Linux环境及开发工具应用实践

-Linux开发工具2

zhaofang@email.buptsse.cn

软件学院　　赵方

目录

为什么要分解工程

- **编写大的工程：**
- 当用户改动一行代码，编译器需要全部重新编译来生成一个新的可执行文件。但如果项目是分开在几个小文件里，改动其中一个文件的时候，别的源文件的目标文件(object files)已经存在，所以没有什么原因去重新编译它们。

  只要通过基本的规划，将一个项目分解成多个小文件可使你更加容易的找到一段代码。
- **层次更清晰**

怎样分解项目

- i) 不要用一个 header 文件指向多个源码文件（例外：程序包 的 header 文件）。
- ii) 如果可以的话，完全可以用多个 header 文件来指向同一个源码文件。
- iii) 不要在多个 header 文件中重复定义信息。
- iv) 在每一个源码文件里， #include 部分应声明了源码文件对应的所有header 文件。

make

建立工程的基本步骤：把你的源码文件一个一个的编译成目标文件的格式，最后把所有的目标文件加上需要的程序包连接成一个可执行文件。

在大型的开发项目中，通常有几十到上百个的源文件，如果每次均手工键入 gcc 命令进行编译的话，则会非常不方便。因此，人们通常利用 make 工具来自动完成编译工作。

make

- 这些工作包括：
- 如果仅修改了某几个源文件，则只重新编译这几个源文件；
- 如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。利用这种自动编译可大大简化开发工作，避免不必要的重新编译。
- Makefile为项目构建了一个依赖信息数据库，在每次编译前检查是否可以找到所有需要文件。

make

- makefile 文件是许多编译器，包括 Windows NT 下的编译器维护编译信息的常用方法，只是在集成开发环境中，用户通过友好的界面修改 makefile 文件而已。默认情况下，GNU make 工具在当前工作目录中按如下顺序搜索 makefile：
- GNUmakefile、makefile、 Makefile
- 在Linux系统中，习惯使用 Makefile 作为 makfile 文件。

make

- make 工具通过一个称为 makefile 的文件来完成并自动维护编译工作。makefile 需要按照某种语法进行编写，其中说明了如何编译各个源文件并连接生成可执行文件，并定义了源文件之间的依赖关系。当修改了其中某个源文件时，如果其他源文件依赖于该文件，则也要重新编译所有依赖该文件的源文件。

基本 makefile 结构

- Make 的工作是读进文本文件 makefile 。makefile 中一般包含如下内容：
- 需要由 make 工具创建的项目，通常是目标文件和可执行文件。通常使用“目标（target）”一词来表示要创建的项目。
- 要创建的项目依赖于哪些文件。
- 创建每个项目时需要运行的命令。

基本 makefile 结构

- 有了这些信息， make 会检查硬盘上的文件，如果 目的文件的时间戳（该文件生成或被改动时的时间）比至少它的一 个依赖文件旧的话， make 就执行相应的命令，以便更新目的文件

简单Makefile例子

```
#=== makefile 开始 ===
myprog : foo.o bar.o
  gcc foo.o bar.o -o myprog

foo.o : foo.c foo.h bar.h
  gcc -c foo.c -o foo.o

bar.o : bar.c bar.h
  gcc -c bar.c -o bar.o
#=== makefile 结束 ===
```

简单Makefile例子

- Myprog是主要目标（要保 证总是最新）。给出规则说明只要文件myprog 比文件foo.o或bar.o中任何一个旧，下一行命令将执行
- 在检查文件 foo.o 和 bar.o 的时间戳之前，它会往下查 找那些把 foo.o 或 bar.o 做为目标文件的规则。它找到关于 foo.o 的规则，该文件的依靠文件是 foo.c, foo.h 和 bar.h 。 它从下面再找不到生成这些依靠文件的规则，它就开始检查磁盘上这些依赖文件的时间戳。如果这些文件中任何一个的时间戳比 foo.o 的新，命令 'gcc -o foo.o foo.c' 将会执行，从而更新 文件 foo.o 。
  接下来对文件 bar.o 做类似的检查，依赖文件在这里是文件 bar.c 和 bar.h

make 工具建立程序的好处

- **make 检查时间戳：**源码文件里一个简单改变都会造成那个文件被重新编译（因 为 .o 文件依靠 .c 文件），进而可执行文件被重新连接（因为 .o 文件被改变了）。
- **不需再记忆复杂的依赖关系：**当改变一个 header 文件时，不再需要记住哪些源码文件依靠它，因为所有的 资料都在 makefile 里。 make 会很轻松的替重新编译所有那 些因依赖这个 header 文件而改变了的源码文件，如有需要，再进行重新连接。

