

实践 CVS 与 Subversion

作者：ZC Miao<hellwolf@seu.edu.cn>

最后修改时间：2006 年 03 月 22 日 09 时

一 关于本文

CVS 和 Subversion 作为优秀的被广泛使用的版本控制系统(Version Control System)，都有着很好的使用文档，关于 CVS 和 Subversion 的文章也有很多，而本文则希望从一个新的角度，通过实践的方式对照着介绍 CVS 和 Subversion 的用法。如果你以前没有接触过 CVS 和 Subversion，那么希望本文能让你走近它们；如果你已经有相关的基础知识，也希望本文能够让你更好的理解这些知识。但是，本文不是 CVS 和 Subversion 的参考手册，所有介绍都是基于所使用实例的需要，更全面的知识还需要你参考相关的文档来了解。

二 CVS 与 Subversion 简介

在源码版本控制领域方面，不乏优秀的商业软件如 ClearCase 和 SourceSafe，但在开源世界中最著名的版本控制软件无疑是 CVS 了。然而，CVS 有一些与生俱来的缺点，在设计之初没有考虑到，而随着使用量和使用状况的大增，慢慢显露了出来。后来，CVS 的主要作者（Fogel 等等）在他们现任公司的资助下开发了 SubVersion，用以替代 CVS。SubVersion 的设计目的就是针对 CVS 的一些弱点进行改进。目前已经有很多知名的开源项目从 CVS 转向 SubVersion 来保存源代码。Subversion 目前的最新版本 1.3，CVS 的最新版本是 1.11。

三 建立仓库(Repository)

无论是 CVS 还是 Subversion，都需要先建立一个存放项目的地方，这个地方就叫做仓库(Repository)。

在 CVS 中，每个文件夹都称为仓库，其中有个特殊的最顶层的仓库包含一个存放管理文件的 CVSROOT 文件夹，于是称这个特殊仓库为 CVSROOT 仓库。而在 Subversion 中一个仓库则可以包含文件夹和文件，也就是说 Subversion 的仓库维护着一个目录树而 CVS 仓库仅仅维护一个文件夹中的文件，也正因为这个原因 CVS 的许多命令都是递归(recursively)进行的——对一个仓库进行的命令将递归向下到所有的子仓库中。

这里就产生了仓库的粒度、权限设置问题。关于仓库划分的粒度，其可以粗到所有项目都使用同一个仓库，也可以是一个仓库包含几个相关的项目。一般的建议是：使用 CVS 时，对于可以使用同一个权限设置和管理设置的所有项目都存放在同一个 CVSROOT 中；而使用 Subversion 时，对每个项目都创建一个 Repository，这是因为 Subversion 对整个 Repository 使用相同的 Revision Number。权限设置一般的原则就是属主和用户组可读写，其他用户则视情况而定。这样，每个项目创建一个单独的用户组，同一个项目的人就应该加入到了这个用户组中。

下面就具体介绍一下在 CVS 和 Subversion 中如何完成如下任务：建立一个 mo-javascripts 项目所需的仓库，项目用户组为 project_mojavascripts,并添加用户 hellwolf 到该项目组；该项目为公开项目，任何人可访问。

1 建立 CVSROOT 仓库

- 1 建立项目文件夹/cvsroot/mojavescrpts/¹。

```
#mkdir /cvsroot/mojavescrpts
```

- 2 添加一个项目用户组 project_mojavescrpts，并将 hellwolf 添加到该组中。

```
#groupadd project_mojavescrpts
#gpaswd -a hellwolf project_mojavescrpts
Adding user hellwolf to group project_mojavescrpts
```

注意，组信息的更改不会对已经运行的程序有影响，所有为了传播新的组信息，你可能需要重新登录一下你的终端或者重启一下相应的程序²。

- 3 设置文件夹的组和权限

为了让所有在项目文件夹中创建的文件和文件夹都属于 project_mojavescrpts 组，则需要设置文件夹的 group stick bit。

```
#chmod 2775 /cvsroot/mojavescrpts/ #2 等价于g+s
#chgrp project_mojavescrpts /cvsroot/mojavescrpts
#ls -ld /cvsroot/mojavescrpts
drwxrwsr-x 2 root project_mojavescrpts 4096 2006-03-15 17:38 /cvsroot/mojavescrpts
```

- 4 初始化 CVSROOT

```
#cvs -d /cvsroot/mojavescrpts init
#ls -l /cvsroot/mojavescrpts
total 4
drwxrwsr-x 3 root project_mojavescrpts 4096 2006-03-15 17:39 CVSROOT
```

其中-d 用来指定 CVS 根目录(CVSROOT)的位置，该参数可以是本地文件夹名字如本例所示，还可以是 pserver 提供的远程目录，或者是 rsh/ssh 提供的远程目录，详细可以参考 CVS 的手册。

这里的 CVSROOT 管理目录是 project_mojavescrpts 组可读可写的，你可能并不希望这样，那么你可以根据实际情况单独调整这个目录的权限。

这样就建立好了一个 CVSROOT 仓库，现在尝试 checkout 和 checkin。

```
$groups #看看是不是属于project_mojavescrpts 组
hellwolf dialout cdrom floppy audio video plugdev project_mojavescrpts
$mkdir mojavescrpts
$cvs -d /cvsroot/mojavescrpts/ co . #checkout 根仓库的方法
cvs checkout: Updating .
cvs checkout: Updating CVSROOT
U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
...
$echo "#test" > test.txt
$cvs add test.txt
cvs add: scheduling file `test.txt' for addition
cvs add: use `cvs commit' to add this file permanently
$ cvs ci
cvs commit: Examining .
cvs commit: Examining CVSROOT
```

1 不要被迷惑，/cvsroot 不是一个 CVSROOT 仓库，其下的文件夹才是 CVSROOT 仓库，因此或许改名叫 cvs-roots 更合适;)

2 至少我不知道有什么其他好办法，如果你知道，请和大家分享一下

```
/cvsroot/mojavescrpts/test.txt,v <-- test.txt
initial revision: 1.1
```

但是现在换一个非 `project_mojavescrpts` 组的用户 `checkout` 的时候问题却出现了：

```
$ cvs -d /cvsroot/mojavescrpts/ co .
cvs checkout: warning: cannot write to history file
/cvsroot/mojavescrpts/CVSR00T/history: Permission denied
cvs checkout: Updating .
cvs checkout: failed to create lock directory for `/cvsroot/mojavescrpts'
(/cvsroot/mojavescrpts/#cvs.lock): Permission denied
cvs checkout: failed to obtain dir lock in repository `/cvsroot/mojavescrpts'
cvs [checkout aborted]: read lock failed - giving up
```

