

Omen

Pragmatic Version Control 实效版本控制

(正式版中文共享版 Ver 1.0)

原著: Dave Thomas; Andy Hunt

翻译: 菠菜

发布: <http://www.Matrix.org.cn>

注明: 本资料由菠菜负责翻译并授权Matrix (<http://www.Matrix.org.cn>)发布, 您可以免费使用和任意传播, 但是您必须遵循以下条款:

1. 不得用于任何商业目的, 包括但不限于出版, 印刷, 销售, 咨询。
2. 您可以传播本资料, 但是传播必须保留本协议说明并不可做任何修改。
3. 不得对本文内容进行修改再传播, 为了保证文章内容和版本的统一, 任何修改建议请在Matrix 论坛讨论, 并由Matrix 组织统一修改, 修改后统一发布修改过后的版本。
4. Matrix 对本授权协议拥有最终解释权。
5. 与本书相关的所有公告都发布在[Matirx的Cvs Maven Ant论坛](#)

Chris (Chris@Matrix.org.cn)

From Matrix

版本变化

NO.	章节	开始日期	结束日期	参与人	文档版本号
1	第一章	2004-05-10	2004-05-15	Melthaw Zhang	0.1.0
2	第二章至第七章	2004-05-15	2004-06-09	Melthaw Zhang	0.5.0
3	第八章	2004-06-09	2004-06-11	Melthaw Zhang	0.6.0
4	第九章	2004-06-11	2004-06-14	Melthaw Zhang	0.7.0
5	第十章	2004-06-14	2004-06-21	Melthaw Zhang	0.8.0
6	重新排版	2005-05-15		Melthaw Zhang	0.8.5

关于 **Matrix** 的最新消息

<p>[公告]Matrix 站衫</p> 	<p>向IT界倡议：援助程序员王俊行动</p> <p>发布UML for Java Programmers中文共享版</p> <p>[公告]Matrix站衫-实物照片</p> <p>[公告]新开数据库专区—Oracle专业论坛</p> <p>Matrix Java大讲坛—产品开发过程</p> <p>[New]Matrix 正式启用Jmatrix</p>
----------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

目 录

关于新手工具箱	6
序言	7
排版约定	7
致谢	8
第一章 引言	9
1.1 Version Control in Action 版本控制活动	10
1.2 各章组织结构	12
第二章 什么是版本控制.....	14
2.1 文件库(Repository).....	14
2.2 到底应该保存什么内容?	15
2.3 工作间和文件操作	17
2.4 项目, 模块和文件	18
2.5 版本的作用到底在哪?	19
2.6 Tags 标签	20
2.7 Branches 分支.....	21
2.8 Merging 合并.....	23
2.9 锁定选项.....	24
2.10 配置管理(CM).....	27
第三章 入门 28	
3.1 安装 CVS	28
3.2 创建文件库 (Repository)	32
3.3 CVS 命令	33
3.4 创建一个简单的项目	34
3.5 开始项目工作	36
3.6 进行修改	37
3.7 更新文件库 (Repository)	39
3.8 冲突发生的时候.....	40

3.9 解决冲突	42
第四章 怎么样版本控制?	48
4.1 我们的基本哲学	48
4.2 组织版本控制系统	49
第五章 访问文件库 (Repository)	50
5.1 安全和用户帐号	51
5.2 CVSROOT:访问目的地参数字符串	52
5.3 设置 ssh 访问模式	53
5.4 使用 pserver 进行连接	54
第六章 常用 CVS 命令	56
6.1 Check Out 命令	56
6.2 保持代码最新的状态	58
6.3 添加文件和目录	60
6.4 忽略某些文件	64
6.5 给文件重命名	65
6.6 重命令目录	67
6.7 查看更改情况	68
6.8 处理合并冲突	71
6.9 提交修改	75
6.10 查看修改的历史信息	76
6.11 取消修改	79
第七章 使用标签和分支	82
7.1 标签, 分支和打标签	82
7.2 创建发布分支	84
7.3 在发布分支上工作	86
7.4 Generating a Release 生成发布	86
7.5 在发布分支修改 bug	88
7.6 开发人员的试验分支	89

7.7 在试验分支上编码	90
7.8 合并试验分支	91
第八章 创建项目	92
8.1 创建初始项目	92
8.2 项目结构.....	94
第九章 使用模块	97
9.1 组织子项目变的很容易	98
9.2 CVS 模块	100
9.3 总结.....	106
第十章 第三方代码.....	107
10.1 包括源码的库	109
10.2 修改第三方代码	112
CVS概要及诀窍	119
其他资源	131
B.1 CVS 在线资源.....	131
B.2 其他 CVS 书籍.....	131
B.3 其他版本控制系统.....	132
B.4 参考书目	134

关于新手工具箱

在过去几年来,我们发现实效编程的许多读者其实真正需要的是在开发的基础下部组织方面得到适当的帮助,这样他们才能更早地形成良好的习惯。很多高级读者完全能够明白这个主题,但是也需要让他们的团队和组织的其他成员信服和得到教育。我想我们已经发现了可以帮助他们的一些重要的东西。

新手实效工具箱包括了三卷书,内容覆盖了现代软件开发的最基础的知识。这系列书籍包括了你最需要掌握的实践,工具,原理,这些知识让你的团队保持积极性和高产的效率。用这些知识武装起来的你和你的团队能够尽早地采纳和形成一些良好的习惯,并由此享受组织良好,安全可靠的项目给你带来的的安全感和舒适感。

《实效版本控制》,也就是本书,描述了在一个项目中怎么样将版本控制作为项目基石来应用的全过程。没有版本控制的项目好比没有 **UNDO** (撤销) 按钮的 **word** (文字) 处理器: 你输入的内容越多,犯错误的代价就越高。实效版本控制讲述的内容完全不同于那些曲折的官僚的,或者冗长的乏味的版本控制过程,而是告诉你怎么样有效地使用版本控制,利用版本控制的优点和安全性。

卷二《实效单元测试》讨论怎么样进行有效地单元测试。单元测试是一个非常基础的技术,在我们编写代码的时候,单元测试为开发者提供了真实的,实时的反馈。许多开发人员曲解了单元测试的含义,并没有真正认识到怎么样利用单元测试来提高我们的工作。

卷三《实效自动控制》覆盖了自动化创建,测试和发布过程中最基本的实践和技术。很少有项目能够随意控制时间,实效自动控制可以让计算机帮你做更多的日常工作,把你从繁琐中解脱出来,将注意力更多地投入到有意义的和有难度的挑战上。

这些书的风格和我们的第一本书完全相似,书籍着笔于你每天劳苦工作中要面对的特殊需要和问题。但这些书并不雷同于那些只提供了整体概况的华而不实的书籍,作者致力于让读者真正地理解书中精髓,让读者在此基础上,纵然有一天面对即使本书没有涉及到的情况,也能临危不惧,从容不迫地提出自己的解决方案,这才是本书的目的。

要查询本书的最新消息以及其他适合开发人员和管理人员的相关资源请访问我们的站点: <http://www.pragmaticprogrammer.com>

祝你愉快!



序言

版本控制做的好，就象呼吸一样，你根本感觉不到它的存在，但是它却能让你的项目保持生气勃勃的活力。然而当我们和世界各地的团队交流访问的时候，我们注意到：没有几个团队的版本控制做对了（甚至很多就根本没有版本控制）。

这里面有很多原因，很多团队抱怨版本控制太复杂了。他们获取基本材料，将材料 **check in** 或者 **check out** 某个中心数据库，但是当需要创建发布的时候，或者需要处理第三方代码的时候，就开始失控了。失败的是，这个时候这些团队要么停止使用版本控制，要么就陷在写满了晦涩难懂的解决过程的一页页指令里面。

其实根本不用这么复杂。在这本书里面我们会给出很多基本的药方，可以治疗版本控制遇到的 90% 的疑难杂症。在这些技巧的指引下，团队可以开始版本控制的愉快之旅。当然你们的不断反馈对我们来说也很重要，有任何错误，忽略的地方，或者建议都可以通过访问我们的网站告诉我们：<http://www.pragmaticprogrammer.com/sk/vc/feedback.html>

排版约定

<i>italic font</i>	Indicates terms that are being defined, or borrowed from another language.
computer font	Computer stuff (file names, terminal sessions, commands, and so on).
	A warning that this material is more advanced, and can safely be skipped on your first reading.
	"Joe the Developer," our cartoon friend, asks a related question that you may find useful.
-d ⇒ <u>Destination</u>	An <i>aide-memoir</i> for a command option (in this case -d).

致谢

写书的其中一个乐趣是邀请朋友对自己的书稿进行评价。让人惊讶的是他们居然都同意了。特别感谢 Steve Berczuk, Vinny Carpenter, Will Gwaltney, Krista Knight, Andy Oliver, Jared Richardson, 和 Mike Stok, 他们提出了宝贵的建议。

Dave Thomas and Andy Hunt

September, 2003

pragprog@pragmaticprogrammer.com

第一章 引言

这本书将对你讲述怎么样通过使用版本控制来提高你的软件开发效率。

版本控制，有时也叫源码控制。假如我们的项目是由三角架支撑着，那么版本控制就是其中的第一条腿。我们认为使用版本控制是所有项目必须的。

不论是对于团队还是个体，版本控制都提供了很多好处。

- 版本控制提供项目级的 **undo**（撤销）功能：没有什么事情是终结版本，任何错误必须很容易回滚。假设你在使用世界上最复杂的文字处理系统。它具备了所有的能想到的功能，就是没有支持 **DELETE**（删除）键。想象你打字的时候得多么的谨慎和缓慢吧，特别是一篇超大的文档的快临近末尾的时候，一个不小心就要重头再来（译者：试想你选中所有的文字，不小心按了 **DELETE** 键，因为没有撤销功能，只好重新录入）。编辑文字和版本控制相同，任何时候都需要回滚，无论是一个小时，一天，还是一周，这让你的团队工作自由快速的工作，而且对于修正错误也非常自信。
- 版本控制允许多人在同一代码上工作，只要遵守一定的控制原则就行。再也不会发生诸如一个人覆盖了另一个人编辑的代码，导致那个人的修改无效这样的情况。
- 版本控制系统保存了过去所作的修改的历史记录。如果你遭遇到一些惊讶的代码，通过版本控制系统可以很容易找出是谁干的，修改了什么，修改的时间，如果幸运的话，还能找出原因。
- 版本控制系统还支持在主线上开发的同时发布多个软件版本。在软件发布的时候也不需要整个团队的停止工作，不需要 ~~冻结~~ 代码。
- 版本控制也是项目级的时间机器（原文：time machine），你可以选择任何一个时间，精确地查看项目在当时的情况。这对研究非常有用，也是重现以前某个有问题的发布版本的基础。

本书将注意力集中在以项目的视角为出发点的版本控制上，而不是简单地罗列出版本控制系统的命令就完事大吉。取而代之的是，我们关注项目成功所需的任务，然后探讨怎么样通过版本控制系统来达到这个目标。

版本控制怎么样实践呢？首先让我们从小故事开始吧。。。

1.1 Version Control in Action 版本控制活动

某天, Fred 走进办公室, 迫不及待地工作起来, 他现在正在做一个名叫 Orinoco 的订书系统(原文: Fred rolls into the office eager to continue working on the new Orinoco book ordering system)。(为什么叫 Orinoco? Fred 的公司喜欢使用河流的名字来作为内部项目的名字)在享用了第一杯咖啡后, Fred 从中心版本控制系统那里更新了项目的本地源码。日志里列出了所有已更新的文件名, 他注意到 Wilma 已经修改了 Orders 的代码。Fred 担心这个改变会影响自己的工作, 碰巧今天 Wilma 在客户那边安装新系统, 所以不能直接询问她。Fred 只好转向求助于版本控制系统, 查询和 Orders 代码变化有关的注释。Wilma 的注释让他松了口气:

```
* Added new deliveryPreferences field to the Order class
```

为了找出更详细的情况, Fred 再次通过版本控制系统查询源码上的实际变化。他注意到 Wilma 添加了两个变量, 都设置了缺省值, 也没有发现其他地方对这些值做了修改。这在将来可能会是个问题, 但是今天现在还不会影响他的工作, 所以 Fred 继续工作起来。

在这些代码基础上, Fred 添加了一个新类和两个测试类。在创建文件的时候, Fred 还添加了要加入到版本控制系统的文件名(原文: Fred adds the names of the files he creates to the version control system as he creates them)。直到提交修改, 这些文件才会添加到版本控制系统里, 只添加名字是为了提醒自己别忘了将来要将文件加进去。

两个小时后, Fred 终于完成了新功能的第一部分。新代码通过了测试, 也不会影响系统的其他部分, 所以 Fred 决定把它们全部 check in 版本控制系统, 让其他成员也能访问这些代码。多年以来, Fred 发现频繁地进行 check in 和 check out 比好几天才做一次要好很多, 因为每次只需要处理两个文件的代码冲突, 这比一周一次但是要处理整个团队修改的代码要强多了。

为什么你从不接电话

就在 Fred 准备开始下一轮编码的时候, 他的电话响了, 是 Wilma 从客户那边打来的。她安装最新发布的版本的时候遇到点问题: 发票打印模块并没有在货运量基础上计算的销售税。客户很着急, 这个问题必须马上解决。

除非你使用版本控制系统...

Fred 和 Wilma 反复检查了的发布版本的名字, 然后 check out 版本控制系统里面该发

布版本的所有文件。并把这些文件放在电脑的一个临时目录下，因为他打算完成工作之后就把这些文件删了。现在在他的电脑上已经有系统的两份源代码：主线版本以及目前和客户端上正在使用的版本。因为就要开始修改错误，他让版本控制系统在他当前工作的代码上打上了标签。（将来错误修改之后他会打上另一个标签，这些标签其实就是他在开发的时候留下的记号。通过在做修改前后使用一致的标签，在将来某个的时候，其他成员就能够准确地找出做了哪些修改。）

为了找出问题，**Fred** 首先写了一个测试。通过测试，问题立即浮现出来，很明显没有人检查涉及到货运的销售税的计算问题。（**Fred** 把这个事情记下来，准备在下次迭代评估会议上要好好谈谈这个事情，**this is something that should never have gone out the door**（待译））**Fred** 一边叹气一边将货运部分添加到税的合计部分里面（原文：**Fred adds the line of code that adds shipping in to the taxable total**），然后编译并测试代码。他重新做了一次快速的整体测试，测试通过后，**Fred** 将已经修改了错误的代码重新 **check in** 版本控制系统。最后，他给这个发布分支添加了一个标签，表示错误已经修改了。跟着他给负责紧急发布版本的 **QA** 发了一个消息。通过他打的标签，他们能够指示构建系统生成包含了他的修正补丁的交付版本。然后 **Fred** 给 **Wilma** 打了个电话，告诉她问题已经修复了，目前工作移交到 **QA** 部门，她很快就可以拿到新版本了。

这个工作中的小插曲结束之后，**Fred** 将刚才的代码从机器上删除了。看起来没有引起混乱，他刚才的修改也很安全地保存到中心服务器。跟着他就担心起来，刚才他在发布分支上发现的错误在当前正在开发的代码上是否也存在呢？一个快捷的方法就是把刚才在发布版本上添加的测试代码添加到正在开发的测试集里面去。于是他让版本控制系统将发布分支上的测试部分给合并到正在开发的相应的文件上。这个合并的操作会将发布分支上的任何修改复制到开发的代码上。最后他再次运行测试，测试没通过，看来问题仍然存在。于是他将在发布分支上的修改错误的代码部分合并到开发的代码上。（他不必用发布分支在本地机上的代码来做合并，所有的修改都可以从中心版本控制系统上拿到（译者：因为这个原因，所以 **Fred** 将刚才的代码从机器上删除了））测试终于通过了，他将这些修改提交到版本控制系统。看来又少了一个折磨大家的 **bug**（译者：这里用 **bug** 比用错误或者问题更符合程序员的语气）了。

问题解决了，**Fred** 重返工作岗位。这是一个愉快的下午，写测试代码，编程，看来有望在下班前搞定。当然在他工作的时候，其他的家伙也在修改代码，因此他使用版本控制系

统将本地的代码和别人的修改进行了同步。快下班的时候，Fred 最后运行了一次测试，然后将修改做了 **check in**，就等二天上班了。

第二天。。

不幸的是，第二天出了点意外。一夜下来 Fred 家里的中央暖气终于坏了，而且 Fred 住在明尼苏达州，碰巧又是二月（译注：二月是明尼苏达州很寒冷的时候），这可不能随便凑合。Fred 打电话到公司说要等维修人员，今天整天都不能去公司了。不过这可不代表他就不用工作了。Fred 通过因特网的安全连接方式访问公司，**check out** 最新的开发源码到自己的笔记本电脑上。因为头天晚上回家之前他将所有的修改都做了 **check in**，现在得到的所有源码都是最新的。虽然裹着毯子，坐在火炉旁，但也可以在家继续工作了。在结束当天的工作的时候他将在笔记本电脑上做的所有修改都做了 **check in**，这样第二天去上班的时候可以拿到和今天同步更新的代码。生活挺好的（要是修中央暖气不付钱就更好了）。