编写 make 规则 (Rules)

- 最明显的（也是最简单的）编写规则的方法是一个一个的查看源码文件，把它们的目标文件做为目的，而C源码文件和被它 #include 的 header 档做为依靠文件。
- 使用 gcc 的时候，用 -M 开关，它会为每一个给它的C文件输出一个规则，把目标文件做为目的，而这个C文件和所有应该被 #include 的 header 文件将做为依靠文件。注意这个规则会加入所有 header 文件，包 括被角括号(`<', `>')和双引号(`"")所包围的文件。
- 由 gcc 输出的规则不会含有命令部分；可以自己写入命令 或者什么也不写，而让make 使用它的隐含的规则 。

makefile 变量（宏）

- makefile 里的变量就像一个环境变量。事实上，环境变量在 make 过程中被解释成 make 的变量。
- makefile变量大小写敏感，一般使用大写字母。
- 以相同的编译选项同时编译多个 C 源文件时，可以使用变量为每个目标的编译指定编译选项，避免乏味。
- 在 makefile 中引用变量的值时，只需变量名之前添加 $ 符号。

makefile 变量的作用

- **贮存一个文件名列表**。生成可执行文件的规则包含一些目标文件名做为依靠。在这个规则的命令行里同样的那些文件被输送给 gcc 做为命令参数。如果在这里使用一个变数来贮存所有的目标文件名，加入新的目标文件会变的简单而且较不易出错。
- 例如：前面程序可以改为：
- OBJECT= foo.o bar.o
- 使用是加上 $符号。
- $(OBJECT)

makefile 变量的作用

- **贮存可执行文件名**。如果用户的项目被用在一个非 gcc 的系统里，或者如果想使用一个不同的编译器，必须将所有使用编译器的地方改成用新的编译器名。但是如果使用一 个变量来代替编译器名，那么只需要改变一个地方，其它所有地方的命令名就都改变了。

makefile 变量的作用

- 贮存编译器旗标。假设你想给你所有的编译命令传递一组 相同的选项（例如 -Wall - O -g）；如果你把这组选项存入一个变量，那么你可以把这个变量放在所有呼叫编译器 的地方。而当你要改变选项的时候，你只需在一个地方改 变这个变量的内容。要设定一个变量，你只要在一行的开始写下这个变量的名字，后面跟一个 = 号，后面跟你要设定的这个变量的值。以后你要引用这个变量，写一个 $ 符号，后面是围在括号里的变量名。

makefile 变量的作用

```
#=== makefile 开始 ===
OBJS = foo.o bar.o
CC = gcc
CFLAGS = -Wall -O -g
myprog : $(OBJS)
$(CC) $(OBJS) -o myprog
foo.o : foo.c foo.h bar.h
$(CC) $(CFLAGS) -c foo.c -o foo.o
bar.o : bar.c bar.h
$(CC) $(CFLAGS) -c bar.c -o bar.o
#=== makefile 结束 ===
```

make 的主要预定义变量

$*　不包含扩展名的目标文件名称
$+　所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。
$<　第一个依赖文件的名称。
$?　所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。
$@　目标的完整名称。
$^　所有的依赖文件，以空格分开，不包含重复的依赖文件。
$%　如果目标是归档成员，则该变量表示目标的归档成员名称。
AR　归档维护程序的名称，默认值为 ar。
ARFLAGS　归档维护程序的选项。

make 的主要预定义变量

- AS　汇编程序的名称，默认值为 as。
- ASFLAGS　汇编程序的选项。
- CC　C 编译器的名称，默认值为 cc。
- CCFLAGS　C 编译器的选项。
- CPP C 预编译器的名称，默认值为 $(CC) -E。
- CPPFLAGS　C 预编译的选项。
- CXX　C++ 编译器的名称，默认值为g++。
- CXXFLAGS　C++ 编译器的选项。
- FC　FORTRAN 编译器的名称，默认值为f77。
- FFLAGS　FORTRAN 编译器的选项

make 的主要预定义变量

```
#=== makefile 开始 ===
OBJS = foo.o bar.o
CC = gcc
CFLAGS = -Wall -O -g
myprog : $(OBJS)
$(CC) $^ -o $@
foo.o : foo.c foo.h bar.h
$(CC) $(CFLAGS) -c $< -o $@
bar.o : bar.c bar.h
$(CC) $(CFLAGS) -c $< -o $@
#=== makefile 结束 ===
```

makefile 里的函数 (Functions)