第一个 **warning** 还可以接受，但是第二个错误就无法忍受了，因为那样将无法得到任何人可以访问的效果，解决办法是修改 `CVSR00T/config` 配置文件，将锁文件换到任何人可写的地方去。

```
$vi CVSR00T/config
[修改 LockDir 一项，比如 LockDir=/var/lock/cvs/mojavescrpts]
$ls -ld /var/lock/cvs/mojavescrpts
drwxrwxrwt  2 root root 4096 2006-03-19 16:19 /var/lock/cvs/mojavescrpts
$cvs ci
cvs commit: Examining .
cvs commit: Examining CVSR00T
/cvsroot/mojavescrpts/CVSR00T/config,v <-- CVSR00T/config
new revision: 1.2; previous revision: 1.1
cvs commit: Rebuilding administrative file database
```

然后再尝试用非 `project_mojavescrpts` 组的用户 `checkout`：

```
cvs -d /cvsroot/mojavescrpts/ co .
cvs checkout: warning: cannot write to history file
/cvsroot/mojavescrpts/CVSR00T/history: Permission denied
cvs checkout: Updating .
U test.txt
cvs checkout: Updating CVSR00T
U CVSR00T/checkoutlist
U CVSR00T/commitinfo
U CVSR00T/config
...
```

现在就已经达到我们的要求了，虽然让任何用户能看见 `CVSR00T` 文件夹不是个好主意，按照前面说的，你可以在这里微调一下 `CVSR00T` 文件夹的权限。

2 建立 Subversion 仓库

1 创建项目文件夹 `/svnroot/mojavescrpts/`

```
#mkdir /svnroot/mojavescrpts/
```

2 添加一个项目用户组 `project_mojavescrpts`，并将 `hellwolf` 添加到该组中。

这一步和 `CVS` 中一样，遂不重复。

3 初始化 Subversion 仓库

```
#svnadmin create --fs-type fsfs /svnroot/mojavescrpts/
```

这里推荐使用 `Subversion` 的新的数据库类型 `fsfs()`，另一种是 `bdb(Berkeley DB)`。

4 重新调整仓库的权限

首先看一下仓库里面都有哪些东西。

```
# ls -F /svnroot/mojavescrpts/
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

表格 1: Subversion 仓库中各文件和文件夹的作用

文件／文件夹	作用
conf/	包含仓库的配置文件
dav/	如果使用 <code>mod_dav_svn</code> ，则用于保存 <code>mod_dav_svn</code> 所需使用的文件
db/	最重要的文件夹，顾名思义这里是存储所有包含版本信息的数据的地方。
format	文件中包含一个整数值，代表当前仓库结构的版本号。目前所使用的是版本 3 的。
hooks/	Subversion 有让每个仓库都保留有自定义行为的能力，这种能力是通过一种叫钩子脚本的东西实现的。这些脚本就存储在这个文件夹中。
locks/	锁文件存放处
README.txt	仓库的说明性文件

从上面也可以看出为了完成任务重要求的效果，我们必须重新设置 **db** 的权限：

```
#chgrp -R project_mojavescrpts /svnroot/mojavescrpts/db/
#chmod -R g+w /svnroot/mojavescrpts/db/
```

这里不需要自己设置组粘滞位(`group sticky bit`)了，因为 `svnadmin create` 已经这么做了。

别忘了还有一个 **locks** 文件夹：

```
#chgrp -R project_mojavescrpts /svnroot/mojavescrpts/locks
#chmod -R g+w /svnroot/mojavescrpts/locks
```

这样就建立了一个 Subversion 的仓库，现在尝试 `checkout` 和 `checkin`：

```
$groups #看看是不是属于project_mojavescrpts 组
hellwolf dialout cdrom floppy audio video plugdev project_mojavescrpts
$svn co file:///svnroot/mojavescrpts/
$cd mojavescrpts
$echo "#test" > test.txt
$svn add test.txt
A      test.txt
$svn ci
Adding      test.txt
Transmitting file data .
Committed revision test.txt.
```

如果你使用一个不属于 `project_mojavescrpts` 组的用户，可以 `checkout`，但是当 `checkin` 的时候会遇到如下错误：

```
svn: Commit failed (details follow):
svn: Can't create directory '/svnroot/mojavescrpts/db/transactions/1-1.txn':
Permission denied
svn: Your commit message was left in a temporary file:
svn:      '/data/home/debian/mojavescrpts/svn-commit.tmp'
```

这正符合我们的要求。

四 版本控制的基本操作

版本控制是项目管理中与人打交道最多的部分，用好版本控制系统则是一个项目成功的基本保证。本部分将仍然以实例的方式介绍用 **Subversion** 和 **cvs** 进行版本控制的基本操作。这次的任务是开发和维护 **mojavescrpts** 项目，**mojave** 是我的一台机器的名字，项目的目的就是维护一组对这台机器进行日常系统管理的脚本。

1 初始化项目

按照 **CVS** 的常规用法，根仓库下面并不直接存放文件，而是首先在其下建立文件夹(**CVS** 术语称其为模块)，建立更仓库下的文件夹的最好方法是用 **import** 命令。比如我们现在需要创建一个 **mirrorutils** 模块，存放镜像用的脚本和配置文件，可以这样：

```
$mkdir mirrorutils
$cd mirrorutils
$cvs -d /cvsroot/mojavescrpts import -m "dummy" mirrorutils dummy-vendor dummy
```

因为 **CVS** 只维护文件的版本信息，所以上面以 **dummy** 开头的信息只是因为 **import** 的语法需要，但不会被记录下来，我们的目的仅仅是创建一个新的模块 **mirrorutils**(由第一个参数指定的名字)罢了。关于 **import** 的详细语法解释在后面[跟踪第三方项目](#)部分中有详细介绍。

而在 **Subversion** 中，则首先需要建立基本的 **Subversion** 目录结构

```
$svn mkdir file:///svnroot/mojavescrpts/trunk \
file:///svnroot/mojavescrpts/tags \
file:///svnroot/mojavescrpts/branches \
-m "basic repository layout"

Committed revision 1.
```

这三个文件夹都是使用 **Subversion** 维护代码的习惯安排方式，分别用于主干代码，标签和分支。在后面会详细介绍这些文件夹的用法。

```
然后和 CVS 部分一样，我们现在在主干代码中创建一个 mirrorutils 文件夹。
$ svn mkdir file:///svnroot/mojavescrpts/trunk/mirrorutils -m "directory to hold
mirror tools and its config files"
$ svn ls file:///svnroot/mojavescrpts/trunk/
mirrorutils/
```

2 导出工作拷贝

前面我们已经创建和初始化了 **mojavescrpts** 的 **CVS** 和 **Subversion** 仓库，为了进行开发，我们首先得获得仓库的一个本地拷贝。在 **CVS** 和 **Subversion** 中都是用的命令 **checkout(co)**。**cvs** 用全局选项 **-d** 或者在 **CVSROOT** 环境变量中指定根仓库的位置。位置可以直接用本地文件夹的路径表示，或者用类似 **:ext:bach@faun.example.org:/usr/local/cvsroot** 这样的参数指定远程文件夹，更多可以参考 **cvs** 手册。**Subversion** 则在 **checkout** 的参数中以 **URI(Uniform Resource Identifier, 统一资源标识符)** 的方式指定 **Subversion** 文件夹的位置，得益于插件化的设计 **Subversion** 理论上可以支持无数种 **URI** 定位模型，目前实际支持的 **URI** 前缀有 **file://**，**http://**，**svn://**，**svn+ssh://**，分别代表本地文件存取，**apache** 的 **mod_dav_svn** 扩展方式存取，**svnserve** 服务器存取和 **ssh** 方式存取。

现在我们面对的情况很简单，仓库是位于本地的，用下面的命令就可以导出 **cvs** 和 **Subversion** 的工作拷贝。