Story-book 项目

The correct use of version control on Fred and Wilma's project was pretty unobtrusive（待译），但是版本控制也对他们的沟通提供了帮助，就算 Wilma 在很远的地方也没问题。Fred 可以调查代码的修改状况，可以在他们的应用程序的多个版本上打补丁。他们的版本控制系统支持离线工作，因此 Fred 的工作是和地点无关的，这样他就可以在家里暖气出问题的时候在家一边工作一边等维修了。因为他们有适当的版本控制（而且又会使用），就算遇到一些没有预料到的事件，即使没有经验，他们也能对付。

使用版本控制给 Fred 和 Wilma 带来了很多好处，譬如可以游刃有余的处理现实生活中很多奇怪的事情。那就是本书所要讲述的。

1.2 各章组织结构

下一章，也就是“**什么是版本控制？**”，将介绍版本控制系统的一些基本概念和术语。这也是很多版本控制系统里采纳的。本书我们将重点讲述一种自由的版本控制软件——CVS。通过长期的发展和积累，CVS 目前可能是应用最广泛的版本控制系统了。

第三章：“**开始使用 CVS**”是一篇 CVS 指南。余下各章就是在项目中使用 CVS 的一组技巧了。这些部分将分为六章，每章都会讲述一系列的技巧。

- 连接 CVS 的不同方式

- 常用的 CVS 命令
- 使用标签和分支处理发布和实验代码
- 创建项目
- 创建子项目
- 处理第三方代码

最后本书以汇总所有的技巧的附录结束，附录还提供了一个简单的资源列表和参考书目。

第二章 什么是版本控制

版本控制系统就是在你开发应用程序的时候用来保存版本变化的地方，而这个版本变化就是指你所编写的东西的版本变化。

基本的部分都很简单。不幸的是，过去几年，形形色色的人们针对不同的版本控制使用了不同的术语。并引起了混淆。因此我们首先定义一些我们将会使用到的术语。

2.1 文件库(Repository)

你也许已经注意到了我们刚才所说的：版本控制系统用来保存你所编写的东西的所有的版本变化的地方。但是我们没有说清楚到底这些东西保存在哪里。实际上，它们都保存在文件库(repository)里面。

几乎在所有的版本控制系统里，文件库（Repository）都是指保存你的项目文件的各版本的主备份的中心。也有些版本控制系统使用数据库(database)这个概念，还有一些使用普通文件，有些呢两者兼备。无论怎么样，很明显的是文件库（Repository）是你的版本控制系统策略的核心组件。你应该将它置于一个安全可靠的机器上。而且应该毋庸置疑的进行有规律的备份。

不同的网络访问方式

不同的版本控制系统的作者对于网络有不同的定义。一些人认为是通过共享的网络设备访问文件库（Repository）里的文件（如 Windows 的共享文件价或者 NFS）。另外一些认为是拥有一个客户端—服务器结构，客户端在此基础上通过网络访问多个文件库（Repository）。这两种方式都有效（尽管前者很难正确地设计出来，如果底层文件系统对锁定的支持并不可靠的话）。然而，你会发现开发和安全问题是导致你使用哪个系统的关键。

如果版本控制系统需要访问共享设备，而且你还需要可以从内部网外访问版本控制系统，那么你需要搞明白你的组织是否允许你这样干。虚拟专用网（VPN）包就提供这种安全访问的服务，不过不是所有的公司能在这上面运作。

CVS 使用 C/S（客户端/服务器）模式进行远程访问。

在过去，文件库（Repository）极其使用者不得不共享一台机器（或者至少要共享一个文件系统），结果使用起来相当的受约束，很难让开发人员在不同的地点，机器或者不同的操作系统上工作。因此，发展至今，大量的版本控制系统支持网络操作：作为开发人员，可以通过网络访问文件库（Repository），这种情况下：文件库（Repository）扮演着服务器的角色，而版本控制工具扮演了客户端的角色。这变的非常强大，无所谓开发人员的位置，只要他们能通过网络访问文件库（Repository），他们就能访问项目的所有代码和历史变化。而且他们可以用安全的方式来访问，甚至可用通过因特网来访问文件库（Repository）。我和 Andy 就经常在旅途中通过因特网访问我们的源码。

这引申出一个有趣的问题。如果你没有网络连接来访问文件库（Repository），而你又急需开发，那怎么办？答案很简单：你必须要有网络才行。一些版本控制系统专门针对网络连接文件库（Repository）而设计，这要求你必须使用网络，如果不通过中心文件库（Repository），完全不能修改代码。另外一些系统支持的访问方式更多一些。本书使用的 CVS 系统就是后者。我们可以在 35000 英尺高的飞机上在笔记本电脑上编辑修改，然后在酒店进行同步更新。是否支持离线/在线是选择版本控制系统的重要指标，因此你要首先搞清楚你所选择的产品是否支持你的工作方式（译者：比方说经常出差的人）。

2.2 到底应该保存什么内容？

项目里的所有东西都保存在文件库（Repository）。那么这里的"东西"精确点讲到底是什么呢？

嗯，很明显你需要创建项目的所有源文件，这可能是 Java 文件，C# 文件，或者 VB 文件，以及任何你在使用的编程语言。实际上，源文件对于版本控制系统如此的重要，很多人因此将版本控制系统称之为：源码控制系统。

源码的确重要，不过很多人忘了其他的需要保存在版本控制系统里的东西。举例来说：假设你是个 Java 程序员，你可能通过使用 Ant 工具来编译你的源码。Ant 通常使用一个名

为 `build.xml` 的脚本文件来控制操作。这个脚本也是创建过程的一部分，如果没有它就不能创建应用程序。因此它也应该保存在版本控制系统里面。

类似地，很多项目使用 `metadata` 元数据来驱动他们的配置管理。因此这些元数据文件也应该保存在文件库（**Repository**）中。依此类推，任何脚本，如你用来创建发布盘，或者 QA 用的测试数据等等，都应该保存在文件库（**Repository**）里。

实际上有个简单的方法来判断那些东西应该而哪里不应该保存在文件库（**Repository**）里面。只要问自己，如果手头上没有最新版本的 `x`，我们是否能够创建交付一个应用程序？如果回答是否，那 `x` 就应该保存在文件库（**Repository**）里。



Joe的问题：什么是生成的制成品？

如果我们把所有需要用于创建项目的东西都需要保存在文件库（**Repository**）里，那么是不是说连生成的文件也要保存在里面？比方说我们运行 `JavaDoc` 生成的源码的 `API` 文档。是不是那些文档也要保存在版本控制系统里面？

答案很简单，那就是“不”。如果生成的文件能够从其他文件再生，那么保存就完全是多余的。有副本难道不好吗？哪倒不是，我的意思是因为我们不想浪费磁盘空间。也不想打乱步骤。如果我们同时保存了源码和文档，那么当源码改变的时候，文档因为得不到及时更新而过期。如果我们忘记及时更新并 `check in`，那么就会在文件库（**Repository**）里出现误导的文档。所以在这个情况吓，我们只想保存信息的单一源头，也就是源码。这个规则同样适用与大部分生成的制品。

实际上一些制品很难重新生成，譬如你可能只有一个工具的许可证，所有的开发人员都要靠它来生成文件，或者你的某个制品需要一个小时才能创建出来。这回总情况下，将这些生成的制品保存在文件库（**Repository**）里就有意义了。因为开发人员没有工具许可证就不能创建文件，或者一个可以创建昂贵制品的高速机器。这些东西都可以 `check in` 文件库（**Repository**），让所有的开发人员都可以直接利用这些现成的文件进行工作。

除了所有用于创建发布版本的文件以外，还应该将一些非代码形式的项目资料置于版本控制之下（），包括项目的（内部和外部）文档。还有重要的邮件，回忆记录，以及你从网

上找来的任何对项目有益的资料。

2.3 工作间和文件操作

文件库（**Repository**）保存了项目的所有文件，但是如果你需要给应用程序添加什么新的特征，这帮助不大。我们需要能够直接访问文件的地方，那就是 **workspace** 工作间。**workspace** 保存了从文件库（**Repository**）拿到到的项目文件的本地拷贝。对于中小型的项目，**workspace** 可能简单的保存了素有代码和其他内容的拷贝。。对于大型项目，你就要对所有的东西进行组织分配，这样所有的开发人员能够在子项目一级上工作，这样可以节省创建的时间，而且还独立于其他的子系统。也有人把 **workspace** 叫做 **working directory** 或者 **working directory**。

我们需要从文件库（**Repository**）获取资料，用于初始化 **workspace**。不同的版本控制系统对于这个操作使用了不同的名字，而最通用的说法（比如 **CVS**）是 **checking out**。当你 **check out** 文件库（**Repository**）的时候，会将文件的本地拷贝放到你的 **workspace** 中。**Check out** 这个操作保证你能拿到想要的文件的最新拷贝，并将这些文件放在一个和文件库（**Repository**）一样结构的目录下。

当你在某个项目上工作的时候，会修改该项目在本地 **workspace** 里的代码。时不时的你也要将修改保存到文件库（**Repository**）。这个过程叫做 **committing**，也就是将你所做的修改提交到文件库（**Repository**）。

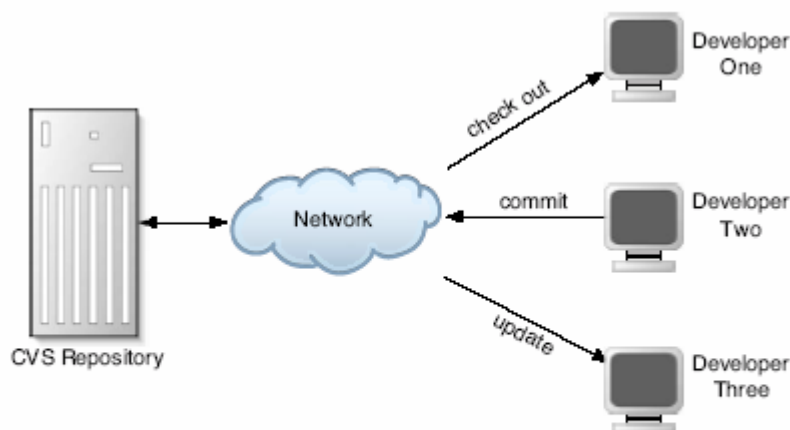


Figure 2.1: Clients and a Repository

当然在你做修改的同时其他队员也在修改。他们也会向文件库（**Repository**）提交他们所做的修改。但是这种袖管改不会影响到你的本地代码，你的本地代码也不会突然变化，因为其他人把修改保存到文件库（**Repository**）。另外一种方法就是通过指示文件库（**Repository**）更新本地代码。在更新的过程中，你会接受到最新的文件集。当然，如果你的同时也在做更新操作，他们也会同步你的修改（是不是有点糊涂了，不管怎么样，有些家伙 **check out** 最新的修改的时候，用术语 **check out** 代表 **update**。因为这种用法非常普遍，因此本书我们也将使用这种用法）。图 2.1(Figure 2.1)显示了这种交互关系。

当然还有一个潜在的问题：如果你和同事同时对同一文件进行修改的时候又怎么办？这取决于你使用的版本控制系统，大部分版本控制系统都有自己的一套方法。我们将在[锁定选项](#)那一节更详细地探讨这个问题。

2.4 项目，模块和文件

到目前为止我们一直谈论的都是怎么样保存，还没有涉及到这些东西是怎么组织起来的。

从最底层的角度来看，大部分版本控制系统处理个体文件。每个文件都按名保存在文件库（**Repository**）里，如果你添加一个名为 **Panel.java** 到文件库（**Repository**），那么其他人可以将 **Panel.java** **check out** 到各自的 **workspace**

不过那非常底层化。一个有代表性的项目通常都会有上百乃至上千个文件，同样地，一

个公司可能会有很多项目。还好几乎所有的版本控制系统都允许你设计文件库（Repository）的结构。从高级的角度来讲，一般按项目组织工作。在每个项目里，又按模块组织工作（通常叫子模块）。举个例子：假设你在 Orinoco 这个项目上工作，这是一个庞大的基于 web 的图书订购系统。所有的用于创建该应用的文件都保存在文件库（Repository）里 Orinoco 项目名下。如果需要，你可以将所有的文件 **check out** 到本地。

项目 Orinoco 被划分为几个很大的独立的模块。例如：某个团队处理信用卡模块，另外的处理订单履行模块。信用卡子项目的人员工作的时候不需要所有的项目文件，他们的代码应该很好地划分出来，这样当他们 **check out** 的时候只希望看到相应的项目的那部分文件。

CVS 提供了让文件库（Repository）管理员把项目划分为模块的功能。一个模块包括了一组文件（一般都保存在一个或多个目录下），而且可以根据模块的名字被 **check out**。模块可以是层次结构式的，但没有规定说一定要这样，相同的文件可以出现在多个模块中。模块甚至提供了在不同项目间共享代码的功能（将文件放到某个模块，然后让其他团队通过引用名字来访问）。

模块提供了更丰富多变的视图，让团队成员按需工作。我们将在第 9 章探讨模块的知识。

2.5 版本的作用到底在哪？

本书是关于版本控制系统的，但是至今我们都在探讨从文件库（Repository）保存和获取文件方面的知识。那么版本的作用到底在哪？

事实背后的真相是：版本控制系统的文件库（Repository）是一个相当智能的系统。不只是保存了每个文件的当前的拷贝，还保存了已经做过 **check in** 文件的每个版本。如果你 **check out** 一个文件，编辑并 **check in** 该文件，文件库（Repository）除了会保存原始的版本外还保存了你修改后的版本。大部分版本控制系统使用简单的数字编号作为文件的版本号。如果你使用的是 CVS，那么某个文件的第一个版本的版本号是 1.1（很快我们就要探讨更复杂的版本编号）。和这些版本号紧密相关的是文件被 **check in** 的日期和时间，以及开发人员可选择性的使用的用于描述修改的注释。

保存修订版本的这种系统非常强大。通过这个功能，版本控制系统能够做到：

- 获取某个特定的修订版本的文件

- 可以 **check out** 两个月前所有的系统源码。
- 告诉你某个特定文件在版本 1.3 和 1.5 之间的变化。

而且你还可以使用修订版本系统撤销错误。如果你在周末的时候发现自己的开发完全走偏差了，你还可以回到岔路口重新来过，将代码完全回滚到周一早上的状态。

不同的修订版本系统对修订版本的编号方式有一些不同。有的版本控制系统对某次 **check in** 涉及到的所有文件设置一个修订版本号，而其他的系统可能给每个文件各自分配一个序列号。CVS 采取后者的方式。举例来说：假设我们从资料 **check out** 三个文件，这三个文件的版本号如下：

```
File1.java    1.10
File2.java    1.7
File3.java    1.9
```

我们编辑了 **File1.java** 和 **File3.java** 两个文件，**File2.java** 保持不变。如果这个时候我们向文件库（Repository）提交这些修改，系统会将变化过文件的修订版本号加 1：

```
File1.java    1.11
File2.java    1.7
File3.java    1.10
```

这意味着你不可以用某个独立的文件版本号，用于跟踪诸如整个项目的发布情况（举例：项目 Orinoco 的版本号 1.3a（译者：也就是说不能用文件版本号作为发布版本号，因为文件版本号和发布版本号并不一一对应））。因为这点，那些刚使用 CVS 的团队常常感到失望。所以我有必要重复一下：CVS 赋予文件的独立的修订版本号不能作为扩展功能的版本号。实际上版本控制系统提供标签（或者其他方式）这样的功能来代替前面所述的扩展版本号。

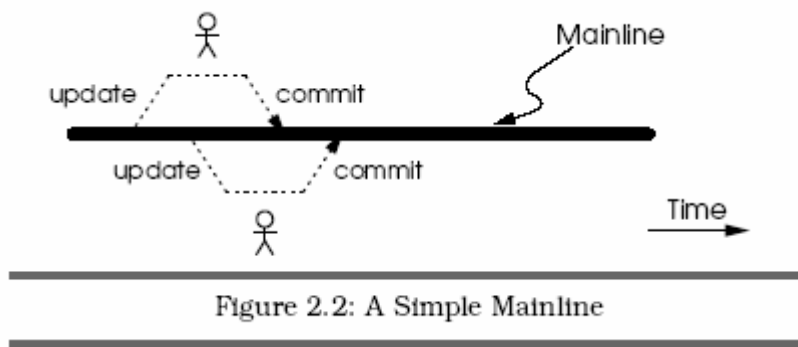
2.6 Tags 标签

尽管这些修订版本很强大，但是人们显然更容易记住“PreRelease2”而不是“1.47”。当不同修订版本号的不同文件组合一起形成某个特定的软件发布的时候，问题就来了。接着前面的例子，我们已准备好将 **File1.java**，**File2.java** 和 **File3.java** 三个文件打包，但是因为每个文件都有一个不同的修订版本号，我们怎么样把这些不同的数字打包到一起呢？

只有使用标签才是解决之道。版本控制系统提供了一个功能：你可以在需要的时候给一组文件（可能是模块，可能是完整的项目）分配一个标签名。如果你给刚才那三个文件分配标签 **PreRelease2**，那么以后可以以相同的名字把它们 **check out**，这样你就获得了版本号