- makefile 里的函数跟它的变量很相似--使用的时候，你用一个 $ 符号跟开括号，函数名，空格后跟一列由逗号分隔的参数，最后 用关括号结束。
- 例如‘wildcard’的函 数，它有一个参数，功能是展开成一列所有符合由其参数描述的文件名，文件间以空格间隔。可以使用命令：
  SOURCES = $(wildcard *.c)
  这行会产生一个所有以 ‘.c' 结尾的文件的列表，然后存入变量 SOURCES 里

makefile 里的函数 (Functions)

- 另一个有用的函数是patsubst（ patten substitude, 匹配替换的缩写） 。
- 它有 3 个参数（第一个是一个需要匹配的 式样，第二个是用什么来替换它，第三个是一个需要被处理的 由空格分隔的字列。例如 OBJS = $(patsubst %.c,%.o,$(SOURCES)) 运行将处理所有在 SOURCES 字列中的字，如果它的 结尾是 ‘.c’，就用 ‘.o’ 把 ‘.c’ 取代。注意这里的 % 符号是匹 配一个或多个字符。

make隐含规则

- 定义如何从不同的依赖文件建立特定类型目标。
- 支持两种类型的隐含规则：

后缀规则（Suffix Rule）：Unix系统通常支持一种基于文件扩展名即文件名后缀的隐含规则。这种后缀规则定义了如何将一个具有特定文件名后缀的文件（例如.c文件），转换成为具有另一种文件名后缀的文件（例如.o文件）：

```
.c:.o
$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

make隐含规则

- 系统中默认的常用文件扩展名及其含义为：
  - .o 目标文件
  - .c C源文件
  - .f FORTRAN源文件
  - .s 汇编源文件
  - .y Yacc-C源语法
  - .l Lex源语法

make隐含规则

模式规则（pattern rules）：定义更加复杂的依赖性规则。在目标名称的前面多了一个 % 号，同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

- %.c:%.o
- $(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<

运行 make

- 直接在 make 命令的后面键入目标名可建立指定的目标；直接运行 make，则建立第一个目标。可以用 make -f mymakefile 这样的命令指定 make 使用特定的 makefile，而不是默认的 makefile 或 Makefile
- make 命令其他选项

  -C DIR　在读取 makefile 之前改变到指定的目录 DIR。

  -f FILE　以指定的 FILE 文件作为makefile

运行 make

make 命令其他选项

-h	显示所有的 make 选项
-i	忽略所有的命令执行错误
-I DIR	当包含其他 makefile 文件时，可利用该选项指定搜索目录
-n	只打印要执行的命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	在处理 makefile 前后，显示工作目录
-W FILE	假定文件 FILE 已经被修改

调试make

- -d选项能够使make在执行命令时打印大量的额外调试信息。
- 输出信息如下：
  - 在重新编译时make需要检查的文件
  - 被比较的文件以及比较的结果
  - 需要被重新生成的文件
  - make想要使用的隐式规则
  - make实际使用的隐式规则以及所执行的命令

常见的make出错信息

- No rule to make target ‘target’. Stop
  - makefile中没有包含创建指定的target所需要的规则，而且也没有合适的默认规则可用。
- ‘target’ is up to date
  - 指定target的相关文件没有变化。
- Target ‘target’ not remade because of errors
  - 在编译target时出错，这一消息仅在使用make的-k选项时才会出现。
- Command: Command not found
  - Make 找不到命令（拼写错误或路径不对）。
- Illegal option – option
  - 在调用make时包含了不能被make识别的选项。

有用的makefile目标

- install
  - 把最终的二进制文件，所支持的库文件或shell脚本及相关的文档移动到文件系统中与之相应的最终位置，并设置文件的权限和属主。
  - 编译程序，运行简单的测试
- uninstall
  - 删除由install目标所安装的那些文件。
- dist
  - 用来生成准备发布的软件包。将删除编译工作目录中旧的二进制文件和目标文件，并创建一个归档文件，以便上载。
- tags
  - 创建或更新程序的标记表。
- installtest或installcheck
  - 验证安装过程。

版本控制工具CVS

