
set tabstop=4
set shiftwidth=4
set softtabstop=4
set smarttab
set ai
set si
```

```
set wrap
set sw=4
set wildmenu
set ruler
set cmdheight=1
set lz
set backspace=eol,start,indent
set whichwrap+=<,>,h,l
set magic
set noerrorbells
set novisualbell
set showmatch
set mat=2
set hlsearch
set ignorecase
set encoding=utf-8
set fileencodings=utf-8
set termencoding=utf-8
set smartindent
set cin
set showmatch
set guioptions-=T
set guioptions-=m
set vb t_vb=
set laststatus=2
set pastetoggle=<F9>
set background=dark
highlight Search ctermbg=black  ctermfg=white guifg=white guibg=black
autocmd BufNewFile *.py,*.cc,*.sh,*.java exec ":call SetTitle()"
func SetTitle()
    if expand("%:e") == 'sh'
        call setline(1, "#!/bin/bash")
        call setline(2, "#Author:oldboy")
        call setline(3, "#Blog:http://oldboy.blog.51cto.com")
        call setline(4, "#Time:".strftime("%F %T"))
        call setline(5, "#Name:".expand("%"))
        call setline(6, "#Version:V1.0")
        call setline(7, "#Description:This is a test script.")
    endif
endfunc
```

说明：如果不方便敲出来，可以在网上搜索一下 .vimrc 的配置，然后对着本文修改即可，或者加入本书前言部分提到的 QQ 群获取。

vim 路径等配置知识的整理见表 16-1。

表 16-1 vim 路径等配置知识

相关配置文件	功能描述
.viminfo	用户使用 vim 的操作历史
.vimrc	当前用户 vim 的配置文件
/etc/vimrc	系统全局 vim 的配置文件
/usr/share/vim/vim74/colors/	配色模板文件存放路径

16.3 让配置文件 .vimrc 生效

将 vim 的配置文件 .vimrc 上传到 Linux 系统的“~”目录下，然后退出 SSH 客户端重新登录，即可应用 .vimrc 里对应的设置。示例如下：

```
[root@oldboy scripts]# ll ~/.vimrc
-rw-r--r-- 1 root root 2595 9月   8 10:52 /root/.vimrc
```

提示：同样适用于普通用户。

重新登录后，当使用 vim 时就会自动加载 .vimrc 设定的配置。

16.4 使用 vim 编辑器进行编码测试

16.4.1 代码自动缩进功能

图 16-1 显示了使用代码自动缩进功能的效果，这个自动缩进的功能非常好用，当输入循环及条件结构语句等代码时，系统会自动将输入语句的关键字及命令代码缩进到合理的位置，可以看到，vim 的配置是以两个空格为缩进宽度（.vimrc 里设置的）的。

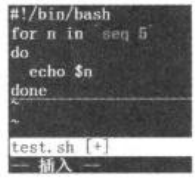


图 16-1 代码自动缩进功能说明

16.4.2 代码颜色高亮显示功能说明

代码颜色高亮显示也是一个非常好的功能，可以通过它区分字符、变量、循环等很多不同的 Shell 脚本元素。例如当编写的代码出现错误时，对应的代码高亮颜色就会和正确时的不同，开发者可以根据代码的高亮颜色对 Shell 脚本进行调试，提升编码的效率，减少编码的错误，图 16-2 是故意把 `seq 5` 中的后反引号去掉后的截图，可以看到整个 for 循环

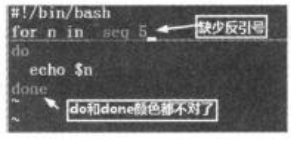


图 16-2 代码颜色高亮显示功能

体的关键字（do 和 done）颜色立刻变得不一致（正常是黄色，现在变成红色）了，从而可以判断出脚本对应不同颜色的代码周边有错误。

16.5　vim 配置文件的自动增加版权功能

当执行“vim oldboy.sh”编辑脚本时，只要是以 .sh 为扩展名的，就会自动增加版权信息功能，如图 16-3 所示。

```
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
#Time:2016-09-09 13:29:14
#Name:13_4_1.sh
#Version:V1.0
#Description:a test script.
```

图 16-3　在脚本开头自动增加版权的功能

16.6　vim 配置文件的代码折叠功能

vim 非常强大，只不过对有些功能需要进行额外配置，下面就演示一下在代码量较大时比较有用的高级功能——代码折叠（依赖 .vimrc 配置，当然也可以以命令模式执行）。

在命令模式下，可以把光标定位到当前的第 2 行，然后执行 zf3j 命令，便可将第 2 行及其下的 3 行缩进，其他缩进也是如此，如图 16-4 所示。

```
 1 #!/bin/bash
 2 +--  4 行: CheckDb(){
 6     count=0
 7 +--  4 行: if [ "${status[${i}]}" != "Yes" -a "${status[${i}]}" != "0" ]
11 done
12 +-- 16 行: if [ $count -ne 0 ];then
```

图 16-4　代码折叠后的效果

若把光标放到对应折叠后的行上，按空格键即可展开上述折叠的行，如图 16-5 所示。

```
 1 #!/bin/bash
 2 CheckDb(){
 3 status=($(awk -F ': ' '/_Running|_Behind/{print $NF}' slave.log))
 4 for((i=0;i<${#status[*]};i++))
 5 do
 6     count=0
 7     if [ "${status[${i}]}" != "Yes" -a "${status[${i}]}" != "0" ]
 8         then
 9         let count+=1
10     fi
11 done
12 +-- 16 行: if [ $count -ne 0 ];then
```

图 16-5　展开折叠代码的效果

16.7　vim 编辑器批量缩进及缩进调整技巧

有时我们从外部复制部分 Shell 代码到当前脚本后发现缩进是乱的，如图 16-6 所示。

```
stop)
        for ((i=0; i< echo ${#VIP[*]} ; i++))
        do
            interface="lo:`echo ${VIP[$i]}|awk -F . '{print $4}'`"
           /sbin/ip addr del ${VIP[$i]}/24 dev lo label $interface
        done
        echo "0" >/proc/sys/net/ipv4/conf/lo/arp_ignore
        echo "0" >/proc/sys/net/ipv4/conf/lo/arp_announce
        echo "0" >/proc/sys/net/ipv4/conf/all/arp_ignore
        echo "0" >/proc/sys/net/ipv4/conf/all/arp_announce
                if [ $RETVAR -eq 0 ];then
                        action "Start LVS Config of RearServer." /bin/true
                    else
                                        action "Start LVS Config of RearServer." /bin/false
                                            fi
        ;;
```

图 16-6　缩进是乱的图形展示

此时可以将 vim 编辑器调整为命令模式（按 Esc 键），然后移动键盘上下键将光标定位到要调整的行开头，如图 16-7 所示。

```
        if [ $RETVAR -eq 0 ];then
                        action "Start LVS Config of RearServer." /bin/true
                    else
                                        action "Start LVS Config of RearServer." /bin/false
                                            fi
```

图 16-7　定位光标图示

接下来输入“v”(可视化缩写)，然后用键盘移动光标选定要调整的多行，如图 16-8 所示。

```
                if [ $RETVAR -eq 0 ];then
                        action "Start LVS Config of RearServer." /bin/true
                    else
                                        action "Start LVS Config of RearServer." /bin/false
                                            fi
        ;;
ipvs_client.sh [+]                                              44,9          86%
-- 可视 --
```

图 16-8　调整缩进命令展示

最后按“=”键即可将代码调整为规整的格式，效果如图 16-9 所示。

```
stop)
        for ((i=0; i< echo ${#VIP[*]} ; i++))
        do
            interface="lo:`echo ${VIP[$i]}|awk -F . '{print $4}'`"
           /sbin/ip addr del ${VIP[$i]}/24 dev lo label $interface
        done
        echo "0" >/proc/sys/net/ipv4/conf/lo/arp_ignore
        echo "0" >/proc/sys/net/ipv4/conf/lo/arp_announce
        echo "0" >/proc/sys/net/ipv4/conf/all/arp_ignore
        echo "0" >/proc/sys/net/ipv4/conf/all/arp_announce
        if [ $RETVAR -eq 0 ];then
          action "Start LVS Config of RearServer." /bin/true
        else
          action "Start LVS Config of RearServer." /bin/false
        fi
        ;;
ipvs_client.sh [+]
缩进了 6 行
```

图 16-9　最后调整后的结果

最后再根据代码需要进行编辑。

16.8　其他 vim 配置文件功能说明

vim 还可以实现显示当前行、显示光标的坐标位置等功能，除此之外，还有搜索、割裂窗口等更多功能，这些就留给读者自己去尝试吧，图 16-10 为 vim 显示当前行及光标的坐标位置图。

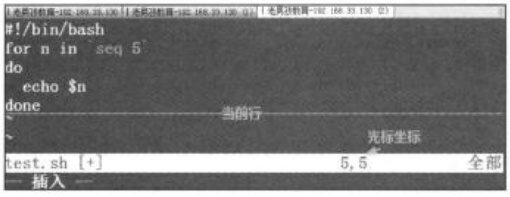


图 16-10　vim 显示当前行及光标的坐标位置图

16.9　vim 编辑器常用操作技巧

老男孩将 vi/vim 编辑器常用的操作技巧整理成表 16-2 中的内容，供读者参考。

表 16-2　vi/vim 编辑器常用操作技巧

命　令	说　明
普通模式：移动光标的操作	
G 或（Shift+g）	将光标移动到文件的最后一行
gg	将光标移动到文件的第一行，等价于 1gg 或 1G
0	数字 0，表示将光标从所在位置移动到当前行的开头
$	从光标所在位置将光标移动到当前行的结尾
n<Enter>	n 为数字，<Enter> 为回车键，表示将光标从当前位置向下移动 n 行
ngg	n 为数字，表示移动到文件的第 n 行，如 11gg 表示移动到第 11 行，可配合“:set nu”查看，同 nG
H	光标移动到当前窗口最上方的那一行
M	光标移动到当前窗口中间的那一行
L	光标移动到当前窗口最下方的那一行
h 或（←）	光标向左移动一个字符
j 或（↓）	光标向下移动一个字符
k 或（↑）	光标向上移动一个字符
l 或（→）	光标向右移动一个字符

（续）

命　令	说　明
普通模式：搜索与替换操作	
/oldboy	从光标位置开始，向下寻找名为 oldboy 的字符串
?oldboy	从光标位置开始，向上寻找名为 oldboy 的字符串
n	从光标位置开始，向下重复前一个搜索的动作
N	从光标位置开始，向上重复前一个搜索的动作
:g/A/s//B/g	把符合 A 的内容全部替换为 B，斜线为分隔符，可以用 @、# 等替代
:%s/A/B/g	把符合 A 的内容全部替换为 B，斜线为分隔符，可以用 @、# 等替代
:n1,n2s/A/B/gc	n1、n2 为数字，表示在第 n1 行和 n2 行间寻找 A，且用 B 替换
普通模式：复制、粘贴、删除等操作	
Yy	复制光标所在的当前行
nyy	n 为数字，表示复制从光标开始向下的 n 行
p/P	p 表示将已复制的数据粘贴到光标的下一行，P 表示粘贴到光标的上一行
dd	删除光标所在的当前行
ndd	n 为数字，表示删除从光标开始向下的 n 行
u	恢复（回滚）前一个执行过的操作
.	点号，重复前一个执行过的动作
进入编辑模式命令	
i	在当前光标所在处插入文字
a	在当前光标所在位置的下一个字符处插入文字
I	在当前所在行的行首第一个非空格符处开始插入文字，和 A 相反
A	在当前所在行的行尾最后一个字符处开始插入文字，和 I 相反
O	在当前所在行的上一行处插入新的一行
o	在当前所在行的下一行处插入新的一行
Esc	退出编辑模式，回到命令模式中
命令行模式	
:wq	退出并保存
:wq!	退出并强制保存，“!”为强制的意思
:q!	强制退出，不保存
:n1,n2 w filename	n1、n2 为数字，表示将 n1 行到 n2 行的内容保存成 filename 这个文件
:n1,n2 co n3	n1、n2 为数字，表示将 n1 行到 n2 行的内容复制到 n3 位置下
:n1,n2 m n3	n1、n2 为数字，表示将 n1 行到 n2 行的内容挪至 n3 位置下
:!command	暂时离开 vi，到命令行模式下执行 command 的显示结果！例如：!ls /etc

（续）

命　令	说　明
:set nu	显示行号
:set nonu	与 set nu 相反，取消行号
:vs filename	垂直分屏显示，同时显示当前文件和 filename 对应文件的内容
:sp filename	水平分屏显示，同时显示当前文件和 filename 对应文件的内容
I + # + Esc	在可视块模式下（按 Ctrl+V 键），一次性注释所选的多行，取消注释可用 :n1,n2s/#//gc
Del	在可视块模式下（按 Ctrl+V 键），一次性删除所选内容
r	在可视块模式下（按 Ctrl+V 键），一次性替换所选内容

Linux

第17章

Linux 信号及 trap 命令的企业应用实践

17.1 信号知识

17.1.1 信号介绍

运行 Shell 脚本程序时，如果按下快捷键 Ctrl+c 或 Ctrl+x（x 为其他字符），程序就会立刻终止运行。

在有些情况下，我们并不希望 Shell 脚本在运行时被信号中断，此时就可以使用屏蔽信号手段，让程序忽略用户输入的信号指令，从而继续运行 Shell 脚本程序。

简单地说，Linux 的信号是由一个整数构成的异步消息，它可以由某个进程发给其他的进程，也可以在用户按下特定键发生某种异常事件时，由系统发给某个进程。

17.1.2 信号列表

在 Linux 下和信号相关的常见命令为 kill 及 trap 命令，本章将重点讲解如何使用 trap 命令，以及如何利用 trap 控制的跳板机脚本来使用信号。

执行 kill -l 或 trap -l 命令，可以列出系统支持的各种信号，多达 64 个，如图 17-1 所示。

```
[root@oldboy scripts]# trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

图 17-1　系统支持的各种信号

下面将对企业实战中 Linux 系统的重要信号进行说明，见表 17-1。

在使用信号名时需要省略 SIG 前缀。

表 17-1　企业实战中 Linux 系统的重要信号及说明

信　号	说　明
HUP(1)	挂起，通常因终端掉线或用户退出而引发
INT(2)	中断，通常因按下 Ctrl+c 组合键而引发
QUIT(3)	退出，通常因按下 Ctrl+\ 组合键而引发
ABRT(6)	中止，通常因某些严重的执行错误而引发
ALRM(14)	报警，通常用来处理超时
TERM(15)	终止，通常在系统关机时发送
TSTP(20)	停止进程的运行，但该信号可以被处理和忽略，通常因按下 Ctrl+z 组合键而引发

通常需要忽略的信号包括 HUP、INT、QUIT、TSTP、TERM 等，对应的信号编号分别为 1、2、3、20、15。Shell 脚本中既可以用数字来代表信号，也可以使用信号的名字来代表信号，如果读者想要了解关于信号的更多知识，可以加本书的 QQ 交流群（本书前言中有提供），群里有详细的文档说明。

17.2　使用 trap 控制信号

trap 命令用于指定在接收到信号后将要采取的行动，信号的相关说明前面已经提到过。trap 命令的一种常见用途是在脚本程序被中断时完成清理工作，或者屏蔽用户非法使用的某些信号。在使用信号名时需要省略 SIG 前缀。可以在命令提示符下输入命令 trap -l 来查看信号的编号及其关联的名称。

trap 命令的参数分为两部分，前一部分是接收到指定信号时将要采取的行动，后一部分是要处理的信号名。

trap 命令的使用语法如下：

```
trap command signal
```

signal 是指接收到的信号，command 是指接收到该信号应采取的行动。也就是：

```
trap '命令；命令' 信号编号
```

或

```
trap '命令；命令' 信号名
```

范例 17-1：测试 trap 命令捕获 Ctrl+c 信号。

```
[root@oldboy ~]# trap 'echo oldboy' 2  #<== 当执行此命令时，按 Ctrl+c 键，就会执行
echo 命令，这里结尾的 2 就是 Ctrl+c 键对应的数字信号。
```

```
[root@oldboy ~]# ^Coldboy    #<== 按 Ctrl+c 键后调用 echo 命令输出结果。

[root@oldboy ~]# ^Coldboy    #<== 按 Ctrl+c 键后调用 echo 命令输出结果。
[root@oldboy ~]# trap "echo oldgirl" INT  #<== 当执行此命令时，按 Ctrl+c 键，就会执行 echo 命令，这里结尾的 INT 就是 Ctrl+c 键对应的信号名称。

[root@oldboy ~]# ^Coldgirl   #<== 按 Ctrl+c 键后调用 echo 命令输出结果。

[root@oldboy ~]# ^Coldgirl   #<== 按 Ctrl+c 键后调用 echo 命令输出结果。
```

用 stty -a 可以列出中断信号与键盘的对应信息，如下：

```
[root@oldboy ~]# stty -a
speed 38400 baud; rows 20; columns 103; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts -cdtrdsr
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc -ixany -imaxbel
-iutf8 opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echoke
```

范例 17-2：测试按下 Ctrl+c 组合键而引发的 INT(2) 信号。

```
[root@oldboy ~]# trap "" 2 #<== 执行动作处为空，此命令可以用来屏蔽与数字对应的 Ctrl+c 信号。
[root@oldboy ~]# #<== 此时执行 Ctrl+c 键无任何反应。
[root@oldboy ~]# trap ":" 2 #<== 恢复 Ctrl+c 信号。
[root@oldboy ~]# #<== 此时可以执行 Ctrl+c 了。
[root@oldboy ~]# trap "echo -n 'you are typing ctrl+c'" 2
[root@oldboy ~]# ^Cyou are typing ctrl+c
[root@oldboy ~]# ^Cyou are typing ctrl+c
[root@oldboy ~]# ^Cyou are typing ctrl+c
[root@oldboy ~]# ^Cyou are typing ctrl+c
```

范例 17-3：同时处理多个信号。

执行任何一个对应信号的事件时，都会执行前面对应的动作，因为动作为空，所以执行后没有任何反应。

```
[root@oldboy ~]# trap "" 1 2 3 20 15  #<== 执行这些数字信号，什么都不做。
[root@oldboy ~]# trap ":" 1 2 3 20 15 #<== 执行这些数字信号，恢复对应功能。
[root@oldboy ~]# ^C
[root@oldboy ~]# trap "" HUP INT QUIT TSTP TERM #<== 执行这些名称信号，什么都不做。
[root@oldboy ~]# trap ":" HUP INT QUIT TSTP TERM #<== 执行这些名称信号，恢复对应功能。
[root@oldboy ~]# trap "" `echo {1..64}`       #<== 屏蔽 1-64 的所有数字信号。
```

17.3　Linux 信号及 trap 命令的生产应用案例

范例 17-4：开发脚本实现触发信号后清理文件功能。

脚本如下：

```
[root@oldboy scripts]# cat 17_4_1.sh
#!/bin/bash
#Author:oldboy training
trap "find /tmp -type f -name "oldboy_*"|xargs rm -f && exit" INT
#<== 捕获 Ctrl+c 键后即执行 find 删除命令。
while true
do
    touch /tmp/oldboy_$(date +%F-%H-%M-%S) #<== 在 /tmp 下创建文件。
    sleep 3                                #<== 休息 3 秒。
    ls -l /tmp/oldboy*                     #<== 查看文件创建的情况。
done
[root@oldboy scripts]# sh 17_4_1.sh #<== 执行脚本后，很快就会在 /tmp 下创建很多文件。
-rw-r--r-- 1 root root 0 9月  22 13:41 /tmp/oldboy_2016-09-22-13-41-08
-rw-r--r-- 1 root root 0 9月  22 13:41 /tmp/oldboy_2016-09-22-13-41-08
-rw-r--r-- 1 root root 0 9月  22 13:41 /tmp/oldboy_2016-09-22-13-41-11
-rw-r--r-- 1 root root 0 9月  22 13:41 /tmp/oldboy_2016-09-22-13-41-08
-rw-r--r-- 1 root root 0 9月  22 13:41 /tmp/oldboy_2016-09-22-13-41-11
-rw-r--r-- 1 root root 0 9月  22 13:41 /tmp/oldboy_2016-09-22-13-41-14
^C #<== 按 Ctrl+c 键后，发现所有的文件都被清理了，说明已经调用了 find 清理命令。
[root@oldboy scripts]# ls -l /tmp/
总用量 4
drwx------ 2 oldboy oldboy 4096 9月  22 13:38 ssh-YSpSS25727
```

范例 17-5：开发脚本，练习使用 QUIT、TSTP、INT 信号。

脚本如下：

```
[root@oldboy scripts]# cat 17_5.sh
#!/bin/bash
#Author:oldboy training
trap 'echo "you art typing Ctrl-c,sorry,script will not terminate."' INT
#<== 捕获 Ctrl+c 键对应的信号后，执行对应快捷键时就会执行 echo 命令。
trap 'echo "you art typing Ctrl-\,sorry,script will not terminate."' QUIT
#<== 捕获 Ctrl+\ 键对应的信号，执行对应快捷键时就会执行 echo 命令。
trap 'echo "you art typing Ctrl-z,sorry,script will not terminate."' TSTP
#<== 捕获 Ctrl+z 键对应的信号，执行对应快捷键时就会执行 echo 命令（实测此处没有执行 echo 命令）。
while true
do
    echo "Now,test signal `date`"
    sleep 5
done
```

测试执行结果如下：

```
[root@oldboy scripts]# sh 17_5.sh
Now,test signal 2016年 09月 22日 星期四 14:03:09 CST
Now,test signal 2016年 09月 22日 星期四 14:03:14 CST
^Cyou art typing Ctrl-c,sorry,script will not terminate.
#<== 按下Ctrl+c键将打印这一行提示。
Now,test signal 2016年 09月 22日 星期四 14:03:15 CST
Now,test signal 2016年 09月 22日 星期四 14:03:20 CST
^\退出
you art typing Ctrl-\,sorry,script will not terminate.
#<== 按下Ctrl+\键将打印这一行提示。
Now,test signal 2016年 09月 22日 星期四 14:03:21 CST
Now,test signal 2016年 09月 22日 星期四 14:03:26 CST
^Z^Z #<== 按下Ctrl+z键没有打印提示，但是程序停止运行了。
```

范例 17-6：开发企业级 Shell 跳板机。要求用户登录到跳板机后只能执行管理员给定的选项动作，不允许以任何形式中断脚本而到跳板机服务器上执行任何系统命令。

参考答案 1：

1）首先做好 ssh 密钥验证（跳板机地址为 192.168.33.128）。

将在所有机器上操作以下命令：

```
[root@oldboy ~]# useradd jump                          #<== 要在所有机器上操作。
[root@oldboy ~]# echo 123456|passwd --stdin jump   #<== 要在所有机器上操作。
Changing password for user jump.
passwd: all authentication tokens updated successfully.
```