为 1.11 的 File1.java，版本号为 1.7 的 File2.java 以及版本号为 1.10 的 File3.java。

使用标签是跟踪项目的重大历史事件的重要手段。在本书的后部分我们将大量使用标签。标签和分支（下一节的主题）从 86 页开始就有各自专门的章节。



2.7 Branches 分支

在正常的开发过程中，很多人都在一个共用代码上同时同作（尽管他们可能在不同的部分上工作）。他们将资料 **check out**，修改，然后 **check in**，所有人共享工作结果。这条代码线通常被称为主线（mainline），如图 2.2(Figure 2.2)。在这个图（以及后面的所有图）里，时间轴方向是由左到右。粗的水平线代表了代码随时间的变化，这是开发的主线。各个开发人员将主线的代码 **check in** 和 **check out** 到各自的工作间。

不过让我们先考虑一下新的发布就要打包的这个时间。一个小的子团队可能正准备要发布的软件，修改最后一个错误，在发布引擎上工作，帮助 QA。在这个关键时期，他们需要可靠的保证，开发人员的任何修改（比方说为以后的版本添加新功能）都可能让他们的工作前功尽弃。

有个办法就是在生成发布的时候冻结新的开发，但是这样就会让其他人完全闲下来。

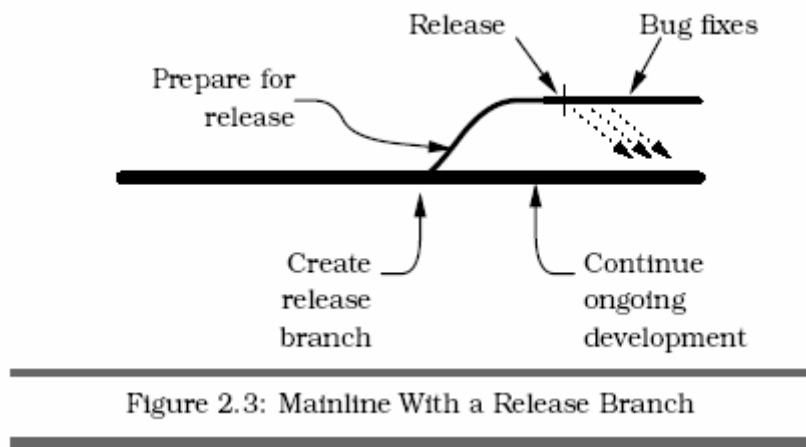
还有一个选择，就是将源代码拷贝到一个空机器上，然后让开发人员只在这台机器上工作。但是如果这样做，如果他们在拷贝后进行的修改又怎么处理？怎么样跟踪这些变化？如果我们发现在发布代码里的错误同样也存在于主线中，那么我们又怎么样有效可靠地将修正部分合并回去？还有就是一旦发布了软件，我们怎么样修改客户发现的错误，我们怎么样保

证能够找到和发布时候相同状态的源码？

更好的选择就是使用版本控制系统提供的分支功能。

分支有点象科幻小说里常见的一种靠事件触发的进行时间分割的机器。从那个点再往前就有两个平行的未来时空。如果某个事件发生了，那么其中的某一个又会被分割。很快你就要面对无数的互斥的宇宙空间（很牛的机器，可以在没有想象力的时候将故事分解，让情节发展下去）。版本控制系统里的分支同样让你创建多个平行的远景，不过里面并不是充斥着异族和宇宙骑士，而是源代码和版本信息。

以一个团队即将发布产品的新版本为例，至此所有的开发成员都在主线上工作，如上一页图 2.2(Figure 2.2)所示。但是发布小分队希望能够和主线独立开来。为了达到这个目的，他们在文件库（Repository）创建了一个分支。从现在直到他们工作结束，发布小分队将在这个分支上进行 check out 和 check in。甚至在应用发布之后，这个分支仍然保持活跃，只要客户报告了错误，他们就要在该发布分支上修补。图 2.3(Figure 2.3)展示了这种情况。



一个分支几乎跟一个完全独立的文件库（Repository）差不多：使用该分支的人员操作和查看源码的操作完全独立于主线上工作的人员。每个分支都有自己的历史变化和独立的修订版本（不过有一点很明显，那就是如果你回到分支创建的那个时间点，会发现分支和主线完全一致）。

这和你创建分支的时候希望的完全一致。在发布分支上工作的分队有一个稳定的代码，

分队在上面进行改进和发布。同时开发的主要人员继续在主线的代码上工作，在发布的时候不再需要冻结代码。还有，当客户报告发布版本上的错误的时候，发布小分队直接访问发布分支，这样既修改了错误，更新了发布，还不会牵扯任何主线上的新代码进来。（译者：新代码意味着潜伏着新的错误）

分支通过标签识别，而且在分支里的文件修订版本号在原来的基础上还有了扩展级。假设你创建分支的时候 `File1.java` 的版本是 `1.14`，那么在分支里面这个文件的版本号变成了 `1.14.2.1`，同时在线里保持 `1.14` 版本号不变。当然如果你在线里对这个文件进行了修改，那么版本就变成了 `1.15`，如果在分支里修改，版本号就变成 `1.14.2.2`。

你可以在分支上再创建分支，不过一般都不需要。我们曾经遇到很多程序员因为一些痛苦的经验而放弃使用分支，因为他们在项目中过度复杂地进行分支。本书里面我们会描述一些简单的模式，不用太复杂就可以应付你需要的所有事情了。

2.8 Merging 合并

回到刚才有很多未来的科幻小说里。为了让情节更有情趣一点，作者通常会让主角通过某个空洞在不同的宇宙间穿梭，`polyphase deconfabulating oscillotrons`, or just a good strong cup of piping hot tea.（待译）

当然你也可以在版本控制里的各个分支间穿梭（要不要茶都无所谓）。尽管每个人都从某个特定的分支里 `check out` 或者 `check in` 某个版本，在某个开发人员的一台机器上可以 `check out` 多个分支的版本（当然是在硬盘的不同目录或文件夹下面）。这样一个开发人员可以在主线上工作的同时，还可以在发布分支上修改错误。

更好的是版本控制系统还支持合并。如果你在发布分支上修改了一个错误，并发现主线上还存在同样的错误。你就可以让版本控制系统找出你修改错误的那些代码，并把其中的变化应用到主线上。这最大限度地减少了在系统的不同版本间进行拷贝和粘贴（修改）的工作。后面我们将会更详细地讨论合并。

2.9 锁定选项

假设 Fred 和 Wilma 这两个开发人员在同一个项目上工作。每个都将项目文件 `check out` 到各自的本地硬盘上，而且两个人都想修改 `File1.java`。如果他们同时进行 `check in` 会发生什么事情呢？

糟糕的一种情况是：版本控制系统接受了 Fred 的修改，接着又接受了 Wilma 对相同文件的修改。因为 Wilma 的版本并没有包括 Fred 的修改，因此最后保存在文件库 (Repository) 里的 Wilma 的版本里根本不会有 Fred 的辛苦工作的成果。

为了阻止这种事情，版本控制系统实现了某种冲突解决系统（也许是个好东西）。有两个很常见的冲突解决版本。

第一种就是严格锁定 (strict locking)。在 strict locking 的版本控制系统里面，所有的被 `check out` 的文件都标示为只读 (read only) 状态。你可以查看这些文件，也可以用于创建应用程序，但是不能编辑和修改。如果你进行修改，就必须得到文件库 (Repository) 的允许：我可以编辑 `File1.java` 文件吗？如果正好没有其他人在编辑这个文件，文件库 (Repository) 机会赋予你修改的权利，并将你的本地文件的权限改为读写状态 (read/write)。然后你就可以编辑了。如果你修改的时候别人也发出修改该文件的请求，他们就会被拒绝。当你完成修改，并把文件 `check in`，你的本地的文件就会重回只读状态，现在轮到别人的编辑了。

第二种冲突的解决方案通常叫做乐观锁定 (optimistic locking)，尽管这其实根本没有使用锁。所有的开发人员能够编辑任何已经被 `check out` 的文件：文件以读写的状态被 `check out`。然而，文件库 (Repository) 不允许你对一个虽然你已经 `check out` 但是被别人更新过的文件做 `check in`。取而代之是，它首先会要求你在 `check in` 之前用文件库 (Repository) 的最新修改版本来更新你的本地文件。这就是这种方案的技巧所在。但是却不是简单地用文件库 (Repository) 的最新版本覆盖你的本地文件，版本控制系统会尝试将文件库 (Repository) 里的修改部分和你的修改进行合并。例如文件 `File1.java`：

```
Line 1  public class File1 {  
-       public String getName() {  
-           return "Wibble";  
-       }  
5       public int getSize() {  
-           return 42;  
-       }  
-   }
```


Wilma 和 Fred 都 check out 这个文件，其中 Fred 对第三行进行了修改：

```
return "WIBBLE";
```

然后他把文件做了 check in。这样 Wilma 的文件就过期了。在不知道 Fred 已经做了 check in 的情况下，Wilma 对第 6 行进行了修改，将返回值由 42 改成了 99。然后她做 check in，版本控制系统告诉她她的文件已经过期了，她需要将文件库（Repository）的变化合并到自己的文件上。下一页的图 2.4(Figure 2.4)表示了这个相当有代表性的冲突。

在 Wilma 合并文件修改的时候，版本控制系统显然很智能，足以分辨出 Fred 的修改并没有覆盖 Wilma 的，因此直接将她的本地文件的第三行进行了更新，而她的修改则得以完整保存在文件中。然后她做了 check in，她的修改不但得到了保存，Fred 的修改也没有受到影响。

如果 Fred 和 Wilma 同时对第三行进行了修改，但是各自的修改不同，这种情况下会出什么事呢？假设 Fred 抢先 check in，他的修改自然先被保存。然后 Wilma 做 check in，她再次获知自己的版本过期了。不过这次她要合并文件库（Repository）的版本的时候发现她修改的那行文件库（Repository）里面也有了变化。看来有冲突了。这个情况下，Wilma 就要查看一些警告信息了，而且她的源文件里的代码冲突的部分将会被标记出来。这个情况下她不得不手工解决这个问题了（解决方法可能是和 Fred 谈谈，找找为什么他们都在同一行代码上进行修改的原因）。

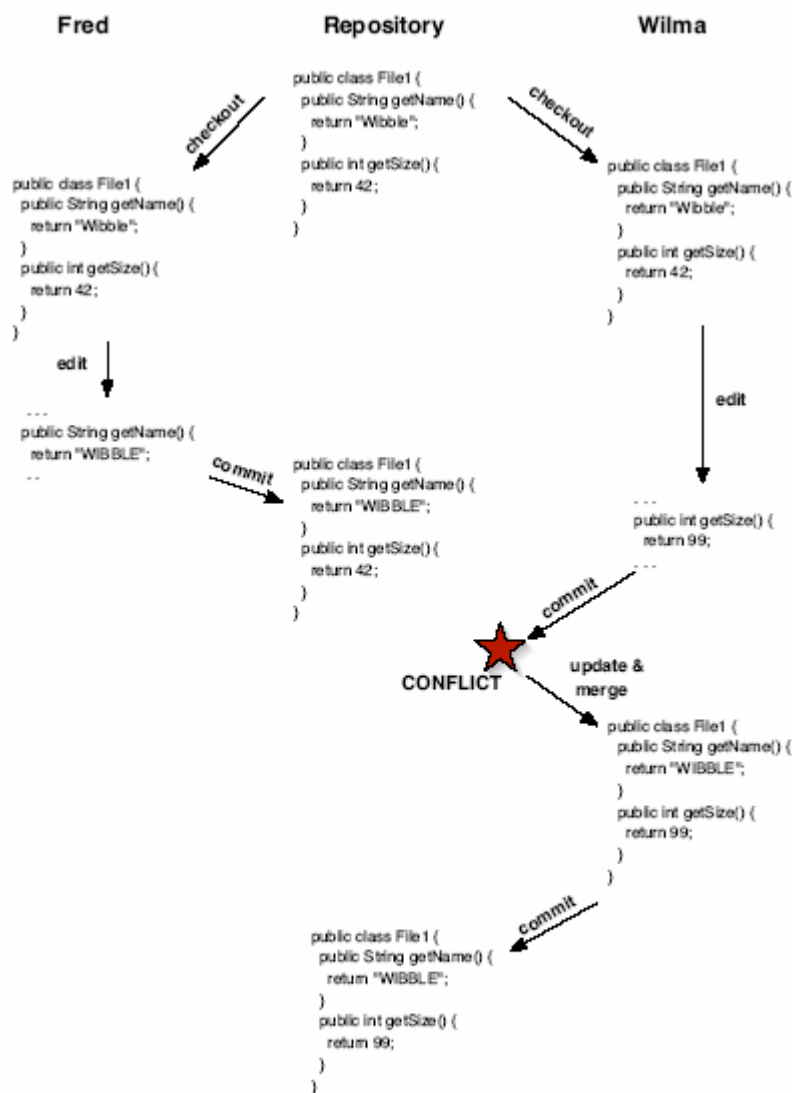


Figure 2.4: Fred and Wilma make changes to the same file, but the conflict is handled by a merge.

通过上面的描述，你可能已经想到乐观锁定是一种有点不顾结果的开发方式；多人同时修改同一文件。通常那些没有尝试过这种方式的人都认为这根本没有用，他们坚持只在实现了严格锁定的版本控制系统上工作。

然而事实是严格锁定导致了更多的无谓的资源争夺，但是却没有特别的回报。一旦你尝试乐观锁定系统（譬如 **CVS**），你就会惊奇地发现很少发生冲突。而且实践证明团队常见的对工作的分割后再分配到开发人员的这种方式，意味着开发人员通常都在代码的不同部分工作，很少有互相干涉的时候。即使他们需要对同一个文件进行编辑，通常也是在这个文件的不同部分。在严格锁定的系统下，某个开发人员必须要等待另一个人完成修改然后 **check in**

之后才能继续。但是在乐观锁定的系统下，这些可以同时进行。在过去两种锁定方式我们都尝试过，强烈推荐大多数团队都使用乐观锁定的版本控制系统。

2.10 配置管理(CM)

有时你听别人说到配置管理系统和软件配置管理系统（一般缩写为 **CM** 和 **SCM**）。第一感觉好像他们在谈论版本控制系统，是不是？的确是，**CM** 实践很大成都依赖于一个适当的好的版本控制。但是版本控制只是配置管理的一个工具。

CM 是项目管理的一套实践，可以让你准确高效的交付软件。要达到这个技术目标，就要通过版本控制，但是也会使用大量的人手控制，以及交叉检查，保证不会有内容被遗忘了。你可以把配置管理当作是识别那些需要交付的内容的一种方法，版本控制呢，可以当作是记录了这些识别物。**CM** 是一个很大（*and to some extent ill-defined*）的课题，在这本书里面就不再详述了。s

从现在开始，我们集中精力到怎么样使用版本控制系统来做好我们的工作。下一章将对某种特定的版本控制系统也就是 **CVS** 进行一个初步介绍。

第三章 入门

在提交你的下一个价值几十万美元的项目到 CVS 之前，让你首先熟悉系统可能是个好主意。本章我们的例子是开发和维护一个小项目，这些都要在一个真实的 CVS 文件库（Repository）上工作。通常来说使用 CVS 的第一步都是非常困难的。

- 首先你需要在你的机器上安装 CVS
- 而且在你使用文件库（Repository）做 check in 某个项目之前，该文件库（Repository）必须先设置好，你必须亲力亲为。

当然和文件库（Repository）进行交互的时候你也有许多选择。你可以选择传统的 CVS 命令行工具，也可以选择使用 GUI 的前端，当然还可以使用你的 IDE 的内置功能。

最后，根据你使用的不同的操作系统，还有很多不同的地方。我们会一边继续一边标志出这些重点。到某个适当的时候我们会把所有的东西总结到一起。

3.1 安装 CVS

很明显在运行 CVS 程序前要先装好软件。该软件除了能管理资料之外，同时还提供了一个命令行工具，用于访问文件库（Repository）。

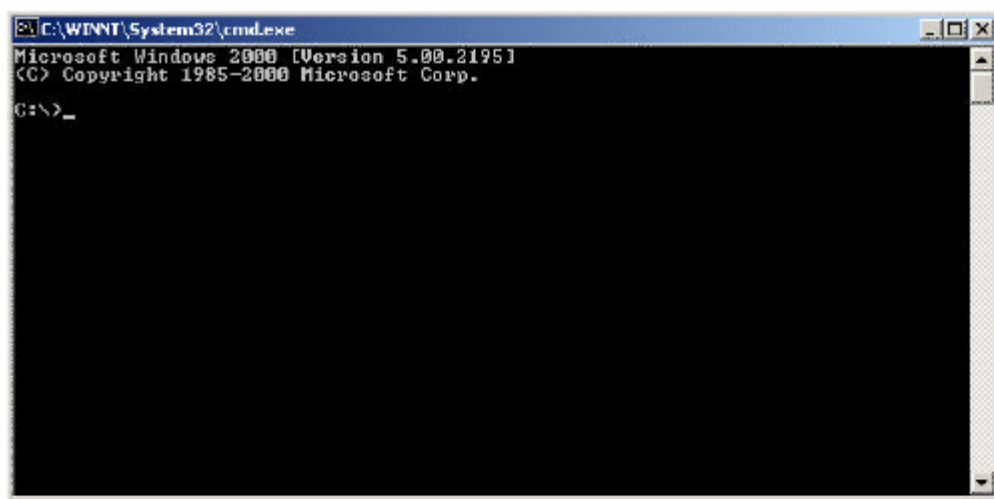


Figure 3.1: Windows command window

第一步就是判断一下是否你的机器已经装上 CVS 了。最简单的方式就是命令行。如果你对命令行很熟悉，你可以跳过这一节。

命令行

命令行是一种直接在计算机上运行你的命令的一种低层次的简便的工具。命令行也是一种强大的工具，但是其含义也相当模糊，如果你使用命令的方式，可能要在房间里昼夜工作才行。