- 版本控制系统是一个软件系统，它能对计算机软件开发中源代码（当然也不仅仅局限于源代码）的版本进行跟踪、管理和控制，可以记录程序源代码的修改过程并维护一个完整的代码修改和更新的历史记录。
- 使用CVS来管理软件项目，在版本更改时只存储不同版本的diff文件。
- 更适合用于多人协作开发的项目管理。
- CVS的特点：
  - 支持远程访问；
  - 用户可以为他要修改的文件编辑标志。
  - 完成项目中跟踪、协调和管理的工作，以保证多人协作开发软件系统的完整性和安全性。

版本控制工具CVS

- CVS 的基本工作思路是这样的：在一台服务器上建立一个仓库，仓库里可以存放许多不同项目的源程序。由仓库管理员统一管理这些源程序。这样，就好象只有一个人在修改文件一样。避免了冲突。每个用户在使用仓库之前，首先要把仓库里的项目文件下载到本地。用户做的任何修改首先都是在本地进行，然后用 CVS 命令进行提交，由 CVS 仓库管理员统一修改。这样就可以做到跟踪文件变化，冲突控制等等。

版本控制工具CVS

- CVS软件，请找到相关的rpm，tgz，deb 等包装上，或者到 http://www.cvshome.org/CVS/Dev/code
- http://www.cyxlic.com
- http://www.loria.fr/~molli/cvs-index.html 下载原程序编译安装

版本控制工具CVS的使用

- 建立CVS服务器
- 建立CVS项目
- 定义模块
- 版本的分支和合并
- 多人协作开发软件

建立CVS服务器

- 在使用CVS进行版本管理控制前，必须首先建立起CVS服务器。
- 为了CVS系统能够正常运行，则必须保证一定的系统资源，一般需要32M就可以正常工作。

建立CVS服务器

- 1）建立CVSROOT目录：
- #groupadd cvs
- #adduser cvsroot
- 2）用cvsroot用户登陆，修改/home/cvsroot的权限，赋予同组人读写的权限。
- $chmod 777
- 3）建立CVS仓库：
- $cvs -d /home/cvsroot init

建立CVS服务器

- 4）以root身份登陆，修改/etc/inetd.conf和/etc/services：
- 在/etc/inetd.conf里加入：
- cvsserver stream tcp nowait root /usr/bin/cvs cvs –allow-root=/home/cvsroot pserver
- 注意：上面所加入的行是单独一整行， /usr/bin/cvs 是CVS版本的命令路径， /home/cvsroot是所建立的CVSROOT的路径。
- 在/etc/services里加入：
- cvsserver 2401/top

建立CVS服务器

- cvsserver是任意的名称，但是不能和已有服务重名，而且要求和/etc/inetd.conf中所加入行的第一项一致。
- 5）添加可以使用CVS服务的用户到cvs组：
- 以root身份修改/etc/group，把需要使用CVS的用户名加到cvs组里，修改后的/etc/group应该有下面这样一行：
- cvs:x:105:laser,gumpwu

建立CVS服务器

- 6）重启inetd使修改生效：
- #killall -HUP inetd
- 这样服务器设置完成，接下来进行客户端的配置。
- 1）设置环境变量CVSROOT:
- $export CVSROOT=:pservser:laser@the_server-name:/home/cvsroot
- 这里pservser是访问方式，laser是可以使用CVS服务器的用户名， the_server-name是CVS服务器的名称或IP地址， /home/cvsroot是CVS服务器的CVSROOT目录。

建立CVS服务器

- 2）登录CVS服务器：
- 输入$cvs login登录，输入口令，口令和在CVS服务器上的口令一样。成功登录后系统将在用户主目录建立一个.cvspass文件，以后就不必输入口令。

建立CVS项目

- 使用CVS进行版本控制的第一步是是在仓库中建立项目。你可以有几种方法：
- 建立一个目录树
- 从其它版本控制系统建立文件
- 从无到有建立一个目录树

建立CVS项目

- 建立一个目录树
- 　　当你开始使用CVS时，你可能已经有几个项目可以使用CVS进行管理了。在这种情况下，最容易的方法就是使用：　“import”命令。下面通过一个例子来讲解如何使用它的。假定现在有一些想放到CVS中的文件在＂wdir＂中，并且你想让它们放在数据仓库中的如下目录：

＂$CVSROOT/yoyodyne/rdir＂

建立CVS项目

- 可以使用如下命令：

```
$cd  wdir
$cvs  inport  -m  "Inported  Sources"  yoyodyne/
rdir  yoyo  start
```

如果没有使用"-m"参数记录一个日志信息，CVS将调用一个编辑器（*译者注：通常是vi）并且提示输入信息。"yoyo"字符串是开发者标笺，"start"是发行标笺。他们没有什么特别的意义，仅仅是因为CVS的需要才放在这里。

建立CVS项目

- 现在可以检查一下它是否有效了，然后可以删除原来的代码目录。