仅在跳板机上操作以下命令：

```
[root@oldboy ~]# su - jump
[jump@oldboy ~]$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa >/dev/null 2>&1
#<== 生成密钥对。
[jump@oldboy ~]$ ssh-copy-id -i ~/.ssh/id_dsa.pub 192.168.33.130
#<== 将公钥分发到其他服务器。
The authenticity of host '192.168.33.130 (192.168.33.130)' can't be established.
RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.33.130' (RSA) to the list of known hosts.
jump@192.168.33.130's password:
Now try logging into the machine, with "ssh '192.168.33.130'", and check in:

    .ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.

[jump@oldboy ~]$ ssh-copy-id -i ~/.ssh/id_dsa.pub 192.168.33.129
#<== 将公钥分发到其他服务器。
The authenticity of host '192.168.33.129 (192.168.33.129)' can't be established.
RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
Are you sure you want to continue connecting (yes/no)? yes
```

```
Warning: Permanently added '192.168.33.129' (RSA) to the list of known hosts.
jump@192.168.33.129's password:
Now try logging into the machine, with "ssh '192.168.33.129'", and check in:

    .ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
```

2）实现传统的远程连接菜单的脚本。

菜单脚本如下：

```
    cat <<menu
      1)oldboy-192.168.33.129
      2)oldgirl-192.168.33.130
      3)exit
menu
```

3）利用 Linux 信号防止用户中断信号在跳板机上操作。

代码如下：

```
function trapper () {
    trap ':' INT  EXIT TSTP TERM HUP  #<== 屏蔽这些信号。
}
```

4）用户登录跳板机后即调用脚本（不能用命令行管理跳板机），并且只能按管理员的要求选单。

以下为实战内容。

将脚本放在跳板机上：

```
    [root@oldboy ~]# echo '[ $UID -ne 0 ] && . /server/scripts/jump.sh' >/etc/
profile.d/jump.sh
    [root@oldboy ~]# cat /etc/profile.d/jump.sh
    [ $UID -ne 0 ] && . /server/scripts/jump.sh

    [root@oldboy scripts]# cat /server/scripts/jump.sh
    #!/bin/sh
    #oldboy training
    trapper() {
        trap ':' INT  EXIT TSTP TERM HUP  #<== 定义需要屏蔽的信号，冒号表示啥都不做。
    }
    main(){
        while :
        do
            trapper
            clear
            cat <<menu
              1)Web01-192.168.33.129
              2)Web02-192.168.33.130
```

```
menu
        read -p "Pls input a num.:" num
        case "$num" in
            1)
                echo 'login in 192.168.33.129.'
                ssh 192.168.33.129
                ;;
            2)
                echo 'login in 192.168.33.130.'
                ssh 192.168.33.130
                ;;
            110)
                read -p "your birthday:" char
                if [ "$char" = "0926" ];then
                    exit
                    sleep 3
                fi
                ;;
            *)
                echo "select error."
        esac
    done
}
main
```

执行效果如下：

```
[root@oldboy ~]# su - jump  #<== 切到普通用户即弹出菜单，工作中直接用 jump 登录，
                                即弹出菜单。
    1)Web01-192.168.33.129
    2)Web02-192.168.33.130
Pls input a num.:
    1)Web01-192.168.33.129
    2)Web02-192.168.33.130
Pls input a num.:1          #<== 选 1，则进入 Web01 服务器。
login in 192.168.33.129.
Last login: Tue Oct 11 17:23:52 2016 from 192.168.33.128
[jump@littleboy ~]$         #<== 按 Ctrl+D 键退出到跳板机服务器，且再次弹出菜单。
    1)Web01-192.168.33.129
    2)Web02-192.168.33.130
Pls input a num.:2          #<== 选 2，则进入 Web02 服务器。
login in 192.168.33.130.
Last login: Wed Oct 12 23:30:14 2016 from 192.168.33.128
[jump@oldgirl ~]$           #<== 按 Ctrl+D 键退出到跳板机服务器，且再次弹出菜单。
    1)Web01-192.168.33.129
    2)Web02-192.168.33.130
Pls input a num.:110        #<== 选 110，则进入跳板机命令提示符
your birthday:0926          #<== 需要输入特别码才能进入，这里是管理员通道，密码要保密呦。
[root@oldboy scripts]#      #<== 跳板机管理命令行。
```

Linux

第18章

Expect 自动化交互式程序应用实践

18.1 Expect 介绍

18.1.1 什么是 Expect

Expect 是一个用来实现自动交互功能的软件套件（Expect is a software suite for automating interactive tools，这是作者的定义），是基于 TCL[⊖]的脚本编程工具语言，方便学习，功能强大。

18.1.2 为什么要使用 Expect

在现今的企业运维中，自动化运维已经成为运维的主流趋势，但是在很多情况下，执行系统命令或程序时，系统会以交互式的形式要求运维人员输入指定的字符串，之后才能继续执行命令。例如，为用户设置密码时，一般情况下就需要手工输入 2 次密码，如下：

```
[root@oldboy ~]# passwd oldboy
Changing password for user oldboy.
New password:          #<== 需要手工输入密码。
Retype new password:   #<== 需要再次手工输入密码。
passwd: all authentication tokens updated successfully.
```

⊖ 全拼为 Tool Command Language，是一种脚本语言，由 John Ousterhout 创建。TCL 功能很强大，经常被用于快速原型开发、脚本编程、GUI 和测试等方面，不过现在用得不多了。

再比如使用 SSH 远程连接服务器时，第一次连接要和系统实现两次交互式输入：

```
[root@oldgirl ~]# ssh root@192.168.33.130
The authenticity of host '192.168.33.130 (192.168.33.130)' can't be established.
RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
Are you sure you want to continue connecting (yes/no)? yes  #<== 需要手工输入yes。
Warning: Permanently added '192.168.33.130' (RSA) to the list of known hosts.
root@192.168.33.130's password:                              #<== 需要手工输入密码。
Last login: Tue Oct 11 00:06:35 2016 from 192.168.33.128
[root@oldboy ~]#
```

经过上面的举例，相信大家已经明白为什么要使用 Expect 程序了。简单地说，Expect 就是用来自动实现与交互式程序通信的，而无需管理员的手工干预。比如 SSH、FTP 远程连接等，正常情况下都需要手工与它们进行交互，而使用 Expect 就可以模拟手工交互的过程，实现自动与远端程序的交互，从而达到自动化运维的目的。

以下是 Expect 的自动交互工作流程简单说明，依次执行如下操作：

spawn 启动指定进程→ expect 获取期待的关键字→ send 向指定进程发送指定字符→进程执行完毕，退出结束。

18.2 安装 Expect 软件

首先，要确保机器可以正常上网，并设置好 yum 安装源，然后执行 yum install expect -y 命令安装 Expect 软件，安装过程如下：

```
[root@oldboy ~]# rpm -qa expect          #<== 检查是否安装。
[root@oldboy ~]# yum install expect -y   #<== 执行安装命令。
[root@oldboy ~]# rpm -qa expect          #<== 再次检查是否安装。
expect-5.44.1.15-5.el6_4.x86_64
```

18.3 小试牛刀：实现 Expect 自动交互功能

首先准备 2 台虚拟机或真实服务器，IP 和主机名列表见表 18-1。

表 18-1 IP 和主机名列表

IP 地址	主机名
192.168.33.128	oldgirl
192.168.33.130	oldboy

在执行下面的例子之前，先在 128 这台服务器上手工执行下如下命令：

```
ssh -p22 root@192.168.33.130 uptime    #<== 连接到 130 上查看负载值。
```

执行结果如下：

```
[root@oldboy ~]# ssh -p22 root@192.168.33.130 uptime
The authenticity of host '192.168.33.130 (192.168.33.130)' can't be established.
RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
Are you sure you want to continue connecting (yes/no)? yes #<== 根据提示手工输入 yes。
Warning: Permanently added '192.168.33.130' (RSA) to the list of known hosts.
root@192.168.33.130's password     #<== 手工输入密码。
 21:20:35 up 1 day,  9:08,  1 user,  load average: 0.08, 0.02, 0.01
[root@oldboy ~]# ssh -p22 root@192.168.33.130 uptime
root@192.168.33.130's password     #<== 手工输入密码。
 21:20:39 up 1 day,  9:08,  1 user,  load average: 0.08, 0.02, 0.01
[root@oldboy ~]# ssh -p22 root@192.168.33.130 uptime
root@192.168.33.130's password     #<== 手工输入密码。
 21:20:43 up 1 day,  9:08,  1 user,  load average: 0.07, 0.02, 0.00
```

可以看到，每次都需要手工输入密码，才能执行 ssh 命令，否则无法执行。

下面就牛刀小试，利用 Expect 的功能实现自动交互，发送密码并执行上述 ssh 命令（注意，由于已经执行过一次 ssh 了，yes 的交互就不会再出现了）。

```
[root@oldboy ~]# cat oldboy.exp       #<== 扩展名使用 exp 代表是 Expect 脚本。
#!/usr/bin/expect       #<== 脚本开头解析器，和 Shell 类似，表示程序使用 Expect 解析。
spawn ssh root@192.168.33.130 uptime #<== 执行 ssh 命令（注意开头必须要有 spawn，
                                          否则无法实现交互）。
expect "*password"      #<== 利用 Expect 获取执行上述 ssh 命令输出的字符串是否为期待的
                             字符串 *password，这里的 * 是通配符。
send "123456\n" #<== 当获取到期待的字符串 *password 时，则发送 123456 密码给系统，\n 为换行。
expect eof      #<== 处理完毕后结束 Expect。
```

执行 Expect 脚本：

```
[root@oldboy ~]# which expect
/usr/bin/expect
[root@oldboy ~]# expect oldboy.exp    #<== 使用 Expect 执行脚本是个好习惯。
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password        #<== 这里再也不需要手工输入密码了。
 21:24:05 up 1 day,  9:12,  1 user,  load average: 0.00, 0.00, 0.00
[root@oldboy ~]# expect oldboy.exp
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password        #<== 这里再也不需要手工输入密码了。
 21:24:08 up 1 day,  9:12,  1 user,  load average: 0.00, 0.00, 0.00
```

此时我们并没有手工输入密码，就已经自动连到远端机器执行 ssh 命令了，这是不是很神奇？接下来老男孩就带领大家一起进入 Expect 程序学习之旅。

18.4　Expect 程序自动交互的重要命令及实践

Expect 程序中的命令是 Expect 的核心，需要重点掌握。

18.4.1 spawn 命令

在 Expect 自动交互程序执行的过程中，spawn 命令是一开始就需要使用的命令，通过 spawn 执行一个命令或程序，之后所有的 Expect 操作都会在这个执行过的命令或程序进程中进行，包括自动交互功能，因此如果没有 spawn 命令，Expect 程序将会无法实现自动交互。

spawn 命令的语法为：

```
spawn [选项] [需要自动交互的命令或程序]
```

例如：

```
spawn ssh root@192.168.33.130 uptime
```

在 spawn 命令的后面，直接加上要执行的命令或程序（例如这里的 ssh 命令）等，除此之外，spawn 还支持如下一些选项。

- -open：表示启动文件进程。
- -ignore：表示忽略某些信号。

提示：这些选项不常用，了解即可，无需深入。

使用 spawn 命令是 Expect 程序实现自动交互工作流程中的第一步，也是最关键的一步。

18.4.2 expect 命令

1. expect 命令语法

在 Expect 自动交互程序的执行过程中，当使用 spawn 命令执行一个命令或程序之后，会提示某些交互式信息，expect 命令的作用就是获取 spawn 命令执行后的信息，看看是否和其事先指定的相匹配，一旦匹配上指定的内容就执行 expect 后面的动作，expect 命令也有一些选项，相对用得较多的是 -re，表示使用正则表达式的方式来匹配。

expect 命令的语法为：

```
expect  表达式  [动作]
```

示例如下：

```
spawn ssh root@192.168.33.130 uptime
expect "*password" {send  "123456\r"}
```

上述命令不能直接在 Linux 命令行中执行，需要放入 Expect 脚本中执行。

2. expect 命令的实践

范例 18-1：执行 ssh 命令远程获取服务器负载值，并要求实现自动输入密码。

方法 1：将 expect 和 send 放在一行。

```
[root@oldboy ~]# cat 18_1_1.exp
#!/usr/bin/expect                              #<== 脚本解释器。
spawn ssh root@192.168.33.130 uptime           #<== 开启 expect 自动交互式，执行 ssh 命令。
expect "*password" {send  "123456\n"}          #<== 如果 ssh 命令输出匹配 *password，
                                                   就发送 123456 给系统。
expect eof     #<== 要想输出结果，还必须加 eof，表示 expect 结束。
```

执行结果如下：

```
[root@oldboy ~]# expect 18_1_1.exp #<== 采用 expect 执行脚本，就相当于使用 sh 执行
                                        Shell 脚本。
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password: #<==Expect 程序自动帮我们输入了密码。
 21:51:14 up 1 day,  9:39,  1 user,  load average: 0.00, 0.00, 0.00
```

从上面的例子可以看出，expect 命令是依附于 spawn 命令的，即通过 spawn 执行 ssh 命令后，系统会提示输入密码，此时的 expect 命令按照事先的配置匹配 ssh 命令执行后的字符串 password，如果匹配到了指定的 password 字符串，则会执行紧随其后包含在 {}（大括号）中的 send 或 exp_send 动作，匹配的动作也可以放在下一行，这样就不需要使用 {}（大括号）了，就像下面这样，实际完成的功能与上面的一样。

方法 2：expect 和 send 放在不同行。

```
[root@oldboy ~]# cat 18_1_2.exp
#!/usr/bin/expect
spawn ssh root@192.168.33.130 uptime
expect "*password:"
send  "123456\n"
expect eof
```

执行结果如下：

```
[root@oldboy ~]# expect 18_1_2.exp
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password:
 21:52:37 up 1 day,  9:40,  1 user,  load average: 0.00, 0.00, 0.00
```

expect 命令还有一种高级用法，即它可以在一个 expect 匹配中多次匹配不同的字符串，并给出不同的处理动作，此时只需要将匹配的所有字符串放在一个 {}（大括号）中就可以了，当然还要借助 exp_continue 指令实现继续匹配。

范例 18-2：执行 ssh 命令远程获取服务器负载值，并自动输入“yes”及用户密码。

```
[root@oldboy ~]# cat 18_2_1.exp
```

```
#!/usr/bin/expect
spawn ssh root@192.168.33.130 uptime
expect {     #<== 起始大括号前要有空格。
    "yes/no"    {exp_send "yes\r";exp_continue} #<==exp_send 和 send 类似。
    "*password" {exp_send "123456\r"}
}
expect eof
```

说明：

1）exp_send 和 send 类似，后面的 \r（回车）和前文的 \n（换行）类似。

2）expect {}，类似多行 expect。

3）匹配多个字符串，需要在每次匹配并执行动作后，加上 exp_continue。

演示如下：

```
[root@oldboy ~]# rm -f ~/.ssh/known_hosts #<== 清除密钥文件，使其出现提示 yes/no 信息。
[root@oldboy ~]# expect 18_2_1.exp
spawn ssh root@192.168.33.130 uptime
The authenticity of host '192.168.33.130 (192.168.33.130)' can't be established.
RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
Are you sure you want to continue connecting (yes/no)? yes  #<==expect 自动输入 yes。
Warning: Permanently added '192.168.33.130' (RSA) to the list of known hosts.
root@192.168.33.130's password:  #<==expect 自动给密码。
 22:03:13 up 1 day,  9:51,  1 user,  load average: 0.00, 0.00, 0.00
 #<== 轻松打印出负载值。
```

范例 18-3：利用 expect 响应 Shell 脚本中的多个 read 读入。

准备数据：利用 read 提示用户输入，故意创造交互式输入，给 expect 出难题。

```
[root@oldboy ~]# cat 18_3_1.sh   #<== 这里是 Shell 脚本！
#!/bin/sh
read -p 'Please input your username:' name
read -p 'Please input your password:' pass
read -p 'Please input your email:' mail
echo -n "your name is $name,"
echo -n "your password is $pass,"
echo "your email is $mail."
```

执行结果如下：

```
[root@oldboy ~]# sh 18_3_1.sh
Please input your username:oldboy          #<== 提示输入，只能手动输入对应字符串。
Please input your password:123456          #<== 提示输入，只能手动输入对应字符串。
Please input your email:31333741@qq.com    #<== 提示输入，只能手动输入对应字符串。
your name is oldboy,your password is 123456,your email is 31333741@qq.com.
```

以下为正式解答：开发 Expect 自动化脚本，根据需求自动输入多个字符串。

```
[root@oldboy ~]# cat 18_3_2.exp
#!/usr/bin/expect
spawn /bin/sh 18_3_1.sh    #<== 执行上述 Shell 脚本，注意这里使用的是相对路径。
expect {
    "username" {exp_send "oldboy\r";exp_continue}
                           #<== 若获取到的是 username 信息，则自动输入 oldboy。
    "*pass*"   {send "123456\r";exp_continue}
                           #<== 若获取到的是 *pass* 信息，则自动输入 123456。
    "*mail*"   {exp_send "31333741@qq.com\r"}
                           #<== 若获取到的是 *mail* 信息，则自动输入邮件地址。
}
expect eof
```

执行结果如下：

```
[root@oldboy ~]# expect 18_3_2.exp       #<== 回车后，无任何人工交互，直接输出结果。
spawn /bin/sh 18_3_1.sh
Please input your username:oldboy        #<== 自动输入需要的字符串。
Please input your password:123456        #<== 自动输入需要的字符串。
Please input your email:31333741@qq.com #<== 自动输入需要的字符串。
your name is oldboy,your password is 123456,your email is 31333741@qq.com.
```

18.4.3　send 命令

在上面的案例中，我们已经看到了 exp_send 和 send 命令的使用方法，这两个命令是 Expect 中的动作命令，用法类似，即在 expect 命令匹配指定的字符串后，发送指定的字符串给系统，这些命令可以支持一些特殊转义符号，例如：\r 表示回车、\n 表示换行、\t 表示制表符等，这些用法与 TCL 中的特殊符号相同，例如前文案例中的 18_3_2.exp。

Send 命令的使用示例如下：

```
#!/usr/bin/expect
spawn /bin/sh 18_3_1.sh
expect {
    "username" {exp_send "oldboy\r";exp_continue}
    "*pass*"   {send "123456\r";exp_continue}
    "*mail*"   {exp_send "31333741@qq.com\r"}
}
expect eof
```

send 命令有几个可用的参数，具体如下。

- -i：指定 spawn_id，用来向不同的 spawn_id 进程发送命令，是进行多程序控制的参数。
- -s：s 代表 slowly，即控制发送的速度，使用的时候要与 expect 中的变量 send_slow 相关联。

18.4.4 exp_continue 命令

前文已经使用过这个 exp_continue 命令，它一般处于 expect 命令中，属于一种动作命令，一般用在匹配多次字符串的动作中，从命令的拼写就可以看出命令的作用，即让 Expect 程序继续匹配的意思，示例见前文案例中的 18_3_2.exp。

exp_continue 命令的使用示例如下：

```
#!/usr/bin/expect
spawn /bin/sh 18_3_1.sh
expect {
    "username" {exp_send "oldboy\r";exp_continue}
    "*pass*"   {send "123456\r";exp_continue}
    "*mail*"   {exp_send "31333741@qq.com\r"}
}
expect eof
```

说明：如果需要一次匹配多个字符串，那么不同的匹配之间就要加上 exp_continue，否则 expect 将不会自动输入指定的字符串。最后一个的结尾就不需要加上 exp_continue 了，因为都匹配完成了。

在这个例子中，匹配第一个字符串 "username" 之后，expect 发送 oldboy 字符串给系统，然后利用 exp_continue 继续匹配下一个字符串 "*pass*"，并自动输入指定字符串 123456，最后匹配 "*mail*"，并输入指定字符串 31333741@qq.com，由于结尾没有 exp_continue，因此匹配结束。

18.4.5 send_user 命令

send_user 命令可用来打印 Expect 脚本信息，类似 Shell 里的 echo 命令，而默认的 send、exp_send 命令都是将字符串输出到 Expect 程序中去，有关 send_user 命令用法的示例如下。

范例 18-4：send_user 命令的使用示例。

```
[root@oldboy ~]# cat 18_4_1.exp
#!/usr/bin/expect
send_user "I am oldboy.\n"        #<==\n 表示换行。
send_user "I am a linuxer,\t"     #<==\t 表示 Tab 键。
send_user "My blog is http://blog.oldboyedu.com\n"
```

执行结果如下：

```
[root@oldboy ~]# expect 18_4_1.exp
I am oldboy.
I am a linuxer, My blog is http://blog.oldboyedu.com
```

像不像 Shell 里的 echo 命令？而且有 echo -e 的功能。

18.4.6　exit 命令

exit 命令的功能类似于 Shell 中的 exit，即直接退出 Expect 脚本，除了最基本的退出脚本功能之外，还可以利用这个命令对脚本做一些关闭前的清理和提示等工作，比如下面的示例。

范例 18-5：exit 功能的实践。

```
[root@oldboy ~]# cat 18_4_2.exp
#!/usr/bin/expect
send_user "I am oldboy.\n"
send_user "I am a linuxer,\t"
send_user "My blog is http://blog.oldboyedu.com\n"
exit -onexit {
    send_user "Good bye.\n"
}
```

执行结果如下：

```
[root@oldboy ~]# expect 18_4_2.exp
I am oldboy.
I am a linuxer,  My blog is http://blog.oldboyedu.com
Good bye.
```

18.4.7　Expect 常用命令总结

相信到这里读者已经掌握了上述 Expect 的常用命令，下面将这些知识进行总结以方便记忆，整理结果如表 18-2。

表 18-2　前文 Expect 常用命令总结

Expect 命令	作　用
spawn	spawn 命令是一个在 Expect 自动交互程序的开始就需要使用的命令，通过 spawn 执行一个命令或程序，之后所有的 Expect 操作都在这个执行过的命令或程序进程中进行，包括自动交互功能
expect	在 Expect 自动交互程序的执行过程中，在使用 spawn 命令执行一个命令或程序之后，会提示某些交互式信息，expect 命令的作用就是获取这些信息，查看是否和其事先指定的信息相匹配，一旦匹配上指定的内容，就执行 expect 后面的动作
send	Expect 中的动作命令，当 expect 匹配了指定的字符串后，发送指定的字符串给系统，这些命令可以支持一些特殊的转义符号，例如：\r 表示回车、\n 表示换行、\t 表示制表符等，还有一个类似的 exp_send 命令
exp_continue	属于一种动作命令，在一个 expect 命令中，用于多次匹配字符串并执行不同的动作中。从命令的拼写格式就可以看出该命令的作用，即让 expect 程序继续匹配
send_user	send_user 命令用来打印 Expect 脚本信息，类似 Shell 里的 echo 命令，并且带 -e 功能
exit	退出 Expect 脚本，以及在退出脚本前做一些关闭前的清理和提示等工作