在 Windows 上，你可以使用开始/运行，然后键入程序的名字 `cmd`，这样就会打开一个命令行窗口（在一些老的 Windows 系统上，可能要键入 `command` 来代替 `cmd`）。然后你应该看到类似图 3.1(Figure 3.1)所示的窗口。

在 Unix 上，你可能已经在命令行模式下工作了。如果你安装了桌面环境，如 Gnome 或者 KDE，可以查找到 `terminal`，`konsole` 或者 `xterm` 应用程序，然后运行它。这样你就会看到类似图 3.2(Figure 3.2)的窗口（如果你使用的是 Mac OS X，你的 `shell` 程序隐藏在应用/工具下面）。

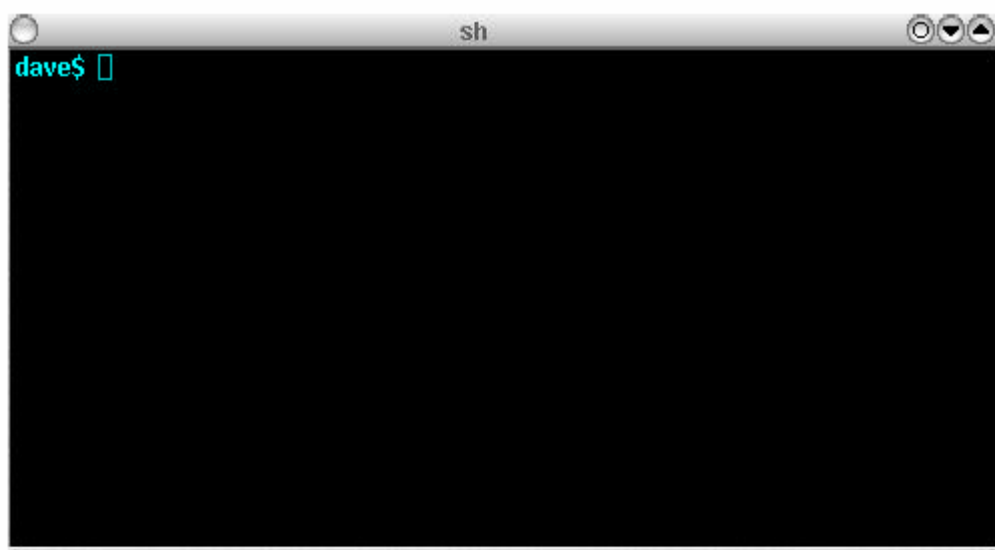


Figure 3.2: Unix command window

你可以在命令行窗口输入命令然后观察命令的输出结果，这些命令是没有 GUI 界面的。例如在刚才创建的命令行窗口里面你可以输入下面的命令，然后按回车（Enter 或者 Return）键。

```
echo Hello
```

你会看到显示 **Hello** 文字，在下面是新的一行提示符，你可以接着输入另外的命令。如图 3.3(Figure 3.3)。



Figure 3.3: Window after executing “echo hello”

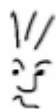
提示符

使用命令行窗口的一个乐趣是你可以根据 **shell** 提供给你的默认的提示符。你可以将时间，当前目录名，用户名以及其他所有的基本的信息都显示在提示符里。不过有点不好的是，这种弹性的设计导致了一点混乱，在看一下上一页的截屏，你会发现 **Windows** 的提示符跟 **Unix** 的提示符完全不一样。

在这本书里，我们会尽力简化我们的例子，例子里使用的提示符将标准化和通用化。我们使用这样的格式：在当前目录名后接一个 **>** 符号。例如下面的例子：

```
work> cvs update
```

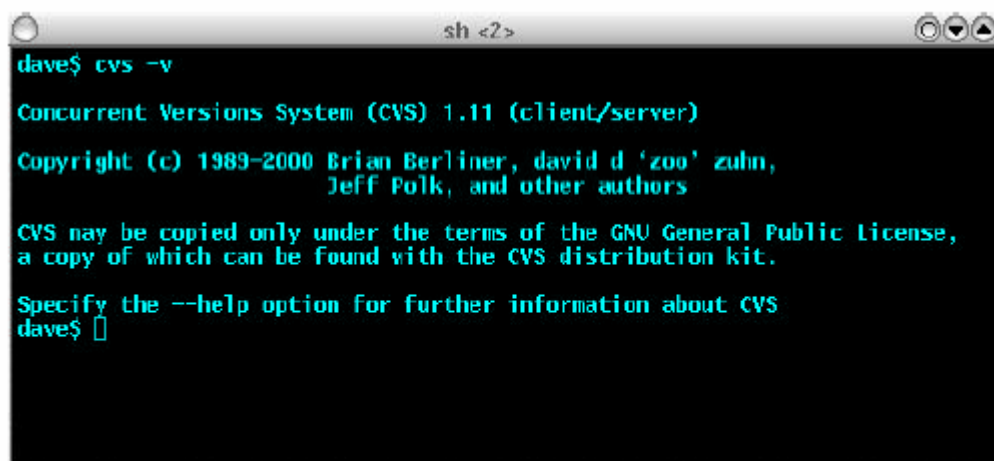
这个的含义是我们在名为 **work** 的目录下运行命令 **cvs update**。将这种提示符应用到你实际使用的操作系统的命令行窗口下应该很简单。



本书使用的命令并非 Windows 或者 Unix 专用的，实际上这些命令在两个操作系统都能工作。唯一的不同是文件的名字：Windows 的文件名使用驱动器的名字以及\符号，而 Unix 使用/符号。只要使用恰当的文件名，一切都可以工作的很好。

CVS 已经安装好了吗？

打开你的计算机的命令行窗口并输入命令"`cvs -v`"（记得按回车键）。如果 CVS 已经正确地安装在你的机器上了，会看到类似于下一页的图 3.4(Figure 3.4)的界面，你可以直接跳到下一节继续学习。



```
sh <2>
dave$ cvs -v
Concurrent Versions System (CVS) 1.11 (client/server)
Copyright (c) 1989-2000 Brian Berliner, david d 'zoo' zuhn,
                        Jeff Polk, and other authors
CVS may be copied only under the terms of the GNU General Public License,
a copy of which can be found with the CVS distribution kit.
Specify the --help option for further information about CVS
dave$
```

Figure 3.4: Determining the CVS Version

如果你的机器上没有装 CVS，就需要安装一下。安装还有点技巧，但是这取决于你的操作系统，还有公司政策（如果你在公司的机器上运行本书提供的练习）。可以直接参考 CVS 的官方网站，这比重新发明轮子要好（译者：也就是自己琢磨怎么安装）。在 CVS 的网站上你可以找到并下载你想要的资料和 CVS 软件。如果你是 Windows 用户，你可以在

cvshome 的下载页找到已经创建好的二进制的安装文件，如果你是 **Unix** 用户，你可以使用网站提供的源码自己创建安装程序，或者（如果你愿意）你可以下载和你的系统对应的已经打好包的二进制文件（例如如果你使用的是 **RedHat** 操作系统，网站提供了 **RPMs** 安装文件）。不管你使用什么样的操作系统，都要记得将 **CVS** 程序路径设置在 **PATH** 环境变量里，这样你才可以在命令行使用 **CVS**。OK，那我们就等你安装了。。。。。

3.2 创建文件库（Repository）

CVS 运行的时候需要文件库（**Repository**）。这一步我们将创建一个我们需要的文件库（**Repository**）。

现在你可能已经能够访问 **CVS** 文件库（**Repository**）了，也许你的公司已经安装了一个。现在开始我们不再关心怎么样安装 **CVS**，而是开始运行我们的 **CVS**。这样我们才能自由自在地进行下去，而不用担心那些乱七八糟的事情。你也许想把文件库（**Repository**）留一段时间。有时候，当你需要通过实验来获知怎么样做某个事情最优的时候，在一个安全的沙箱（**sandbox**）里做再好不过了。

在创建文件库（**Repository**）前你还要下一个决定：到底把文件库（**Repository**）放在硬盘的什么位置？你机器上的 **CVS** 文件库（**Repository**）由一个具有层次结构的文件和目录（有规则的文件系统）组成。所有你需要做的就是告诉 **CVS** 这个层次结构的最顶点的位置。后面的例子我们将建设你使用目录 **sandbox** 作为你的文件库（**Repository**）。Windows 的用户可以参考 **C:\sandbox** 目录，Unix 的用户可以参考他们的 **home** 目录的 **~/sandbox** 目录。

最简单的创建文件库（**Repository**）的方式是在命令行运行 **cvs init** 命令：

```
Unix:      cvs -d ~/sandbox init
Windows:   cvs -d C:\sandbox init
```

参数 **-d** 告诉 **CVS** 文件库（**Repository**）的位置（注意是小写 **-d**）

Destination known fact that .repository. starts with a silent and invisible

letter .d.). You can think of the -d option as defining the destination of CVS commands.

只要你愿意，你可以列出刚才创建的文件库（Repository）目录下的内容，你会发现里面包括了一个名为 CVSROOT 的子目录（该目录下有一些管理文件）。恭喜恭喜，你现在已经是一个 CVS 管理员了。

现在我们要给文件库（Repository）添加一个项目。但是你要牢牢记住不要直接在文件库（Repository）里创建文件，你只能通过 CVS 命令来操作文件库（Repository）。

3.3 CVS 命令

到现在为止我们还一直通过命令行模式使用 CVS。如果你喜欢漂亮一点，好多开发者已经为 CVS 写了 GUI 的界面，这样你不用在命令行上而是通过点击就可以完成同样的操作。一些开发人员喜欢这种界面风格，另外的则喜欢底层命令。

在开源的产品里面，最流行的前端应用非 WinCvs 和 Tortoise CVS 莫属。特别是后者非常有趣，因为它支持在 Windows 浏览器上使用 CVS 客户端。还要提到的是在 Tortoise CVS 网站上，有个支持在 Visual Studio 使用 CVS 的项目，不过这个项目直到 2003 年 9 月还处于 beta 测试阶段。

如果你通常使用 IDE 环境进行开发，你应该检查一下你的 IDE 是否直接支持 CVS。实际上很多都支持（包括最流行的开源 IDE 工具 Eclipse），这样可以节省很多时间。你可以详细查查你使用的 IDE 的文档。

说完这些有意思的前端工具，最重要还是要记住理解底层的命令行接口（译者：也就是 CVS 命令）是很重要的，当你需要让你的开发任务自动化的时候，自动化工具需要直接同文件库（Repository）打交道，而这是通过命令行接口进行的。出于那个原因（），下面我们将向大家展示 CVS 的命令行接口。

3.4 创建一个简单的项目

这一节我们要往文件库（**Repository**）里放一个新项目（每个好的项目都有一个名字，我们把这个项目叫做 **Sesame**）。我们要创建两个文件，然后把这两个文件导入（**import**）文件库（**Repository**）的项目里面。（项目的正式名 **Sesame** 是以大写字母 **S** 开头的，但是再文件库（**Repository**）里的项目名一般都用小写字母）

那么，假设我们马上就要开始在 **Sesame** 上工作了。不过现在文件库（**Repository**）里还没有项目，因为我们还没有什么东西可以放进去了。动手吧。首先在你的机器上创建一个临时目录（我们用的是 **tmpdir**）。然后，用你最喜欢的编辑器或者 **IDE**，在这个目录下创建两个文件：**Color.txt** 和 **Number.txt**。

文件 **Color.txt**:

```
black  
brown  
red  
orange  
yellow  
green
```

文件 **Number.txt**:

```
zero  
one  
two  
three  
four
```

唔，我好像听到有人说这一点都不象程序的源代码。记住：文件库（**Repository**）是可以保存我们项目里的所有东西的。说不定我们在写一个儿童的教育程序呢，这些文件包含的是某个程序的数据（假设这样吧）。

现在我们要通知 **CVS**：我们要把这些文件导入文件库（**Repository**）的一个新项目里

了。想要完成这个事情，就要使用 `cvs import` 命令了。不幸地是我们在本书前还没有使用过 `import` 呢，这可是 CVS 命令里最重要的参数哦。不过我们现在还不会涉及太多细节，在第 99 页我们才会详细的讲到 `cvs import`。

为了导入刚才我们创建的两个文件，首先要定位到包括它们的临时目录下。

```
tmpdir> cvs -d C:\sandbox import -m "" sesame sesame initial
N sesame/Color.txt
N sesame/Number.txt
No conflicts created by this import
```

如果是 Unix 则是：

```
tmpdir> cvs -d ~/sandbox import -m "" sesame sesame initial
N sesame/Color.txt
N sesame/Number.txt
No conflicts created by this import
```



这里的参数 `-d` 和 `cvs init` 命令里的含义相同。它让 CVS 知道文件库 (Repository) 在哪里。关键字 `import` 则让 CVS 知道我们要导入一个项目。

Message a project. The `-m` and the empty string that follow it lets you associate a log message with this import. There are circumstances where this is useful (particularly when dealing with third-party code), but for now an empty log message is fine.

跟在后面的参数 `sesame` 是文件库 (Repository) 中项目的名字。这个名字也是你将来要引用到的，所以一定要选好。

后面两个参数是标签，我们现在暂时不管它们。

注意到在 CVS 做 `import` 的时候，还对做了什么记录在日志。对于我们这个例子，它显示了我们的两个文件的名称，名称前面还有一个字符 `N`。这代表它们都是新文件的意思，当然还表示这两个文件已经加入到文件库 (Repository) 了。

现在我们已经把这些文件安全地保存在文件库 (Repository) 里了。如果我们够猛（或者够傻），现在就可以把临时目录下的所有文件给删了。不过那些谨慎的（也是实际的）开

发人员可能会在删除之前先验证一下：我们的文件到底是不是正确地保存在文件库（Repository）了。最容易的方式就是从 CVS 中 `check out` 项目 `Sesame` 的文件，保存到你的本地工作目录下。一旦确定文件一个不少，而且看上去都对了，就可以删除原来的文件了。下一节就告诉你怎么做。

3.5 开始项目工作

你是不是在新项目（比方说我们刚才创建的项目 `Sesame`）上工作，或者是加入一个运行了数个月，拥有上千个文件的项目，这些都不重要。刚开始对于项目文件的操作都一样。

- 决定将工作文件保存在你本地机的位置。
- 将项目从文件库（Repository）`check out`，保存在本地

第一个决定相当简单：我们一般在机器上有个一个目录叫做 `work`。然后将所有的项目文件 `check out`，放在这个目录下。对于简单的项目，我们一般直接 `check out`，然后放在 `work/` 下。对于那些更复杂的，就要创建一个本地动作空间（workspace）。现在假设我们在一个简单的项目上工作。如果我们有三个独立的项目，分别叫做 `poppy`，`sesame` 和 `sunflower`，`check out` 到本地，最后的目录结构如图 3.5(Figure 3.5)。

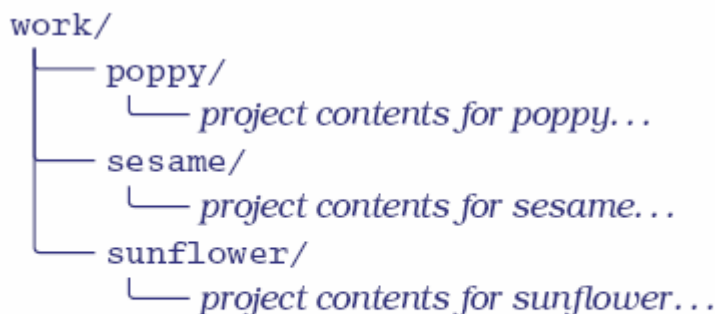


Figure 3.5: A Checked-out Tree

如果你还没有工作空间，那就开始创建一个工作目录，可以直接从命令行或者用文件管理器（File Manager）。

```
Unix:      mkdir ~/work
Windows:  mkdir \work
```

现在我们要把项目 `check out` 到这个目录下。选项 `-d` 让 `CVS` 知道去哪里找文件库 (Repository), `co` 就是 `check out` 的意思, `sesame` 是项目名。

```
Unix:      cd ~/work
           cvs -d ~/sandbox co sesame
Windows:   cd \work
           cvs -d C:\sandbox co sesame
```

然后你应该能看到类似下面的输出:

```
work> cvs -d ~/sandbox co sesame
cvs checkout: Updating sesame
U sesame/Color.txt
U sesame/Number.txt
```

现在你已经拥有 `Sesame` 项目的一份本地拷贝, 包括了刚才我们在初始阶段导入的两个文件。现在开始, 我们可以在这些文件上工作了, 因为这些文件已经在 `CVS` 的管理之下。在检查了文件的正确性之后, 我们删除了临时目录下的原始文件。并把这些文件的控制权移交给我们的版本控制系统, 如果同时保留机器上的原始文件和 `CVS` 管理的文件就太容易搞混淆了。我们将 `sesame` 作为当前目录, 并在此下面操作那些已经 `check out` 的文件。