```
$cd
$mv  dir  dir.orig
$cvs  checkout  yoyodyne/dir
$diff  -r  dir.orig  yoyodyne/dir
$rm  -r  dir.orig.
```

为了避免偶然进入原来的目录中去编辑文件，删除原来的代码是个好主意。当然，在你删除之前保留一份备份到其它地方也是明智之举。

“checkout”命令能使用模块名作为参数（正如前面所有例子）或是一个相对于$CVSROOT的路径，如同上面的例子。应当检查CVS目录中的权限情况是否合适，应当使它们属于某一个特定的组。

- 如果想＂import＂的一些文件是二进制代码，你可以使用一些特殊的方法表明这些文件是否 是二进制文件。

建立CVS项目

- 从其它版本控制系统建立文件
- 如果有一个其它版本控制系统维护的项目，例如RCS，也许希望把这些文件放到CVS中，并且要保留这些文件的历史。
- 从RCS：
  如果你使用RCS，找到RCS文件，通常一个文件名叫"foo.c"的文件会有"RCS/foo.c,v"的RCS文件。（但它有可能在其它地方，请看RCS的文档以得到相关信息）。
  如果文件目录在CVS中不存在，那在CVS中创建它。然后把这些文件拷贝到CVS的仓库目录中（在仓库中的名字必须是带",v"的原文件；这些文件直接放在CVS中的这个目录下，而非"RCS"子目录中）。这是在CVS中一个为数不多的直接进入CVS仓库直接操作的情况，而没使用CVS命令。
  然后就可以把它们在新的目录下"checkout"了。
  当把RCS文件移动CVS中时，RCS文件应在未被锁定的状态，否则移动操作时CVS将会出现一些问题。

建立CVS项目

- 从其它版本控制工具
  从SCCS：
  有一个＂sccsarcs＂的脚本文件可以做把SCCS的文件转化成RCS文件，这个文件放在CVS原代码目录的＂contrib＂目录中。注意：
  你必须在一台同时安装了RCS和SCCS的机器上运行它。并且，正如其它在＂contrib.＂目录中的其它脚本一样。（你的方法也许是变化多端的）
  从PVCS：
  在＂contrb＂中有一个叫＂pves-to-rcs＂的脚本可以转换PVCS到RCS文件。你必须在一台同时有PVCS和RCS的机器上运行它。

建立CVS项目

- 从无到有建立一个目录树
- 建立一个新的项目，最容易的方法是建立一个空的目录树，如下所示：

```
$mkdir  tc
$mkdir  tc/man
$mkdir  tc/testing
```

在这之后，你可以"import"这个（空）目录到仓库中去。

```
$cd  tc
$cvs  import  -m  "created  directory  structure"yoyodyne/
dir  yoyo  start
```

然后，当新增一个文件时，增加文件（或目录）到仓库中。请检查$CVSROOT中的权限是否正确。

定义模块

- 在＂moduyes＂文件中定义模块。这不是严格需要的，但模块能把相关的目录和文件容易关联起来。下面的例子可以充分演示如何定义模块。

1. 得到模块文件的工作拷贝。

```
$cvs checkout CVSROOT/modules
$cd CVSROOT
```

2. 编辑这个文件并写入定义模块的行。你可以使用下面的行定义模块＂tc＂：