18.5 Expect 程序变量

18.5.1 普通变量

Expect 中的变量定义、使用方法与 TCL 语言中的变量基本相同。

定义变量的基本语法如下：

```
set 变量名 变量值
```

示例如下：

```
set  password  "123456"
```

打印变量的基本语法如下：

```
puts       $变量名
```

范例 18-6：定义及输出变量。

```
[root@oldboy ~]# cat 18_6_1.exp
#!/usr/bin/expect
set password "123456"
puts $password
send_user "$password\n"        #<== send_user 也可以打印输出。
```

执行结果如下：

```
[root@oldboy ~]# expect 18_6_1.exp
123456
123456
```

18.5.2 特殊参数变量

在 Expect 里也有与 Shell 脚本里的 $0、$1、$# 等类似的特殊参数变量，用于接收及控制 Expect 脚本传参。

在 Expect 中 $argv 表示参数数组，可以使用 [lindex $argv n] 接收 Expect 脚本传参，n 从 0 开始，分别表示第一个 [lindex $argv 0] 参数、第二个 [lindex $argv 1] 参数、第三个 [lindex $argv 2] 参数……

范例 18-7：定义及输出特殊参数变量。

```
[root@oldboy ~]# cat 18_7_1.exp
#!/usr/bin/expect
#define var
set file [lindex $argv 0]      #<== 相当于 Shell 里脚本传参的 $1。
set host [lindex $argv 1]      #<== 相当于 Shell 里脚本传参的 $2。
set dir  [lindex $argv 2]      #<== 相当于 Shell 里脚本传参的 $3。
send_user "$file\t$host\t$dir\n"
```

```
puts "$file\t$host\t$dir\n"
```

执行结果如下：

```
[root@oldboy ~]# expect 18_7_1.exp oldboy.log 192.168.33.130 /tmp
#<== 必须要给参数呦!
oldboy.log      192.168.33.130  /tmp
oldboy.log      192.168.33.130  /tmp
```

Expect 接收参数的方式和 bash 脚本的方式有些区别，bash 是通过 $0 ... $n 这种方式来接收的，而 Expect 是通过 set < 变量名称 > [lindex $argv <param index>] 来接收的，例如 set file [lindex $argv 0]。

除了基本的位置参数外，Expect 也支持其他的特殊参数，例如：$argc 表示传参的个数，$argv0 表示脚本的名字。

范例 18-8：针对 Expect 脚本传参的个数及脚本名参数的实践。

```
[root@oldboy ~]# cat 18_8_1.exp
#!/usr/bin/expect
set file [lindex $argv 0]
set host [lindex $argv 1]
set dir  [lindex $argv 2]
puts "$file\t$host\t$dir"
puts $argc
puts $argv0
```

执行结果如下：

```
[root@oldboy ~]# expect 18_8_1.exp oldgir.txt 10.0.0.3 /opt
oldgir.txt      10.0.0.3        /opt    #<== 这是脚本后面的三个参数。
3                                       #<== 这是参数的总个数（$argc 的结果）。
18_8_1.exp                              #<== 这是脚本的名字（$argv0 的结果）。
```

18.6　Expect 程序中的 if 条件语句

Expect 程序中 if 条件语句的基本语法为：

```
if { 条件表达式 } {
    指令
}
```

或

```
if { 条件表达式 } {
    指令
} else {
    指令
}
```

说明：if 关键字后面要有空格，else 关键字前后都要有空格，{ 条件表达式 } 大括号里面靠近大括号处可以没有空格，将指令括起来的起始大括号“{”前要有空格。

范例 18-9：使用 if 语句判断脚本传参的个数，如果不符则给予提示。

```
[root@oldboy ~]# cat 18_9_1.exp
#!/usr/bin/expect
if { $argc != 3 } {       #<==$argc 为传参的个数，相当于 Shell 里的 $#。
    send_user "usage: expect $argv0 file host dir\n" #<== 给予提示，$argv0 代表
                                                          脚本的名字。
    exit                                              #<== 退出脚本。
}
#define var
set file [lindex $argv 0]
set host [lindex $argv 1]
set dir  [lindex $argv 2]
puts "$file\t$host\t$dir"
```

执行结果如下：

```
[root@oldboy ~]# expect 18_9_1.exp
usage: expect 18_9_1.exp file host dir  #<==18_9_1.exp 就是 $argv0 输出的结果。
[root@oldboy ~]# expect 18_9_1.exp oldboy.log 192.168.33.130 /home/oldboy
#<== 传三个参数。
oldboy.log      192.168.33.130  /home/oldboy #<== 这是脚本后面的三个参数。
```

范例 18-10：使用 if 语句判断脚本传参的个数，不管是否符合都给予提示。

```
[root@oldboy ~]# cat 18_10_1.exp
#!/usr/bin/expect
if {$argc != 26} {
    puts "bad."
} else {
    puts "good."
}
```

执行结果如下：

```
[root@oldboy ~]# expect 18_10_1.exp
bad.
[root@oldboy ~]# expect 18_10_1.exp {a..z}
good.
```

本章的目的并不是带领读者彻底精通 Expect 语言，而是指导读者解决运维管理中的交互问题，实现自动化运维，因此，请读者不要过多地纠结于 Expect 语言，而应多关注前文讲解的自动化交互的知识。

18.7　Expect 中的关键字

Expect 中的特殊关键字用于匹配过程，代表某些特殊的含义或状态，一般只用于 Expect 命令中而不能在 Expect 命令外面单独使用。

18.7.1　eof 关键字

eof（end-of-file）关键字用于匹配结束符，前面已经使用过 eof 这个关键字了，本节不再过多重复。例如：

```
[root@oldboy ~]# cat 18_1_1.exp
#!/usr/bin/expect
spawn ssh root@192.168.33.130 uptime
expect "*password" {send  "123456\n"}
expect eof
[root@oldboy ~]# cat 18_2_1.exp
#!/usr/bin/expect
spawn ssh root@192.168.33.130 uptime
expect {
    "yes/no"    {exp_send "yes\r";exp_continue}
    "*password" {exp_send "123456\r"}
}
expect eof
```

18.7.2　timeout 关键字

timeout 是 Expect 中的一个控制时间的关键字变量，它是一个全局性的时间控制开关，可以通过为这个变量赋值来规定整个 Expect 操作的时间，注意这个变量是服务于 Expect 全局的，而不是某一条命令，即使命令没有任何错误，到了时间仍然会激活这个变量，此外，到时间后还会激活一个处理及提示信息开关，下面来看看它的实际使用方法。

范例 18-11：timeout 超时功能实践。

```
[root@oldboy ~]# cat 18_11_1.exp
#!/usr/bin/expect
spawn ssh root@192.168.33.130 uptime
set timeout 30      #<== 设置 30 秒超时。
expect "yes/no"     {exp_send "yes\r";exp_continue}
expect  timeout     {puts "Request timeout by oldboy.";return}
                    #<== 当到达 30 秒后就超时，打印指定输出后退出。
```

执行结果如下：

```
[root@oldboy ~]# expect 18_11_1.exp
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password: Request timeout by oldboy.
```

上面的处理中，首先将 timeout 变量设置为 30 秒，此时 Expect 脚本的执行只要超

过了 30 秒，就会直接执行结尾的 timeout 动作，打印一个信息，停止运行脚本，读者还可以做更多的其他事情。

在 expect{} 的用法中，还可以使用下面的 timeout 语法：

```
[root@oldboy ~]# cat 18_11_2.exp
#!/usr/bin/expect
spawn ssh root@192.168.33.130 uptime
expect {
    -timeout 3
    "yes/no" {exp_send "yes\r";exp_continue}
    timeout  {puts "Request timeout by oldboy.";return}
}
```

执行结果如下：

```
[root@oldboy ~]# expect 18_11_2.exp
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password: Request timeout by oldboy.
```

timeout 变量设置为 0，表示立即超时，为 -1 则表示永不超时。

18.8 企业生产场景下的 Expect 案例

环境准备：首先准备 3 台虚拟机或真实服务器，IP 和主机名列表见表 18-3。

表 18-3 IP 和主机名

IP 地址	主机名	角　色
192.168.33.128	oldgirl	管理机
192.168.33.129	littleboy	被管理机 1
192.168.33.130	oldboy	被管理机 2

18.8.1 批量执行命令

说明：以下全部脚本都在管理机上运行。

范例 18-12：开发 Expect 脚本实现自动交互式批量执行命令。

1）实现 Expect 自动交互的脚本：

```
[root@oldboy ~]# cat 18_12_1.exp
#!/usr/bin/expect
if { $argc != 2 } {
    puts "usage: expect $argv0 ip command"
    exit
}
```

```
#define var
set ip  [lindex $argv 0]
set cmd [lindex $argv 1]
set password "123456"
#
spawn ssh root@$ip $cmd
expect {
    "yes/no"    {send "yes\r";exp_continue}
    "*password" {send "$password\r"}
}
expect eof
```

执行结果如下：

```
[root@oldboy ~]# expect 18_12_1.exp 192.168.33.128 uptime
spawn ssh root@192.168.33.128 uptime
root@192.168.33.128's password:
 14:20:53 up 16:26,  2 users,  load average: 0.07, 0.03, 0.01
[root@oldboy ~]# expect 18_12_1.exp 192.168.33.128 "free -m"
spawn ssh root@192.168.33.128 free -m
root@192.168.33.128's password:
             total       used       free     shared    buffers     cached
Mem:           981        492        488          0         42        293
-/+ buffers/cache:        156        824
Swap:          767          0        767
```

2）利用 Shell 循环执行 Expect 脚本命令：

```
[root@oldboy ~]# cat 18_12_2.sh
#!/bin/sh
if [ $# -ne 1 ]
  then
    echo $"USAGE:$0 cmd"
    exit 1
fi
cmd=$1
for n in 128 129 130
do
    expect 18_12_1.exp 192.168.33.$n "$cmd"  #<== 带双引号接收带参数的命令。
done
```

查看所有机器负载的执行结果：

```
[root@oldboy ~]# sh 18_12_2.sh uptime
spawn ssh root@192.168.33.128 uptime
root@192.168.33.128's password:
 14:32:04 up 16:38,  2 users,  load average: 0.00, 0.00, 0.00
spawn ssh root@192.168.33.129 uptime
root@192.168.33.129's password:
```

```
 11:58:01 up 26 min,  2 users,  load average: 0.00, 0.00, 0.00
spawn ssh root@192.168.33.130 uptime
root@192.168.33.130's password:
 11:58:08 up 1 day, 15:48,  2 users,  load average: 0.00, 0.00, 0.00
```

查看所有机器内存的执行结果：

```
[root@oldboy ~]# sh 18_12_2.sh "free -m" #<== 带双引号传入带参数的命令。
spawn ssh root@192.168.33.128 free -m
root@192.168.33.128's password:
             total       used       free     shared    buffers     cached
Mem:           981        493        487          0         42        293
-/+ buffers/cache:        156        824
Swap:          767          0        767
spawn ssh root@192.168.33.129 free -m
root@192.168.33.129's password:
             total       used       free     shared    buffers     cached
Mem:           981        178        803          0         10         52
-/+ buffers/cache:        114        866
Swap:          767          0        767
spawn ssh root@192.168.33.130 free -m
root@192.168.33.130's password:
             total       used       free     shared    buffers     cached
Mem:           981        505        475          0         91        294
-/+ buffers/cache:        119        861
Swap:          767          0        767
```

如果遇到连接 SSH 反应慢的问题，请在所有被管理的机器上提前执行如下命令：

```
sed -ir '13iUseDNS no\nGSSAPIAuthentication no\n' /etc/ssh/sshd_config
/etc/init.d/sshd reload
```

以其中一台为例，所有机器的操作过程都如下所示：

```
[root@littleboy ~]# sed -ir '13iUseDNS no\nGSSAPIAuthentication no\n' /
etc/ssh/sshd_config
[root@littleboy ~]# /etc/init.d/sshd restart
Stopping sshd:                                             [  OK  ]
Starting sshd:                                             [  OK  ]
```

18.8.2 批量发送文件

范例 18-13：开发 Expect 脚本以实现自动交互式批量发送文件或目录。

1）实现 Expect 自动交互的脚本：

```
[root@oldboy ~]# cat 18_13_1.exp
#!/usr/bin/expect
if { $argc != 3 } {
    puts "usage: expect $argv0 file host dir"
```

```
    exit
}

#define var
set file [lindex $argv 0]
set host [lindex $argv 1]
set dir  [lindex $argv 2]
set password "123456"
spawn scp -P22 -rp $file root@$host:$dir
expect {
    "yes/no"    {send "yes\r";exp_continue}
    "*password" {send "$password\r"}
}
expect eof
```

执行结果如下：

```
[root@oldboy ~]# expect 18_13_1.exp
usage: expect 18_12_2.exp file host dir
[root@oldboy ~]# expect 18_13_1.exp /etc/hosts 192.168.33.130 /home/oldboy
spawn scp -P22 -rp /etc/hosts root@192.168.33.130:/home/oldboy
root@192.168.33.130's password:
hosts                                         100%  158     0.2KB/s   00:00
```

2）利用 Shell 循环执行 Expect 脚本命令：

```
[root@oldboy ~]# cat 18_13_2.sh
#!/bin/sh
if [ $# -ne 2 ]
  then
    echo $"USAGE:$0 file dir"
    exit 1
fi
file=$1
dir=$2
for n in 128 129 130
do
    expect 18_13_1.exp $file 192.168.33.$n  $dir
done
```

将 /etc/hosts 文件批量发送到所有服务器指定的 /opt 目录：

```
[root@oldboy ~]# sh 18_13_2.sh
USAGE:18_13_2.sh file dir
[root@oldboy ~]# sh 18_13_2.sh /etc/hosts /opt
spawn scp -P22 -rp /etc/hosts root@192.168.33.128:/opt
root@192.168.33.128's password:
hosts                                         100%  158     0.2KB/s   00:00
spawn scp -P22 -rp /etc/hosts root@192.168.33.129:/opt
root@192.168.33.129's password:
```

```
hosts                                          100%  158     0.2KB/s   00:00
spawn scp -P22 -rp /etc/hosts root@192.168.33.130:/opt
root@192.168.33.130's password:
hosts                                          100%  158     0.2KB/s   00:00
```

将 /server/scripts 目录批量发送到所有服务器指定的 /tmp 目录：

```
[root@oldboy ~]# sh 18_13_2.sh /server/scripts /tmp/
spawn scp -P22 -rp /server/scripts root@192.168.33.128:/tmp/
root@192.168.33.128's password:
exec.sh                                        100%   32     0.0KB/s   00:00
spawn scp -P22 -rp /server/scripts root@192.168.33.129:/tmp/
root@192.168.33.129's password:
exec.sh                                        100%   32     0.0KB/s   00:00
spawn scp -P22 -rp /server/scripts root@192.168.33.130:/tmp/
root@192.168.33.130's password:
exec.sh                                        100%   32     0.0KB/s   00:00
```

18.8.3 批量执行 Shell 脚本

范例 18-14：开发 Expect 脚本以实现自动交互式批量执行 Shell 脚本。

实际上本范例就是先发送脚本文件，然后再远程执行脚本的过程。

1）准备脚本文件：

```
[root@oldboy ~]# rpm -qa httpd
[root@oldboy ~]# echo "yum install httpd -y" >/server/scripts/yum.sh
[root@oldboy ~]# cat /server/scripts/yum.sh
yum install httpd -y
```

2）使用**范例 18-13** 的脚本发送要执行的脚本文件到所有机器：

```
[root@oldboy ~]# sh 18_12_2.sh
USAGE:18_12_2.sh cmd
[root@oldboy ~]# sh 18_13_2.sh
USAGE:18_13_2.sh file dir
[root@oldboy ~]# sh 18_13_2.sh /server/scripts/yum.sh /tmp/
spawn scp -P22 -rp /server/scripts/yum.sh root@192.168.33.128:/tmp/
root@192.168.33.128's password:
yum.sh                                         100%   21     0.0KB/s   00:00
spawn scp -P22 -rp /server/scripts/yum.sh root@192.168.33.129:/tmp/
root@192.168.33.129's password:
yum.sh                                         100%   21     0.0KB/s   00:00
spawn scp -P22 -rp /server/scripts/yum.sh root@192.168.33.130:/tmp/
root@192.168.33.130's password:
yum.sh                                         100%   21     0.0KB/s   00:00
```

3）使用范例 18-12 的脚本远程连接所有服务器并远程执行脚本：

```
[root@oldboy ~]# sh 18_12_2.sh "source /tmp/yum.sh" #<== 推荐使用 source 远程
                                                          执行脚本。
```

```
spawn ssh root@192.168.33.128 source /tmp/yum.sh
root@192.168.33.128's password:
Loaded plugins: fastestmirror, security
Setting up Install Process
Loading mirror speeds from cached hostfile
 * base: mirrors.tuna.tsinghua.edu.cn
 * extras: mirrors.yun-idc.com
...省略若干...
---> Package httpd-tools.x86_64 0:2.2.15-54.el6.centos will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package         Arch        Version                     Repository    Size
================================================================================
Installing:
 httpd           x86_64      2.2.15-54.el6.centos        updates      833 k
Installing for dependencies:
...省略若干...
    Installing : httpd-2.2.15-54.el6.centos.x86_64                        3/3
    Verifying  : httpd-tools-2.2.15-54.el6.centos.x86_64                  1/3
    Verifying  : httpd-2.2.15-54.el6.centos.x86_64                        2/3
    Verifying  : apr-util-ldap-1.3.9-3.el6_0.1.x86_64                     3/3

Installed:
    httpd.x86_64 0:2.2.15-54.el6.centos

Dependency Installed:
    apr-util-ldap.x86_64 0:1.3.9-3.el6_0.1
    httpd-tools.x86_64 0:2.2.15-54.el6.centos

Complete!
spawn ssh root@192.168.33.129 source /tmp/yum.sh
root@192.168.33.128's password:
Loaded plugins: fastestmirror, security
Setting up Install Process
Loading mirror speeds from cached hostfile
 * base: mirrors.tuna.tsinghua.edu.cn
 * extras: mirrors.yun-idc.com
...省略若干...
---> Package httpd-tools.x86_64 0:2.2.15-54.el6.centos will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package         Arch        Version                     Repository    Size
================================================================================
Installing:
```

```
 httpd            x86_64       2.2.15-54.el6.centos       updates       833 k
Installing for dependencies:
... 省略若干 ...
    Installing : httpd-2.2.15-54.el6.centos.x86_64                        3/3
    Verifying  : httpd-tools-2.2.15-54.el6.centos.x86_64                  1/3
    Verifying  : httpd-2.2.15-54.el6.centos.x86_64                        2/3
    Verifying  : apr-util-ldap-1.3.9-3.el6_0.1.x86_64                     3/3

Installed:
    httpd.x86_64 0:2.2.15-54.el6.centos

Dependency Installed:
    apr-util-ldap.x86_64 0:1.3.9-3.el6_0.1
    httpd-tools.x86_64 0:2.2.15-54.el6.centos

Complete!
spawn ssh root@192.168.33.130 source /tmp/yum.sh
root@192.168.33.128's password:
Loaded plugins: fastestmirror, security
Setting up Install Process
Loading mirror speeds from cached hostfile
 * base: mirrors.tuna.tsinghua.edu.cn
 * extras: mirrors.yun-idc.com
... 省略若干 ...
---> Package httpd-tools.x86_64 0:2.2.15-54.el6.centos will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package         Arch        Version                     Repository     Size
================================================================================
Installing:
 httpd           x86_64      2.2.15-54.el6.centos        updates       833 k
Installing for dependencies:
... 省略若干 ...
    Installing : httpd-2.2.15-54.el6.centos.x86_64                        3/3
    Verifying  : httpd-tools-2.2.15-54.el6.centos.x86_64                  1/3
    Verifying  : httpd-2.2.15-54.el6.centos.x86_64                        2/3
    Verifying  : apr-util-ldap-1.3.9-3.el6_0.1.x86_64                     3/3

Installed:
    httpd.x86_64 0:2.2.15-54.el6.centos

Dependency Installed:
    apr-util-ldap.x86_64 0:1.3.9-3.el6_0.1
    httpd-tools.x86_64 0:2.2.15-54.el6.centos

Complete!
```

18.8.4　自动化部署 SSH 密钥认证 +ansible 的项目实战

范例 18-15：批量分发 SSH 密钥并建立 ansible 批量管理环境。

1）本地生成密钥对，代码如下：

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa >/dev/null 2>&1
```

2）开发 Expect 脚本自动化交互分发公钥到所有的服务器：

```
[root@oldboy ~]# cat 18_15_1.exp
#!/usr/bin/expect
# create by oldboy
if { $argc != 2 } {
    send_user "usage: expect expect.exp file host\n"
    exit
}
#define var
set file [lindex $argv 0]
set host [lindex $argv 1]
set password "123456"
#start exec command
spawn ssh-copy-id -i $file "-p 22 root@$host"
expect {
    "yes/no"    {send "yes\r";exp_continue}
    "*password" {send "$password\r"}
}
expect eof
```

3）开发 Shell 脚本循环执行 Expect 脚本：

```
[root@oldboy ~]# cat 18_15_2.sh
#!/bin/sh
for n in 128 129 130
do
    expect 18_15_1.exp ~/.ssh/id_dsa.pub 192.168.33.$n
done
```

操作过程如下：

```
[root@oldboy ~]# rm -fr ~/.ssh/
[root@oldboy ~]# ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa >/dev/null 2>&1
[root@oldboy ~]# sh 18_15_2.sh
spawn ssh-copy-id -i /root/.ssh/id_dsa.pub -p 22 root@192.168.33.128
The authenticity of host '192.168.33.128 (192.168.33.128)' can't be established.
RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.33.128' (RSA) to the list of known hosts.
root@192.168.33.128's password:
Now try logging into the machine, with "ssh '-p 22 root@192.168.33.128'",
and check in:
```

```
        .ssh/authorized_keys

    to make sure we haven't added extra keys that you weren't expecting.

    spawn ssh-copy-id -i /root/.ssh/id_dsa.pub -p 22 root@192.168.33.129
    The authenticity of host '192.168.33.129 (192.168.33.129)' can't be established.
    RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
    Are you sure you want to continue connecting (yes/no)? yes
    Warning: Permanently added '192.168.33.129' (RSA) to the list of known hosts.
    root@192.168.33.129's password:
    Now try logging into the machine, with "ssh '-p 22 root@192.168.33.129'",
and check in:

        .ssh/authorized_keys

    to make sure we haven't added extra keys that you weren't expecting.

    spawn ssh-copy-id -i /root/.ssh/id_dsa.pub -p 22 root@192.168.33.130
    The authenticity of host '192.168.33.130 (192.168.33.130)' can't be established.
    RSA key fingerprint is fd:2c:0b:81:b0:95:c3:33:c1:45:6a:1c:16:2f:b3:9a.
    Are you sure you want to continue connecting (yes/no)? yes
    Warning: Permanently added '192.168.33.130' (RSA) to the list of known hosts.
    root@192.168.33.130's password:
    Now try logging into the machine, with "ssh '-p 22 root@192.168.33.130'",
and check in:

        .ssh/authorized_keys

    to make sure we haven't added extra keys that you weren't expecting.
```

4）实现无密码且不需要 Expect 就可以批量管理：

```
[root@oldboy ~]# cat exec.sh
ssh 192.168.33.128 uptime
ssh 192.168.33.129 uptime
ssh 192.168.33.130 uptime
```

执行结果如下：

```
[root@oldboy ~]# sh exec.sh
 15:22:33 up 17:28,  2 users,  load average: 0.00, 0.00, 0.00
 12:48:30 up  1:17,  2 users,  load average: 0.45, 0.11, 0.04
 12:48:38 up 1 day, 16:38,  2 users,  load average: 0.00, 0.00, 0.00
```

5）配合 ansible（Python 开发的基于 SSH 的批量管理工具）实现自动化运维管理。首先安装 ansible，命令如下：

```
yum install epel-release  -y
```

```
yum install ansible -y
```

然后编辑 ansible 的主机配置文件 hosts，添加主机组 oldboy：

```
[root@oldboy ~]# tail -4 /etc/ansible/hosts
[oldboy]
192.168.33.128
192.168.33.129
192.168.33.130
```

接着使用 ansible 并借助 Expect 建立好的 SSH 密钥认证执行命令，过程和结果如图 18-1 所示。

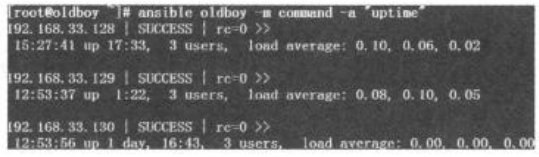


图 18-1　命令执行过程和结果

ansible 自动化管理超出了本书的内容范围，因此不再过多讲解，更多内容可以了解老男孩的 Linux 教学课程或博客上的内容。

18.9　本章小节

老男孩如是说：Expect 程序的功能远不止本文介绍的这些，本文主要从运维工作实战的角度，给大家讲解自动化运维中用 Shell 脚本难以实现的交互式问题的解决方案，更多关于 Expect 的内容，请参考相关资料。对于一般的企业运维人员，掌握本章所讲的这些，就已经够用了，类似的交互工具还有 sshpass、ansible 等。

Linux

第19章

企业 Shell 面试题及企业运维实战案例

首先要恭喜看到此章的所有读者，如果前 18 章你都能够掌握，那么搞定本章的试题和案例，将不再是难事。

本章所讲的内容是 IT 运维中常见的企业面试题及企业实战案例，在老男孩以往的教学中，对这些案例都是不给答案的，而是由学生自己来完成，并让他们在班级里上百名学生面前进行讲解，以使他们真正掌握 Shell 编程。现在作为压轴戏，将这些面试题和读者分享，强烈建议读者在看本章的内容时，尽量思考并自行完成试题，之后再参考答案，就当是做一套综合考试题吧，看看经过前 18 章的学习，自己对 Shell 脚本知识到底掌握得如何。

本章的考试题（无答案）可从此网址获取：http://oldboy.blog.51cto.com/2561410/1632876。

19.1 企业 Shell 面试题案例

19.1.1 面试题 1：批量生成随机字符文件名

使用 for 循环在 /oldboy 目录下批量创建 10 个 html 文件，其中每个文件需要包含 10 个随机小写字母加固定字符串 oldboy，名称示例如下：

```
[root@oldboy scripts]# ls /oldboy
apquvdpqbk_oldboy.html  mpyogpsmwj_oldboy.html  txynzwofgg_oldboy.html
bmqiwhfpgv_oldboy.html  mtrzobsprf_oldboy.html  vjxmlflawa_oldboy.html
jhjdcjnjxc_oldboy.html  qeztkkmewn_oldboy.html
```

```
jpvirsnjld_oldboy.html  ruscyxwxai_oldboy.html
```

（1）问题分析

本题考察的第一个知识点就是生成 10 个随机小写字母，产生随机数的方法在第 11.5 节已经详细讲解过了，这里采取 openssl 命令的方法来实现，操作结果如下：

```
[root@oldboy scripts]# openssl rand -base64 40          #<== 生成 40 位随机数。
js0k6Mex+c1XicknYocxNTxJZXwfn25xgXeJMQwXLZvGoye80SmcXA==
[root@oldboy scripts]# openssl rand -base64 40|sed 's#[^a-z]##g'
#<== 利用 sed 替换掉非小写字符。
xgyecpgowfdamqhdwzrivw
[root@oldboy scripts]# openssl rand -base64 40|sed 's#[^a-z]##g'|cut -c 2-11
#<== 利用 cut 取 10 位。
butvlgpssj
```

其次，本题考察 for 循环的使用，最终脚本答案见如下参考解答。

（2）参考解答

```
[root@oldboy scripts]# cat 19_1.sh
#!/bin/sh
Path=/oldboy                                #<== 定义生成文件的路径。
[ -d "$Path" ]||mkdir -p $Path              #<== 如果定义的路径不存在则创建。
for n in `seq 10`                           #<==for 循环 10 次，即创建 10 个文件。
do
    random=$(openssl rand -base64 40|sed 's#[^a-z]##g'|cut -c 2-11)
                                            #<== 将 10 位随机字符赋值给变量。
    touch $Path/${random}_oldboy.html       #<== 根据题意生成所需要的文件。
done
```

（3）执行结果

```
[root@oldboy scripts]# ls /oldboy
apquvdpqbk_oldboy.html  mpyogpsmwj_oldboy.html  txynzwofgg_oldboy.html
bmqiwhfpgv_oldboy.html  mtrzobsprf_oldboy.html  vjxmlflawa_oldboy.html
jhjdcjnjxc_oldboy.html  qeztkkmewn_oldboy.html
jpvirsnjld_oldboy.html  ruscyxwxai_oldboy.html
```

19.1.2　面试题 2：批量改名

将 19.1.1 节所得文件名中的 oldboy 字符串全部改成 oldgirl（最好用 for 循环实现），并且将扩展名 html 全部改成大写。

（1）问题分析

本题考察的知识点是对文件进行批量改名，在前面 11.2 节已讲解过类似案例。

（2）参考解答

方法 1：

```
[root@oldboy scripts]# cat 19_2_1.sh
```

```
#!/bin/sh
Filename=_oldgirl.HTML  #<== 定义替换后的目标字符串。
Dirname="/oldboy"       #<== 定义操作目录。
cd $Dirname||exit 1     #<== 切换到指定目录下，如果不成功则退出。
for n in `ls`           #<== 遍历当前目录获取文件名列表，并循环处理。
do
    name=$(echo ${n}|awk -F '_' '{print $1}')   #<== 定义替换后的目标。
    mv $n ${name}${Filename}                    #<== 实际 mv 改名操作命令。
done
```

方法 2：非循环实现方法，4.3.3 节也讲过类似的知识。

```
[root@oldboy scripts]# cat 19_2_2.sh
#!/bin/sh
Path="/oldboy"
cd $Path && \
ls|awk -F '_' '{print "mv "$0" "$1"_oldgirl.HTML"}'|bash
#<== 拼接 mv 操作命令后执行。
```

方法 3：专业的事交给专业的“人”做。

```
[root@oldboy scripts]# rename oldboy.html oldgirl.HTML *.html
```

（3）执行结果（以方法 1 为例的执行结果）

```
[root@oldboy scripts]# sh 19_2_1.sh
[root@oldboy scripts]# ls /oldboy
apquvdpqbk_oldgirl.HTML  mpyogpsmwj_oldgirl.HTML  txynzwofgg_oldgirl.HTML
bmqiwhfpgv_oldgirl.HTML  mtrzobsprf_oldgirl.HTML  vjxmlflawa_oldgirl.HTML
jhjdcjnjxc_oldgirl.HTML  qeztkkmewn_oldgirl.HTML
jpvirsnjld_oldgirl.HTML  ruscyxwxai_oldgirl.HTML
```

19.1.3 面试题 3：批量创建特殊要求用户

批量创建 10 个系统账号 oldboy01-oldboy10 并设置密码（密码为随机数，要求是字符和数字等的混合）。

不用 for 循环的实现思路可参考：http://user.qzone.qq.com/49000448/blog/1422183723
本题的详细答案请参见第 11 章**范例 11-14**，此处仅作为 Shell 案例收集整理。

19.1.4 面试题 4：扫描网络内存活主机

写一个 Shell 脚本，判断 10.0.0.0/24 网络里，当前在线的 IP 有哪些？

（1）问题分析

判断 IP 在线最常见的方法就是使用 ping 命令，其实还有一个不错的命令就是 nmap，这个 nmap 命令有可能需要执行括号内的命令安装（yum install nmap -y）。

（2）参考解答

方法 1：并发 ping 方案（该方法来自老男孩教育运维 26 期的曹同学）

```
[root@oldboy scripts]# cat 19_5_1.sh
#!/bin/sh
CMD="ping -W 2 -c 2"                    #<== 定义 ping 命令及参数。
Ip="10.0.0."                            #<==IP 前半部分。
for n in $(seq 254)                     #<==IP 后半部分，1-254。
do
    {
    $CMD $Ip$n &> /dev/null             #<== 对指定 IP 实行 ping 政策。
    if [ $? -eq 0 ];then                #<== 如果返回值为 0，则证明该 IP 对应的主机存在。
        echo "$Ip$n is ok"              #<== 打印提示。
    fi
    }&                                  #<== Shell 的并发检测功能，批量 ping，快速返回结果。
done
```

提示：这里的 Shell 并发功能值得读者格外注意。

以下为 ping 方案脚本执行结果，几乎在 1 ~ 2 秒内返回全部结果。

```
[root@oldboy scripts]# sh 19_5_1.sh
[root@oldboy scripts]# 10.0.0.8 is ok  #<== 因并发问题导致输出看起来异常，非错误。
10.0.0.3 is ok
10.0.0.2 is ok
10.0.0.1 is ok
10.0.0.31 is ok
```

方法 2：nmap 方案（该方法由老男孩教育运维 26 期的曹同学提供）

```
[root@oldboy scripts]# cat 19_5_2.sh
#!/bin/sh
CMD="nmap -sP "
Ip="10.0.0.0/24"
CMD2="nmap -sS"
$CMD $Ip|awk '/Nmap scan report for/ {print $NF}'
```

以下为 nmap 方案相关脚本的执行结果，几乎在 1 ~ 2 秒内返回全部结果。

```
[root@oldboy scripts]# sh 19_5_2.sh
(10.0.0.1)
(10.0.0.2)
(10.0.0.3)
(10.0.0.8)
10.0.0.31
```

19.1.5　面试题 5：解决 DOS 攻击

写一个 Shell 脚本以解决 DOS 攻击生产的问题。

请根据 Web 日志或网络连接数，监控当某个 IP 并发连接数或短时内 PV 达到 100

（读者可根据实际情况来设定）时，即调用防火墙命令封掉对应的 IP。防火墙命令为："iptables -I INPUT -s IP -j DROP"（IP 为要封的地址）。

本题的详细答案可参见第 10 章**范例 10-10**，此处仅作为 Shell 案例收集整理。

19.1.6 面试题 6：MySQL 数据库分库备份

请用脚本实现对 MySQL 数据库的分库备份。

本题的详细答案见第 11 章**范例 11-11**，此处仅作为 Shell 案例收集整理。

19.1.7 面试题 7：MySQL 数据库分库分表备份

请用脚本实现对 MySQL 数据库的分库加分表备份。

本题的详细答案见第 11 章**范例 11-12**，此处仅作为 Shell 案例收集整理。

19.1.8 面试题 8：筛选符合长度的单词

利用 bash for 循环打印下面这句话中字母数不大于 6 的单词（某企业面试真题）。

```
I am oldboy teacher welcome to oldboy training class
```

本题的详细答案见第 13 章**范例 13-4**，此处仅作为 Shell 案例收集整理。

19.1.9 面试题 9：MySQL 主从复制异常监控

开发一个守护进程脚本，每 30 秒监控一次 MySQL 主从复制是否异常（包括不同步及延迟），如果异常，则发送短信并发送邮件给管理员存档。

本题的详细答案见第 13 章**范例 13-6**，此处仅作为 Shell 案例收集整理。

19.1.10 面试题 10：比较整数大小

综合实战案例：开发 Shell 脚本分别实现以脚本传参及 read 读入的方式比较 2 个整数大小。用条件表达式（禁止 if）进行判断并以屏幕输出的方式告知用户比较的结果。注意：一共要开发 2 个脚本。当使用脚本传参及 read 读入的方式时，需要对变量是否为数字，以及传参个数是否正确给予提示。

本题的详细答案见第 6 章范例 6-35，此处仅作为 Shell 案例集中收集整理。

19.1.11 面试题 11：菜单自动化软件部署

综合实例：打印选择菜单，按照选择一键安装不同的 Web 服务。

示例菜单：

```
[root@oldboy scripts]# sh menu.sh
    1.[install lamp]
```

```
2.[install lnmp]
3.[exit]
pls input the num you want:
```

要求：

1）当用户输入 1 时，输出“start installing lamp.”的提示，然后执行 /server/scripts/lamp.sh，脚本内容输出“lamp is installed”后退出脚本，也就是实际工作中使用的 lamp 一键安装脚本。

2）当用户输入 2 时，输出“start installing lnmp.”的提示，然后执行 /server/scripts/lnmp.sh，输出“lnmp is installed”后退出脚本，也就是实际工作中使用的 lnmp 一键安装脚本。

3）当输入 3 时，退出当前菜单及脚本。

4）当输入任何其他字符时，给出提示“Input error”后退出脚本。

5）对要执行的脚本进行相关的条件判断，例如：脚本文件是否存在，是否可执行等判断，尽量用上前面讲解过的知识点。

本题的详细答案见第 6 章范例 6-36，此处仅作为 Shell 案例集中收集整理。

19.1.12　面试题 12：Web 及 MySQL 服务异常监测

用 if 条件语句实现对 Nginx Web 服务及 MySQL 数据库服务是否正常的检测，如果服务未启动，则启动相应服务。

本题的详细答案见第 7 章范例 7-4，此处仅作为 Shell 案例集中收集整理。

19.1.13　面试题 13：监控 Memcached 缓存服务

监控 Memcached 缓存服务是否正常，模拟用户（Web 客户端）检测。

使用 nc 命令加上 set/get 来模拟检测。

（1）问题分析

要想写出相应的脚本，必须要对 Memcached 服务很熟练，此部分内容可参考老男孩已经出版的图书《跟老男孩学 Linux 运维：Web 集群实战》一书的第 13 章。

（2）环境准备

```
[root@oldboy scripts]# yum install memcached -y
[root@oldboy scripts]# yum install nc -y    #<== 用于探测端口及异地传输文件的工具。
[root@oldboy scripts]# memcached -uroot -m 16 -p 11211  #<== 启动 Memached 服务。
[root@oldboy scripts]# netstat -lntup|grep memcache
tcp    0    0 0.0.0.0:11211           0.0.0.0:*    LISTEN    57033/memcached
tcp    0    0 :::11211                :::*         LISTEN    57033/memcached
udp    0    0 0.0.0.0:11211    0.0.0.0:*                     57033/memcached
udp    0    0 :::11211         :::*                          57033/memcached
```

（3）参考解答

脚本如下：

```
[root@oldboy scripts]# cat 19_13_1.sh
#!/bin/sh
if [ `netstat -lntup|grep 11211|wc -l` -lt 1 ]
#<== 若端口对应的行数小于 1，则说明服务未启动。
  then
    echo "Memcached Service is error."          #<== 报错后，退出。
    exit 1
fi
printf "del key\r\n"|nc 127.0.0.1 11211 &>/dev/null #<== 删除缓存中的 key 及对应的值。
printf "set key 0 0 10 \r\noldboy1234\r\n"|nc 127.0.0.1 11211 &>/dev/null
                                                  #<== 添加新值。
McValues=`printf "get key\r\n"|nc 127.0.0.1 11211|grep oldboy1234|wc -l`
                                                  #<== 查看新值。
if [ $McValues -eq 1 ]                            #<== 如果查到了新值，则进入判断。
  then
    echo "Memcached status is ok"
else
    echo "Memcached status is bad"
fi
```

（4）执行结果

```
[root@oldboy scripts]# sh 19_13_1.sh
Memcached status is ok
[root@oldboy scripts]# pkill memcached
[root@oldboy scripts]# sh 19_13_1.sh
Memcached Service is error.
[root@oldboy scripts]# memcached -uroot -m16m -d
[root@oldboy scripts]# sh 19_13_1.sh
Memcached status is ok
```

19.1.14 面试题 14：开发脚本实现入侵检测与报警

监控 Web 站点目录（/var/html/www）下的所有文件是否被恶意篡改（文件内容被更改了），如果有则打印改动的文件名（发邮件），定时任务每 3 分钟执行一次。

（1）问题分析

1）首先要说明的是，思考过程的积累比实际代码开发的能力积累更重要。

2）什么是恶意篡改，只要是未经过许可的改动都是篡改。

3）文件内容被改动了会有如下特征。

- 大小可能会变化。
- 修改时间会变化。
- 文件内容会变化，利用 md5sum 指纹校验。

❑ 增加或删除文件，比对每次检测前后的文件数量。

（2）参考解答

本题主要采用 md5sum 的方法来实现。

第一步，在企业网站发布代码之后，即对所有网站数据建立初始指纹库和文件库，这个步骤很重要，没有基础的指纹库，无法进行入侵检测。

以 /var/html/www 作为站点目录为例。

1）建立测试数据：

```
[root@oldboy scripts]# mkdir /var/html/www -p            #<== 创建站点目录。
[root@oldboy scripts]# cp -a /etc/a* /var/html/www/      #<== 复制少量测试数据。
[root@oldboy scripts]# cp -a /etc/b* /var/html/www/      #<== 复制少量测试数据。
[root@oldboy scripts]# ls /var/html/www/                 #<== 检查。
abrt  adjtime  aliases.db  alternatives  ansible      at.deny  audit            bashrc
acpi  aliases  alsa        anacrontab    asound.conf  audisp   bash_completion.d  blkid
```

2）建立初始的文件指纹库：

```
[root@oldboy scripts]# find /var/html/www -type f|xargs md5sum >/opt/
zhiwen.db.ori    #<== 建立文件内容指纹库。
[root@oldboy scripts]# tail /opt/zhiwen.db.ori
c74c148efcf0db7a55b4095628d72708  /var/html/www/asound.conf
d14759ce4246bcd47bc11e17dd8def64  /var/html/www/acpi/actions/power.sh
c7e90504ce3c63d143c9bf4383ad3d02  /var/html/www/acpi/events/video.conf
f29f9bfd00dd416c97db8f738e5a9237  /var/html/www/acpi/events/power.conf
2bba2f3012bdee732a5cfb7edf3a52a2  /var/html/www/audisp/audispd.conf
199eaa1e43fa9139f0910bdb64fd219e  /var/html/www/audisp/plugins.d/af_unix.conf
b011b799f8cf6918611c9949bef00c19  /var/html/www/audisp/plugins.d/syslog.conf
68b329da9893e34099c7d8ad5cb9c940  /var/html/www/at.deny
209d77d1471935df4ac5e4277207a6bb  /var/html/www/bashrc
039e9bb0c206d57c616c8fab3cb82c2a  /var/html/www/aliases
```

3）建立初始的文件库：

```
[root@oldboy scripts]# find /var/html/www -type f >/opt/wenjian.db.ori
#<== 建立文件数量和名字库。
[root@oldboy scripts]# tail /opt/wenjian.db.ori
/var/html/www/asound.conf
/var/html/www/acpi/actions/power.sh
/var/html/www/acpi/events/video.conf
/var/html/www/acpi/events/power.conf
/var/html/www/audisp/audispd.conf
/var/html/www/audisp/plugins.d/af_unix.conf
/var/html/www/audisp/plugins.d/syslog.conf
/var/html/www/at.deny
/var/html/www/bashrc
/var/html/www/aliases
```

第二步，检测文件内容和文件数量变化。

1）检测文件内容变化：

```
    [root@oldboy scripts]# echo oldboy >>/var/html/www/audisp/plugins.d/af_
unix.conf                                                    #<== 篡改文件。
    [root@oldboy scripts]# export LANG=en                    #<== 调整字符集。
    [root@oldboy scripts]# md5sum -c --quiet /opt/zhiwen.db.ori
    #<== 检查所有文件内容是否变化。
    /var/html/www/audisp/plugins.d/af_unix.conf: FAILED    #<== 变化的会被打印出来。
    md5sum: WARNING: 1 of 33 computed checksums did NOT match     #<== 综合提示。
```

2）检测文件数量变化：

```
    [root@oldboy scripts]# echo oldgirl.txt >/var/html/www/test.txt
    #<== 模拟增加新文件。
    [root@oldboy scripts]# md5sum -c --quiet /opt/zhiwen.db.ori
    #<== 利用指纹库无法检测新增文件。
    /var/html/www/audisp/plugins.d/af_unix.conf: FAILED
    md5sum: WARNING: 1 of 33 computed checksums did NOT match
    [root@oldboy scripts]# find /var/html/www -type f >/opt/wenjian.db_curr.ori
    #<== 获取检测前的所有文件数量及文件名。
    [root@oldboy scripts]# diff /opt/wenjian.db*     #<== 采用 diff 命令比较。
    18d17
    < /var/html/www/test.txt     #<==test.txt 就是新增的，怎么样，还可以吧。
```

第三步，开发检查指纹识别脚本。

首先，人工做如下操作：

```
    find /var/html/www -type f|xargs md5sum >/opt/zhiwen.db.ori
    find /var/html/www -type f >/opt/wenjian.db.ori
```

脚本检测会以上述两个命令获取的结果为原始的正确依据，如下

```
    [root@oldboy scripts]# cat 19_14_1.sh
    #!/bin/bash
    RETVAL=0                                  #<== 状态初始化。
    export LANG=en                            #<== 调整字符集。
    CHECK_DIR=/var/html/www                   #<== 定义要监测的站点目录。
    [ -e $CHECK_DIR ]||exit 1                 #<== 如果目录不存在则退出脚本。