```
$cvs -d /cvsroot/mojavescrpts co -d mirrorutils_cvs mirrorutils
```

```
cvcs checkout: Updating mirrorutils_cvcs
```

cvcs checkout 命令的 **-d** 是指定导出的目录，默认是当前文件夹，第一个参数是需要导出的模块名，在这里是 **mirrorutils**。

```
$svn co file:///svnroot/mojavescrpts/trunk/mirrorutils mirrorutils_svn
Checked out revision 2.
$ ls
mirrorutils_cvcs  mirrorutils_svn
```

Subversion 的命令很简洁，第一个参数指定需要导出的文件夹或文件，其中 **file:///svnroot/mojavescrpts** 是 Subversion 仓库，**trunk** 是其中的文件夹（按照习惯，这是主干代码的意思），**mirrorutils** 是 **trunk** 下的一个文件夹；第二个参数重新命名导出的文件夹，否则你将得到一个 **mirrorutils** 文件夹。

3 开发代码

有了本地的拷贝后，就可以进行代码的开发了。进行的操作无非就是：修改文件，添加文件(夹)，移动文件(夹)，删除文件(夹)。

修改文件没什么好多说的，**fire up your editor;**)

首先是添加文件夹，假设现在我们需要在 **mirrorutils** 中添加一个文件夹 **Conf.d** 用来存放用来做镜像脚本的配置文件。

在 **cvcs** 中首先在本地新建文件夹，然后使用 **add** 命令

```
$mkdir Conf.d
$cvcs add Conf.d
Directory /cvsroot/mojavescrpts/mirrorutils/Conf.d added to the repository
```

在 Subversion 中使用 **mkdir** 命令。

```
$svn mkdir Conf.d
A      Conf.d
```

在 **CVS** 中添加文件夹的操作立即完成了；而在 **Subversion** 中文件夹被标记为 **A**，在下次 **checkin** 时才真正添加。

然后添加一个叫 **mirror.pl** 的脚本。

```
#编写 mirror.pl 脚本并保存
$cvcs add mirror.pl
cvcs add: scheduling file `mirror.pl' for addition
cvcs add: use `cvcs commit' to add this file permanently
$svn add mirror.pl
A      mirror.pl
```

这一次 **CVS** 和 **Subversion** 都没有直接添加文件，需要提交时生效。

现在假设我们要改名 **mirror.pl** 文件为 **mirrors.pl**。在 **cvcs** 中没有直接没有相应的命令，常用的方法如下

```
cvcs$ mv mirror.pl mirrors.pl
cvcs$ cvcs rm mirror.pl
cvcs remove: removed `mirror.pl'
cvcs$ cvcs add mirrors.pl
cvcs add: scheduling file `mirrors.pl' for addition
cvcs add: use `cvcs commit' to add this file permanently
```

在 **cvcs** 中移动文件夹的操作也是这样，但是 **cvcs** 中删除目录的操作有其局限性，在后面会说道。

而在 Subversion 中，则提供了 **mv** 命令，但前提是被移动文件是最新的且没有做过本地修改，这是因为 **mv** 移动的是仓库中的文件而不是本地的文件。因为我们的 **mirror.pl** 还没有提交到仓库，所以还不能运行 **mv** 命令。在 Subversion 中移动文件夹和移动文件是相同的操作过程。

删除文件在 **cvs** 和 Subversion 中都是使用 **rm** 命令。在 **cvs** 中必须先删除本地的文件，或者在 **rm** 中指定 **-f** 选项。在 Subversion 中必须确保本地文件没有做过修改，否则会拒绝删除操作以防止丢失你的工作成果。

至于删除文件夹，在 Subversion 中和删除文件一样操作。而在 **cvs** 中，却无法真正的删除一个文件夹，究其原因，**cvs** 只能维护文件的历史信息而文件夹没有历史信息。所以在 **cvs** 中经常会留下些已经没有任何文件的空文件夹——如果你觉得这些空文件夹难看，你可以在 **checkout** 或者 **update** 的时候指定 **-P** 选项，意思是自动剪裁(**prune**)掉本地拷贝中空的仓库，但在仓库中文件夹还是存在的。

4 复查和撤销更改

在 Subversion 中可用 **svn status** 和 **svn diff** 命令

```
$svn status
A      mirror.pl
$svn diff | head
Index: mirror.pl
=====
--- mirror.pl      (revision 0)
+++ mirror.pl      (revision 0)
@@ -0,0 +1,201 @@
+# (C) 2006 ZC Miao (hellwolf@seu.edu.cn)
+# $Id$
+# Summary : handy script to make mirrors.
+# $Author$
+# $Date$
```

在 **cvs** 中也可用 **cvs diff** 可以查看所作的修改，但是要达到 **svn status** 的效果，则必须运行 **cvs update**，**cvs update** 将在下面介绍。

如果你对你所作的更改不满意，那么可以撤销更改回到修改前的版本(并不一定是仓库中的最新版本)。在 **CVS** 中需要中，可以先删除该文件，然后用 **update** 命令找回该文件；在 Subversion 中也可以使用这种方法，不过更优雅的方案是使用 **svn revert** 命令。

```
$echo bomb! > fedora-core-5-updates.conf
$svn revert fedora-core-5-updates.conf
Reverted 'fedora-core-5-updates.conf'
$head -n1 fedora-core-5-updates.conf
# Summary : Configuration for fedora core 5 updates
```

在编辑过程中，你也可能会发现需要一个更早以前版本的文件了。在 **CVS** 中需要通过一个比较 **tricky** 的方法：

```
$cvs update -p -r 1.1 file1 >file1
=====
Checking out file1
RCS:  /tmp/cvs-sanity/cvsroot/first-dir/Attic/file1,v
VERS: 1.1
*****
```

-r 指定需要导出的文件的版本号或者标签号，**-p** 是将文件内容导出到标准输出，而我们将它导出到 **file1** 文件中。

在 Subversion 中，方法要依然要优雅些：

```
$svn cp --revision 3 \  
svn+ssh://mojave/svnroot/mojavescrpts/trunk/mirrorutils/Conf.d ./
```

该命令将 **revision 3** 版本的主干中的 **Conf.d** 拷贝到当前目录中。

5 同步合作者的修改

如果有多个开发者同时开发的话，你必须在提交修改前和仓库同步一下，已合并来自他人的修改。在 **CVS** 和 **Subversion** 中这条命令都叫 **update**。

在 **CVS** 中，如果仓库中有新的文件夹被添加了，为了得到新的文件夹，还需要使用 **update** 的 **-d** 选项。