```
tc yoyodyne/tc
```

3. 提交你的更改到仓库

```
$cvs commit -m ＂Added tc module.＂modules
```

4. 发行模块文件

```
$cd
$cvs release -d CVSROOT
```

分支与合并

- CVS允许你独立出一个派生的代码到一个分离的开发版本---分支。当你改变一个分支中的文件时，这些更改不会出现在主开发版本和其它分支版本中。

  在这之后你可以使用合并（merging）把这些变更从一个分支移动到另一个分支（或主开发版本）。合并涉及到使用“cvs update -j”命令，合并这些变更到一个工作目录。 你可以确认（commit）这次版本，并且因此影响到拷贝这些变更到其它的分支。

多个开发者同时工作

- 当多个开发者同时参与一个项目时，常常会发生冲突。一般经常发生的情况是两个人想同时编辑一个文件的时候。它的解决方法之一是文件锁定或是使用保留式的checkout，这种方法允许一个文件一次只允许一个人编辑。这是一些版本控制系统的唯一解决方式，包括RCS和SCCS。现在在CVS通常使用保留式checkout的方法是使用＂CVS　admin-1＂命令。下面将讲述可以使用适当的方法来避免两个人同时编辑一个文件，而非使用软件的方式强迫达到这一点。

多个开发者同时工作

- 在CVS中默认的方法是＂unreserved　checkout＂--非保留式的导出。在这种方法下，开发者可以同时在他们的工作拷贝中编辑一个文件。第一个提交工作的没有一种自动的方法可以知道另一个人在编辑文件。另一个人可能会在试图提交时得到一个错误信息。他们必须使用CVS命令使他们的工作拷贝同仓库的内容保持更新。这个操作是自动的。
  CVS可以支持facilitate多种不同的通信机制，而不会强迫去遵守某种规则，如＂resered checkouts＂那样。以下的部分描述这些不同的方式是如何工作的，和选择多种方式之间涉及到的一些问题。

多个开发者同时工作

- 文件状态
  基于你对导出的文件使用过的操作，和这些文件在仓库中的版本使用过的操作，我们可以
  把一个文件分为几个状态。这个状态可以由 " status " 命令得到，它们是：
  up-to-date：
  对于正在使用的这个分支，文件同仓库中的最后版本保持一致。
  Locally　Modified：
  你修改过文件，但没有 " commit " 。

多个开发者同时工作

- Locally added：
  使用了“add”命令增加文件，但没有
  “commit " Locally Removed：
  你使用了 " remove " 命令，但没有 " commit "
  Needs checkout：
  其他人提交了一个更新的版本。这个状态的名字有些误导，你应当使用 " update " 而非 " checkout " 来更新新的版本。

多个开发者同时工作

- Needs Patch：
  象“Needs checkout”一样，但CVS服务将只给出Patch（补丁）文件，而非整个文件。而给出Patch和给出整个文件的效果是相同的。
  Needs Merge：
  一些人提交了一个更新版本，而你却修改过这些文件。
  File had conflicts on merge：
  这同“Locally Modified”相象，只是“update＂命令给出了一个冲突信息。

多个开发者同时工作

- Unkown：
  CVS不知道关于这个文件的情况.例如，你创建了一个新文件，而没有使用＂add＂命令为了帮助弄清楚文件的状态，＂status＂也报告工作版本（working vevision），这是这个文件是从哪个版本来的，另外还报告＂仓库版本＂（Repository vevision）。这是这个文件在仓库中的这个版本分支的最后版本。

多个开发者同时工作

- 你应当把 " update " 和 " status " 命令放在一起来认识。你使用 " update " 使你的文件更新到最新状态，你使用 " status " 命令来得到 " update " 命令将会做何种操作。（当然，仓库中的状态将可能会在你运行update之前变化）。事实上，如果你想使用一个命令得到比使用 " status " 正式的状态信息，你可以使用：

  $cvs -n -q -update

  这里 " -n " 选择项表示不真正执行update，而只显示状态； " -q " 选择项表示不打印每个目录的名字。

多个开发者同时工作

- 使一个文件更新到最版本
  当你想更新或是合并一个，使用update命令。对于一个不是最新版本的文件，这个命令大略等同于 " checkout " 命令：最新版本从仓库中提出并放到工作目录中。
  当你使用 " update " 命令时，你修改过的文件在任何情况下不会受到损害。如果在仓库中没有更新的版本， " update " 时你的代码没有任何影响。当你编辑过一个文件，并且仓库中有更新版本，那么 " update " 将合并所有的变更到你的工作目录。

多个开发者同时工作

- 例如，想象一个你导出了一个1.4版的文件并且开始编辑它，在某一时候其他人提交了1.5版，然后又提交了1.6版，如果你运行update命令，CVS将把1.4版到1.6版之间的变更放到你的文件中。如果在1.4版和1.6版之间的改变太靠近于的你一些变更的话，那么一个"覆盖"（"overlop"）冲突就发生了。在这种情况下将输出一个警告信息，然后结果保留的文件中包含了有冲突代码的两个版本，由特别的符号所分隔开。