    ZhiWenDbOri="/opt/zhiwen.db.ori"          #<== 定义原始指纹库路径。
    FileCountDbOri="/opt/wenjian.db.ori"      #<== 定义原始文件库路径。
    ErrLog="/opt/err.log"                     #<== 定义检测后的内容日志。

    [ -e $ZhiWenDbOri ]||exit 1               #<== 如果原始指纹库不存在则退出脚本。
    [ -e $FileCountDbOri ]||exit 1            #<== 如果原始文件库不存在则退出脚本。

    #judge file content
    echo "[root@oldboy scripts]# md5sum -c --quiet /opt/zhiwen.db.ori">$ErrLog
```

```
                                                    #<== 打印检测命令。
md5sum -c --quiet /opt/zhiwen.db.ori &>>$ErrLog #<== 实际执行检测命令。
RETVAL=$?                                           #<== 收集返回值。

#com file count
find $CHECK_DIR -type f >/opt/wenjian.db_curr.ori   #<== 实际执行检测命令，获取
                                                        最新文件数量等。
echo "[root@oldboy scripts]# diff /opt/wenjian.db*" &>>$ErrLog #<== 打印检测命令。
diff /opt/wenjian.db* &>>$ErrLog #<== 实际执行检测命令，比对文件数量及文件名变化情况。

if [ $RETVAL -ne 0 -o `diff /opt/wenjian.db*|wc -l` -ne 0 ]
#<== 如果返回值不为 0，或者比对结果行数不为 0，则进入判断。
  then
    mail -s "`uname -n` $(date +%F) err" 31333741-@qq.com <$ErrLog  #<== 发送邮件。
else
    echo "Sites dir is ok"|mail -s "`uname -n` $(date +%F) is ok" 31333741-@qq.com
fi
```

提示：如果要使用邮件则请参考本书前面章节介绍的 mail 发送相关内容进行配置。

然后，利用定时任务检查，命令如下：

```
[root@oldboy scripts]# crontab -l|tail -2
# ids monitor site dir and file change by oldboy at 20161111
*/3 * * * * /bin/sh /server/scripts/19_14_1.sh >/dev/null 2>&1
```

现在来思考一下，在企业中一般什么文件需要做指纹验证呢？

系统命令、用户文件、配置文件、启动文件等重要文件，都要监控起来，另外，在实际工作中应对所有的用户操作做日志审计，让所有人的所有操作无处遁形，起到威慑和监督的作用，从而减少被当作“黑锅侠”的风险。

19.1.15　面试题 15：开发 Rsync 服务启动脚本

写出网络服务独立进程模式下 Rsync 的系统启动脚本，例如：/etc/init.d/rsyncd {start|stop|restart}。

要求：

- 要使用系统函数库技巧。
- 要用函数，不能将一堆代码混在一起。
- 可被 chkconfig 管理。

本题的详细答案见第 7 章范例 7-9 及范例 7-10，此处仅作为 Shell 案例收集整理。

19.1.16　面试题 16：开发 MySQL 多实例启动脚本

已知 MySQL 多实例启动命令为：mysqld_safe --defaults-file=/data/3306/my.cnf &

停止命令为：mysqladmin -u root -poldboy123 -S /data/3306/mysql.sock shutdown

请完成 MySQL 多实例启动脚本的编写。

要求：用函数、case 语句、if 语句等实现。

（1）问题分析

要想写出脚本，必须要对 MySQL 服务很熟练，此部分内容可参考老男孩已经出版的图书《跟老男孩学 Linux 运维：Web 集群实战》一书的第 9 章。

（2）参考解答

脚本如下：

```
[root@oldboy scripts]# cat 19_16_1.sh
#!/bin/sh
#################################################
#this scripts is created by oldboy at 2007-06-09
#blog:http://oldboy.blog.51cto.com
#################################################
#init
Port=3306
MysqlUser="root"
MysqlPass="oldboy123"
CmdPath="/application/mysql/bin"

#startup function
start()
{
    if [  `netstat -lnt|grep "$Port"|wc -l` -eq 0 ]
      then
        printf "Starting MySQL...\n"
        /bin/sh ${CmdPath}/mysqld_safe --defaults-file=/data/${Port}/my.cnf
2>&1 > /dev/null &
    else
        printf "MySQL is running...\n"
    fi
}

#stop function
stop()
{
    if [ ! `netstat -lnt|grep "$Port"|wc -l` -eq 0 ]
      then
        printf "Stoping MySQL...\n"
        ${CmdPath}/mysqladmin -u ${MysqlUser} -p${MysqlPass} -S /data/${Port}/
mysql.sock shutdown
    else
       printf "MySQL is stopped...\n"
    fi
}
```

```
#restart function
restart()
{
    printf "Restarting MySQL...\n"
    stop
    sleep 2
    start
}

case "$1" in
start)
    start
    ;;
stop)
    stop
    ;;
restart)
    restart
    ;;
*)
    printf "Usage: $0 {start|stop|restart}\n"
esac
```

19.1.17　面试题 17：开发学生实践抓阄脚本

老男孩培训的学生有了去企业项目实践的机会，但是，名额有限，仅限 3 人（班长带队）。

因此需要开发一个抓阄的程序来挑选学生，具体要求如下：

1）执行脚本后，输入想去的同学的英文名字全拼，产生随机数（01 ~ 99 之间的数字），数字越大就越有机会去参加项目实践，对于前面已经抓到的数字，下次不能再出现。

2）输入第一个名字之后，屏幕输出信息，并将名字和数字记录到文件里，**程序不能退出，继续等待别的学生输入**。

（1）参考解答

脚本如下：

```
[root@oldboy scripts]# cat 19_17_1.sh
#!/bin/bash
# http://oldboy.blog.51cto.com/2561410/1308647
FileLog=/tmp/zhuajiu.log          #<== 定义最终抓阄后的结果日志。
[ -f "$FileLog" ]||touch $FileLog  #<== 如果不存在日志文件，则创建一个空文件。
function Check_Name(){            #<== 提示用户输入名字函数，判断用户的输入是否重复。
    while true                    #<== 进入 while 无限循环。
    do
```

```
        read -p "please input your English name: " name #<== 提示并读入用户输入的名字。
        if [ -n "$name" -a "$(grep -w "$name" $FileLog|wc -l)" -eq 0 ]; then
                #<== 如果名字不为空，并且日志文件里没有重名的，则进入判断。
            flag=1  #<== 将 flag 标志设置为 1，表示可以继续执行后面的内容，后面将以此为条件
                    进行判断操作。
            break   #<== 满足输入条件且没有重名，则跳出循环，为了避免无限循环，一定要给一
                    个跳出的条件及命令。
        else
            echo "The name your input is null or alread exist"
                #<== 如果 if 条件不成立，则给出提示。
            continue #<== 重复 while 循环，即让用户重新输入名字。
        fi
    done
}
function Product_RandomNum(){   #<== 该函数生成随机数，并判断随机数是否重复。
    if [ $flag -eq 1 ];then     #<== 当 flag 变量为 1 时，表示用户输入的名字合法了。
        while true #<== 继续无限循环，生成随机数。
        do
            RandomNum=$(expr $RANDOM % 99  + 1)  #<== 生成 1 ~ 99 的随机数。
            if [ $(grep -w "$RandomNum" $FileLog|wc -l ) -ne 1 ];then
                #<== 如果随机数没被人抽到，则进入判断。
                echo "$name,your num is ${RandomNum}." | tee-a $FileLog
                #<== 屏幕输出，并将名字和数字记录。
                flag1=0 #<== 这里将 flag1 设置为 0 的目的还是为了给出条件，供下面的脚本进行判断。
            else
                flag1=1 #<== 这里将 flag1 设置为 0 的目的还是为了给出条件，供下面的脚本进行判断。
            fi
            if [ $flag1 -eq 1 ];then #<== 如果 flag1 为 1，则表示随机数重复了。
                Product_RandomNum #<== 既然随机数重复了，那么就重新执行随机数生成函数。
            else
                Check_Name          #<== 如果 flag1 不为 1，则表示这个人抽取随机数成功，
                                        给出提示供下一个小伙伴继续抽取随机数。
            fi
        done
    fi
}
function main(){ #<== 主函数调用上述其他函数。
    Check_Name
    Product_RandomNum
}
main              #<== 总执行入口（跟老男孩学习就要做到专业和规范，加油）。
```

（2）执行结果

```
[root@oldboy scripts]# >/tmp/zhuajiu.log
[root@oldboy scripts]# sh 19_17_1.sh
please input your English name: oldboy
oldboy,your num is 72.
please input your English name: oldgirl
```

```
oldgirl,your num is 17.
please input your English name: oldboy
The name your input is null or alread exist
please input your English name: oldgirl
The name your input is null or alread exist
please input your English name: tingting
tingting,your num is 23.
please input your English name: lili
lili,your num is 19.
please input your English name: ^C

[root@oldboy scripts]# sort -t" " -rn -k4 /tmp/zhuajiu.log
oldboy,your num is 72.
tingting,your num is 23.
lili,your num is 19.
oldgirl,your num is 17.
```

提示：清除日志后，可以重新开始。

此题有众多网友参与解答，因此把本题的答案地址发给大家，里面包含了大量的评论和答案：

http://oldboy.blog.51cto.com/2561410/1308647

19.1.18　面试题 18：破解 RANDOM 随机数

已知下面的字符串是 RANDOM 随机数变量经过 md5sum 处理后，再截取一部分连续字符串的结果，请破解这些字符串在使用 md5sum 处理前所对应的数字。

21029299

00205d1c

a3da1677

1f6d12dd

890684b

方法 1（由老男孩教育 28 期的学生实现）：

```
[root@oldbog scripts]# cat 19_18_1.sh
#!/bin/sh
array=( #<== 把字符串放到数组里。
21029299
00205d1c
a3da1677
1f6d12dd
890684b
)
```

```
Path=/tmp/md5.txt
Num=0
funGetMd5() {
    [ -f "$Path" ]||touch $Path
    rowNum=$(wc -l < $Path)
    if [ $rowNum -ne 32768 ];then
        > $Path
        for ((Num=0;Num<=32767;Num++))
        do
            {
            Stat=$(echo $Num|md5sum|cut -c 1-8)
            echo "$Stat $Num" >> $Path #<== 建立数字和md5sum后内容
                                              的对应关系。
            }&

         done
    else
         return 0
    fi
}

funFindMd5() {
    word=$(echo "${array[@]}"| sed -r 's# |\n#|#g')  #<== 取出所有数组元素
                                                        并用 | 分隔开。
    grep -E "$word" $Path   #<== 同时过滤包含所有不同字符串的内容。
}
funcMain(){
    funGetMd5
    funFindMd5
}
funcMain
```

提示：本题采用不一样的函数命名方式，供读者学习参考。

19.1.19 面试题 19：批量检查多个网站地址是否正常

批量检查多个网站地址是否正常。

要求：

1）使用 Shell 数组方法实现，检测策略尽量模拟用户访问。

2）每 10 秒钟做一次所有的检测，对于无法访问的网址输出报警。

3）待检测的地址如下：

http://blog.oldboyedu.com

http://blog.etiantian.org

http://oldboy.blog.51cto.com

http://10.0.0.7

本题的详细答案见第 13 章范例 13-5，此处仅作为 Shell 案例收集整理。

19.1.20　面试题 20：单词及字母去重排序

用 Shell 处理以下的内容[⊖]

the squid project provides a number of resources to assist users design,implement and support squid installations. Please browse the documentation and support sections for more infomation,by oldboy training.

- ❑ 按单词出现的频率降序排序！
- ❑ 按字母出现的频率降序排序！

这道题适合用 awk 进行处理。

（1）数据准备

```
    [root@oldboy scripts]# cat oldboy.txt
    the squid project provides a number of resources to assist users
design,implement and support squid installations. Please browse the documentation
and support sections for more infomation,by the oldboy training and oldboy,but
not oldgirl.
```

（2）按单词出现频率降序排序的参考解答

方法 1：awk 数组法

```
    [root@oldboy scripts]# awk -F "[,. ]" '{for(i=1;i<=NF;i++)array[$i]++}
END{for(key in array)print array[key],key|"sort -nr"}' oldboy.txt|column -t
    3  the
    3  and
    2  support
    2  squid
    2  oldboy
    2
    1  users
    1  training
    1  to
    ... 省略若干 ...
```

方法 2：sort 排序法

```
    [root@oldboy scripts]# tr "[ ,.]" "\n"<oldboy.txt|grep -v "^$"|sort|uniq
-c|sort -rn
          3 the
          3 and
          2 support
          2 squid
```

⊖ 该面试题见 http://oldboy.blog.51cto.com/2561410/1686891。

```
          2 oldboy
          1 users
          1 training
          1 to
       ... 省略若干 ...
```

方法 3：tr 配合 awk 数组法

```
    [root@oldboy scripts]# tr "[ ,.]" "\n"<oldboy.txt|sed '/^$/d'|awk
'{++S[$0]}END{for(key in S)print S[key]" " key|"sort -rn"}'
    3 the
    3 and
    2 support
    2 squid
    2 oldboy
    1 users
    1 training
    1 to
    ... 省略若干 ...
```

（3）按字母出现频率降序排序的参考解答

方法 1：awk 数组法

```
    [root@oldboy scripts]# tr "{ |,|.}" "\n"<oldboy.txt|awk -F "" '{for(i=1;
i<=NF;i++)array[$i]++}END{for(key in array)print array[key],key|"sort -nr"}'
    22 o
    19 s
    18 t
    18 e
    16 n
    15 i
    13 r
    11 d
    11 a
    ... 省略若干 ...
```

方法 2：grep 及 sort 排序法

```
    [root@oldboy scripts]# grep -o "[^ ]" oldboy.txt|sort|uniq -c|sort -rn -k1
        22 o
        19 s
        18 t
        18 e
        16 n
        15 i
        13 r
        11 d
        11 a
    ... 省略若干 ...
```

19.1.21　面试题 21：开发脚本管理服务端 LVS

请在 LVS 负载均衡主节点上，开发管理 LVS 服务的脚本 ip_vs。

实现：利用 ipvsadm 可以启动并配置好 LVS 服务，脚本如下：/etc/init.d/lvs{start|stop|restart}

（1）问题分析

此题要求读者对 LVS 软件有一定的了解，此部分内容可以参考老男孩网上课程内容或相关书籍。另外，如果进行实际测试，需要提前执行“yum install ipvsadm -y”安装 LVS 软件。

（2）参考解答

```
[root@oldboy scripts]# cat 19_21_1.sh
#!/bin/bash
# Written by oldboy ??31333741@qq.com
# QQ:31333741
# description: Config Director vip and ipvs
. /etc/init.d/functions
VIP=10.0.0.3
INTERFACE=eth0
SubINTERFACE=${INTERFACE}:`echo $VIP|cut -d. -f4`
PORT=80
GW=10.0.0.254
RETVAR=0

IP=/sbin/ip
ROUTE=/sbin/route
IPVSADM=/sbin/ipvsadm
ARPING=/sbin/arping

RIPS=( #<== 定义 realserver 节点 IP 数组。
        10.0.0.7
        10.0.0.8
     )

function usage (){
    echo "Usgae : $0 {start|stop|restart}"
    return 1
}

function ipvsStart (){ #<== 配置 ipvs。
    $IP addr add $VIP/24 dev ${INTERFACE} label $SubINTERFACE #<== 添加 VIP。
    $ROUTE add -host $VIP dev $SubINTERFACE #<== 添加 VIP 对应主机路由。
    $IPVSADM -C
    $IPVSADM -A -t $VIP:$PORT -s wrr -p 60  #<== 生成 ipvs 实例。
    for ((i=0; i<`echo ${#RIPS[*]}`; i++))
    do
```

```
        $IPVSADM -a -t $VIP:$PORT -r ${RIPS[$i]}:$PORT -g -w 1
        #<== 添加节点。
    done
    RETVAR=$?

    # update MAC
    $ARPING -c 1 -I ${INTERFACE} -s $VIP $GW &>/dev/null #<== 更新 arp 表。
    if [ $RETVAR -eq 0 ]
      then
        action "Ipvsadm started." /bin/true
    else
        action "Ipvsadm started." /bin/false
    fi
    return $RETVAR
}

function ipvsStop (){ #<== 停止 lpvs。
    $IPVSADM -C
    $IPVSADM -Z
    $IP addr del $VIP/24 dev ${INTERFACE} label $SubINTERFACE &>/dev/null
    #<== 删除 VIP。
    RETVAR=$?
    $ROUTE del -host $VIP dev $SubINTERFACE &>/dev/null  #<== 删除路由。
    $ARPING -c 1 -I ${INTERFACE} -s $VIP $GW  >/dev/null 2>&1 #<== 更新 arp 表。
    if [ $RETVAR -eq 0 ]
      then
        action "Ipvsadm stopped." /bin/true
    else
        action "Ipvsadm stopped" /bin/false
    fi
    return $RETVAR
}

main ()
{
    #judge argv num
    if [ $# -ne 1 ]; then
        usage $0
    fi
    case "$1" in
        start)
            ipvsStart
            ;;
        stop)
            ipvsStop
            ;;
        restart)
            ipvsStop
            ipvsStart
```

```
            ;;
        *)
            usage $0
            ;;
    esac
}
#start operating
main $*
```

（3）执行结果（见图 19-1）

```
[root@oldgirl scripts]# sh 19_21_1.sh start
Ipvsadm start.                                             [确定]
[root@oldgirl scripts]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.0.0.3:80 wrr persistent 60
  -> 10.0.0.7:80                  Route   1      0          0
  -> 10.0.0.8:80                  Route   1      0          0
[root@oldgirl scripts]# sh 19_21_1.sh stop
Ipvsadm start.                                             [确定]
[root@oldgirl scripts]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
```

图 19-1　管理 LVS 服务的脚本执行结果

19.1.22　面试题 22：LVS 节点健康检查及管理脚本

请在 LVS 负载均衡主节点上，模拟 keepalived 健康检查功能管理 LVS 节点，当节点挂掉时从服务器池中将其剔除，好了后再将其加到服务器池中来。

参考答案

脚本如下：

```
[root@oldboy scripts]# cat 19_22_1.sh
#!/bin/sh
#created by oldboy 201308
IPVSADM=/sbin/ipvsadm
VIP=10.0.0.3
PORT=80
RIPS=(
10.0.0.7
10.0.0.8
)
while true
do
    for((i=0;i<${#RIPS[*]};i++))
    do
        PORT_COUNT=`nmap ${RIPS[$i]} -p $PORT|grep open|wc -l`
                                        #<== 检测节点，是否正常。
        if [ $PORT_COUNT -ne 1 ];then #<== 如果节点不通，
            if [ `$IPVSADM -Ln|grep ${RIPS[$i]}|wc -l` -ne 0 ];then
```

```
                                #<== 并且确实没有从池中剔除，
                    $IPVSADM -d -t $VIP:$PORT -r ${RIPS[$i]}:$PORT >/dev/null 2>&1
                                #<== 则删除不正常的节点。
                fi
            else                #<== 如果节点正常。
                if [ `$IPVSADM -Ln|grep ${RIPS[$i]}|wc -l` -eq 0 ];then
                                #<== 如果节点池中没有 RS，
                    $IPVSADM -a -t $VIP:$PORT -r ${RIPS[$i]}:$PORT >/dev/null 2>&1
                                #<== 则添加对应的节点。
                fi
            fi
        done
        sleep 5
    done
```

19.1.23 面试题 23：LVS 客户端配置脚本

请在 LVS 客户端节点上，开发 LVS 客户端设置 VIP 及抑制 ARP 的管理脚本。

实现：/etc/init.d/lvsclient {start|stop|restart}

（1）参考解答

脚本如下：

```
[root@oldboy scripts]# cat 19_23_1.sh
#!/bin/bash
# Written by oldboy
# description: Config realserver lo and apply noarp
RETVAR=0
VIP=(
        10.0.0.3
        10.0.0.4
    )

. /etc/init.d/functions

case "$1" in
    start)
        for ((i=0; i<`echo ${#VIP[*]}`; i++))
        do
            interface="lo:`echo ${VIP[$i]}|awk -F . '{print $4}'`"
            /sbin/ip addr add ${VIP[$i]}/24 dev lo label $interface
            REVTAR=$?
        done
        echo "1" >/proc/sys/net/ipv4/conf/lo/arp_ignore
        echo "2" >/proc/sys/net/ipv4/conf/lo/arp_announce
        echo "1" >/proc/sys/net/ipv4/conf/all/arp_ignore
        echo "2" >/proc/sys/net/ipv4/conf/all/arp_announce
        if [ $RETVAR -eq 0 ];then
                action "Start LVS Config of RearServer." /bin/true
```

```
            else
                action "Start LVS Config of RearServer." /bin/false
            fi
            ;;
        stop)
            for ((i=0; i<`echo ${#VIP[*]}`; i++))
            do
                interface="lo:`echo ${VIP[$i]}|awk -F . '{print $4}'`"
                /sbin/ip addr del ${VIP[$i]}/24 dev lo label $interface >/dev/
null 2>&1
            done
            echo "0" >/proc/sys/net/ipv4/conf/lo/arp_ignore
            echo "0" >/proc/sys/net/ipv4/conf/lo/arp_announce
            echo "0" >/proc/sys/net/ipv4/conf/all/arp_ignore
            echo "0" >/proc/sys/net/ipv4/conf/all/arp_announce
            if [ $RETVAR -eq 0 ];then
                action "Close LVS Config of RearServer." /bin/true
            else
                action "Close LVS Config of RearServer." /bin/false
            fi
            ;;
        *)
            echo "Usage: $0 {start|stop}"
            exit 1
    esac
    exit $RETVAR
```

（2）执行结果

```
[root@oldboy  scripts]# sh 19_23_1.sh start
Start LVS Config of RearServer.                                [ 确定 ]
[root@oldboy scripts]# ifconfig|tail -8
lo:3      Link encap:Local Loopback
          inet addr:10.0.0.3  Mask:255.255.255.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1

lo:4      Link encap:Local Loopback
          inet addr:10.0.0.4  Mask:255.255.255.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1

[root@oldboy scripts]# sh 19_23_1.sh stop
Close LVS Config of RearServer.                                [ 确定 ]
```

19.1.24　面试题 24：模拟 keepalived 软件高可用

请在 LVS 服务端备用节点上，模拟 keepalived vrrp 功能，监听主节点，如果主节点不可访问，则启动备节点并配置 LVS 服务，接管主节点的资源并对用户提供服务（提醒：注意 ARP 缓存），提示此题要借助 19.1.21 的功能。

（1）问题分析

此题实际上要实现两部分功能。

1）监测主节点是否宕机，可以使用 ping 或 nmap 命令。

2）如果主节点宕机，则调用管理 LVS 的服务脚本，执行 LVS 配置。

（2）参考解答

1）监测主节点是否宕机及管理 LVS 的服务脚本如下：