3.6 进行修改

尽管我们已经很辛苦的工作, 客户还是抱怨, 我们提供的颜色跟完整的调色板差太远。于是, 打开你最喜欢的编辑器, 在文件后面添加了四行内容。

文件 `Color.txt`:

```
black
brown
red
orange
yellow
green
blue
purple
gray
white
```

add these lines

在我们保存修改之前, 先看看 `CVS` 是怎么认定我们项目的状态的。我们可以使用 `cvs`

`status` 命令来查看一个或多个文件的状态。

```
work/sesame> cvs status Color.txt
=====
File: Color.txt          Status: Locally Modified
Working revision:       1.1 Thu Apr 17 17:03:13 2003
Repository revision:    1.1 /Users/.../sesame/Color.txt,v
Sticky Tag:             (none)
Sticky Date:            (none)
Sticky Options:         (none)
```

这里重要的一行是状态这行：CVS 识别出这个文件已经在本地修改过了（但是这些袖管改还没有保存到文件库（Repository））。

如果我们总是一小步一小步工作，要记住对个别文件的修改还是很容易的。不过一旦我们忘记了某个文件为啥被修改（或者我们只想反复检查），就可以使用 `cvs diff` 命令来显示本地文件和其文件库（Repository）里的版本之间的区别。

```
work/sesame> cvs diff Color.txt
Index: Color.txt
=====
RCS file: /Users/dave/sandbox/sesame/Color.txt,v
retrieving revision 1.1
diff -r1.1 Color.txt
6a7,10
> blue
> purple
> gray
> white
```

命令执行后输出了很多信息。其中第一行告诉我们正在做检查的文件名。这有两个作用，第一，如果我们使用命令检查多个文件，这可以帮助我们确定当前的位置，还有就是在打补丁的时候要用的（不过那不是我们现在要看的）。

在=号组成的一行的下面三行则告诉我们文件库（Repository）文件的名称和版本号，以及用于生成文件区别的低层命令。

6a7,10 有点神秘，其实是在第 6 行后面从 7 行到 10 行都是我们新添加的。跟着，后面的日志每行都以>符号开头，表示我们实际添加的内容。CVS 还有一个功能是以两边对比的形式同时显示本地和文件库（Repository）上的文件。

```
work/sesame> cvs diff --side-by-side Color.txt
Index: Color.txt
=====
RCS file: /Users/dave/sandbox/sesame/Color.txt,v
retrieving revision 1.1
diff --side-by-side -r1.1 Color.txt
black                                black
brown                               brown
red                                 red
orange                             orange
yellow                             yellow
green                              green
                                   > blue
                                   > purple
                                   > gray
                                   > white
```

CVS GUI 前端工具具有一个独一无二的优势，那就是如果你使用这样的工具，你会发现你能够使用不同颜色来显示文件之间的区别。

3.7 更新文件库（Repository）

修改完成（当然也经过了测试），我们准备好要将我们的最新版本保存到文件库（Repository）。在一个只有一个人的项目（例如 Sesame）里，这非常简单。用 `cvs commit` 命令就可以了。

```
work/sesame> cvs commit -m "Client wants 4 more colors"
cvs commit: Examining .
Checking in Color.txt;
/Users/dave/sandbox/sesame/Color.txt,v <-- Color.txt
new revision: 1.2; previous revision: 1.1
done
```

提交（commit）功能用于向文件库（Repository）保存我们所做的修改，选项 `-m` 用于附加一个对修改的说明信息。如果你没有指定 `-m` 选项，CVS 会弹出一个编辑窗口，让你输入信息，这对于第一次使用这个的来说有点措手不及。

尽管如此，我让 CVS 提交项目 Sesame 的所有文件，CVS 很智能，发现文件 `Number.txt` 并没有改变，因此只有文件 `Color.txt` 更新新版本（1.2）。

提交之后，查询状态功能显示文件库（Repository）真正更新了。

```
work/sesame> cvs status Color.txt
=====
File: Color.txt           Status: Up-to-date
Working revision:         1.2 Thu Apr 17 17:26:17 2003
Repository revision:      1.2 /Users/.../sesame/Color.txt,v
Sticky Tag:               (none)
Sticky Date:              (none)
Sticky Options:           (none)
```

我们也能查看文件的历史变化（CVS 称这为文件的日志 log）

```
work/sesame> cvs log Color.txt
RCS file: /Users/dave/sandbox/sesame/Color.txt,v
Working file: Color.txt
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;      selected revisions: 2
description:
-----
revision 1.2

date: 2003/04/17 18:24:56; author: dave; state: Exp; lines: +4 -0
Client wants 4 more colors
-----
revision 1.1
date: 2003/04/17 17:11:36; author: dave; state: Exp;
=====
```

3.8 冲突发生的时候

每个人第一次听说 CVS 不支持文件锁定编辑的时候都不安起来。他们担心如果两个人同时编辑同一个文件的时候会发生什么事情。这一节我们会解除所有疑点（希望这次你的不安能够消除）。要做到这点我们需要另一个用户（这样我们才能有多个用户同时编辑一个文件）。不幸的是，我们克隆人工具正在运行，这样只好请你来模拟了。

在处理冲突的时候，CVS 并不知道用户情况，它只关心在不同的工作空间的文件是否一致。这意味着我们只需要简单的 **check out** 新的一份项目文件就能模拟出第二个用户，我们只需要放在和第一份不同的空间就可以了。我们第一次 **check out** 我们的项目的时候，CVS 将它放在 **sesame**（也就是项目名）目录下。要重新 **check out**，我们需要改变默认值。

只有一条规则：别把第二份项目文件 **check out** 到当前的 **sesame** 目录下。可以 **check out** 到和当前目录并行的目录下，假设是叫 **aladdin**。要做到这点，需要使用 **-d** 选项来指定新的目录名。

```
Unix:      cd ~/work
           cvs -d ~/sandbox co -d aladdin sesame
Windows:   cd \work
           cvs -d C:\sandbox co -d aladdin sesame
```

CVS 输出后面的结果：

```
cvs checkout: Updating aladdin
U aladdin/Color.txt
U aladdin/Number.txt
```

首先让我们先做一个快速的全面检查：改变一个目录下的一个文件，**check in**，然后让 CVS 更新另一个目录的本地文件。

首先，编辑 **sesame** 目录下的 **Number.txt** 文件，添加两行（**five** 和 **six**）：

文件 **Number.txt**:

```
zero
one
two
three
four
five
six
```

现在把文件 **check in** 文件库（Repository）

```
work/sesame> cvs commit -m "Customer needed more numbers"
cvs commit: Examining .
Checking in Number.txt;
/Users/dave/sandbox/sesame/Number.txt,v <-- Number.txt
new revision: 1.2; previous revision: 1.1
done
```

在 **aladdin** 目录下的 **Number.txt** 已经过期了（因为文件库（Repository）里面已经有了更新的版本）。现在让我们去检查一下。

```
work/sesame> cd ../aladdin
work/aladdin> cvs status Number.txt
=====
File: Number.txt      Status: Needs Patch
Working revision:     1.1 Thu Apr 17 17:11:36 2003
Repository revision:  1.2 /Users/.../sesame/Number.txt,v
Sticky Tag:           (none)
Sticky Date:          (none)
Sticky Options:       (none)
```

Status 这一行告诉我们：我们的文件需要修补（Needs Patch）更新最新版本。在此之前，我们要让 CVS 告诉我们在我们的版本和文件库（Repository）的当前版本之间的区别（如果更行文件库（Repository）可能会影响你当前工作的材料，你就会推迟更新）。我们再一次用到 `cvs diff` 命令。

```
work/aladdin> cvs diff -rHEAD Number.txt
Index: Number.txt
=====
RCS file: /Users/dave/sandbox/sesame/Number.txt,v
retrieving revision 1.2
retrieving revision 1.1
diff -r1.2 -r1.1
6,7d5
< five
< six
```

选项 `-rHEAD` 告诉 CVS：我们想比较拿我们本地的 `Number.txt` 和文件库（Repository）的最新版本比较（head 分支）。在 `6,7d5` 这一行后面，添加了两行（不再有什么惊奇的了）。如果我们没有指定 `-r` 标志，CVS 将拿我们的本地文件和文件库（Repository）的版本比较（这个例子里是 1.1 版本）。因为我们并没有改变 `Aladdin`，所以不会显示出有什么改变。

我们能够将 `sesame` 下的修改合并到 `aladdin` 下。

```
work/aladdin> cvs update
cvs update: Updating .
U Number.txt
```

这段表明 CVS 已经更新了本地的 `Number.txt` 文件。如果查看一下，会发现已经多出了两行。

3.9 解决冲突

那么，如果两个人同时编辑同一个文件会发生什么事呢？最终会出现两个情况。第一个种是各自的修改没有交迭的地方。模拟这种情况不需要太大力气，如下。

首先，编辑 `sesame` 目录下的 `Number.txt` 文件，将第一行的字母都改为大写。

文件 `Number.txt` (in `sesame`):

```
ZERO
one
two
three
four
five
six
```

现在编辑 `aladdin` 目录下的 `Number.txt` 文件。这次将文件的最后一行的字母都改为大写。

文件 `Number.txt` (in `aladdin`):

```
zero
one
two
three
four
five
SIX
```

我们刚才做的就是模拟两个用户改变同一文件的情况。现在这些修改都是独立的，因为文件库（**Repository**）知道两个用户的情况，我们可以向文件库（**Repository**）提交修改了。投币决定先让 `Aladdin` 将他修改的版本 `check in`。

```
work/aladdin> cvs commit -m "Make 'six' important"
cvs commit: Examining .
Checking in Number.txt;
/Users/dave/sandbox/sesame/Number.txt,v <-- Number.txt
new revision: 1.3; previous revision: 1.2
done
```



过了一小会儿，`sesame` 也 `check in` 了。（注意，这个版本是将第一行的字母改为大写）

```
work/sesame> cvs commit -m "Zero needs emphasizing"
cvs commit: Examining .
cvs commit: Up-to-date check failed for 'Number.txt'
cvs [commit aborted]: correct above errors first!
```

啊哈！CVS 使用了 errors 这个字眼，而且最后还以感叹号结束了。真是厄运。

或许不是。让我们试试 CVS 暗示的，用文件库（Repository）将我们的本地版本更新。

记得我们的文件有大写的 zero，而文件库（Repository）的是大写的 six。

```
work/sesame> cvs update
cvs update: Updating .
RCS file: /Users/dave/sandbox/sesame/Number.txt,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into Number.txt
M Number.txt
```

注意附加信息。CVS 告诉我们不能简单地更行我们的本地文件，取而代之，CVS 合并了文件库（Repository）和我们的修改。现在让我们看看合并后的本地版本：

文件 Number.txt:

```
ZERO
one
two
three
four
five
SIX
```

太神奇了！现在我们的版本同时包括了我们自己的修改和 Aladdin 的修改。同时编辑同一个文件，CVS 解决了这个问题。

在得意洋洋之前，记住我们的本地修改还没有保存到文件库（Repository）呢。我们让 CVS 提交我们的修改，成功了，因为我们的本地版本已经包括了文件库（Repository）里的最新的变化。

```
work/sesame> cvs commit -m "Zero needs emphasizing"
cvs commit: Examining .
Checking in Number.txt;
/Users/dave/sandbox/sesame/Number.txt,v <-- Number.txt
new revision: 1.4; previous revision: 1.3
done
```

下次 Aladdin 更新的时候，他也会跟我们一样。

```
work/sesame> cd ../aladdin
work/aladdin> cvs update
cvs update: Updating .
U Number.txt
```

Butting Heads.When Changes Clash

前面的例子里面，两个虚拟的开发人员的修改并没有交迭。如果两个开发人员同时修改同一文件的同一行又会怎么样呢？让我们来找找答案吧。

定位到 **sesame** 目录下，把文件 **Number.txt** 的第二行 **one** 改为 **ichi**，先不急着把这个 **check in**，先到 **aladdin** 目录下，将同样一行从 **one** 改为 **uno**，然后假设 Aladdin 又一次得到 **check in** 的优先权。

```
work/aladdin> cvs commit -m "User likes Italian one"
cvs commit: Examining .
Checking in Number.txt;
/Users/dave/sandbox/sesame/Number.txt,v <-- Number.txt
new revision: 1.5; previous revision: 1.4
done
```

现在回到 **sesame** 目录下。记住我们是在假设模拟两个独立的用户，我们要假装不知道 Aladdin 所做的修改，而且还要尽力把我们的修改 **check in**。

```
work/sesame> cvs commit -m "One should be Japanese"
cvs commit: Examining .
cvs commit: Up-to-date check failed for 'Number.txt'
cvs [commit aborted]: correct above errors first!
```

这个消息以前就见过了，意思是我们需要先从文件库（Repository）更新。

```
work/sesame> cvs update
cvs update: Updating .
RCS file: /Users/dave/sandbox/sesame/Number.txt,v
retrieving revision 1.4
retrieving revision 1.5
Merging differences between 1.4 and 1.5 into Number.txt
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in Number.txt
C Number.txt
```

现在这次有点惊慌：CVS 告诉我们在将其文件库（Repository）的版本合并到我们的本地修改的时候发生了冲突。我们的努力白费了吗？不

冲突发生的时候，主要是因为两个开发人员发生了歧义。在这个例子里面，一个开发人员想将这一行改为意大利文，另一个却想改为日文。想一下就很明白了，这是交流上的中断引起的，团队里有问题（至少是在团队的过程里）。不管什么原因，我们剩下就是：这一行到底应该是什么样的？CVS 并没有客户热线，因此 CVS 也解决不了问题，而是在文件里添加了特别的注释来表示冲突是什么。在这个例子里如果看看 Number.txt 的内容，就会看到：

文件 Number.txt:

```
ZERO
<<<<<< Number.txt
ichi
=====
uno
>>>>>> 1.5
two
three
four
five
SIX
```

<<<<<< 和 >>>>>>之间表示冲突发生的位置。在中间我们能同时看到自己的修改和与文件库（Repository）冲突的那部分变化。时间可以验证一切。第一件要做的就是找出谁修改了文件库（Repository）的版本。Cvs log 命令能够显示一个或多个文件的历史变化，它能帮助我们找出发生了什么事情。

```
work/sesame> cvs log -r1.5 Number.txt
RCS file: /Users/dave/sandbox/sesame/Number.txt,v
Working file: Number.txt
head: 1.5
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 5;      selected revisions: 1
description:
-----
revision 1.5
date: 2003/04/22 18:22:07;  author: dave;  state: Exp;  lines: +1 -1
User likes Italian one
=====
```

最后两行，我们可以看到进行修改的作者的名字，以及做 check in 的时候加的注释。

We

wander over and ask him about the change. 还是给客户打了个电话才解决了这个问题：客户希望单词 **one** 是用日文，而单词 **two** 用意大利文。**Aladdin** 肯定听错了。有了这个信息，现在我们就能够解决问题了。编辑 **sesame** 目录下的 **Number.txt** 文件，首先删除 CVS 关于冲突的记号，然后根据用户需要进行修改：

文件 **Number.txt (sesame)**:

```
ZERO
ichi
due
three
four
five
SIX
```

在解决了冲突之后，现在我们能够提交文件了。

```
work/sesame> cvs commit -m "One is Japanese, two Italian"
cvs commit: Examining .
Checking in Number.txt;
/Users/dave/sandbox/sesame/Number.txt,v <-- Number.txt
new revision: 1.6; previous revision: 1.5
done
```

CVS 竟然帮我们发现了一个理解错误。解决了这个问题，打击都很高兴。乐观锁定完全名副其实。而且，只是一点点惊慌而已，我们还需要强调这种冲突在真实的项目里发生的机会非常小。

然而，还有一点值得注意的是 CVS 并不会心灵感应。两个人以完全不同的方式修改同样的错误很有可能发生。如果这种修改没有引起冲突，CVS 完全能够愉快的接受两者，甚至他根本搞不清楚两者的修改是同样的代码。没有冲突以为着你根本不会践踏别人的修改（文本级），但是你还是应该坚持依靠单元测试来保证这些修改有效。

这就是我们的 CVS 快速之旅的所有内容。不过你应该保留你的沙箱(sandbox)文件库 (Repository)。后面，你会发现你要试验一些特别的功能的时候，在真实的项目文件库 (Repository) 里实践之前先在这个里面实践还是很有帮助的。

第四章 怎么样版本控制？

尽管版本控制在理论上听起来很美，但是很多团队都没有采用。因为有时候理论并不能转化为实践。当然读读那些写了什么诸如生成发布分支的书籍也是很好的，不过玩玩这些书最后就成了讲怎么样把 CV 的命令敲对。

另一个问题是有些团队有时候过度滥用版本控制系统，关键了过于复杂的结构来管理源码，最后即使是最简单的任务也要通过一串复杂的用法才能完成。结果怎么样？这些团队最终还是放弃了（根据我们的经验是很快就放弃），使用版本控制系统看起来还是很值得争辩的。