```
$cvs update -d  
cvs update: Updating .  
A mirror.pl  
cvs update: Updating Conf.d  
U Conf.d/fedora-core-5-updates.conf
```

cvs update 的输出中，第一个字符代表的是文件的状态，分别是：

U 该文件在仓库中被他人修改（或新建），现在本地拷贝已更新到最新文件

P 类似于 **U**，不过是用补丁的方式，通常是为了减少产生的网络带宽

A 该文件需要被添加进仓库中

R 该文件需要从仓库中删除

M 有两种意思，一种是 **Modified**，代表本地文件有过修改，另一种是 **merged**，代表本地已经成功合并仓库中文件的更改。如果是后者，**cvs** 会输出一些额外的信息，并且会备份你的文件。

C 合并过程发生了冲突，后面将介绍如何处理这种情况

? 该文件不属于 **cvs** 管辖范围

```
$svn update  
A Conf.d  
A Conf.d/fedora-core-5-updates.conf  
Updated to revision 5.
```

Subversion 的输出也是在开头显示文件状态，

A 文件(夹)被添加进本地拷贝（和 **CVS** 的 **A** 不同，请注意）

D 文件(夹)从本地拷贝中删除

U 从仓库中更新了最新的文件版本

C 仓库中的文件和本地同时做了修改，但是两个修改无法合并

G 仓库中的文件和本地同时做了修改，两个修改被成功合并

6 处理冲突

为了模拟冲突处理的情景，我对刚才 **update** 得到的 **Conf.d/fedora-core-5-updates.conf** 文件进行一些人为的修改，然后假设另一个人也对这个文件进行了一些更改，而这两个更改是无法合并的。只要对方提交了更改，那么我在 **update** 的时候就会得到冲突的通知。在 **CVS** 中：

```
$cvs update  
cvs update: Updating .  
RCS file: /cvsroot/mojavescrpts/mirrorutils/Conf.d/fedora-core-5-  
updates.conf,v retrieving revision 1.1  
retrieving revision 1.2
```



```
Merging differences between 1.1 and 1.2 into fedora-core-5-updates.conf
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in fedora-core-5-updates.conf
C fedora-core-5-updates.conf
```

在 Subversion 中：

```
$svn update
C fedora-core-5-updates.conf
Updated to revision 6.
$ls
fedora-core-5-updates.conf      fedora-core-5-updates.conf.r5
fedora-core-5-updates.conf.mine fedora-core-5-updates.conf.r6
```

解决 CVS 中的冲突的过程是，打开有冲突的文件，寻找类似这样的段落：

```
<<<<<< fedora-core-5-updates.conf
                protocol    => 'ftp',
=====
                protocol    => 'https',
>>>>>> 1.2
```

其中上半部分是本地的修改，下半部分是仓库中的最新更改。然后通过日志寻找下面部分的修改人

```
$cvs log fedora-core-5-updates.conf
...
-----
revision 1.2
date: 2006-03-22 09:15:10 +0000;  author: jacky;  state: Exp;  lines: +4 -4
http to https
-----
...
```

联系该修改人，沟通成功后，保存你们的商议结果，然后就可以提交了。如果修改过程中你需要重新回顾你的修改而不想有那些符号的干扰，可以找类似 `#fedora-core-5-updates.conf.1.1` 这样的文件名的文件。记住，CVS 解决冲突的机制就是，一旦冲突文件的最后修改时间被更新，那么 CVS 就认为你已经解决了冲突。

解决 Subversion 中的冲突的过程是，打开冲突文件，依然和可以找到 CVS 中那样格式的冲突段落。还是使用 `svn log`，沟通对方后，修改文件。Subversion 的设计可谓考虑周到，为了防止你搞的头昏，还附带了好多辅助文件，文件名都是冲突文件名加后缀的形式。比如 `.mine` 文件是自己的修改版本，`.r5` 是修定号为 5 的文件。在你确定解决了冲突之后和 CVS 不同，此时提交 Subversion 会报错

```
$ svn ci fedora-core-5-updates.conf
svn: Commit failed (details follow):
svn: Aborting commit: '/data/home/hellwolf/tmp/mirrorutils_svn/Conf.d/fedora-core-5-updates.conf' remains in conflict
```

此时你需要显式的告诉 Subversion 你已经解决了该冲突

```
$ svn resolved fedora-core-5-updates.conf
Resolved conflicted state of 'fedora-core-5-updates.conf'
$ ls
fedora-core-5-updates.conf
```

这样冲突就解决了。

7 提交修改

提交修改很简单，用 `cvs ci` 和 `svn ci` 命令就可以了。如果没有用 `-m` 参数指定日志内容，

那么会打开一个编辑器让你输入日志。可以通过 `EDITOR` 环境变量告诉 `CVS` 和 `Subversion` 你所喜欢的编辑器。

8 给代码打标签(tag)

在经过了一段时间的开发后，项目到达了一个里程碑阶段，你可能想记录这一阶段的代码的状态，那么你就需要给代码打上标签。

在 `CVS` 中使用 `cvs tag` 命令：

```
$cvs tag mirrorutils_rel_0_0_1
cvs tag: Tagging .
T mirror.pl
cvs tag: Tagging Conf.d
T Conf.d/fedora-core-5-updates.conf
```

这份本地拷贝的标签就叫 `mirrorutils_rel_0_0_1`，标签名不能以数字开头以和版本号有索区别。今后如果你想导出这个阶段的代码，那么只要在 `co` 命令和 `update` 命令中使用 `-r mirrorutils_rel_0_0_1` 参数：

```
$cvs -d /cvsroot/mojavescripts/ co -r mirrorutils_rel_0_0_1 \
-d mirrorutils_rel_0_0_1 mirrorutils
cvs checkout: Updating mirrorutils_rel_0_0_1
U mirrorutils_rel_0_0_1/mirror.pl
cvs checkout: Updating mirrorutils_rel_0_0_1/Conf.d
U mirrorutils_rel_0_0_1/Conf.d/fedora-core-5-updates.conf
```

在 `Subversion` 中，则采用的是拷贝文件夹的方式，还记得我们在[初始化项目](#)一节介绍的 `Subversion` 的基本目录结构吗？是的，这时候 `tags` 文件夹就要派上用场了：