版本控制工具CVS

- 建立一个新的CVS项目：

1. 进入到你的已有项目的目录：$cd cvstest
2. 运行命令： $cvs import -m “this is a cvstest project” stest v_0_0_1 start
   说明：import 表示向cvs仓库输入项目文件； -m参数后面的字串是描述文本； cvstest 是项目名称； v_0_0_1是这个分支的总标记；start 是每次 import 标识文件的输入层次的标记。

- 运行下面的命令： $cvs checkout cvstest
- 从仓库中检索出cvstest项目的源文件

版本控制工具CVS

- 如果你已经做过一次checkout了，那么不需要重新checkout，只需要进入cvstest项目的目录，更新一下就行了：
- $cd cvstest
$cvs update

版本控制工具CVS

- 打印出状态报告：
- ==========
  =======File: foo.c Status: Up-to-date
  Working revision: 1.1.1.1 'Some Date'
  Repository revision: 1.2 /home/cvsroot/cvstest/foo.c,v
  Sticky Tag: (none)
  Sticky Date: (none)
  Sticky Options: (none)

版本控制工具CVS

- 这里最重要的就是 Status 栏，可能有四种状态：
  Up-to-date：表明你获得的文件是最新的；
  Locally Modified：表明你曾经修改过该文件，但还没有提交，你的版本比仓库里的新；
  Needing Patch：表明其它人已经修改过该文件并且已经提交了！你的版本比仓库里的旧；
  Needs Merge：表明你曾经修改过该文件，但是别人也修改了这个文件，而且还提交给仓库了！

版本控制工具CVS

- 1，你对某个文件做了修改，比如说改了ceo.c，加了一行程序：printf(“where can I find VC to cheat!”);
  改完之后你要把修改提交给仓库，用命令：
  $cvs commit -m “add a complain” ceo.c
  或者就是：
  $cvs commit -m “worry about money”
  让cvs帮你检查哪个文件需要提交。

版本控制工具CVS

- 2，当开始工作时，可能先做：
  $cvs status

- 如果发现有人改了ceo.c，可以更新：
  $cvs update ceo.c
  或：
  $cvs update
  把ceo.c这个文件更新为最新版本

版本控制工具CVS

- 还有一些命令，比如要增加一个文件garbage_china_concept_stocks_list：
- $cvs add arbage_china_concept_stocks_list
- 然后还需要做：
- $cvs commit garbage_china_concert_stocks_list

软件包安装工具

- 软件安装工具rpm
- RPM 有五个基本的操作 模式(不包括包的编译): 安装，卸载，升级，查询，校验
- 要了解完整的细节和选项，可以使用 **rpm --help**

安装

- 典型的 RPM 有着类似 foo-1.0-1.i386.rpm这样的名称, 其中指明了包名 (foo), 版本号 (1.0), 发行号 (1), 和硬件平台 (i386)。安装一个软件包只需简单的键入以下命令:
- # rpm -ivh foo-1.0-1.i386.rpm
- foo ## #
- RPM 将会打印出软件包的名字(并不一定要与文件名相同 ）, 然后打印出一连串的 # 号以表示安装进度。

安装

- 尽管通常是使用
- **rpm -ivh foo-1.0-1.i386.rpm** 来安装包, 但也可以用
- **rpm -Uvh foo-1.0-1.i386.rpm** 来替代。**-U** 是包升级参数, 也可以用来安装新包。

软件包已被安装

- 如果软件包已被安装，会出现以下信息：
- **# rpm -ivh foo-1.0-1.i386.rpm**
- foo package foo-1.0-1 is already installed error: foo-1.0-1.i386.rpm cannot be installed #
- 如果仍要安装该包，可以在命令行中使用 **--replacepkgs** 选项，这样 RPM 将忽略该错误信息：
- **# rpm -ivh --replacepkgs foo-1.0-1.i386.rpm**
- foo
- ## #

文件冲突

- 如果要安装的软件包中有一个文件已在安装其它包时被安装，会显示以下信息：
- **# rpm -ivh foo-1.0-1.i386.rpm**
- foo /usr/bin/foo conflicts with file from bar-1.0-1 error:
- foo-1.0-1.i386.rpm cannot be installed #
- 要想让RPM 忽略该错误信息，请使用 **--replacefiles** 命令行选项：
- **# rpm -ivh --replacefiles foo-1.0-1.i386.rpm**