本书的其余章节将着笔于这些问题。将通过一种简单的方式来组织你的版本控制系统，以及向大家展现一套应付一个团队每天都需要做的所有事情的基本实践。我们一开始就假设你将这些基本的实践作为一套诀窍来用，只要你想做出点成绩就要按照它们的指示来做，尽量不要太偏移这些诀窍。如果你发现遇到我们没有谈到过的情况，在做之间一定要好好想想。也许你并不需要那样。

通过任何诀窍，你很快就会发现使用它们越来越舒适。是时候慢慢展开实践了。建设你不打算在实际的项目文件库（Repository）中直接实践一些新用法，可以在测试文件库（Repository）（例如我们上一章那样的）里面配置好，然后在里面做。

4.1 我们的基本哲学

stop doing it).我们认为版本控制是三个基本技术实践之一，每个团队都需要精通这三个技术（另外两个是单元测试和自动化）。每个团队都应该在任何时候使用版本控制系统管理他们任何产出。我们会尽力使它们简单化，明显化，和轻量化（如果不这样，大家最终还是会放弃）。

简单化的意思就是做事情的时候能有多简单就多简单。Check in 是一个简单的操作，最基本的操作应该只有一两个动作。创建一个新的定制的发布是一个相对复杂点的概念，因此步骤多点都没关系。当时也应该尽量简单。版本控制也应该是明显的，我们需要分配事情，这样我们所做的以及我们正在处理的软件版本都应该是清晰的。在直接面对源码的时候不应该有任何需要猜测的东西。

最后，我们正在描述的是轻量级的进程，我们并不希望版本控制阻碍我们正常工作。

4.2 组织版本控制系统

组织 CVS 文件库（Repository）中的源码的基本规则如下：

- 在开始前，你需要建立一个安全有效的访问文件库（Repository）的方式。
- 一旦获得连接，有一套简单的 CVS 命令是你每天都要用的。
- 你们公司开发的项目都应该以 CVS 模块的形式被访问。你只能通过一个点将项目的所有源码 check out。
- 如果项目下包括了可以独立工作的子组件，如果你希望在项目间共享组件，这些组件应该保存在模块里。
- 如果你的项目和第三方（如供应商或者开源项目）代码合作，需要将这些作为资源进行管理。
- 开发人员应该及时地使用标签来标识出具有重要意义的事件，包括发布，修复错误，以及重要代码试验开始的时刻。

第五章 访问文件库 (Repository)

在描述所有独特的 CVS 技术前，有一个重要的步骤：你真真切切需要能够访问文件库 (Repository)。

至此我们所有的试验都是在本地文件库 (Repository) 上进行的，而且用户都是在同一台机器上。不过这可不太多可能一个团队工作的有效方式。你需要一个中心文件库 (Repository)，每个用户都可以从自己的机器上通过网络来访问它。本章我们将讨论连接文件库 (Repository) 的不同方式，并给出一个选择适合你的团队的指导性建议。

CVS 提供相当多的通过网路访问文件库的选择，这很容易把你搞糊涂。所以，让我们从一些可靠的方面着手：如果你一开始选择了错误的方法也不会造成多大的问题。连接方法一点不会影响文件库，你可以随意在不同方法间切换，完全不会影响你的工作。

第二，有一点很重要，需要你记住：你不必只选择一个连接方法。你的客户端可以部分通过一种连接方式，而其他的通过完全不同的另一种方式。实际上，如果你有部分开发人员在房间里而其他的通过因特网访问文件库，这才是最佳的组织方法。

大部分远程 CVS 文件库的访问方式包括两种：pserver 或 external。在 pserver 模式下，CVS 在文件库机器上运行一个服务器进程，所有的客户端都连接这台机器。这种方式，就好像 CVS 是一个 web 服务器或者 ftp 服务器一样：它自己处理连接的细节以及管理安全。

Pserver 模式有一些优点：

- 设置相对简单。
- 强制式只读用户（使用者只能 check out 或者 update，但是不能 update）。
- 支持匿名访问（一种普遍在开源项目中应用的机制，可以授权未知的用户访问文件库）

但是 pserver 模式也有些缺点：

- 使用专用的网络端口，很多防火墙都不让通过。
- 使用非常简单的口令加密，文本是通过明文传输的。
- 需要独立的管理员（也就是说，如果你已经对访问文件库的机器的用户进行了远程访问（其他方面的）的管理，你还需要重新对 CVS 用户做同样的工作）。

因为这个问题，我们推荐 pserver 只用于对文件库进行远程的匿名访。

使用 **ext** 方式访问文件库和 **pserver** 模式有些许不同。**CVS** 使用已经存在的操作系统命令在客户端和服务端设置一个数据管道。缺省的 **CVS** 扩展使用一个叫 **rsh**（也就是 **remote shell**）的程序。**Rsh** 被开发出来可以追溯到实验室之间的串联线缆联网时代，当时每个网络用户都是被信任的，这有点令人遗憾（译者：**rsh** 是远程执行 **shell** 里面的命令。这是很古老的技术，已经不能满足现代网络的安全需求）。**Rsh** 的确方便但是不是特别安全，而且 **rsh** 还不允许你通过公众因特网访问。

幸运的是还有一个安全的插件兼容的别的选择，那就是 **ssh**。我们强烈推荐所有对 **CVS** 文件库进行扩展访问的都通过 **ssh** 通道进行。如果你想用 **rsh** 作为内部网的访问方式，尽管做。只是要确定如果你在酒店的房间工作，而且需要访问文件库的时候，你就应该切换到 **ssh**（好消息是因为 **ssh** 和 **rsh** 是兼容的，你可以使用同一份文件库的工作文件的版本，我们一分种内就能看到）。

总结如下：

- 对于内部网或者任何工作方法。我们推荐使用 **ssh** 通道，只是因为你可能只想对所有的访问都用一个方法。
- 对于有规则的扩展访问，使用 **ssh** 通道。
- 使用 **pserver** 的方式提供对文件库的匿名公共访问。

那么在决定了使用哪种连接方法后，你怎么实际使用呢？这有点依赖你所使用的工具。这里我们将向你展示怎么样从命令行连接，如果你使用的是 **GUI** 或者 **IDE** 工具，有关细节请参考其相关文档。

5.1 安全和用户帐号

CVS 有用户的概念：所谓用户就是访问并修改文件库的使用者。无所谓你是使用 **pserver** 还是 **ssh** 方式，你将有一个能够访问 **CVS** 的用户 **id**。

通过 **ssh** 模式，你使用一个有效的用户 **id** 登录到你的服务器上。那就是说每个 **CVS** 用户在这个服务器上必须有一个用户帐号（在 **Unix** 下，你必须已经在 **/etc/passwd** 文件里登记）。

而 **pserver** 模式的弹性就更大。你可以设置和服务器用户相应的帐号，也可以设置独立于操作系统用户的 **CVS** 专用帐号。文件库管理员完成所有这些用户名映射（使用在文件库

的 CVSROOT 模块下的 `passwd` 文件)。有关配置的细节超出本书的范围 (Reader Ray Schneider 提醒我们有一些帮助管理 `pserver` 的免费工具可以使用, 包括 `cvsadmin` 和 `cvspadm`)。

任一一种方式, 文件库管理员都可能设置单个用户, 让所有的人都用这个用户来访问文件库, 文件库管理员在尝试一种欺骗的行为。尽管这样也可以, 不过的确不是个好主意。CVS 的一个好处是可以跟踪谁做了什么, 一年后你都能够通过查看日志找出在某个特别的源文件里的某一行是某人修改的。如果每个人都用一个 CVS 帐号 `cvsuser`, 那么就失去了意义。

使用独立的用户 `id` 意味着文件库管理员可以针对文件库的各个部分设置访问控制权。例如一种公共策略: 给所有的团队成员设置完全访问他们项目代码的权利, 但是他们对于其他团队的项目代码只有只读的权限。尽管设置这种访问控制权限超出了本书的范围, 你还是需要很清楚的识别出每个用户, 这样才能实现这种设置。在服务器上使用单个用户帐号会破坏这种访问控制规则。

5.2 CVSROOT:访问目的地参数字符串

CVS 使用类似于 URL 的形式来指定文件库的抵制。这个字符串, 有时叫做 CVSROOT, 包括访问类型, 跟着是用户, 服务器, 文件库的名字以及地址。这个字符串的语法有点乱:

:type:user@server:repository_location

你输入的不同的字段的值取决于你想访问的类型, 是 `ssh` 还是 `pserver`。不久我们就会研究细节。首先我们要知道怎么样把这些值传给 CVS。

将 CVSROOT 传给 CVS 命令

很多 CVS 命令都接受 `-d` 参数, 让你显示地指定 CVSROOT 的值。在我们第一次创建我们的沙箱文件库的时候已经见识过这个功能。

```
work> cvs -d ~/sandbox checkout project
```

这同样也适合更复杂的客户服务器连接方法。例如: 如果你在使用 `pserver` 模式连接 `xyz.com` 服务器的 `/var/cvs` 目录下的文件库, 并且假设你在那个机器上的帐号是 `wilma`, 你可以用以下命令进行 `check out`:

```
work> cvs -d :pserver:wilma@xyz.com:/var/cvs checkout proj1
```

记住每次使用 CVS 的时候都要输入这样的字符串可不容易。幸运的是你不必这样。当你第一次设置好 workspace，或者当你使用其他需要显示地指定文件库位置的命令的时候，你设置好一个默认的要使用的文件库，这样可以节省一些输入。命令行用户可以通过设置 CVSROOT（也可能会用到 CVS RSH）环境变量来做到这一点。CVSROOT 告诉 CVS 默认使用的文件库（等价于 -d 参数）。CVS RSH 告诉 CVS 当连接方法为 ext 的时候使用的哪个程序。我们将用 ssh。

在 Windos 下设置这些环境变量的方式是：右键点击“我的电脑”，选择“高级”面板，点击“环境变量”。对于单一交互会话（single interactive session），你也可以使用命令行（shell）：

```
C:\> set CVSROOT=:pserver:wilma@xyz.com:/var/cvs
```

在 Unix 下，取决你使用的 shell。对于 bash，zsh，以及类似的 shell，添加如下的行到你的 profile 文件里面（export 命令设置了全局环境变量，否则，一旦你的 profile 文件执行完毕这个变量就消失了）。

```
export CVSROOT=:pserver:wilma@xyz.com:/var/cvs
```

在 C shell 下，你可以使用 setenv 命令来完成同样的工作，而不是 export 命令。

一旦你在一个 check out 的目录下工作，CVS 默认会自动使用这个目录对应的的文件库。

5.3 设置 ssh 访问模式

在和 CVS 使用 ssh 通道之前，你需要设置好能够工作的 ssh 环境，这样你才能在你的客户机和 CVS 文件库之间进行交流。你可以购买 ssh 的商业版本（如来自 www.ssh.com 的产品），也可以采用开源版本（如 www.openssh.com）。怎样设置 ssh 超出了本书的范围，对我们来说假设你已经安装好了，而且你也可以使用命令登录了。

```
ssh -l user my.server.machine
```

为了联合使用 CVS 和 ssh，你需要知道你在服务器上的用户名，服务器的机器名，以及服务器上文件库目录的位置。（一个服务器能够处理多个文件库：这个阶段你可以挑选一个你想用的）具备了这些信息，你就可以指定远程机器和文件库，设置 CVSROOT 环境变

量了。需要指定访问类型为 **ext**，让 **CVS** 使用外部程序建立网络通道。因为你在使用外部程序来访问 **CVS**，因此你需要通过设置 **CVS RSH** 环境变量来告诉它。这个例子我们将使用 **ssh**。

要以用户 **dave** 访问 **my.repository.com** 上的文件库 **/var/repository**，设置环境变量如下：

```
CVSROOT :ext:dave@my.repository.com:/var/repository
CVS_RSH  ssh
```

在 **Windows** 下，你可以使用 **GUI** 界面来设置环境变量（前面讲过），你也可以从命令行设置。**Windows** 的命令如下：

```
C:\> set CVS_RSH=ssh
C:\> set CVSROOT=:ext:dave@my.repository.com:/var/repository
```

在 **Unix** 下，添加如下的信息到你的 **profile** 文件：

```
export CVS_RSH=ssh
export CVSROOT=:ext:dave@my.repository.com:/var/repository
```

一旦这些设置完成，你的 **cv**s 命令就能够自动通过一个安全的加密的通道连接到服务器了。

5.4 使用 **pserver** 进行连接

如果你决定使用 **pserver** 访问文件库，你的文件库管理员将为你在服务器上设置一个 **CVS** 口令。

和 **ssh** 方式一样，要想让客户端知道怎么找到文件库的位置，最容易的方式可能是设置 **CVSROOT** 环境变量。只是这次你要使用的是 **pserver** 访问方法，而不是 **ext**。在使用 **pserver** 的时候就没有必要设置 **CVS RSH** 环境变量了。

```
CVSROOT=:pserver:dave@my.repository.com:/var/repository
export CVSROOT
```

但是在你开始使用 **CVS** 命令前，你要先登录（没什么惊奇的，用 **cv**s login 命令）：

```
cv
```

s login

系统会提示你输入口令（文件库管理员应该已经分配给你了）。你只需要输入一次就可以了，因为 **CVS** 会话能够记住你的口令。如果你（端正地）关心这个的安全，你可以使用

`cvs logout` 命令，这样 CVS 会忘记你的口令。不过，如果你是真正关心安全问题，无论如何你都应该使用 `ssh`。

第六章 常用 CVS 命令

在开始那一章我们进行了 CVS 探索，包括创建了一个虚拟的项目，试验一些简单的命令。本章我们将更深入一点。我们会在这里呈现一套诀窍：你每天工作必备的 CVS 命令。

本章并非详尽。本书后面我们会着眼于更多的高级主题，诸如发布管理，工作间，第三方代码的管理等等。然而，本章的命令和技术将能够解决你使用 CVS 的工作中的 90% 的问题。

这些例子假设你已经设置好可运行的文件库，以及引用文件库的相应环境变量。实际上，后者将是第一个诀窍的主题。

6.1 Check Out 命令

`cvs checkout` 命令(常常缩写为 `cvs co`)将文件库的部分内容放到你的本地 `workspace`。(第——页始我们会更详细讨论 `workspace` 管理)

最简单的是使用同一个名字将一个或多个模块或子模块 `check out` 到本地目录。下面的命令是将文件库的 `client` 和 `server` 模块的内容 `check out` 出来，并作为子目录保存在 `work` 目录下。

```
> cd work
work> cvs co client server
```

在__页关于子模块的章节，我们将教你怎么样只 `check out` 一个目录下的部分文件。下面的命令是将保存在 `client/templates` 下的文件 `check out` 出来，并保存在 `work/client/templates` 目录下。

```
work> cvs co client/templates
```

默认情况下 CVS `check out` 默认分支的头(通常就是主线 `mainline`)。你可以使用 `-r` 或者 `-D` 选项改变这个默认值。

`-r` 选项让你 `check out` 一个指定的修订版本。这个修订版本可以用绝对的版本号来指定(如 1.34)，也可以用标签。因为 CVS 为每个文件保存独立的修订版本历史，因此基于某个绝对的版本号将整个项目 `check out` 通常没什么意义(版本号为 1.34 的文件 `File1.java`

可能是版本号为 1.34 的文件 `File2.java` 后两个月才建立的)。可能你要用标签来代替。后面我们会更详细地讨论标签，基本上标签是你为一组文件当前的状态赋予的一个象征性的名字。除了你创建的标签外，CVS 还提供了两个标签：`HEAD` 达标文件库最近的版本，`BASE` 则指你最近 `check out` 到当前目录的版本。除非你被一大堆标签和分支（第七章里会讲分支）搞的一团糟，否这没有必要担心 `HEAD`，因为这是绝大部分命令的默认值。

```
work> cvs co -r REL_1_34 client
```

`-r` 标签根据版本号 `check out` 文件，`-D` 标签则根据日期 `check out`。跟在 `-D` 选项后面的日期有多种格式，包括 `ISO8601`，因特网电邮标准，以及其他各种缩写形式，详情参考表 6.1。

一旦你 `check out` 某个特定修订版本的目录，那个版本是 `sticky`。也就是说在这个目录下所有的后续工作都应用到这个版本上。只有你根据一个分支 `check out` 的版本才有意义（如发布分支，第 89 页我们会更深地讨论）。如果你在一个 `sticky` 的标签上运行 `cvs status` 命令，你会看到那个标签列出来了。

Specification	Examples
ISO8601	2003-06-04
	20030604
	2003-06-04 20:12
	2003-06-04T20:12
	2003-06-04 20:12Z
	2003-06-05 20:12:00-0500
E-Mail format	Mon Jun 9 17:12:56 CDT 2003
	Mon, Jun 9 17:12:56 2003
	Jun 9 17:12:56 2003
	June 9, 2003
Relative	1 day ago
	27 minutes ago
	last monday
	yesterday
	third week ago

Table 6.1: Sample date specifications accepted by the CVS `-D` option.

sticky 标签是 CVS 的一种机制，你可以在多个版本上工作。同作与文件关联的标签，CVS 知道你在过去某个特定时间点上的文件上工作，或者是在文件库树结构下的某个特定的分支上工作。因为这个功能，在你下次 **check in** 的时候 CVS 不会覆盖主线 **head** 的文件版本。