```
[root@oldboy scripts]# cat keepalived_lvs.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
VIP=10.0.0.3
PORT=80
ipvs_tools=`rpm -qa ipvsadm|wc -l`
if [ $ipvs_tools -ne 1 ]
  then
    yum install ipvsadm -y
fi

while true
do
    ping -w2 -c2 ${VIP} >/dev/null 2>&1
    if [ $? -ne 0  ];then
        /bin/sh ./19_21_1.sh start >/dev/null 2>&1
    else
        /bin/sh ./19_21_1.sh stop >/dev/null 2>&1
    fi
    sleep 5
done
```

2）管理 LVS 的服务脚本，见 19.1.21 节。

19.1.25 面试题 25：编写正（或长）方形图形

请用 Shell 或 Python 编写一个正（或长）方形，接收用户输入的数字。

（1）参考解答

方法 1：

```
[root@oldgboy scripts]# cat 19_25_1.sh
#!/bin/bash
read -p "Please Enter a number:" Line
for ((i=1; i<=$Line; i++))
do
    for ((m=1;m<=$(($Line+1));m++))
    do
        echo -n "*"
```

```
    done
    for ((h=1; h<=$(($Line-1)); h++))
    do
        echo -n '*'
    done
    echo
done
```

方法 2：

```
[root@oldboy scripts]# cat 19_25_2.sh
#!/bin/bash
read -p "Please Enter a number:" Line
for ((i=1; i<=$Line; i++))
do
    for ((m=1;m<=$(($Line));m++))
    do
        echo -n "■"
    done
    echo
done
```

（2）执行结果（如图 19-2 所示）

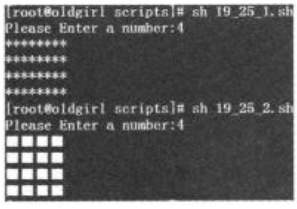


图 19-2　正方形图形输出

19.1.26　面试题 26：编写等腰三角形图形字符

请用 Shell 或 Python 编写一个等腰三角形，接收用户输入的数字。

（1）参考解答

脚本如下：

```
[root@oldboy scripts]# cat 19_26_1.sh
#!/bin/bash
read -p "Please Enter a number:" Line
for ((i=1; i<=Line; i++))
do
    for ((j=$Line-$i; j>0; j--));
    do
        echo -n ' '
    done
```

```
    for ((h=1; h<=$((2*$i-1)); h++))
    do
        echo -n '*'
    done
    echo
done
```

（2）执行结果（如图 19-3 所示）

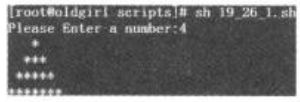


图 19-3　等腰三角形图形输出

19.1.27　面试题 27：编写直角梯形图形字符

请用 Shell 或 Python 编写一个画直角梯形的程序，接收用户输入的参数 n(n>2）和 m。

（1）参考解答

方法 1：

```
[root@oldboy scripts]# cat 19_27_1.sh
#!/bin/sh
if [ $# -ne 2 ];then
    echo $"USAGE:$0 num1(>2) num2"
    exit 1
fi

for n in `seq $1 $2`
do
    for((m=1;m<=$n;m++))
    do
        echo -n "*"
    done
    echo
done
```

方法 2：

```
[root@oldboy scripts]# cat 19_27_2.sh
#!/bin/bash
#!/bin/sh
if [ $# -ne 2 ];then
    echo $"USAGE:$0 num1(>2) num2"
    exit 1
fi

Line_start=$1
Line_end=$2
```

```
#4 6
for ((i=0; i<=$(($2-$1)); i++))
do
    for ((h=1; h<=$(($1+$i)); h++))
    do
        echo -n '*'
    done
    echo
done
```

方法 3：awk 方法（精通 awk，请关注老男孩的第三本图书《Linux 三剑客实战》）

```
    [root@oldboy scripts]# awk   'BEGIN{for(i=ARGV[1];i<=ARGV[2];i++)
for(j=1;j<=i;j++)printf ("%s",j==i?"*\n":"*")}'  4 6
    ****
    *****
    ******
```

（2）执行结果（如图 19-4 所示）

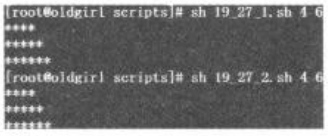


图 19-4　直角梯形图形输出

19.1.28　面试题 28：51CTO 博文爬虫脚本

老男孩教育培训机构需求如下：

请把 http://oldboy.blog.51cto.com 地址中的所有博文，按照时间倒序列表如下：

2013-09-13 运维就是一场没有硝烟的战争

http://oldboy.blog.51cto.com/2561410/1296694

2016-04-17 运维人员写项目方案及推进项目的基本流程思路

http://oldboy.blog.51cto.com/2561410/1764820

附加高级要求：

生成 html 页面，并设置超链接。

结果如下：

http://oldboy.blog.51cto.com/2561410/1862041

（1）问题分析

此题属于一个小的 Shell 爬虫项目实战，需要读者根据需求，分析涉及的网页地址规律，下载所有网页，再分析内容规律，最后过滤出所需要的内容，生成 html 页面。

（2）参考答案

以下脚本来自 31 期的同学王梅西：

```
    #!/bin/sh
    PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
    HTMLFILE=/home/oldboy/html
    HTTP=http://oldboy.blog.51cto.com/all/2561410
    NUM=$(curl $HTTP |awk -F "[ /]" '/页数/ {print $(NF-3)}')
    [ -d $HTMLFILE ]||mkdir $HTMLFILE -p
    echo -e "<b><h1>老男孩51CTO博客文章html整理版</h1></b>\n<b><h3>老男孩教育运维
脱产班31期王梅西出品</h3></b>" >$HTMLFILE/blog_oldboy_$(date +%F).html
    for((i=$NUM;i>0;i--))
    do
        curl $HTTP/page/$i|egrep "<li><span>|<em"|awk '{if(NR%2==0){printf
$0 "\n"}else{printf $0}}'|awk -F '["<>]+' '{print "<a href=\"http://oldboy.
blog.51cto.com"$9"\">",$14,$10,"</a> <br>"}'|sort -n >>$HTMLFILE/blog_
oldboy_$(date +%F).html
    done
```

（3）最终结果

因内容很长，所以此处不再列出，仅提供地址或扫二维码观看。

http://oldboy.blog.51cto.com/2561410/1862041

19.1.29 面试题 29：Nginx 负载节点状态监测

开发通过 Web 界面展示监控 Nginx 代理节点的状态，效果图如图 19-5 所示，当节点宕机时，以红色展示，当节点正常时以绿色展示。

图 19-5 Nginx 负载节点状态监测示例图

（1）问题分析

要想写出此题的脚本，必须要对 Nginx 服务很熟练，此部分内容可参考老男孩已经出版的图书《跟老男孩学 Linux 运维：Web 集群实战》。

（2）参考解答

以下脚本来自老男孩 19 期的学员刘磊：

```
[root@oldboy scripts]# cat 19_30_1.sh
#!/bin/bash
RIPS=(
    10.0.0.7
    10.0.0.8
    )

file_location=/var/html/test.html
[ -e "$file_location" ]||mkdir `dirname $file_location` -p

function web_result {
    rs=`curl -I -s $1|awk 'NR==1{print $2}'`
    return $rs
}

function new_row {
    cat >> $file_location <<eof
    <tr>
    <td bgcolor="$4">$1</td>
    <td bgcolor="$4">$2</td>
    <td bgcolor="$4">$3</td>
    </tr>

eof
}

function auto_html {
    web_result $2
    rs=$?
    if [ $rs -eq 200 ]
      then
        new_row $1 $2 up green
    else
        new_row $1 $2 down red
    fi

}
function main(){

    while true
    do
```

```
    cat >> $file_location <<eof
    <h4>Oldboy Nginx Service Status Of RS :</h4>
    <meta http-equiv="refresh" content="1">
    <table border="1">
    <tr>
    <th>NO:</th>
    <th>IP:</th>
    <th>Status:</th>
    </tr>

eof

    for ((i=0;i<${#RIPS[*]};i++)); do
        auto_html $i ${RIPS[$i]}
    done

    cat >> $file_location <<eof
    </table>
eof

    sleep 2
    > $file_location
    done
}
main $*
```

（3）执行结果

生成如下 html 并且会自动刷新：

```
[root@oldboy scripts]# cat /var/html/test.html
<h4>Oldboy Nginx Service Status Of RS :</h4>
<meta http-equiv="refresh" content="1">
<table border="1">
<tr>
<th>NO:</th>
<th>IP:</th>
<th>Status:</th>
</tr>

<tr>
<td bgcolor="red">0</td>
<td bgcolor="red">10.0.0.7</td>
<td bgcolor="red">down</td>
</tr>

<tr>
<td bgcolor="red">1</td>
<td bgcolor="red">10.0.0.8</td>
<td bgcolor="red">down</td>
```

```
</tr>

</table>
```

将 /var/html 作为站点目录将上述生成的 html 文件放到 HTTP 服务站点目录 /var/html 下面，浏览器访问 http:// 地址 /test.html 即可实现本题的功能，最终结果如图 19-6 和图 19-7 所示：

Oldboy Nginx Service Status Of RS :

NO:	IP:	Status:
0	10.0.0.7	down
1	10.0.0.8	down

图 19-6　宕机状态图

Oldboy Nginx Service Status Of RS :

NO:	IP:	Status:
0	10.0.0.7	up
1	10.0.0.8	up

图 19-7　正常状态图

更多答案可参考 http://oldboy.blog.51cto.com/2561410/1589685。

19.2　Shell 经典程序案例：哄老婆和女孩的神器

19.2.1　功能简介

作为 IT 人员，经常会被人觉得不够浪漫、不懂情趣，其实不然。我们可以用我们的技能创造出 IT 人员独有的浪漫。下面介绍的 girlLove 脚本就可以展现 IT 人员的浪漫。girlLove 本质上是一个简易的问答系统，通过设置不同的问题和答案来实现“浪漫”的效果。读者可通过改写该脚本，轻松地实现一个基于 Linux **终端的调查系统或考试系统**等。

girlLove 脚本可以展示为如下几个部分：文字特效（poetrys）、问题（questions）、问题选项（bakans）、答案（answers）和提示（tips）。这些内容都保存在 Shell 数组（girlLove.txt 文件）中，并且是一一对应的关系，在主程序 girlLove.sh 中通过 while 循环逐个展示出来。以上各部分的具体内容都可以在 girlLove.txt 文件中进行设定，设定的选项数量和用户的屏幕相关，如果读者显示屏幕过小，选项数量多了则有可能产生位置偏移等影响。

除了 girlLove 之外，老男孩也提供了如何利用运维思想追女友的课程（精品免费视频）[⊖]，看过的所有小伙伴都说收获很大。

19.2.2　使用方法

脚本使用方法如下：

⊖ 《跟着老男孩学习如何运用运维思想追到心仪的女朋友》，见 http://edu.51cto.com/course/course_id-5907.html。

```
[root@oldboy ~]# tar xf girlLove.tar.gz        #<== 获取到工具软件包，解压。
[root@oldboy ~]# cd girlLove/
[root@oldboy girlLove]# tree
.
├── girlLove.sh                                #<== 实现脚本
└── girlLove.txt                               #<== 特效及问答模板
0 directories, 2 files
[root@oldboy girlLove]# sh girlLove.sh lili    #<== 结果展示见下文。
```

> 提示：解压之后，也可以编辑 girlLove.txt 设置自己的问题、问题选项、答案和提示，注意 girlLove.txt 内容的字符串格式，也是需要有语法的。

19.2.3 girlLove 工具内容模板

这部分内容以 girlLove.txt 文件呈现，本质上这个文件也是一个 Shell 脚本，因此读者改动时请注意格式，下面是 girlLove.txt 文件的内容展现。

老男孩追求女朋友的问答模板如下：

```
# 老男孩追求女朋友的浪漫模板
# 文字特效 (poetrys)
poetrys=(
"                                    " \
"㊣——^-^o 中华人民共和国 o^-^——㊣  " \
"┃ 谨祝：                          ┃  " \
"┃     $girlname 小盆友 天天开心！          ┃  " \
"┃                    老男孩 烦 ┃" \
"㊣——^-^o 中华人民共和国 o^-^——㊣" \
"                                    " \
"嘟嘟 o o O O●哇靠！！！快让开】" \
"    ┌——┐亲亲！开车罗\` 坐好啊 " \
"  ┌┘五档┃老公！开慢点\` 我兴奋   " \
"  └⊙—⊙┘。o o O 轧死了不赔！" )
# 问题集合 (question)
questions=( "1、坐在你旁边的是你什么人？ " \
            "2、你男朋友老家是哪个地方的？ " \
            "3、你男朋友会做下列哪种饭？ " \
            "4、你最爱看下列哪一个电视剧？ " \
            "5、你男朋友最喜欢吃什么？ " \
            "6、你男朋友休闲的时候最喜欢干什么？ " \
            "7、休闲时你最喜欢他陪你做什么 " \
            "8、今年的生日礼物你最想要啥？ " \
            "9、你计划啥时候和他一起领证？ " \
            "10、领证后你想去哪度蜜月？ " \
            "11、结婚后，你希望财务归谁管？(最后一道题了)")
# 问题选项 (bakans)
bakans=( "A. 男朋友 B. 普通朋友 C. 男闺蜜  D. 以上都不是" \
```

```
            "A. 铁岭 B. 沈阳  C. 四平  D. 以上都不是 " \
            "A. 蛋炒饭  B. 鸡蛋羹 C. 煮米饭  D. 以上都是 " \
            "A. 电视剧  B. 电影  C. 话剧  D. 二人转  " \
            "A. 大葱  B. 白菜  C. 排骨  D. 辣椒  " \
            "A. 打台球  B. 看书  C. 睡觉  D. 听歌  " \
            "A. 看电视剧  B. 逛街  C. 旅游  D. 一起起床  " \
            "A. 钻戒  B. 手机  C. 包  D. 高跟鞋  " \
            "A.6 个月内  B.12 个月内  C.24 个月以内  D. 没想好  " \
            "A. 马尔代夫  B. 巴厘岛  C. 海南三亚  D. 去大城市铁岭  " \
            "A. 老公  B. 老婆  C. 共同管理  D. 没想好  ")

# 问题答案 (answers)
answers=( A A A A A A A A A A)

# 问题提示 (tips)
tips=(    "Dear, 选 A 啊, 妹子啊 555.." \
          "Dear, 选 A 啊, 大城市铁岭, 你该知道的! " \
          "Dear, 选 A 啊, 亲, 蛋炒饭是老男孩拿手的呦! " \
          "Dear, 选 A 啊, 肯定是电视剧, 我比你清楚哦 ..." \
          "Dear, 选 A 啊, 绝对是大葱, 你比我还清楚哦 " \
          "Dear, 选 A 啊, 台球啊, 哥打台球时老帅了 ..." \
          "Dear, 选 A 啊, 我陪你做你最喜欢的 ..." \
          "Dear, 选 A 啊, 钻戒必须的, 而且是 1 克拉的 ..." \
          "Dear, 选 A 啊, 这么好的男朋友, 要抓住啊 ..." \
          "Dear, 选 A 啊, 马尔代夫不是你的愿望么? " \
          "Dear, 选 A 啊, 老公不但会赚钱, 还会理财呦! ")
```

19.2.4　girlLove 工具的 Shell 源码注释

```
#!/bin/sh
# by oldboy training.
# http://oldboy.blog.51cto.com

# 设置老婆、女朋友或潜在女朋友的名字, 用来在终端展示, $1 是执行脚本时传参的她的名字。
girlname="$1"

# stty size 命令的作用是打印用户当前屏幕终端行数和列数 (位置)。
# pos_stdy: 设置脚本内容输出位置为屏幕上下位置 (终端 Y 轴长度) 的 2/3。
pos_stdy="$(($(stty size|cut -d' ' -f1)/3*2))"

# pos_stdx: 设置脚本内容输出位置为屏幕左右位置 (终端 X 轴长度) 的 1/2。
pos_stdx="$(($(stty size|cut -d' ' -f2)/2))"

# total_stdy: 获取系统终端 Y 轴总长度。
total_stdy="$(($(stty size|cut -d' ' -f1)))"

# total_stdx: 获取系统终端 X 轴总长度。
total_stdx="$(($(stty size|cut -d' ' -f2)))"
```

```
logo="本节目为北京老男孩 IT 教育出品，祝天下所有有情人终成眷属！"

# 以下 good 和 decl 两个变量是答题之后送给女朋友的结束语。
good="${girlname}，你太棒了，完美答对！"
decl="这辈子最疯狂的事，就是爱上了你，我会好好爱你的，请让我守候你一辈子！"

# info 变量，是开始答题的提示。
info="亲，$girlname，这是我送给你的最特别的礼物，请选择 A-D 并按下回车开始答题吧。"
head="答题进度："

# 如果 girlLove.txt 文件不存在，则退出程序。
[ -f ./girlLove.txt ]||exit 1

# 读入 girlLove.txt 文件中所设置的变量。
. ./girlLove.txt

# 用户输入帮助函数。
function usage(){
    echo $"Usage:$0 mm_name"
    exit 1
}
# start 函数用于设置答题前的祝福语。
function start(){
    # 设置红色背景，有关颜色和背景等内容请参考第 9 章的相关知识。
    printf "\e[40m"
    # 清屏。
    clear
    # 打印 logo 变量设置的祝福语，持续 2 秒。
    printf  "\r\e[10;30H\E[33m${logo}\E[0m\n"
    sleep 2

    # 继续打印 logo 变量设置的祝福语，闪烁 2 秒。
    printf  "\r\e[10;30H\E[33;5m${logo}\E[0m\n"
    sleep 2

    # 打印黑色背景，清屏，后面就开始正题了。
    printf "\e[40m"
    clear
}

# print_xy 函数用来控制字符串（提示和问答）的打印位置。
# 参数 1（$1）：为要打印的字符串。
# 参数 2（$2）：根据参数 2 来选择不同的位置计算公式，对于不同类型的字符串，位置计算公式不同。
# 参数 3（$3）：用来控制字符在 Y 轴的打印位置。
# 参数 4（$4）：用来控制字符在 X 轴的打印位置。
function print_xy(){
    # 参数个数为 0 即退出。
```

```
        if [ $# -eq 0 ]; then
            return 1
        fi

        # 位置控制辅助数字。
        len=32

        # 如果参数个数小于 2，则设定不同的位置坐标。
        if [ $# -lt 2 ]; then
            pos="\e[${pos_stdy};$((${pos_stdx} - ${len}))H"
        fi
        # 根据传入的第二个参数的不同确定不同的屏幕坐标位置。
        case "$2" in
            -) # 如果第二个参数为 -（减号），则定义如下的坐标位置，读者可以根据需求进行调整。
                pos="\e[$((${pos_stdy} - $3));$((${pos_stdx} - ${len}))H"
                ;;
            +) # 如果第二个参数为 +（加号），则定义如下的坐标位置，读者可以根据需求进行调整。
                pos="\e[$((${pos_stdy} + $3));$((${pos_stdx} - ${len}))H"
                ;;
            lu) # 如果第二个参数为 lu，则定义如下的坐标位置，读者可以根据需求进行调整。
                pos="\e[$((${pos_stdy} - $3));$((${pos_stdx} - $4))H"
                ;;
            ld) # 如果第二个参数为 ld 号，则定义如下的坐标位置，读者可以根据需求进行调整。
                pos="\e[$((${pos_stdy} + $3));$((${pos_stdx} - $4))H"
        esac
        # 打印第一个参数的内容到指定的屏幕位置。
        echo -ne "${pos}$1"
    }
    # 答题结束以后打印一个红色的大背景，带着主人公的表决心信息。
    function waiting(){
        local i=1
        # 通过 while 循环实现 /// 转圈的动画效果。
        while [ $i -gt 0 ]
        do
            for j in '-' '\\' '|' '/'
            do
                # 打印前面若干个 / 特效符号 + decl 变量中的内容。
                echo -ne "\033[1m\033[$pos_stdy;$((${pos_stdx}/3))H$j$j$j\033[4m\
033[32m${decl}"
                # 打印后面若干个 / 特效符号。
                echo -ne "\033[24m\033[?25l$j$j$j"
                # 打印前面若干个 / 特效符号 + good 变量中的内容。
                echo -ne "\033[1m\033[$(($pos_stdy-2));$((${pos_stdx}/3))H$j$j$j\033[4m\
033[32m${good}"
                # 打印后面若干个 / 特效符号。
                echo -ne "\033[24m\033[?25l$j$j$j"
                usleep 100000
```

```
        done
        ((i++))
    done
}
# 设置答题问答相关的内容函数。
function print_info(){
    # 在 1/2 X 轴处减去 32，在 2/3 Y 轴的位置处打印如下字符串（格式化界面）。
    # 即 pos="\e[${pos_stdy};$(($(pos_stdx} - ${len}))H" 变量所在位置。
    print_xy "*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*"
    # 紧接着上面字符串位置打印 $info 信息（具体位置同上，参考 print_xy 函数内容）。
    print_xy "\E[33m${info}\E[0m"  + 1

    # 在终端底部中间位置打印 $info 信息。
    #printf "\r\e[${total_stdy};$((($(total_stdx} - ${#info}*2)/2))H${info}\n"

    # 定义偏移量为数字 20，读者可根据实际显示效果随意调整。
    offset=20

    # 定义数组序列，从 0 开始。
    seq=0

    # 循环问题，为求效果，问题数要比 $poetrys 变量行数多 1。
    while [ ${seq} -lt ${#poetrys[@]} ]
    do
        sleep 0
        isanswers=0

        # 打印问题，传入 4 个参数，读入数组内的问题及传入参数控制字符串在屏幕的位置。
        print_xy "${questions[$seq]}" 1d 2 $offset

        # 打印问题选项，传入 4 个参数，读入数组内的选项及传入参数控制字符串在屏幕的位置。
        print_xy "${bakans[$seq]}" 1d 3 $((${offset}-2))

        # 打印回答栏。
        print_xy " 答: " 1d 4 $offset

        # 读取终端答题者的内容输入到变量 ans。
        read ans

        # 光标上移 3 行，并清除从光标到行尾的内容（清除问题行字符）。
        echo -e "\033[3A\r\033[K"

        # 清除问题选项行字符。
        echo -e "\033[K"

        # 清除回答栏字符。
        echo -e "\033[K"
```

```
            # 如果输入的字母和预设的答案不同，则继续循环该问题，大小写都可以。
            if [ "$ans" != "${answers[$seq]}" -a "`echo $ans|tr a-d A-D`" !=
"${answers[$seq]}" ]; then
                # 打印 -----，格式化界面。----- 下面会显示该问题的 tip。
                print_xy "------------------------------------------------" + 5