```
$svn cp file:///svnroot/mojavescripts/trunk \
file:///svnroot/mojavescripts/tags/mirrorutils_rel_0_0_1 \
-m "tagged mirrorutils_rel_0_0_1"
Committed revision 7.
```

`-m` 指定日志信息。是的就这么简单。相信导出具有特定 `tag` 的版本的代码的方法你也就自然会了。

9 分支和合并分支(branch)

在项目进程中，一般有两种情况下需要分支。

一种称为发布分支(**Release Branch**)，比如上面的 `mirrorutils` 发布了 `0.0.1` 的代码，标签为 `mirrorutils_rel_0_0_1`，收到了很多用户的反馈，这其中就有很多的 `Bug` 报告，为了不影响主干代码的开发进程，需要一个专门的 `0.0.1` 的分支用于修正 `0.0.1` 的 `Bug`。而且这种修改很多还需要合并到主干代码中。

还有一种称为特性分支(**Feature Branch**)，比如有人想对 `mirrorutils` 进行规模相对较大的改造，这种改造对主干代码会有很大的影响，那么必须将这个开发和主干开发的过程用分支的方式并行起来，等到这个分支成熟后再合并到主干中。

那么这里就需要两种操作：分支和合并分支。

首先介绍在 `CVS` 中方法。

`cvs` 用 `cvs tag -b` 来打分支标签。继续上面的例子，为了从 `mirrorutils_rel_0_0_1` 中分出一个 **Release Branch**，我们需要首先导出一份标记为 `mirrorutils_rel_0_0_1` 的代码。记住即使你的本地拷贝和 `mirrorutils_rel_0_0_1` 的代码完全一致，你也需要让你的代码的标记是

mirrorutils_rel_0_0_1 的。前面我们已经导出该版本代码到了 mirrorutils_rel_0_0_1 文件夹中，那么我们现在只需要对其打上分支标签就好了：

```
$cd mirrorutils_rel_0_0_1/
$ cvs tag -b mirrorutils_rel_0_0_1_branch
cvs tag: Tagging .
T mirror.pl
cvs tag: Tagging Conf.d
T Conf.d/fedora-core-5-updates.conf
```

然后用 cvs update 让本地拷贝进入到 mirrorutils_rel_0_0_1_branch 分支中：

```
$ cvs update -r mirrorutils_rel_0_0_1_branch
cvs update: Updating .
cvs update: Updating Conf.d
$ cvs status -v mirror.pl
=====
File: mirror.pl      Status: Up-to-date

Working revision:    1.1      Wed Mar 22 09:51:39 2006
Repository revision: 1.1      /cvsroot/mojavescripts/mirrorutils/mirror.pl,v
Sticky Tag:          mirrorutils_rel_0_0_1_branch (branch: 1.1.2)
Sticky Date:         (none)
Sticky Options:      (none)

Existing Tags:
    mirrorutils_rel_0_0_1_branch    (branch: 1.1.2)
    mirrorutils_rel_0_0_1          (revision: 1.1)
```

Sticky Tag 说明今后该本地拷贝中的修改都将进入到 mirrorutils_rel_0_0_1_branch 分支而不是主干中。

在 Subversion 中，这一过程还是通过拷贝文件夹来完成的，这次用到了 branches 文件夹：

```
$svn cp file:///svnroot/mojavescripts/tags/mirrorutils_rel_0_0_1 \
file:///svnroot/mojavescripts/branches/mirrorutils_rel_0_0_1 \
-m "release branch for mirrorutils_rel_0_0_1"

Committed revision 8.
$svn ls file:///svnroot/mojavescripts/branches/
mirrorutils_rel_0_0_1/
```

经过一段时间后，0.0.1 的代码修复了很多 Bug，而主干代码将吸收这些 Bug 的修复，这里就要用到分支合并的技术了。关于分支合并在[跟踪第三方项目](#)中将通过实例化的方式详细将你介绍。

五 跟踪第三方项目

在自由软件世界里，代码随处不在，即便不是程序员也常常要对得到的代码进行修修补补以适应自己的需求。但经常出现的一个矛盾是：自由软件的更新频繁，每次更新后如何保存以前的更改呢？这就需要版本控制系统的帮助了，而 CVS 和 Subversion 则有一套很成熟的“定式”应对这种需求。下面就分别介绍 CVS 和 Subversion 跟踪项目的方法。

1 用 CVS 跟踪第三方项目

你可以专门建立一个 CVSROOT 仓库用来跟踪你所关注的项目，比如我有个 CVSROOT 仓库为/data/share/software/cvsroot/mybranch，以后只要我需要对什么新软件进行一些跟踪和修改的时候，我就把该软件作为一个新仓库存放到我的 mybranch CVSROOT 仓库中。

今后每次该软件有新的更新，就导入新的代码，并且 CVS 会尽力合并你所作的更改，如果合并失败还会提醒你该合并不能自动进行（CVS 术语为 **conflict**）。下面以 **gftp** 为例，说明用 CVS 跟踪项目的过程。

首先，设置 **CVSROOT** 环境变量，这样可以在 **cvs** 命令中省去使用 **-d** 指定 **CVSROOT** 的部分。

```
$export CVSROOT=/data/share/softwares/cvsroot/mybranch
```

然后导入 **gftp 2.0.17** 的代码。这里涉及到 **cvs import** 命令，该命令的一般格式为：**cvs import [-m msg] repository vendor-tag release-tags**。其中，**-m** 为日志信息，如果不指定，**cvs** 会打开 **vi** 或者 **\$EDITOR** 指定的编辑器要求你输入相应的日志信息；**repository** 为 **CVSROOT** 仓库下的仓库名称，如果不存在则由 **CVS** 为你新建，你甚至用表示当前目录的小数点符号“.”作为 **repository** 名字，这样文件就被直接存放在 **CVSROOT** 仓库下；**vendor-tag** 用来标识你所取的代码的来源，这里被命名为 **gftp_devel**，表示导入的是来自 **gftp** 的官方代码；**release-tags** 使用 **rel_2_0_17**，代表 **release 2.0.17**。

```
$tar jxvf /data/share/softwares/class/net/gftp/gftp-2.0.17.tar.bz2
$cd gftp-2.0.17
$cvs import -m "import src from 2.0.17" gftp gftp_devel rel_2_0_17
$ls $CVSROOT/gftp/
ABOUT-NLS,v      config.h.in,v     docs              Makefile.am,v    README,v
acinclude.m4,v    config.rpath,v    gftp.spec.in,v    Makefile.in,v    src
aclocal.m4,v      config.sub,v      gftp.spec,v       missing,v         stamp-h.in,v
AUTHORS,v         configure.in,v    install-sh,v      mkinstalldirs,v  THANKS,v

ChangeLog-old,v   configure,v       INSTALL,v         NEWS,v           TODO,v
ChangeLog,v       COPYING,v        intl             po
config.guess,v    debian          lib              README.html,v
$cd ..
$rm -rf gftp-2.0.17
```

现在你就可以 **checkout** 了，进入你的工作目录，**hack it now!**在这里，为了演示需要，我在 **About** 中添加一个标签 **Cvsnow**，并且在 **ChangeLog** 里面做相应的记录。

```
$cd ~/tmp
$cvs -d $CVSROOT co gftp
```

虽然设置过了 **\$CVSROOT**，但是这里还是指定一下，以让导出的 **CVS** 仓库记住自己的位置，以后无论什么时候只要在 **gftp** 目录里就都可以不指定 **\$CVSROOT** 变量了(你看看 **CVS/Repository** 文件的内容就明白了)。