如果你 **check out** 多个发布分支到同一 **workspace**，你可能不想让 CVS 自己选择目录名（否则 **client** 的 **REL_1_34** 版本会覆盖当前版本）。你用 **-d** 选项可以指定目录（本例将使用名为 **rel1.34** 的目录名）。

```
work> cvs co client
work> cvs co -r REL_1_34
```



6.2 保持代码最新的状态

如果你不是项目唯一的成员，在你工作的同时别人更新文件库的机会还是挺大的。频繁地将他们的修改和你工作的版本进行合并是个好主意，你越长时间做一次，发生冲突的机会就越大。代表性的，一天下来我们常常每个小时都会更新一次我们的工作文件。

在工作目录下运行 **cvs update** 命令。它将目录下的所有文件和文件库的进行同步更新。不过默认地它不会创建任何新目录（自最后一次 **check out** 以来文件库里添加的目录），如果要特别做到这点，就要加 **-d** 选项。下面的命令更新了 **client** 项目下的所有文件和目录。

```
work> cd client
work/client> cvs update -d
```

你也能够选择只更新你的 **check out** 树的一部分。如果你在项目的子目录下执行该命令，那么在那个目录下的文件才会得到更新。这能节约时间，但是也可能让你在一个不一致的文件集上工作（译者：只有部分文件更新了）。

你也能够在命令行上指定要更新的一个和多个文件或目录的名字。

```
work/client> cvs update File1.java templates
```

在更新的过程中，CVS 会跟踪它进入的所有的目录的名字，而且还会显示出每个文件的具有重大意义的活动的状态。例如下面是更新目录（该目录下包括书籍《Pragmatic Starter Kit》）时产生的日志信息。

```
StarterKit> cvs update
? SourceCode/tmpdoc.ilg
? SourceCode/tmpdoc.toc
cvs server: Updating .
RCS file: /home/CVSR00T/PP/doc/StarterKit/pragprog.sty,v

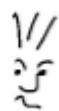
retrieving revision 1.16
retrieving revision 1.17
Merging differences between 1.16 and 1.17 into pragprog.sty
M pragprog.sty
cvs server: Updating SourceCode
A SourceCode/CommonCommands.tip
M SourceCode/HowTo.tip
A SourceCode/Releases.tip
cvs server: Updating SourceCode/images
cvs server: Updating UnitTest
P UnitTest/DesignIssues.tip
U UnitTest/InAProject.tip
P UnitTest/Introduction.tip
cvs server: Updating UnitTest/code
U UnitTest/code/Age.java
U UnitTest/code/TestMyStack.java
U UnitTest/code/testdata.txt
cvs server: Updating UnitTest/code/rev1
cvs server: Updating UnitTest/code/rev2
cvs server: Updating UnitTest/code/rev3
cvs server: Updating util
```

问号开头的句子表示本地 **workspace** 里未知的文件（相对 CVS）。——页的 6.4 节有关于删除这些跟踪日志的信息。以 **A** 开头的行表示本地新增加但是还没有提交到文件库的文件，**M** 开头的行表示文件在本地被修改过。**U** 和 **P** 开头的行表示过期的文件，因为文件库的文件比本地文件更新。第 134 页关于诀窍的章节列出了完整的所有的标记的含义。

注意日志显示两个作者都修改过文件 **pragprog.sty**。在这个例子里，CVS 能够协调两者的修改，将 Andy 的修改合并到 Dave 的本地修改。有时候修改交迭的时候，CVS 会用给文件以 **C** 的标记，表示出现了合并的冲突。详情请参考第 75 页 6.8 节关于处理合并冲突的部分。

cvs update 命令的输出相当冗长，你可以使用全局的 CVS 选项 **-q**（应该写在 **update** 之前）将输出减小。

```
work/client> cvs -q update -d
```



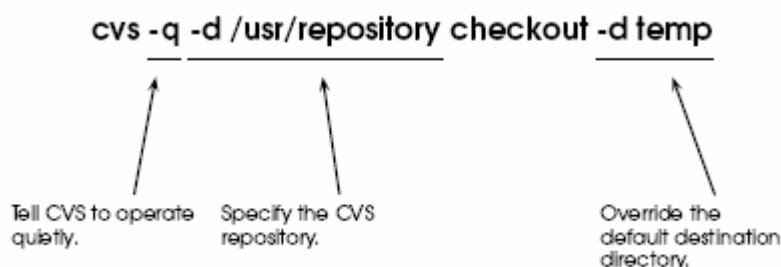
Joe 的问题。。。

在某些方面，CVS 更象一个子系统。你输入以下命令的时候：

cvs **check**out

字首的 **cv**s 表示你在使用 CVS，而 **check**out 表示你要执行的子命令。

因为这个，CVS 有两个独立的指定选项的位置。对于 CVS 子系统的全局选项（如指定文件库位置的 **-d** 选项，还有减少日志输入的 **-q** 选项）应该直接出现在 **cv**s 命令后面。而那些特定的子命令专用的选项就应该出现在子命令的名字后面。比如 **check**out 子命令的 **-d** 选项，用于重新指定 **check**out 的目的地。后面这个 CVS 命令的意思是 **check** **out** 位于 **/usr/repository** 的文件库，并把结果保存在 **temp** 目录下，要求以 **quiet** 的方式输出日志。



6.3 添加文件和目录

cvs **add** 命令告诉 CVS 要向文件库添加的文件和目录。

```
proj> mkdir timelib
proj> cvs add timelib
Directory /Users/.../proj/timelib added to the repository
proj> cd timelib
proj/timelib> #.. create and edit file Time.java ...
proj/timelib> cvs add Time.java
cvs add: scheduling file 'Time.java' for addition
cvs add: use 'cvs commit' to add this file permanently
```

如果你愿意，你还能够使用 **-m** 选项添加一个创建的信息（尽管很少使用）。更有用的是 **-kb** 选项，表示某个文件是二进制文件。

CVS 和二进制(Binary)文件

CVS 设计的主要目的是处理文本文件：如程序文件，XML 文件等等。这就是说 CVS

具备一些智能：

- 它能够按修改的行保存每个修订版本，而不是按每个修改的整个文件来保存。
- 能够处理操作系统间不同的行结束标志（特别是 Unix 使用的 `newline`（新行）标记和 Dox 或 Window 使用的 `carriage return`（硬换行）/`newline`（新行）的区别）
- 通过代替某些关键字，可以给文件添加注释（我们不推荐使用的一个特征，请参考后面）

但是如果给 CVS 管理一个二进制文件（可能是一个 DLL，或者是 Word 文档），这些特征都没有用了。

- CVS 用于计算文件之间的不同之处的工具并不能用于二进制文件。
- 二进制文件并没有行结束标记。将任何包括一个新行的字节修改为包括硬换行和新行的两个字节都可能打乱文件。
- 如果一个二进制文件碰巧包含了类似 CVS 关键字的序列，很可能错误地去扩展它，这样你就可能破坏了文件格式。



Joe的问题...：为什么不用CVS关键字？

如果你使用某个魔力的指令序列，如`$Author$`和`Log`，在你每次 `check out` 该文件的时候，CVS 能够自动用附加文本填充它们。关键字`$Author$`能够改变为提交修订版本的那个人的名字。而关键字`Log`提示 CVS 向文件里面插入一条完整的修改日志。

初初看来，这好像是个好注意：你给文件添加了文档，而且还没有给你增加负担。那到底有什么错？

关键字扩展有三个问题，一个是哲学上的，两个是实践上的。

哲学问题就是你在复制信息。所有能通过关键字插入文件的信息都已经保存在 CVS 里面（否则 CVS 页不能把这个加在第一位置）。那么为什么不直接让 CVS 做呢。那样你可以得到保证更新的授权信息。

第二个问题是这些源文件的额外信息阻碍了文件的可读性。我们曾经看到文件的头上居然又两到三页全是日志信息，而最后只有一行可读的代码。代码是用来读的，不能有任何阻碍阅读的东西。

第三个问题是一旦你在 CVS 文件里使用关键字，合并不同分支之间的代码或者在文件

库之间移动文件就变得很笨拙。你不得不记住怎么样在正确地时机使用用正确的选项，不知为什么每次创建一个主发布版本的时候这些东西都忘的一干二净。

所以关键字扩展并没有多少好处，但是坏处却不少，我们建议不要使用这个功能。

为了处理这个问题，CVS 使用了一个技术。只要你在添加一个的时候指定 **-kb** 选项，告诉 CVS 把这个文件当作二进制文件处理。对于每个版本，CVS 都将保存整个完整的文件，而且绝对不会处理行结束标记和关键字。因此，在添加二进制文件的时候，记住使用 **-kb** 是很重要的。

```
work> cvs add -kb DataFormat.doc
```

然而只是因为它的重要性并不以为着我们总是要记着这个，幸运的是（一旦忘记）我们有机会恢复。

如果我们在 check in 之前忘记了使用 **-kb** 选项，我们可以重新添加一次，而这次记者使用正确的标记就可以了。这有一点技巧性，在第二此添加之前要先移动该文件，如果文件还存在于我们的工作目录，我们就不能移动它。因此为了达到这个不致，我们需要先使用 Unix 的 mv 命令临时来修改文件的名字（相当于 Windows 的 ren）。

```
work> cvs add DataFormat.doc #<-- forgot the -kb option
cvs add: scheduling file 'DataFormat.doc' for addition
cvs add: use 'cvs commit' to add this file permanently
work> mv DataFormat.doc Temp.doc
work> cvs remove DataFormat.doc
cvs remove: removed 'DataFormat.doc'
work> mv Temp.doc DataFormat.doc
work> cvs add -kb DataFormat.doc #<-- use the option
cvs add: scheduling file 'DataFormat.doc' for addition
cvs add: use 'cvs commit' to add this file permanently
work> cvs commit -m "Add new data format document"
```

如果在 check in 之后我们才认识到自己的错误，就需要更多的技巧。最安全的修补方法是直接在文件库里修改标志，然后使用已知的二进制文件的工作拷贝来更新文件库。假设我们忘记使用 **-kb** 标记来提交 DataFormat.doc 文件，下面就是我们需要补救的。

```
work> # reset the flag in the repository
work> cvs admin -kb DataFormat.doc
work> # then reset the flags in our workspace
work> cvs update -A DataFormat.doc
work> # copy a known good copy over this file
work> cp ~/docs/DataFormat.doc DataFormat.doc
work> # and save this back in the repository
work> cvs commit -m "Reset -kb flag"
```

如果你花了大量时间来处理二进制文件，你可能需要考虑一下 **cvs wrappers** 功能，

`cvswrappers` 可以让 CVS 根据文件名指定文件的特性。下一节我们将讲讲这个功能，但是你也可以跳过这一节，完全没问题。

File Characteristics and cvswrappers

CVS wrapper 功能可以让你根据文件名设置文件属性。有三种方式指定这个 wrapper：在文件库的文件 `CVSROOT/cvswrappers` 里面。

- 在每个用户的 `cvswrappers` 文件里面（注意开头的。）
- 这个文件必须保存在你的 `home` 目录下。对于我们来说，这看起来象是一个带疑问的实践，如果特别指定了一组文件名，这将影响文件库的全局，不只是文件库的一个用户而已。
- 通过给 `cvs import` 和 `cvs update` 命令的 `-W` 命令行选项。这对于一次性特别有用，特别是在 `import` 已经存在的目录树的时候。

在所有的案例里面，你都可以指定匹配一个或多个文件名的模式，然后选择应用到那些和模式匹配的文件的 CVS 选项。例如，你可能打算 `import` 一个目录树，该目录树下包括了一个巨大的 `Java.jar` 文件和微软的 `Word` 文档的文件。这些文件都是二进制文件，那么应该使用 `-kb` 选项。

有两种处理方式。如果我们只想添加某些类型的文件到文件库，可以在 `cvs import` 上使用 `-W` 选项。你需要在命令行指定 `-W` 两次，每次都是指定要匹配的模式。也有一些相当神秘的，如在命令行将引号和星号传递给 CVS。对我们专用的 `shell`，命令如下：

```
myproj> cvs import -W "*.jar -k 'b'" -W "*.doc -k 'b'" \  
-m '' myproj ...
```

使用 `-W` 选项的问题是很容易在下一次做 `import` 的时候忘记使用了。因为这样，你可能想在 `cvswrappers` 文件里设置一些永久选项，并保存在 `CVSROOT` 文件库下。那首先，将 `CVSROOT` 下的文件都 `check out`。

```
tmp> cvs co CVSROOT
cvs checkout: Updating CVSROOT
U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
U CVSROOT/cvswrappers
U CVSROOT/editinfo
U CVSROOT/
U CVSROOT/
U CVSROOT/
U CVSROOT/notify
U CVSROOT/rcsinfo
U CVSROOT/taginfo
U CVSROOT/verifymsg
```

现在改变当前目录到 check out 的 CVSROOT 目录下, 并编辑 cvswrappers 文件。给 .jar 和 .doc 文件添加了 -kb 选项, 用你最喜欢的编辑器在文件末尾添加下面两行:

```
*.jar -k 'b'
*.doc -k 'b'
```

现在向文件库提交修改。

```
tmp/CVSROOT> cvs commit -m "Make all .doc/.jar files binary"
cvs commit: Examining .
Checking in cvswrappers;
/Users/dave/sandbox/repo/CVSROOT/cvswrappers,v <-- cvswrappers
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
CVSROOT> cd ..
tmp> cvs release -d CVSROOT
U cvsignore
U cvswrappers
U logininfo
U modules
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'CVSROOT': yes
```

6.4 忽略某些文件

在开发的过程中, 常常会生成很多中间(临时)文件。C 程序员生成 object 文件, Java 程序员生成 class 文件, 我们则到处生成临时文件。类似的临时文件不应该保存在文件库里, 需要的时候应该可以根据源文件重新生成。但是, 如果这些文件四处分散, 都是都是, 每次 update 或者 commit 的时候 CVS 就会抱怨。幸运的是有个简单的方法让 CVS 不会被这些文件骚扰。

如果一个目录包含了一个名为 .cvsignore (注意开头的.) 的文件, CVS 将会把该文件的内容作为该目录下忽略的文件列表。在 .cvsignore 文件里的每一行可以是一个单独的文件

也可以是匹配多个文件的模式。例如你当前在 `client` 目录下，你想让 CVS 忽略一个临时的源文件： `Dummy.java`，以及所有的 `class` 和 `log` 文件，你可以创建包括一下内容的 `.cvsignore` 文件：

```
Dummy.java
*.class
*.log
```

然后需要提供一个选项。如果你将 `.cvsignore` 文件也 `check in` CVS，其他的开发折下次 `check out` 之后，也会忽略同一目录下的相同文件。如果你不将它 `check in`，那么它包含规则只对你有效。通常我们推荐 `check in` 该文件，意思是说所有的人都在同样的环境下工作。

```
work/client> cvs add .cvsignore
work/client> cvs commit -m "Ignore Dummy, log, class files" \
               .cvsignore
```

6.5 给文件重命名

CVS 从不会忘记。这在大部分时间都很好，只有当需要给文件改名字的时候便痛苦起来（我们也会看到，要给目录改名字甚至更痛苦）。

你不可以使用 CVS 直接给文件改名。而是要给你本地的 `workspace` 的文件的工作版本改名，然后让 CVS 从文件库中删除旧文件，并将改名的文件重新添加进去。

例如，我们假设要将文件名 `Contacts.java` 改为 `ContactMgr.java`。需要按照以下步骤进行：

```
proj> cvs -q update -d
proj> mv Contacts.java ContactMgr.java
proj> cvs remove Contacts.java
cvs remove: scheduling 'Contacts.java' for removal
cvs remove: use 'cvs commit' to remove this file permanently
proj> cvs add ContactMgr.java
cvs add: scheduling file 'ContactMgr.java' for addition
cvs add: use 'cvs commit' to add this file permanently
proj> cvs commit -m "Rename Contacts.java to ContactMgr"
cvs commit: Examining .
cvs commit: Examining timelib
RCS file: /Users/dave/sandbox/proj/ContactMgr.java,v
done
Checking in ContactMgr.java;
/Users/.../proj/ContactMgr.java,v <-- ContactMgr.java
initial revision: 1.1
done
Removing Contacts.java;
/Users/.../proj/Contacts.java,v <-- Contacts.java
new revision: delete; previous revision: 1.6
done
```

这样做对两边都有影响。首先是这个新文件的版本要从 1.1 重新开始。所有的修订历史只跟 **Contacts.java** 有关，而 **ContactMgr.java** 完全是空白的历史。

这个的原因是 **Contacts.java** 文件仍然存在于文件库中，它只是被移动到一个特殊的不会参与 CVS 操作的空间（叫做 **Attic**）。从版本 1.7 起，这个文件都不存在了。然而，要记住 CVS 的一个工作是扮演时间机器。如果我想创建昨天的软件版本，我希望能够重新看到 **Contacts.java** 文件的出现。

因为更名后的文件从 1.1 开始，它不会包括原来的历史。要获得修订历史（日志信息，修改信息等等）你需要告诉 CVS 需要查看原来的文件名。

类似的，尽管这个文件已经不在我们的 **workspace** 里面，我们仍然能够从 CVS 查询其状态。结果告诉我们该文件从 1.7 版本开始就移到 **Attic** 里面，已经不存在于我们的 **workspace**。

```
proj> cvs status Contacts.java
=====
File: no file Contacts.java                      Status: Up-to-date
Working revision:    No entry for Contacts.java
Repository revision: 1.7 /Users/.../proj/Attic/Contacts.java,v
```