                # 显示该问题的 tip
                print_xy "${tips[$seq]}" + 7

                # 显示提示后，等待 3 秒后返回。
                sleep 3

                # 将光标移到行首，并清除光标到行尾的字符。
                echo -e "\r\033[K"

                # 光标上移 3 行，并清除光标到行尾的字符。
                echo -e "\033[3A\r\033[K"
                continue
            fi

            # 问题序号 + 1。
            seq=`expr ${seq} + 1`

            # 获取 poetrys 的倒数第 seq + 1 行。
            curseq=`expr ${#poetrys[@]} - ${seq}`

            # 打印 poetrys 的倒数第 seq + 1 行。
            print_xy "${poetrys[${curseq}]}" lu $seq $offset

            # total 定义左右位置及进度条总长度。
            total=$[${total_stdx} - ${#head}*10]
            # 答题的进度条长度。
            per=$[${seq}*${total}/${#poetrys[@]}]
            shengyu=$[${total} - ${per}]
            # 打印答题进度条。
            printf "\r\e[${total_stdy};19H${head}\e[43m%${per}s\e[41m%${shengyu}
s\e [00m" "" "";
        done
        printf "\r\e[$((${total_stdy}));19H \E[33m    恭喜我心中最美的 ${girlname} 全部
答对 \E[0m";

        # 设置屏幕红色背景。
        printf "\e[41m"
        # 清屏。
        clear
    }
```

```
# 主函数开始调用所有函数执行整个脚本。
function main(){
    if [ $# -ne 1 ]; then
        usage
    fi
    start        #打印答题前的祝福。
    print_info   #打印问答提示及调用问答函数打印更多信息、包括进度条等。
    waiting      #答题结束前的内心表达语，带有特效符号。
}
# 主函数开始执行，$* 为脚本传参，其实就是一个你要追的女孩子的名字。
main $*
```

19.2.5 girlLove 最终结果展示

具体展示结果见下面地址中的动画 http://oldboy.blog.51cto.com/2561410/1864839

特别说明，追女孩神器案例来自于腾讯高级工程师 Fork 的案例拓展。

Linux

第20章

子 Shell 及 Shell 嵌套模式知识应用

前面 19 章所讲的 Shell 内容是老男孩在教学中会重点讲解的内容，但老男孩发现已经毕业参加工作的学生在写复杂的 Shell（多个 Shell 相互调用）时，对 Shell 的执行模式及子 Shell 相关知识还是有些模糊不清，因此老男孩猜想读者在看完前面的章节后可能也会遇到此类问题，因而特意用一章来说明子 Shell 及 Shell 嵌套模式知识应用。

20.1 子 Shell 的知识及实践说明

20.1.1 什么是子 Shell

子 Shell 的本质可以理解为 Shell 的子进程，子进程的概念是由父进程的概念引申而来的，在 Linux 系统中，系统运行的应用程序几乎都是从 init（Pid 为 1 的进程）进程派生而来的，所有这些应用程序都可以视为 init 进程的子进程，而 init 则为它们的父进程，通过执行 pstree -a 命令就可以看到 init 及系统中其他进程的进程树信息：

```
[root@oldgirl ~]# pstree -a
init
  ├─crond
  ├─login
  │   └─bash
  ├─mingetty /dev/tty2
  ├─mingetty /dev/tty3
  ├─rsyslogd -i /var/run/syslogd.pid -c 5
```

```
|    ├─{rsyslogd}
|    ├─{rsyslogd}
|    └─{rsyslogd}
├─sshd
|    └─sshd
|        └─sshd
|            └─bash
|                └─sudo su -
|                    └─su -
|                        └─bash
|                            └─pstree -a    #<== 这就是老男孩刚刚执行的显示
|                                              进程树的命令。
└─udevd -d
    ├─udevd -d
    └─udevd -d
```

对于 Shell 的子进程来说，它是一个从父级 Shell 进程派生而来的新的 Shell 进程，我们将这种新的 Shell 进程称为这个父级 Shell 的子 Shell。

Shell 脚本是从上至下、从左至右依次执行每一行的命令及语句的，即执行完一个命令之后再执行下一个。如果在 Shell 脚本中遇到子脚本（即脚本嵌套），就会先执行子脚本的内容，完成后再返回父脚本继续执行父脚本内后续的命令及语句。

通常情况下，当 Shell 脚本执行时，会向系统内核请求启动一个新的进程，以便在该进程中执行脚本的命令及子 Shell 脚本，基本流程如图 20-1 所示，此部分在第 2 章中已有阐述。

图 20-1　Shell 脚本的基本执行流程

20.1.2　子 Shell 的常见产生途径及特点

常见子 Shell 的产生途径主要包括以下 4 种。

1. 带“&”提交后台作业

下面通过 Shell 脚本来实现一个由“&”产生的子 Shell，脚本如下：

```
[root@oldboy scripts]# cat 20_1_1.sh
#!/bin/bash
# Author:oldboy training
# Blog:http://oldboy.blog.51cto.com
parent_var="Parent"              #<== 定义父 Shell 变量 parent_var 并赋值标志性字符。
echo "Shell Start:ParentShell Level: $BASH_SUBSHELL"
```

```
#<== 输出父 Shell 层级，BASH_SUBSHELL 为系统环境变量
{                                   #<== 通过花括号包裹需要被“&”影响的命令集。
    echo "SubShell Level: $BASH_SUBSHELL" #<== 输出子 Shell 的层级，和父 Shell 层级对比
    sub_var="Sub"                   #<== 定义子 Shell 的变量 sub_var，并赋值标志性字符。
    echo "sub_var=$sub_var"         #<== 输出子 Shell 的变量字符串。
    echo "parent_var=$parent_var"
#<== 在子 Shell 中输出父 Shell 定义的变量，看子 Shell 是否可以从父 Shell 继承变量 parent_
var 值。
    sleep 2                         #<== 休息 2s，用于分辨此处的子 Shell 执行是不是异步的。
    echo "SubShell is over."        #<== 输出标识代表子 Shell 结束。
} &                                 #<== 通过“&”产生子 Shell，执行子 Shell 的同时跳回父 Shell。
echo "Now ParentShell start again."  #<== 子 Shell 代码下面紧接着代码，目的是判断
                                          此子 Shell 是异步执行还是同步执行。
echo "Shell Over:ParentShell Level: $BASH_SUBSHELL"
#<== 脚本即将结束时输出父 Shell 的层级。
if [ -z "$sub_var" ];then           #<== 判断，如果子 Shell 中定义的变量 sub_var 为空，
    echo "sub_var is not defined in ParentShell"
                                    #<== 则说明子 Shell 变量无法被父 Shell 引用。
else
    echo "sub_var is defined in ParentShell "
                                    #<== 否则说明子 Shell 变量可以被父 Shell 引用。
fi
```

执行结果如下：

```
[root@oldboy scripts]# sh 20_1_1.sh
Shell Start:ParentShell Level: 0  #<== 打印父 Shell 层级，层级为 0。
SubShell Level: 1                 #<== 开始执行子 Shell，并打印子 Shell 层级，层级为 1。
sub_var=Sub                       #<== 打印子 Shell 的变量字符串。
parent_var=Parent                 #<== 打印父 Shell 的变量字符串，证明子 Shell 可以
                                       从父 Shell 继承变量 parent_var 值。

Now ParentShell start again.      #<== 父 Shell 继续执行。
Shell Over:ParentShell Level: 0   #<== 脚本即将结束时输出父 Shell 的层级，层级为 0。
sub_var is not defined in ParentShell     #<== 输出此提示，证明子 Shell 变量无法
                                               被父 Shell 引用。
[root@oldboy scripts]# SubShell is over.  #<== 子 Shell 执行结束，因为休息了 3s，所
以子 Shell 最后才输出结束标识，也因此证明加“&”符号的子 Shell 是异步执行的，使得 Shell 脚本具
备了并发执行的功能，这个子 Shell 的应用案例具体可参考 19.1.4 节。
```

根据 20_1_1.sh 执行的输出结果，可以得出以下结论：

- 在 Shell 中使用“&”可以产生子 Shell。
- 由“&”产生的子 Shell 可以直接引用父 Shell 定义的本地变量。
- 由“&”产生的子 Shell 中定义的变量不能被父 Shell 引用。
- 在 shell 中使用“&”可以实现其他程序的多线程并发功能（见 19.1.4 节）。

2. 使用“管道”功能

下面通过脚本再来实现一个由“管道”产生的子 Shell，这个脚本中的绝大部分都

和上一个脚本一样，因此，只对差异的部分做注释：

```
[root@oldboy scripts]# cat 20_1_2.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
parent_var="Parent"
echo "Shell Start:ParentShell Level: $BASH_SUBSHELL"
echo "" |\                              #<== 管道、换行，这是和 20_1_1.sh 的区别之一。
{
    echo "SubShell Level: $BASH_SUBSHELL"
    sub_var="Sub"
    echo "sub_var=$sub_var"
    echo "parent_var=$parent_var"
    sleep 2
    echo "SubShell is over."
}                                     #<== 去掉了 “&” 符号。
sleep 1
echo "Now ParentShell start again."
echo "Shell Over:ParentShell Level: $BASH_SUBSHELL"
if [ -z "$sub_var" ];then
    echo "sub_var is not defined in ParentShell"
else
    echo "sub_var is defined in ParentShell"
fi
```

执行结果如下：

```
[root@oldboy scripts]# sh 20_1_2.sh
Shell Start:ParentShell Level: 0
SubShell Level: 1
sub_var=Sub
parent_var=Parent
SubShell is over.
Now ParentShell start again.
Shell Over:ParentShell Level: 0
sub_var is not defined in ParentShell
```

根据 20_1_2.sh 执行的输出结果，可以得出以下结论：

- 在 Shell 中使用管道可以产生子 Shell。
- 由管道产生的子 Shell 可以直接引用父 Shell 定义的本地变量。
- 由管道产生的子 Shell 中定义的变量不能被父 Shell 引用。
- 由管道产生的子 Shell 不能异步执行，只能在执行完毕后才能返回到父 Shell 环境。

3. 使用 “()” 功能

下面通过脚本实现一个由 “()” 产生的子 Shell，脚本代码如下：

```
[root@oldboy scripts]# cat 20_1_3.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
parent_var="Parent"
echo "Shell Start:ParentShell Level: $BASH_SUBSHELL"
(                                          #<== 使用小括号，将命令集括起来。
    echo "SubShell Level: $BASH_SUBSHELL"
    sub_var="Sub"
    echo "sub_var=$sub_var"
    echo "parent_var=$parent_var"
    sleep 2
    echo "SubShell is over."
    )                                      #<== 小括号结束。
sleep 1
echo "Now ParentShell start again."
echo "Shell Over:ParentShell Level: $BASH_SUBSHELL"
if [ -z "$sub_var" ];then
    echo "sub_var is not defined in ParentShell"
else
    echo "sub_var is defined in ParentShell"
fi
```

执行结果如下：

```
[root@oldboy scripts]# sh 20_1_3.sh
Shell Start:ParentShell Level: 0
SubShell Level: 1
sub_var=Sub
parent_var=Parent
SubShell is over.
Now ParentShell start again.
Shell Over:ParentShell Level: 0
sub_var is not defined in ParentShell
```

根据 20_1_3.sh 执行的输出结果，可以得出以下结论：

- 在 Shell 中使用 () 可以产生子 Shell。
- 由 () 产生的子 Shell 可以直接引用父 Shell 定义的本地变量。
- 由 () 产生的子 Shell 中定义的变量不能被父 Shell 引用。
- 由 () 产生的子 Shell 不能异步执行，只有在执行完毕后才能返回到父 Shell 环境。

4. 通过调用外部 Shell 脚本产生子 Shell

最后再来实现一个由外部 Shell 脚本产生的子 Shell，父脚本代码如下：

```
[root@oldboy scripts]# cat 20_1_4_ParentShell.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
```

```
parent_var="Parent"
export parent_env_var="Parent Env"    #<== 定义父 Shell 环境变量 parent_env_var。
echo "Shell Start:ParentShell Level: $BASH_SUBSHELL"
sh ./20_1_4_SubShell.sh               #<== 执行脚本，脚本内容如下。
sleep 1
echo "Now ParentShell start again."
echo "Shell Over:ParentShell Level: $BASH_SUBSHELL"
if [ -z "$sub_var" ];then
    echo "sub_var is not defined in ParentShell"
else
    echo "sub_var is defined in ParentShell"
fi
```

子 Shell 脚本的代码如下：

```
[root@oldboy scripts]# cat 20_1_4_SubShell.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
echo "SubShell Level: $BASH_SUBSHELL"
sub_var="Sub"
echo "sub_var=$sub_var"
echo "parent_var="$parent_var""
echo "parent_env_var=$parent_env_var"

sleep 2
echo "SubShell is over."
```

执行结果如下：

```
[root@oldboy scripts]# sh 20_1_4_ParentShell.sh
Shell Start:ParentShell Level: 0
SubShell Level: 0
sub_var=Sub
parent_var=                    #<== 子 Shell 无法引用父 Shell 变量，因此等号后面为空。
parent_env_var=Parent Env      #<== 子 Shell 可以引用父 Shell 定义的全局环境变量，
                                    因此等号后面有内容。
SubShell is over.
Now ParentShell start again.
Shell Over:ParentShell Level: 0
sub_var is not defined in ParentShell
```

根据 20_1_4_ParentShell.sh 执行的输出结果，可以得出以下结论：

- 在 Shell 中使用 () 可以产生子 Shell。
- 由 () 产生的子 Shell 可以直接引用父 Shell 定义的本地变量。
- 调用外部 Shell 脚本产生的子 Shell 可以直接引用父 Shell 定义的环境变量。
- 调用外部 Shell 脚本产生的子 Shell 中定义的变量（或者环境变量）不能被父 Shell 引用。

- 调用外部 Shell 脚本产生的子 Shell 不能异步执行，只有执行完毕后才能返回到父 Shell 环境。

20.2　子 Shell 在企业应用中的“坑”

在实际生产场景中，经常会遇到不经意间使用了子 Shell 而又没注意到，因此引发一些不该出现的“坑”的情况。

20.2.1　使用管道与 while 循环时遭遇的“坑”

在实际工作中经常会遇到处理多行文本的需求，在 Shell 中，处理多行文本时使用 while 循环会比较方便。

例如，在 subshell.sh 中创建一个函数，函数里使用 while read line 逐行读取 /etc/passwd 文件的内容，并将每一行的内容逐一赋值给一个数组，最终输出数组下标为 0 的内容。同时如果处理过程中遇到了空行，则立即退出循环，并返回值 1，否则返回默认值 0。

首先来看一下 /etc/passwd 里的内容，可以看到文件里有数行内容，且其中有一行为“空”：

```
[root@oldboy scripts]# head /etc/passwd >/tmp/passwd
[root@oldboy scripts]# sed -i '5i \ ' /tmp/passwd
[root@oldboy scripts]# cat /tmp/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin

lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
```

接下来执行过程的主体实现脚本，这其实就是一个有“坑”的脚本，而你可能会浑然不觉：

```
[root@oldboy scripts]# cat 20_1_5.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
#!/bin/bash

function readPasswd()
```

```
{
    local retval=0                              #<== 定义返回值变量并将其初始化为 0。
    local count=0                               #<== 定义 count 变量，用于数组下标。
    cat /tmp/passwd|while read line             #<== 使用 cat 读取文件，管道用 while 循环，
                                                    每次读取一行。
    do

        array[$count]="$line" #<== 将每次读出的行内容赋值（子 Shell 内）给数组，下标为 count。
        if [ -z "${array[$count]}" ];then       #<== 判断如果数组当前元素内容为空，
            retval=1 && break  #<== 则赋值（子 Shell 内）retval=1，且退出 while 循环。
        fi
        ((count++))                             #<==count 自增 1。

    done
    return $retval                              #<== 以 retval 的值作为函数返回值返回。
}
function main(){
    readPasswd                                  #<== 调用 readPasswd 函数。
    echo "retval = $?"                          #<== 输出函数执行返回值。
    echo "array[0] = ${array[0]}"               #<== 输出 /tmp/passwd 第一行的内容，
                                                    即数组中第一个元素的内容。
}
main                                            #<== 开始执行脚本的内容。
```

执行结果如下：

```
[root@oldboy scripts]# sh 20_1_5.sh
retval = 0                                  #<== 返回值输出为 0。
array[0] =                                  #<== 数组元素内容为空。
```

可以看到，执行结果和预期结果不同，为什么得不到预期的结果呢？

查看 /tmp/passwd 文件的内容，发现其中一行为空，但 readPasswd 函数中对空行做了特殊的判断和处理，即如果发现空行，则将 retval 的值赋为 1，且退出 while 循环，所以 readPasswd 的最终返回值应该为 1，而实际打印输出的 retval 最终值却为 0，且 /tmp/passwd 文件的第一行是非空行，因此，不可能是把空行赋给了第一个数组元素 array[0]，但实际结果是打印 array[0] 时却得到了空行。

那么问题到底出在哪里呢？回过头来再仔细研读一下脚本中的代码，细心的读者就会发现 while read line 那一行开头使用了管道，前面说过，使用管道会产生子 Shell，子 Shell 的特性之一就是在子 Shell 中定义的变量（数组也是变量）无法被父 Shell 引用，即不能被 readPasswd 这个函数所引用，而 array[0]=xxxx 和 retval=1 的赋值均是在子 Shell 中完成的，所以当 while 循环执行完毕退回到函数主体时，retval=1 的赋值随即失效，函数主体继续保持 retval=0 的原始赋值。而 array 数组仅在子 Shell 中定义、赋值过，所以当退回到函数主体这个父 Shell 时，array 数组被父 Shell 认为从未定义过，因此打印输出为空。

20.2.2　解决 while 循环遭遇的“坑”

下面换一种方法，不使用管道来实现和上述脚本同样的功能，脚本代码如下：

```
[root@oldboy scripts]# cat 20_1_6.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
#!/bin/bash

function readPasswd()
{
    local retval=0
    local count=0
    while read line                                    #<== 用 while 循环读取文件，弃用管道。
    do
        array[$count]="$line"
        if [ -z "${array[$count]}" ];then
            retval=1 && break
        fi
        ((count++))
    done</tmp/passwd                                   #<== 使用输入重定向的方法读入文件。
    return $retval
}
function main(){
    readPasswd
    echo "retval = $?"
    echo "array[0] = ${array[0]}"
}
main
```

> 提示：有关 while 循环按行读文件的多种方式，请参考本书的 10.5 节。

执行结果如下：

```
[root@oldboy scripts]# sh 20_1_6.sh                #<== 返回值输出为 1
retval = 1
array[0] = root:x:0:0:root:/root:/bin/bash         #<== 输出了数组中的第一个元素，
                                                       以及文件的第一行。
```

这次改用了输入重定向的方法将 /tmp/passwd 的内容输入给 read，实现对数据源的获取，而这一次的脚本执行结果与上一次截然不同，retval 输出的值为 1，打印输出 array[0] 的内容也和 /tmp/passwd 第一行的内容相同，执行结果和预期结果完全一致。

20.3 Shell 调用脚本的模式说明

随着 Shell 脚本的广泛应用，它所使用的场合也变得多种多样，并且编写脚本的代码量也有所增加，有可能一个脚本的代码量可能会达到上百行甚至更多，且多种功能会被整合在一个脚本中，但是企业里的系统架构往往非常庞大，系统应用程序种类繁多，如果用一个复杂的大脚本控制多个程序，显然效果不太理想，也不符合程序架构解耦的发展趋势。因此，对于每个不同的程序，编写不同的脚本进行控制管理就是运维人员常用的方式了，即通过一个主脚本（父脚本）对所有其他功能性的脚本进行统一调用，这就产生了嵌套脚本（子 Shell 脚本的一种）的概念。

在主脚本中嵌套脚本的方式有很多，常见的为 fork、exec、source 三种模式，这三种调用脚本的方式还是有一定的区别，本节将通过下面的说明让读者更清晰地明白这三种调用脚本模式的不同。

20.3.1 fork 模式调用脚本知识

fork 模式是最普通的脚本调用方式，即直接在父脚本里面用“/bin/sh /directory/script.sh”来调用脚本，或者在命令行中给 script.sh 脚本文件设置执行权限，然后使用 /directory/script.sh 来调用脚本。

使用上述方式调用脚本的时候，和 20.1.2 节中的第 4 点说明一致。系统会开启一个 SubShell（子 Shell）执行调用的脚本，SubShell 执行的时候 ParentShell 还在，SubShell 执行完毕后返回到 ParentShell。最后的结论是 SubShell 可以从 ParentShell 继承环境变量，但是默认情况下 SubShell 中的环境变量不能带回 ParentShell。

就好比图 20-2，在当前 Shell 中直接执行某个脚本后，脚本会进入到此 Shell 的子 Shell 中，完成脚本执行过程，当前的 Shell 称为父 Shell，在子 Shell 中可以调用自身的环境变量信息，并且也可以调用父 Shell 的环境变量信息，但子 Shell 的环境变量信息不能被父 Shell 使用。

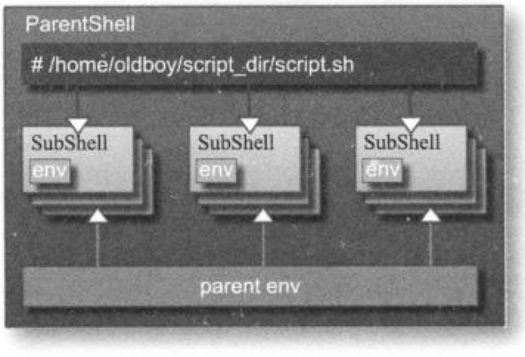


图 20-2 ParentShell 与 SubShell 变量继承关系图

执行方式说明：

```
/directory/script.sh              #<== 对脚本赋予执行权限，直接执行脚本。
/bin/sh /directory/script.sh      #<== 在不赋予执行权限时，利用执行解释器执行脚本。
```

21.3.2 exec 模式调用脚本

exec 模式与 fork 模式调用脚本的方式不同，不需要新开一个 Subshell 来执行被调

用的脚本。被调用的脚本与父脚本在同一个 Shell 内执行，但是使用 exec 调用一个新脚本以后，父脚本中 exec 执行之后的脚本内容就不会再执行了，这就是 exec 和 source 的区别。

执行方式说明：