- foo ## #

未解决依赖关系

- 一个 RPM 包可能会 "依赖" 其它软件包，也就是说要求在安装了特定的软件包之后才能安装该软件包。 如果在安装这个软件包时未解决这种存在的依赖关系，会看到：
- # rpm -ivh bar-1.0-1.i386.rpm
- failed dependencies:
- foo is needed by bar-1.0-1 #
- 只有先安装完所依赖的软件包，才能解决这个问题。 如果想强制安装 (这不是个好办法，因为安装后的软件包未必能正常运行)，可以使用 **--nodeps** 命令行选项。

卸载

- 卸载软件包：
- # rpm -e foo
- #
- 请注意
- 注意这里使用软件包的 *名字* “foo”，而不是原始软件包的文件名："foo-1.0-1.i386.rpm"。卸载软件包时，需要用原始包的实际文件名替换 foo 包名。
- 在卸载某个软件包时，可能会发生依赖关系错误，这说明其它包与此包之间有安装依赖关系。例如：
- # rpm -e foo
- removing these packages would break dependencies:
- foo is needed by bar-1.0-1
- #
- 要使 RPM 在卸载此包时忽略该错误，使用 **--nodeps** 命令行选项。

升级

- 升级软件包和安装软件包十分类似。
- **# rpm -Uvh foo-2.0-1.i386.rpm**
- foo
- ## # RPM 将自动卸载已安装的老版本的 foo 软件包，用户无法看到有关信息。事实上用户可能总是使用 **-U** 来安装软件包，因为即便以往未安装过该软件包，也能正常运行。
- 因为 RPM 执行智能化的软件包升级，自动处理配置文件，会显示如下信息：
- saving /etc/foo.conf as /etc/foo.conf.rpmsave

升级

- 在使用旧版本的 RPM 软件包来升级新版本的软件时，会产生以下信息：
- **# rpm -Uvh foo-1.0-1.i386.rpm**
- foo package foo-2.0-1 (which is newer) is already installed
- error: foo-1.0-1.i386.rpm cannot be installed
- #
- 要使用 RPM 强行“升级”，请使用 --oldpackage 命令行参数：
- **# rpm -Uvh --oldpackage foo-1.0-1.i386.rpm**
- foo
  ##

更新

- 更新与包升级相似：
- # rpm -Fvh foo-1.2-1.i386.rpm
- foo ##
- #
- RPM 更新选项，检查命令行中指明的包版本与安装在系统中的包版本是否一致。 当 RPM 更新选项处理完已安装包的新版本时，该包会升级到新版本。 但是，RPM 更新选项无法安装系统目前没有的软件包。 这与 RPM 升级不同，升级选项能够安装软件包，无论旧版本的包是否已安装。
- RPM 更新选项可以很好的更新一个软件包或一组软件包。 如果用户下载了大量的软件包，但只想升级系统中已有的包时，RPM 更新选项会非常有用。 只需简单的键入：
- # rpm -Fvh *.rpm
- RPM 工具会自动升级那些已经安装好的包。

查询

- 使用命令 **rpm -q**来查询已安装软件包的数据库。简单的使用命令 **rpm -q foo** 会打印出foo软件包的包名、版本号和发行号:
- # rpm -q foo
- foo-2.0-1
- #

查询选项

- 用户还可以使用以下选项与 **-q** 连用，来指明要查询哪些软件包的信息。
- -a 查询所有已安装的软件包。
- -f *<file>* 将查询包含有文件 *<file>*的软件包。
- -p *<packagefile>* 查询软件包文件名为 *<packagefile>*的包。

查询选项

- 以下选项可以出选择感兴趣的信息，加以显示。
- -i 显示软件包信息，如描述、发行号、大小、编译日期、安装日期、硬件平台、以及其它一些各类信息。
- -l 列出软件包中包含的文件。
- -s 显示软件包中所有文件的状态。
- -d 列出被标注为文档的文件（如，man 手册、 info 信息、README，等等）。
- -c 列出被标注为配置文件的文件。这些文件是需要在安装完毕后加以定制的，如（sendmail.cf, passwd, inittab, 等）。

校验

- 包校验是比较自软件包中安装的文件信息和软件包中的原始文件是否相同信息。与其它校验相同，包校验将比较文件的长度、校验和、许可、类型、文件属主和群组。
- 使用 **rpm -V** 命令进行包校验。用户可以配合使用各个包选择选项，来列出校验包的查询结果。 简单地使用 **rpm -V foo** 可以校验 foo 包中原始安装时的所有文件。

校验

- 例如:
- 校验包含特定文件的软件包:
- **rpm –Vf /bin/vi**
- 校验所有已安装的软件包:
- **rpm -Va**
- 用 RPM 包文件校验已安装的软件包: **rpm -Vp foo-1.0-1.i386.rpm**

Linux常用函数

- Linux系统的库函数
- 网络编程函数

- 编程规范

Thank You !

zhaofang@email.buptsse.cn