```
$cd gftp
$ls
ABOUT-NLS      config.h.in      debian          lib             README
acinclude.m4    config.rpath     docs           Makefile.am     README.html
aclocal.m4      config.sub       gftp.spec      Makefile.in     src
AUTHORS         configure        gftp.spec.in   missing         stamp-h.in
ChangeLog       configure.in     INSTALL        mkinstalldirs  THANKS
ChangeLog-old   COPYING         install-sh     NEWS           TODO
config.guess    cvs             intl           po
$vi src/gtk/menu-items.c
[...编写添加一个 about 对话框标签项的代码并保存退出...]
$./configure --prefix=$HOME/bin/root/;make;make install
$~/bin/root/bin/gftp
```

运行后，点击 **Help->About**，出现如下对话框：

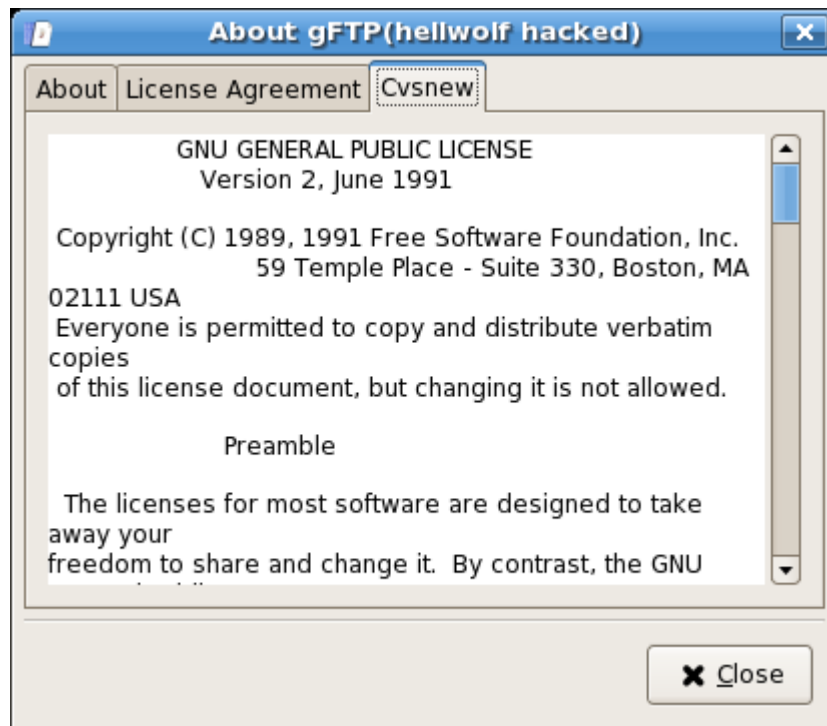


插图 1: Hack 过的 gftp About 对话框

说明测试成功了，接下来提交要提交更改了

```
$vi ChangeLog #在更改成功后再添加一个 ChangeLog Entry 是个好习惯
[...添加一个 ChangeLog Entry...]
$cvcs ci
[...在编辑器中输入一些日志内容...]
```

大功告成，对已经 merge 的上游代码打个 tag。请记住，这个 tag 非常重要，后面要用到。方法就是利用 tag 的 -r 选项。顺便提及一下，在所有用到 tag 的地方，如果用的是 branch_tag 则使用的是 branch 上的最新的“树叶”，在这里就是我们最新导入的代码所打的 tag rel_2_0_17。-F 选项的作用是，即使其后的标签已经存在，一样要重新设置标签。

```
$cvcs tag -r gftp_devel -F last_merged_upstream
```

现在可以用 cvs log 命令来查看一下文件的修订过程了，以 src/gtk/menu-items.c 为例

```
$cvcs log src/gtk/menu-items.c
RCS file: /data/share/softwares/cvsroot/mybranch/gftp/src/gtk/menu-items.c,v
Working file: src/gtk/menu-items.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
    last_merged_upstream: 1.1.1.1
    rel_2_0_17: 1.1.1.1
    gftp_devel: 1.1.1
keyword substitution: kv
total revisions: 3;      selected revisions: 3
description:
-----
revision 1.2
date: 2006/03/16 03:23:16;  author: hellwolf;  state: Exp;  lines: +7 -2
Hacked the about dialog(new title and add a label)
-----
revision 1.1
date: 2006/03/15 13:28:47;  author: hellwolf;  state: Exp;
```

```
branches: 1.1.1;
Initial revision
-----
revision 1.1.1.1
date: 2006/03/15 13:28:47; author: hellwolf; state: Exp; lines: +0 -0
import src from 2.0.17
=====
```

用一张图可以形象的表示该文件的修订历史：

```

+-----+
! 1.1.1.1 (rel_2_0_17, ! <- vendor branch(1.1.1), 即我们指定的 gftp_devel 分支
!last_merged_upstream) !
+-----+
/
/
+-----+ +-----+
! 1.1 !----! 1.2 ! <- 主干代码(trunk)
+-----+ +-----+

```

关于 **import** 究竟做了些什么，在后面介绍了多代码源(**multiple vendor**)的情况时再做分析。

在享受了一段时间你亲手 **hack** 过的代码后，你得知 **gftp 2.0.18** 已经发布了，难道又要重新更改了？不需要，现在就来看看 **cvcs** 的神奇之处吧。

```
$tar jxvf /data/share/softwares/class/net/gftp/gftp-2.0.18.tar.bz2
$cd gftp-2.0.18
$cvcs import -m "import src from 2.0.18" gftp gftp_devel rel_2_0_18
```

在 **import** 的结束后，可能会出现这样的提示：

```
No conflicts created by this import
```

或者

```
2 conflicts created by this import.
Use the following command to help the merge:

    cvs checkout -j<prev_rel_tag> -jrel_2_0_18 gftp_n
```

这个你不用担心，关于这个输出的意思后面再解释。你现在所需要关心的是怎样合并两个版本间的差异。合并的方法如下：

```
$cvcs update -kk -j last_merged_upstream -j gftp_devel
```

命令的意思是合并从 **last_merged_upstream** 到 **gftp_devel**³ 以来的所有变化。这里使用了 **-kk**，意思关闭 **CVS** 对文件中的 **CVS** 变量的展开，如 **\$Id\$** 将被完全保留而不展开，这样可以在合并过程中减少很多不必要的 **conflict**⁴。很明显，你一下子 得到了很多输出，没关系，忽略他们，在来一次 **update** 就可以把关键性息看得更轻楚（这一次 **update** 对于本地代码不会有实际改变）

```
$cvcs update
cvs update: Updating .
C ChangeLog
M gftp.spec
...
M src/gtk/menu-items.c
...
```

这里就是关键信息了。**M** 代表 **merged**，**C** 代表 **conflict**。**Cvs** 是这样完成合并的，首先

3 还记得吗？这是一个 **branch tag**,在这里就是代表所在 **branch** 的最新的部分，也就是 **rel_2_0_18**

4 此选项有粘滞性(**Sticky**)，即今后所有的操作都将附带有 **-kk** 选项

cvcs 检查本地文件是否和第一个-j 指定的文件已经有区别，如果没有，则可以放心合并两个-j 之间的更改，如果有，则分两种情况：你所作的本地更改和两个-j 指定的更改没有冲突可以合并则称 **M** 否则称 **C**。至于这种冲突是怎么定义的，你大可不关心，你所需要做的就是享受这近乎神奇的算法。

然后用前面介绍过的方法解决了冲突后，先别急着 **checkin**,编译测试一下。