```
exec /directory/script.sh
```

21.3.3　source 模式调用脚本

source 模式与 fork 模式的区别是不会新开一个 Subshell 来执行被调用的脚本，而是在同一个 Shell 中执行，所以在被调用的脚本中声明的变量和环境变量都可以在主（父）脚本中获取和使用，此部分知识在本书的第 2 章中已有讲解。

source 模式与 exec 模式相比，最大的不同之处是使用 source 调用一个新脚本以后，父脚本中 source 命令行之后的内容在子脚本执行完毕后依然会被执行。

执行方式说明：

```
source /directory/script.sh   #<== 使用 source 不容易被误解，而 "." 和 "./" 相近，
                                   容易被误解。
. /directory/script.sh        #<== "." 和 source 命令的功能是等价的。
```

20.4　Shell 调用脚本的 3 种不同实践方法

20.4.1　开发测试不同模式区别的 Shell 脚本

本节将通过下面的几个脚本来说明 3 种调用方式的不同。编写不同脚本的作用是一个脚本作为父 Shell 脚本，即 ParentShell.sh，另一个脚本为在父 Shell 脚本中执行的嵌入 Shell 脚本（子 Shell 脚本），即 Subshell.sh。

通过在执行过程中选择嵌入 Shell 脚本的不同执行方式，可以看出 fork、exec、source 三种调用执行脚本模式的区别。

1）编写对内容加颜色的函数，放在 /etc/init.d/functions 里。

为了让不同模式的对比更清晰，先写一个对内容加颜色的函数，并放在 /etc/init.d/functions 里，代码如下：

```
[root@oldboy scripts]# grep -A 300 oldboy /etc/init.d/functions
# by oldboy
plus_color(){
    RED_COLOR='\E[1;31m'
    GREEN_COLOR='\E[1;32m'
    YELLOW_COLOR='\E[1;33m'
    BLUE_COLOR='\E[1;36m'
    PINK='\E[1;35m'
    RES='\E[0m'
```

```
    if [ $# -ne 2 ];then
        echo "Usage $0 content {red|yellow|blue|green}"
        exit
    fi
    case "$2" in
        red|RED)
            echo -e  "${RED_COLOR}$1${RES}"
            ;;
        yellow|YELLOW)
            echo -e  "${YELLOW_COLOR}$1${RES}"
            ;;
        green|GREEN)
            echo -e  "${GREEN_COLOR}$1${RES}"
            ;;
        blue|BLUE)
            echo -e  "${BLUE_COLOR}$1${RES}"
            ;;
        pink|PINK)
            echo -e  "${PINK_COLOR}$1${RES}"
            ;;
        *)
            echo "Usage $0 content {red|yellow|blue|green}"
            exit
    esac
}
```

提示：此部分内容在第 9 章已经详细讲解过，此处不再赘述。

2）编写特殊的 ParentShell.sh 脚本，作为执行程序的父（主）脚本，为了让读者看得更清楚，本脚本应用了上文的 plus_color 函数库，使得输出结果更清晰：

```
[root@oldboy scripts]# cat ParentShell.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
. /etc/init.d/functions
function usage(){                                    #<== 定义帮助函数。
    echo "Usage:$0 {exec|source|fork}"
    exit 1
}
function ParentFun(){                                #<== 定义父脚本主要信息的输出。
    plus_color "ParentShell start." red              #<== 打印父脚本开始提示。
    export ParentVar="Parent"                        #<== 在父脚本中定义环境变量。
    echo "PID for ParentShell.sh before "$1":`plus_color "$$" "green"`"
                                                     #<== 打印父脚本 PID。
    echo "ParentShell.sh: \$ParentVar is `plus_color "$ParentVar" "green"`"
                                                     #<== 输出父脚本环境变量信息。
```

```
    case "$1" in                                    #<== 利用 $1 接收不同的模式。
        exec)                                       #<== 如果匹配 exec,
            echo "using exec…"
            exec ./SubShell.sh ;;                   #<== 则通过 exec 调用子 Shell 脚本。
        source)                                     #<== 如果匹配 source,
            echo "using source…"
            source ./SubShell.sh ;;                 #<== 则通过 source 调用子 Shell 脚本。
        fork)                                       #<== 如果匹配 fork,
            echo "using source…"
            /bin/sh ./SubShell.sh ;;                #<== 则通过 sh(fork 模式) 调用子 Shell 脚本。
        *)                                          #<== 如果匹配其他,
            usage                                   #<== 则给出正确使用方法。
    esac
    echo "PID for ParentShell.sh after "$1":`plus_color "$$" "green"`"
                                                    #<== 打印父脚本 PID。
    echo "ParentShell.sh: Get: \$SUB_VAR=`plus_color "$SUB_VAR" "blue"`"
                                                    #<== 打印子脚本变量信息。
    plus_color "ParentShell stop." red              #<== 父脚本结束提示。
}
function main(){ #<== 主函数
    if [ $# -ne 1 ];then
        usage
    fi
    ParentFun $* #<==$* 接收函数外传参 (调用脚本模式字符串), 传给函数内的 $1。
}
main $*          #<==$* 接受脚本传参, 转到函数里的 $* 或 $1, 注意此处的 $* 不要用引号。
```

说明：

- 把 ParentShell.sh 脚本看作父 Shell 脚本，并在父 Shell 脚本中以不同的模式调用 SubShell.sh 脚本。
- 对比父脚本及子脚本执行后的 PID 输出，看三种方式的调用默认嵌套脚本后父子脚本 PID 的区别。
- 观察不同模式下 ParentShell.sh 脚本和 SubShell.sh 脚本之间的变量引用情况。

3）编写脚本 SubShell.sh，这代表子 Shell 脚本，具体如下：

```
[root@oldboy scripts]# cat SubShell.sh
#!/bin/bash
#Author:oldboy training
#Blog:http://oldboy.blog.51cto.com
. /etc/init.d/functions      #<== 加载函数库, 主要是下文使用的 plus_color 函数。
plus_color "SubShell start." yellow   #<== 打印子 Shell 开始提示。
echo "PID for SubShell.sh: `plus_color "$$" "blue"`" #<== 打印子 Shell 的 PID 信息。
echo -e "SubShell.sh get \$ParentVar=`plus_color "$ParentVar" "blue"`"
                                      #<== 打印父 Shell 变量。
```

```
export SUB_VAR="Sub"                    #<== 定义子 Shell 变量。
echo "SubShell.sh: \$SUB_VAR=`plus_color "$SUB_VAR" "blue"`"
                                        #<== 输出子 Shell 变量。
plus_color "SubShell stop." yellow #<== 打印子 Shell 结束信息。
```

说明：

- 在嵌套脚本 SubShell.sh 中，输出脚本执行的 PID 信息，并通过输出变量确认能否调用父脚本中的变量信息。
- 确认嵌套脚本 SubShell.sh 中的环境变量信息是否会被父脚本引用。

20.4.2 对比 fork 模式与 source 模式的区别

执行 20.4.1 节中的测试脚本，并分别传入 fork 及 source 参数后，得出下面 fork 模式与 source 模式的对比信息，如图 20-3 所示。

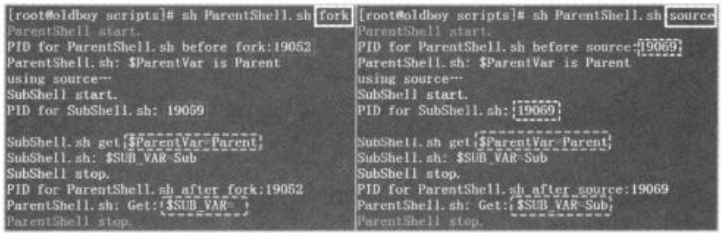


图 20-3 测试脚本传 fork 及 source 参数后输出信息的对比

根据图 20-3 左边的信息，对于 fork 模式执行的输出结果，可以得出以下结论：

- 父脚本执行后的 PID 信息（本例中为 19052）与嵌套脚本（子 Shell 脚本）执行后的 PID 信息（本例中为 19059）不同，说明 fork 模式调用脚本确实产生了子 Shell。
- 父脚本的变量信息会被嵌入的脚本（子 Shell）引用，“SubShell.sh get $ParentVar=Parent”中等号右边的 Parent 即为调用父脚本变量后输出的结果。
- 在嵌入的脚本（子 Shell）中定义的变量信息无法被父脚本引用，“ ParentShell.sh: Get: $SUB_VAR=”中等号右边内容为空，表示没有引用到父脚本的变量。

根据图 20-3 右边的信息，对于 source 模式执行的输出结果，可以得出以下结论：

- 父脚本执行后的 PID 信息（本例中为 19069）与嵌套脚本（子 Shell 脚本）执行后的 PID 信息（本例中为 19069）一致，说明 source 模式调用脚本不会产生子 Shell，而是在同一个 Shell 里执行，此项与 fork 模式不同。
- 父脚本的变量信息会被嵌入的脚本（子 Shell）引用，“SubShell.sh get $ParentVar=Parent”中等号右边的 Parent 即为调用父脚本变量后输出的结果，此项与 fork 模式相同。

- 在嵌入的脚本（子 Shell）中定义的变量信息可以被父脚本引用，“ParentShell.sh: Get: $SUB_VAR=Sub”中等号右边内容为 Sub，表示引用到了子 Shell 脚本中的变量，此项与 fork 模式不同。

20.4.3　对比 exec 模式与 source 模式的区别

同样，执行 20.4.1 节中的测试脚本并分别传入 exec 及 source 参数后，得出下面 exec 模式与 source 模式的对比信息，如图 20-4 所示。

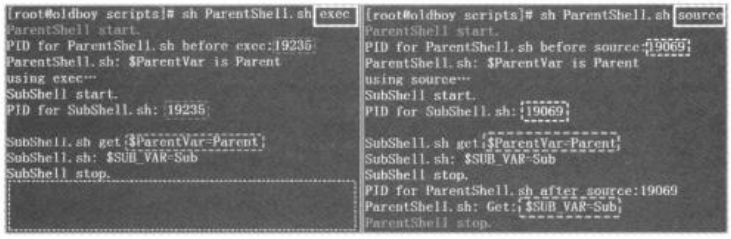


图 20-4　测试脚本传 exec 及 source 参数后的输出信息对比

根据图 20-4 左边的信息，对于 exec 模式执行的输出结果，可以得出以下结论：

- 父脚本执行后的 PID 信息（本例中为 19235）与嵌套脚本（子 Shell 脚本）执行后的 PID 信息（本例中为 19235）一致，说明 exec 模式调用脚本同样不会产生子 Shell，而是在同一个 Shell 里执行，此项与 source 模式相同。
- 父脚本的变量信息会被嵌入的脚本（子 Shell）引用，“SubShell.sh get $ParentVar=Parent”中等号右边的 Parent 即为调用父脚本变量后输出的结果，此项与 fork 及 source 模式都相同。
- 利用 exec 模式执行嵌入脚本（子 Shell 脚本）的问题是，在执行完嵌入脚本后，紧接着嵌入脚本后的所有父脚本命令将不再执行，而是直接退出父脚本，此项与 fork 及 source 模式都不同。

20.5　Shell 调用脚本 3 种不同模式的应用场景

1. fork 模式调用脚本的应用场景

fork 模式调用脚本主要应用于常规嵌套脚本执行的场合，嵌套的脚本只是执行相应的命令操作，不会生成相应的进程信息，父脚本不需要引用嵌套的脚本内的变量及函数等信息，其次在嵌套脚本中定义的变量及函数等不会影响到父脚本中相同的信息定义。

2. exec 模式调用脚本的应用场景

exec 模式调用脚本需要应用于嵌套脚本在主脚本的末尾执行的场合，因此，此种模式的应用并不多见，并且可以被 source 模式完全取代。

3. source 模式调用脚本的应用场景

source 模式调用脚本是比较重要且最常用的一种嵌套方式，主要应用之一是执行嵌套脚本启动某些服务程序。例如：在利用嵌套脚本启动 tomcat 程序并生成 PID 程序文件时，如果选择 fork 模式，那么生成的 PID 文件信息就和执行“ps -ef”命令输出的 PID 信息不一致，这将会导致执行 kill `cat tomcat_pid` 命令时，不能正确关闭 tomcat 程序，而选择 source 模式可以解决此问题。

source 模式调用脚本的另外一个应用就是使得嵌套脚本中的变量及函数等信息被父脚本使用，从而实现更多的业务处理。

附录

Linux 重要命令汇总

线上查询及帮助命令（2 个）

命　令	功能说明
man	查看命令帮助，命令的词典，更复杂的还有 info，但不常用
help	查看 Linux 内置命令的帮助，比如 cd 命令

文件和目录操作命令（18 个）

命　令	功能说明
ls	全拼 list，功能是列出目录的内容及其内容属性信息
cd	全拼 change directory，功能是从当前工作目录切换到指定的工作目录
cp	全拼 copy，其功能为复制文件或目录
find	查找的意思，用于查找目录及目录下的文件
mkdir	全拼 make directories，其功能是创建目录
mv	全拼 move，其功能是移动或重命名文件
pwd	全拼 print working directory，其功能是显示当前工作目录的绝对路径
rename	用于重命名文件
rm	全拼 remove，其功能是删除一个或多个文件或目录
rmdir	全拼 remove empty directories，其功能是删除空目录
touch	创建新的空文件，改变已有文件的时间戳属性
tree	其功能是以树形结构显示目录下的内容
basename	显示文件名或目录名
dirname	显示文件或目录路径

（续）

命　令	功能说明
chattr	改变文件的扩展属性
lsattr	查看文件扩展属性
file	显示文件的类型
md5sum	计算和校验文件的 MD5 值

查看文件及内容处理命令（21 个）

命　令	功能说明
cat	全拼 concatenate，其功能是用于连接多个文件并且打印输出或重定向到指定文件中
tac	tac 是 cat 的反向拼写，因此该命令的功能为反向显示文件内容
more	分页显示文件内容
less	分页显示文件内容，more 命令的相反用法
head	显示文件内容的头部
tail	显示文件内容的尾部
cut	将文件的每一行按指定分隔符分割并输出
split	分割文件为不同的小片段
paste	按行合并文件内容
sort	对文件的文本内容进行排序
uniq	去除重复行
wc	统计文件的行数、单词数或字节数
iconv	转换文件的编码格式
dos2unix	将 DOS 格式文件转换成 UNIX 格式
diff	全拼 difference，比较文件的差异，常用于文本文件
vimdiff	命令行可视化文件比较工具，常用于文本文件
rev	反向输出文件内容
grep/egrep	过滤字符串，三剑客老三
join	按两个文件的相同字段进行合并
tr	替换或删除字符
vi/vim	命令行文本编辑器

文件压缩及解压缩命令（4 个）

命　令	功能说明
tar	打包压缩
unzip	解压文件
gzip	gzip 压缩工具
zip	压缩工具

信息显示命令（11 个）

命　令	功能说明
uname	显示操作系统相关信息的命令
hostname	显示或设置当前系统的主机名
dmesg	显示开机信息，用于诊断系统故障
uptime	显示系统运行时间及负载
stat	显示文件或文件系统的状态
du	计算磁盘空间的使用情况
df	报告文件系统磁盘空间的使用情况
top	实时显示系统资源的使用情况
free	查看系统内存
date	显示与设置系统时间
cal	查看日历等时间信息

搜索文件命令（4 个）

命　令	功能说明
which	查找二进制命令，按环境变量 PATH 路径查找
find	从磁盘遍历查找文件或目录
whereis	查找二进制命令，按环境变量 PATH 路径查找
locate	从数据库（/var/lib/mlocate/mlocate.db）查找命令，使用 updatedb 更新库

用户管理命令（10 个）

命　令	功能说明
useradd	添加用户
usermod	修改系统已经存在的用户属性
userdel	删除用户
groupadd	添加用户组
passwd	修改用户密码
chage	修改用户密码有效期限
id	查看用户的 uid、gid 及其所归属的用户组
su	切换用户身份
visudo	编辑 /etc/sudoers 文件的专属命令
sudo	以另外一个用户身份（默认 root 用户）执行事先在 sudoers 文件中允许的命令

基础网络操作命令（11 个）

命　令	功能说明
telnet	使用 TELNET 协议远程登录
ssh	使用 SSH 加密协议远程登录
scp	全拼为 secure copy，用于在不同主机之间复制文件
wget	命令行下载文件
ping	测试主机之间网络的连通性
route	显示和设置 Linux 系统的路由表
ifconfig	查看、配置、启用或禁用网络接口的命令
ifup	启动网卡
ifdown	关闭网卡
netstat	查看网络状态
ss	查看网络状态

深入网络操作命令（9 个）

命　令	功能说明
nmap	网络扫描命令
lsof	全名为 list open files，即列举系统中已经被打开的文件
mail	发送和接收邮件
mutt	邮件管理命令
nslookup	交互式查询互联网 DNS 服务器的命令
dig	查找 DNS 解析过程
host	查询 DNS 的命令
traceroute	追踪数据传输路由的状况
tcpdump	命令行的抓包工具

有关磁盘与文件系统的命令（16 个）

命　令	功能说明
mount	挂载文件系统
umount	卸载文件系统
fsck	检查并修复 Linux 文件系统
dd	转换或复制文件
dumpe2fs	导出 ext2/ext3/ext4 文件系统信息
dump	ext2/ext3/ext4 文件系统备份工具
fdisk	磁盘分区命令，适用于 2TB 以下的磁盘分区
parted	磁盘分区命令，没有磁盘大小的限制，常用于 2TB 以上的磁盘分区
mkfs	格式化创建 Linux 文件系统

（续）

命　令	功能说明
partprobe	更新内核的硬盘分区表信息
e2fsck	检查 ext2/ext3/ext4 类型文件系统
mkswap	创建 Linux 交换分区
swapon	启用交换分区
swapoff	关闭交换分区
sync	将内存缓冲区内的数据写入磁盘
resize2fs	调整 ext2/ext3/ext4 文件系统的大小

系统及用户权限相关命令（4 个）

命　令	功能说明
chmod	改变文件或目录权限
chown	改变文件或目录的属主和属组
chgrp	更改文件用户组
umask	显示或设置权限掩码

查看系统用户登陆信息的命令（7 个）

命　令	功能说明
whoami	显示当前有效的用户名称，相当于执行 id -un 命令
who	显示目前登录系统的用户信息
w	显示已经登陆系统的用户列表，并显示用户正在执行的指令
last	显示登入系统的用户
lastlog	显示系统中所有用户最近一次登录的信息
users	显示当前登录系统的所有用户的用户列表
finger	查找并显示用户信息

内置命令及其他（19 个）

命　令	功能说明
echo	打印变量，或者直接输出指定的字符串
printf	将结果格式化输出到标准输出
rpm	管理 rpm 包的命令
yum	自动化、简单化地管理 rpm 包的命令
watch	周期性地执行给定的命令，并将命令的输出以全屏的方式显示
alias	设置系统别名
unalias	取消系统别名

（续）

命　令	功能说明
date	查看或设置系统时间
clear	清除屏幕，简称清屏
history	查看命令执行的历史纪录
eject	弹出光驱
time	计算命令执行的时间
nc	功能强大的网络工具
xargs	将标准输入转换成命令行参数
exec	调用并执行指令的命令
export	设置或显示环境变量
unset	删除变量或函数
type	用于判断另外一个命令是否为内置命令
bc	命令行科学计算器

系统管理与性能监视命令（9 个）

命　令	功能说明
chkconfig	管理 Linux 系统开机启动项
vmstat	虚拟内存统计
mpstat	显示各个可用 CPU 的状态统计
iostat	统计系统 IO
sar	全面获取系统的 CPU、运行队列、磁盘 I/O、分页（交换区）、内存、CPU 中断和网络等性能数据
ipcs	用于报告 Linux 中进程间通信设施的状态，显示的信息包括消息列表、共享内存和信号量的信息
ipcrm	用来删除一个或更多的消息队列、信号量集或共享内存标识
strace	用于诊断、调试 Linux 用户空间的跟踪器，也可用于监控用户空间进程和内核的交互，比如系统调用、信号传递、进程状态变更等
ltrace	命令会跟踪进程的库函数调用，并显现出哪个库函数被调用

关机 / 重启 / 注销和查看系统信息的命令（6 个）

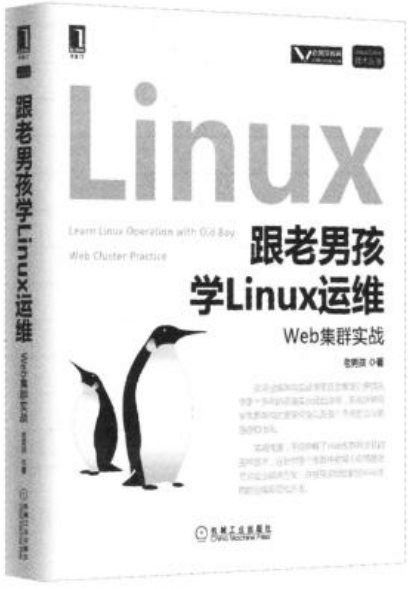
命　令	功能说明
shutdown	关机
halt	关机
poweroff	关闭电源
logout	退出当前登录的 Shell
exit	退出当前登录的 Shell
Ctrl+D	退出当前登录的 Shell 的快捷键

进程管理相关命令（15 个）

命　令	功能说明
bg	将一个在后台暂停的命令变成继续执行（在后台执行）
fg	将后台中的命令调至前台继续运行
jobs	查看当前有多少命令在后台运行
kill	终止进程
killall	通过进程名终止进程
pkill	通过进程名终止进程
crontab	定时任务命令
ps	显示进程的快照
pstree	树形显示进程
nice	调整程序运行的优先级
nohup	忽略挂起信号运行指定的命令
pgrep	查找匹配条件的进程
runlevel	查看系统当前的运行级别
init	切换运行级别
service	启动、停止、重新启动和关闭系统服务，还可以显示所有系统服务的当前状态

有关命令的细致讲解，请关注老男孩后续的新书《跟老男孩学 Linux 运维：Linux 命令实战》一书。

推荐阅读



跟老男孩学linux运维：web集群实战

书号：978-7-111-52983-5 作者：老男孩 定价：99.00元

资深运维架构实战专家及教育培训界顶尖专家十多年的运维实战经验总结，系统讲解网站集群架构的框架模型以及各个节点的企业级搭建和优化

本书不仅讲解了Web集群所涉及的各种技术，还针对整个集群中的每个网络服务节点给出解决方案，并指导你细致掌握Web集群的运维规范和方法，实战性强。

互联网运维涉及的知识面非常广，本书涵盖了构架一个Web网站集群所需要的基础知识，以及常用的Web集群开源软件使用实践。通过本书的实战指导，能够帮助新人很快上手搭建一个完整的Web集群架构网站，并掌握相关的知识点，从而胜任企业的运维工作。