```
./configure --prefix=$HOME/bin/root/;make;make install
```

熟悉的对话框有出现了，合并更改是如此简单，是不是很感动呢？好了，现在可以 **checkin** 了，并且别忘了一点——更新你的 **ChangeLog** 和 **last_merged_upstream** 标签。

```
$vi ChangeLog
[...添加一个ChangeLog Entry...]
$cvcs ci
[...在编辑器中输入一些日志内容...]
$cvcs tag -r gftp_devel -F last_merged_upstream
```

再次提醒一定要使用 **-F** 选项以移动已经设置过的标签。

现在再来看看 **src/gtk/menu-items.c** 这个文件的状态吧：

```
$cvcs log src/gtk/menu-items.c
=====
File: menu-items.c      Status: Up-to-date

Working revision:      1.3      Fri Mar 17 08:53:48 2006
Repository revision: 1.3
/data/share/softwares/cvsroot/mybranch/gftp/src/gtk/menu-items.c,v
Sticky Tag:            (none)
Sticky Date:           (none)
Sticky Options:        -kk

Existing Tags:
    last_merged_upstream      (revision: 1.1.1.2)
    rel_2_0_18                (revision: 1.1.1.2)
    rel_2_0_17                (revision: 1.1.1.1)
    gftp_devel                (branch: 1.1.1)
```

和我们希望的一样 **last_merged_upstream** 已经到了 **rel_2_0_18** 所在的位置。一次对上游代码的完美合并也就完成了。现在你应该已经知道了 **cvcs** 跟踪第三方代码的基本方法了。

现在再简单讨论一下多代码源(**multiple vendor**)的情况。在讨论之前有必要首先分析一下 **import** 究竟为我们做了什么。

在上面的 **import** 过程中，你可能已经发现，**cvcs** 用我们在 **import** 命令中提供的 **vendor name** 作为一个分支标记(**branch tag**)，并且分支号为 **1.1.1**。这个 **1.1.1** 是特殊的一个分支号，专门保留给默认的 **vendor** 使用，所以正常情况下你用 **tag -b** 打分支标记的时候是不会得到 **1.1.1** 的分支号的。每次 **import** 一个文件的时候，**cvcs** 首先检查在主干(**trunk**,版本号为 **1.x**)中是否已经有这个文件。如果没有，则在主干中新建这个文件，并且赋予版本号 **1.1**，日志信息自动设置为“**Initial revision**”，并在输出中显示 **N filename**；如果已经有，而且没有在本本地做过修改，则输出 **U filename**，否则输出 **C filename**。最后将文件在 **vendor branch** 中的版本号加 **1**——如果是新文件则初始为 **1.1.1.1**，如果以前是 **1.1.1.1** 则新版本为 **1.1.1.2**。

现在设想这样的情况，一个项目中有个子文件夹 **doc/**是由一个单独的团队维护和发布的，你需要同时跟踪两个团队发布的代码，那么上面的方法就行不通了。解决办法是有的，你注意到上面的过程中 **cvcs** 为我们设置了默认的分支号 **1.1.1**，那么为了维护多个代码源(**vendor**)我们可以在 **import** 命令中用 **-b** 选项手动指定分支号，格式为 **x.y.z**。比如：

```
$cvcs import -b 1.1.2 -m "import doc from 1.0.1" pp/doc pp_doc_vendor doc_rel_1_0_1
```

记住，1.1.2 必须不属于任何已经打过的分支标记(branch tag)，cvs 不会为你检查这种情况。以后每当有新的 doc 发布的时候，方法也类似，只要换一下日志内容和 release_tag 就可以了。

2 用 Subversion 跟踪第三方项目

还是以 gftp 为例。首先为 gftp 建立一个 Subversion 仓库。

```
$mkdir /data/share/softwares/svn/gftp
$svnadmin create /data/share/softwares/svns/gftp
```

Subversion 的一个特点就是仓库维护的是一个目录树，标记(tag)、分支(branch)、合并(merge)等操作都是针对其下的虚拟目录进行的操作，这种概念的一致性让 Subversion 相当的灵活，在后面你一定会慢慢地体会到这一点。为了维护 gftp 的上游代码，按照习惯——不是规定——我们在仓库下建立一个 vendor 目录和 vendor/gftp_devel 目录。

```
$svn mkdir -m "store vendor's code" file:///data/share/softwares/svn/gftp/vendor/
Committed revision 1.
$svn mkdir -m "store gftp's upstream sources" \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel
Committed revision 2.
$svn ls -R file:///data/share/softwares/svn/gftp/
vendor/
vendor/gftp_devel/
```

现在可以导入 gftp 的代码了。

```
$tar jxvf /data/share/softwares/class/net/gftp/gftp-2.0.17.tar.bz2
$svn import -m "import src from gftp 2.0.17" gftp-2.0.17 \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current
Adding      gftp-2.0.17/configure
Adding      gftp-2.0.17/config.rpath
Adding      gftp-2.0.17/debian
Adding      gftp-2.0.17/debian/control
...
```

这里的 current 文件夹有点类似于上面的 cvs 中的 last_merged_upstream 标签。然后将 current 拷贝为 gftp-2.0.17，就好比给代码打了个标签一样。

```
$svn cp file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/gftp-2.0.17 \
-m "tagging gftp-2.0.17"
Committed revision 4.
```

同样，我需要一个由我个人维护的分支来对上游代码进行自定义，按照习惯，这个分支称作 trunk。创建方法——没错，还是 cp。

```
$svn cp file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current \
file:///data/share/softwares/svn/gftp/trunk \
-m "hacked gftp"
Committed revision 5.
```

现在就可以 checkout 我们的主干代码了。checkout(缩写 co)基本语法为：checkout URL[@REV]... [PATH]。[]中的部分是可选的。

```
$svn co file:///data/share/softwares/svn/gftp/trunk gftp
A      gftp/configure
A      gftp/debian
A      gftp/debian/control
```



```
A    gftp/debian/gftp-text.links
...
Checked out revision 5.
$cd gftp
```

和 **cvs** 部分一样，修改 **ChangeLog** 和 **src/gtk/menu-items.c** 文件，修改完成后，进行编译和运行。这时你还可以用 **svn diff** 查看一下所作的修改，确定无误后就可以 **checkin** 了。

```
$svn diff ChangeLog
Index: ChangeLog
=====
--- ChangeLog      (revision 5)
+++ ChangeLog      (working copy)
@@ -1,3 +1,7 @@
+2006-03-20  ZC Miao  <hellwolf@cocteau.freehell.org>
+
+    * hacked about dialog
+
+2004-03-28  gettextize  <bug-gnu-gettext@gnu.org>
+
+    * Makefile.am (SUBDIRS): Add intl.
$svn ci
[...在打开的编辑器中输入日志内容...]
Sending          ChangeLog
Sending          src/gtk/menu-items.c
Transmitting file data ..
Committed revision 6.
```

一切都很顺利，现在 **2.0.18** 的代码来了，按照 **cvs** 的思维，你可能认为是用 **import** 再次导入代码。

```
$svn import -m "import src from 2.0.18" gftp-2.0.18 \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current
svn: File already exists: filesystem '/data/share/softwares/svn/gftp/db',
transaction '6-1', path '/vendor/gftp_devel/current/configure'
```

很不幸，这次你错了。**Subversion** 的 **import** 的作用相对有限，仅仅是递归的将本地文件夹拷贝到 **Subversion** 的目录树中，没有维护和记录两次 **import** 之间差异的能力。一种比较容易想到的方法是，**checkout** 出 **current** 的代码，然后删除其下的所有文件(除了 **.svn** 文件夹)，将 **gftp-2.0.18** 的代码完全移动过来，然后 **checkin**。这方法似乎不错，但是，仔细一想，如果有文件在更新过程中被删除或者有新文件添加了呢？手动 **svn add**, **svn rm**？老天，这对一个略有规模的项目来说会成为一个多头痛的事情阿！更不幸的消息是，**Subversion** 没有处理这个问题的直接机制，一个普遍使用的方法是使用一个称之为 **svn_load_dirs.pl** 的脚本⁵来协助完成这个过程，记得在这之前先保存一下上次合并的 **current**：

```
$svn cp \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/last_merged_upstream
[...输入日志...]
Committed revision 8.
$ /usr/share/doc/subversion-1.3.0/svn_load_dirs.pl \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/ \
current \
gftp-2.0.18/
```

第一个参数是 **Subversion** 仓库中的一个文件夹位置，第二个参数是一个相对第一个参数位置的 **Subversion** 文件夹，第三个参数是需要导入的代码。运行后，脚本会询问你：

5 请查询你的发行版是如何提供该脚本的，据我所知道的：在 **debian** 中该脚本随 **subversion-tools** 软件包提供，你可以用 **apt-get** 安装之；在 **Fedora Core** 中该脚本在 **/usr/share/doc/subversion-*/svn_load_dirs.pl** 处找到。下文将以 **Fedora Core** 为例。

```

    Deleted      Added
  0 stamp-h.in_ lib/fsplib/COPYING
  1 -----      depcomp
  2 -----      po/en_CA.gmo
  3 -----      po/en_CA.po
...
Enter two indexes for each column to rename, (R)elist, or (F)inish:这里输入 f, 回车
...

```

这样就完成了一次导入新代码的过程，现在就打一下标签吧：

```

$svn cp file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/gftp-2.0.18 \
-m "tagging gftp-2.0.18"

```

其实，前两步过程可以合并为一步，`svn_load_dirs.pl` 脚本有个参数 `-t`，作用就是在导入完代码后自动帮你打一个标签，标签的位置仍然是相对于第一个参数，如上面的过程就可以合并为⁶

```

$/usr/share/doc/subversion-1.3.0/svn_load_dirs.pl \
-t gftp-2.0.18 \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/ \
current \
gftp-2.0.18/

```

导入代码完成后，现在就是要合并更改了。

回到你的工作目录，运行下列命令：

```

$cd ~/tmp/gftp
$svn merge \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/last_merged_upstream
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current
D    stamp-h.in
UU   configure
U    debian/changelog
...

```

很长的一段输出，我们关心的是哪些文件出了 **conflict** 了，用下面的方式可以看到

```

$svn status | egrep '^C'
C      ChangeLog

```

不出所料，还是 **ChangeLog** 文件出现了 **conflict**，修改后，用下面的命令显式的通知 Subversion 我们已经解决了 **conflict**

```

$svn resolved ChangeLog
Resolved conflicted state of 'ChangeLog'

```

然后依然是进行编译和测试，确认成功后 **checkin**，然后别忘了重新标记一下 **last_merged_upstream** 标签

```

$svn ci
...
Committed revision 10.
$svn rm \
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/last_merged_upstream \
-m 'delete old tag'
Committed revision 11.

```

6 在写本文的时候，我遇到了一个 `svn_load_dirs.pl` 的 bug，就是当有文件名含有 '@' 符号时，load 过程将出错，暂时的解决办法是修改 `svn_load_dirs.pl`，搜索 `my @command = ($svn, 'propget', 'svn:eol-style', $upd_file);` 一行，将其改为 `my @command = ($svn, 'propget', 'svn:eol-style', $upd_file, '--revprop', '-r', 'HEAD');`；我已经提交了该 bug，但愿在你看到本文时已经没有问题了。

```
$svn cp \  
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/current \  
file:///data/share/softwares/svn/gftp/vendor/gftp_devel/last_merged_upstream \  
-m "new merged upstream tag"  
  
Committed revision 12.
```

用 Subversion 跟踪第三方的代码就是这么简单。和 CVS 一样，我们再考虑一下多代码源的情况。得益于 Subversion 在概念上的一致性，不难发现每多一个 vendor 只要在 vendor/ 目录下添加一个 vendor name 文件夹就可以了——每次有新代码来一样可以用 svn_load_dir.pl 脚本，导入代码结束后一样是用 merge 命令合并两次导入的代码的修改部分。这里就不再详细叙述其过程了。

六 建立网络合作环境简介

本节简单介绍一下在网络上建立开发合作环境的基本方法。

1 结合 ssh(CVS 和 Subversion)

利用 ssh 进行网络合作开发是目前一种很流行的方式，如 savane(<http://gna.org/>) 这样的成熟的自由软件服务器集成方案就是用的 ssh 方式。这主要是得益于 ssh 的私钥/公钥认真的安全性和方便性。关于配置 ssh 的私钥/公钥认证方式请参考网络上相关文章。配置好后，CVS 需要设置 CVS_RSH 为 ssh，然后根仓库的指定方式用

```
-d :ext: user@server:/path/to/cvsroot
```

在 Subversion 中则用 svn+ssh://的 URI 前缀，比如

```
svn+ssh://svn.sample.net/repos/
```

2 CVS 建立网络合作环境的其他方法

CVS 还有一种常见的网络合作方式是用 pserver，其优点是用户管理可以独立于系统的用户数据库，但也带来了很多安全问题。

3 Subversion 建立网络合作环境的其他方法

Subversion 也有一个独立的服务端程序称为 svnserve，通过它，客户可以指定 svn://这样的 URI 前缀来访问仓库。还有一种更常见的方式是用 apache 的 mod_dav_svn 扩展，利用该扩展，客户甚至可以直接使用 http://这个 URI 前缀来访问仓库。

七 结语

介绍了这么多，相信你对 CVS 和 Subversion 也有了自己的看法，最后希望本文对你有所帮助。