我们尝试一下 **check out** 修订版本 1.6（它实际存在的上一版本）

```
proj> cvs update -r1.6 Contacts.java
U Contacts.java
proj> cvs status Contacts.java
=====
File: Contacts.java      Status: Up-to-date
Working revision:       1.6 Thu Jun 12 23:57:33 2003
Repository revision:    1.6 /Users/.../proj/Attic/Contacts.java,v
Sticky Tag:             1.6
Sticky Date:            (none)
Sticky Options:         (none)
```

如我们所料，成功返回了。但是这在 `workspace` 里面是一个时空错误，因为当前的 `workspace` 已经包括了一个最新的文件。因此我们在 `cvs update` 命令里使用 `-A` 选项来进行整理。`-A` 选项有效地将 `workspace` 带回到和文件库对应的状态。

```
proj> cvs update -A
cvs update: Updating .
cvs update: warning: Contacts.java is not (any longer) pertinent
cvs update: Updating timelib
```

如果以后我们创建一个名为 `Contacts.java` 的新文件（和以前删除的重名）会发生什么情况呢？That won't phase CVS at all, CVS 会直接向文件库以该文件的下一个版本号（1.8）添加该文件。如果你接下来就向 CVS 要 1.7 以前的版本，你就会得到保存在 `Attic` 里的旧版本。如果你要最新版本（或者你只用默认的最新版本），你就会得到那个新文件。

6.6 重命名目录

在 CVS 里面修改文件名还是相当简单的。修改目录名的步骤如下的模式，但是它们有点笨拙：

- 创建新目录
- 将新目录添加到 CVS
- 将旧的目录下的文件移动到新目录。
- 在旧的目录下运行 `cvs remove` 命令，告诉 CVS 目录下的文件不在存在那里了。
- 在新目录下使用 `cvs add` 命令添加该目录下的文件。
- 提交修改，并执行带 `-P` 选项的 `cvs update` 命令，从你的 `workspace` 删除旧目录（`-P` 选项的意思是删除空目录）

我们的例子是把一个名为 `timelib` 的目录（下面包括了一个文件 `Time.java`）更名为 `timelibrary`。在这里例子里，我们使用 Unix shell 命令 `mkdir`, `mv` 和 `ls`，分别用于创建目录，

移动文件，列出目录下的内容。在 Windows 席面，创建目录的名字也叫 `mkdir`，`ren` 命令用于改文件名，`dir` 命令用于列出目录下的内容。

```
proj> mkdir timelibrary
proj> cvs add timelibrary
Directory /Users/.../proj/timelibrary added to the repository
proj> mv timelib/Time.java timelibrary
proj> cvs remove timelib/Time.java
cvs remove: scheduling 'timelib/Time.java' for removal
cvs remove: use 'cvs commit' to remove this file permanently
proj> cvs add timelibrary/Time.java
cvs add: scheduling file 'timelibrary/Time.java' for addition
cvs add: use 'cvs commit' to add this file permanently
proj> cvs commit -m "Rename timelib/ to timelibrary/"
cvs commit: Examining .
cvs commit: Examining timelib
cvs commit: Examining timelibrary
Removing timelib/Time.java;
/Users/dave/sandbox/proj/timelib/Time.java,v <-- Time.java
new revision: delete; previous revision: 1.1
done
RCS file: /Users/.../proj/timelibrary/Time.java,v
done
Checking in timelibrary/Time.java;
/Users/.../proj/timelibrary/Time.java,v <-- Time.java
initial revision: 1.1
done

proj> cvs update -P
cvs update: Updating .
cvs update: Updating timelib
cvs update: Updating timelibrary
proj> ls
CVS                ContactMgr.java Numbers.txt      timelibrary
```

很明显这不是一个最佳的过程，缺乏内置的改名的功能是 CVS 最大的缺点。

当然还有另外的一些小问题。如果有人将来加入，并 `check out` 项目的一份新的版本，他们会得到 `Time.java` 文件，并正确地出现在 `timelibrary` 目录下，但是他们还会得到一个空的 `timelib` 目录，这是被删除的文件占用的地方。你可以通过使用带 `-P` 选项的 `cvs update` 命令删除这个目录，或者在 `check out` 的时候就使用 `-P` 选项。

6.7 查看更改情况

`cvs diff` 命令可以让你查看文件不同版本间的区别。你也可以比较文件库里的版本和本地修改后的版本，而且你还可以查看文件库的文件的两个版本间的区别。

最简单 `cvs diff` 可以显示你对文件的修改。

```
work/client> cvs diff File1.java
Index: File1.java
=====
RCS file: /Users/dave/sandbox/proj/File1.java,v
retrieving revision 1.2
diff -r1.2 File1.java
10c10,12
<         total += amount;
---
>         if (amount.isPositive()) {
>             total += amount;
>         }
```

输出显示我们上次 `check out` 文件 `File1.java` 的版本是 1.2，跟着就修改了该文件，并在一个加法语句前添加了一个 `if` 条件语句。

有人发现 `context diffs` 更有可读性，他们显示的不是改变，而是在文件修改的位置的前后的部分。只需要在 `cvs diff` 后面添加 `-c` 即可。

```
work/client> cvs diff -c File1.java
Index: File1.java
=====
RCS file: /Users/dave/sandbox/repo/proj/File1.java,v
retrieving revision 1.2
diff -c -r1.2 File1.java
*** File1.java 2003/06/10 19:52:36 1.2
--- File1.java 2003/06/10 19:53:20
*****
*** 7,13 ****

    public void addInterestPayment(Money amount) {
!         total += amount;
    }
--- 7,15 ----

    public void addInterestPayment(Money amount) {
!         if (amount.isPositive()) {
!             total += amount;
!         }
    }
```

回到第 36 页，那里写了 `cvs diff` 的另一种格式。你可以用 `--side-by-side` 选项进行边对边的比较。

在 `cvs diff` 的各种形式里面还隐藏着一个 `gotcha`，这个命令显示了当前在你的 `workspace` 里面的文件和你 `check out` 文件之间的区别。但是，如果有人后来修改了这个文件并做了 `check in`，你看不到这些变化。我们很快就会看到怎样处理这种情况。

在不同版本间进行比较

在使用 `checkout` 和 `update` 命令的时候我们已经见识过 CVS 的 `-r` 和 `-D` 选项。`-r` 选项让我们指定一个修订版本或者标签，`-D` 选项让我们指定一个日期。这些选项同样适合 `cvs diff`

Matrix-与 Java 共舞 (<http://www.Matrix.org.cn>)

命令。这些选项你可以指定一次或者两次。如果使用一次，那就是让 CVS 查找出在文件库的版本和本地工作文件之间的区别。如果使用两次，CVS 就列出文件库里两个不同版本之间的区别（这种情况下不会显示本地修改）。

例如在第 60 页我们向大家展示过一个 `cvs update` 命令的例子，本地文件已经修改，同样在文件库的也是。我们很好奇在我们 `check out` 这个文件之后，文件库里文件被修改成什么样了。日志消息如下：

```
doc/StarterKit> cvs update
cvs server: Updating .
RCS file: /home/CVSR00T/PP/doc/StarterKit/pragprog.sty,v
retrieving revision 1.16
retrieving revision 1.17
Merging differences between 1.16 and 1.17 into pragprog.sty
M pragprog.sty
```

我们可以通过日志里的修订版本号检查其区别。

```
doc/StarterKit> cvs diff -r1.16 -r1.17 pragprog.sty
Index: pragprog.sty
=====
RCS file: /home/CVSR00T/PP/doc/StarterKit/pragprog.sty,v
retrieving revision 1.16
retrieving revision 1.17
diff -r1.16 -r1.17
211a211,216
> %
> % Place holder for Exercise package
> %
> \newenvironment{EXERCISES}{}{}
> \newenvironment{EXERCISE}{\par{}\hrulefill\EXERCISE:\par}{}
> \newenvironment{ANSWER}{\par{}ANSWER:\par}{}

```

通过指定 `cvs diff` 命令的两个版本号参数，我们可以看到 Andy 添加了三个，用于 LATEX 宏（我们用于格式化本书）训练。

前面我们说过一个伴随 `cvs diff` 的通用的 `gotcha`，它并不是默认的比较你的本地版本和文件库的最新版本之间的变化。要做到这点，我们需要用一个特殊的标签 `HEAD`，这个标签总是和文件库最新的版本关联。


```
work/client> cvs diff -r HEAD File1.java
Index: File1.java
=====
RCS file: /Users/dave/sandbox/proj/File1.java,v
retrieving revision 1.3
diff -r1.3 File1.java
10c10,12
<      total += amount;
---
>      if (amount.isPositive()) {
>          total += amount;
>      }
14,16c16
<      public Money getTotal() {
<          return total;
<      }
---
```

变化和补丁

如果你曾经在开源的社区里花时间来开发，你肯定遇到过全世界给代码打补丁的家伙。这些补丁基于 CVS 生成的 **diffs**，而且最终是非常有用的。也许你正在一个开源库上工作，也许你需要修改。这些库都保存在 **SourceForge** 上，**SourceForge** 为开源的开发者提供了免费的 CVS 文件库以及其他的 service。作为一个公用的成员，**SourceForge** 允许你从文件库 **check out** 项目的源码，但是如果你不是开发成员，你就不能 **check in**。

这就是补丁。让 CVS 例出你所做的所有修改（**cvs diff** 命令）。将包括了修改的文件打包发邮件给库维护人员，他们将使用补丁程序将这些补丁打到他们代码里。这里我们用到的唯一的新的功能是在 **cvs diff** 命令上使用 **-u** 选项。

```
oslibrary> cvs diff -u >diff.list
```

然后你就可以把这个文件发给维护人员，他们会将这些补丁打到他们的源文件上。

```
oslibrary> patch -p0 <diff.list
```

就算不是开源的，补丁也非常有用。你可以利用补丁的形式向项目的其他成员发送建议性质的修改。如果你的客户端拥有的你源码，你甚至可以发布非常紧急的修复补丁，诸如时不时半夜三点就会来一次那种。最好记得把你修改的 **check in** 文件库。

6.8 处理合并冲突

CVS 不会锁定文件，一个项目里的每个人都能在任何时候修改任何文件。CVS 的一个

特征有点象让人通宵达旦的工作。什么可以阻止两个人同时编辑同一个文件？工作的成果会丢失吗？回答很简单，就是没有什么可以阻止，工作成果也不会丢失。如果他们编辑的是同一文件不同部分，CVS 能够恰当地将修改合并到一起，一切都会继续下去。

然后有时候两个人会编辑同一文件的同一部分(尽管这种发生的机会比你想想的要小的多)。当那种情况发生的时候，CVS 不能自动进行合并，CVS 不知道该保存哪个的修改。在这种情况下，CVS 宣告两个文件版本发生冲突，并将问题留给人手去解决。为了举例说明一个冲突，我们再次动用我们的老朋友 Numbers.txt。这次我们会将这个文件 check out 到两个独立的 workspace。

```
work> cvs co -d proj1 project
cvs checkout: Updating proj1
U proj1/Numbers.txt
work> cvs co -d proj2 project
cvs checkout: Updating proj2
U proj2/Numbers.txt
```

在 proj1 目录下，我们修改 Numbers.txt 文件的第一行，文如下：

文件 Numbers.txt:

```
ONE
two
three
```

然后将这个文件 check in。

```
proj1> #... edit ...
proj1> cvs commit -m "Make 'one' uppercase"
cvs commit: Examining .
Checking in Numbers.txt;

/Users/dave/sandbox/proj/Numbers.txt,v <-- Numbers.txt
new revision: 1.2; previous revision: 1.1
done
```

现在我们转战 proj2。记住我们要制造一个合并冲突，因此假设我们不知道别人已经修改了我们要工作的文件。在 proj2 里面，我们修改文件 Numbers.txt，将第一行改为 One

```
proj2> cvs commit -m "Capitalize 'One'"
cvs commit: Examining .
cvs commit: Up-to-date check failed for 'Numbers.txt'
cvs [commit aborted]: correct above errors first!
```

到现在为止一切都还好。不过在更新之前还不能做 check in，因此我们要先做 cvs update。


```
proj2> cvs commit -m "Capitalize 'One'"
cvs commit: Examining .
cvs commit: Up-to-date check failed for 'Numbers.txt'
cvs [commit aborted]: correct above errors first!
```

CVS 告诉我们两个事情：存在合并冲突，需要我们修复。

解决冲突

在解决合并冲突的时候第一需要回答的问题是：首先为什么这种情况发生了？这并不是某个人的过失问题，常常是交流方面的问题。为什么两个开发人员同时变价同一文件的同一行代码？

有时候能找出好的原因。也许两个人同时发现了同样的问题，两个人都想解决这个问题。或者也许他们都添加了使用相同的数据结构的某种功能，同时为那种结构添加字段。这些都是事出有因，它们都会导致冲突。

但是冲突常常都是大家工作不得力，没有让别人知道工作进展的如何而造成的。因此我们强烈推荐如果你遇到一个无法合理解释的冲突的时候，你应该重点强调一下，在下次会议的时候一定要谈谈这个问题。我们的目的是讨论引起这种情况的原因，提出增进交流的方式，才能减少在将来发生类似的事情的机会。

现在一切都还好，但是你仍然遗留了一个冲突。CVS 将文件的这部分用>>>和<<<符号括起来。

文件 Numbers.txt:

<pre><<<<<< Numbers.txt One ===== ONE >>>>>> 1.2 two three</pre>	<div style="display: inline-block; vertical-align: middle; margin-right: 10px;">}</div> <div style="display: inline-block; vertical-align: middle; text-align: left;"><p><i>Your local changes</i></p><p><i>Changes in the repository</i></p></div>
------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

你可以看到我们自己的新的修改(One)和原来的修改(ONE)。在很多方面这都象内容比较的输出结果（第 71 页）。

我们现在不得不决定怎么样去修复它。在真实的世界里，这涉及到和另外一个修改这个地方的人进行协调的问题。如果只是简单随意地将其他的人工作抛开，用你的修改代替别人的修改，这可能会严重伤害你和同事的感情，下一次项目聚餐恐怕就很难邀请到人家了。

解决方案有很多种方式:

- 你决定放弃自己的修改, 使用文件库的版本。这种情况下, 只需要删除自己的文件, 然后进行 `update` 就可以了。

```
proj2> rm Numbers.txt
proj2> cvs update
cvs update: Updating .
cvs update: warning: Numbers.txt was lost
U Numbers.txt
```

- 如果你想使用两个版本, 那么你需要进行手工编辑。简单地包括冲突的标记的文件, 按自己的需要修改, 并确定删除了冲突的标记。例如, 在我们这个例子里第一行只能是 `First`, 不能是 `One` 或者 `ONE`。
- 你也可以决定保持自己的修改, 放弃文件库的修改。如果你碰巧还有编辑器缓冲区里的文件版本, 只需要保存文件, 然后进行 `cvs commit` 就可以了。这个版本就保存在文件库里。如果你不想让它无所事事, 你不得使用下一个技术。

文件 `Numbers.txt`:

```
<<<<<< Numbers.txt
One
=====
ONE
>>>>>> 1.2
two
three

⇒

First
Two
Three
```

冲突和代码布局

在评论本书草稿的时候, Andy Oliver 提出了关于冲突和代码格式的很好的观点问题如果象下面这样:

Fred 习惯代码用两个字符来进行缩进格式, 而 Wilma 喜欢四个字符的宽度。

Fred:

```
for (i = 0; i < max; i++) {  
    if (values[i] < 0) {  
        process(values[i]);  
    }  
}
```

Wilma:

```
for (i = 0; i < max; i++) {  
    if (values[i] < 0) {  
        process(values[i]);  
    }  
}
```

某一天 **Fred** 碰巧要编辑 **Wilma** 的代码，他不喜欢这种缩进的格式。于是他用他的编辑器将整个文件的缩进改为两个字符为单位。然后对某一行进行了小小的修改，保存文件，并向文件库提交修改。

问题是 **CVS** 关心的：文件的每一行都被修改了。如果 **Wilma**（或者任何其他人）做了某些修改，他会得到合并冲突的警告，因为 **Fred** 修改了缩进单元，这意味着文件库的每一行和 **Wilma** 的 **workspace** 的已经不一样了。

现在你可以考虑：你可以使用 **-b** 选项，让 **cv diff** 命令在比较文件的时候忽略 **whitespace** 的变化。然而这并不改变你已经修改了整个文件的事实，那些对文件进行了本地修改的人员会在下次 **update** 的时候发生合并冲突。

规则很简单：千万别修改一个共享文件的布局。如果你确实需要修改缩进，首先确定团队的其他人没有修改这个文件。然后修改布局，并 **check in** 修改过的文件。然后让其他人进行 **update**，这样他们就会在新的版本上工作了。这样减少冲突的机会，也减少了别人发给你表示憎恶的邮件。

6.9 提交修改

在你进行了一系列的修改后（在理想的世界里，要保证能够通过测试，不会破坏任何事情），你会将这些修改保存到文件库。我们在这本书里已经多次这样做，你要使用 **cv commit** 命令。

然而我们更愿意在每次提交的时候推荐一种更复杂一点的命令序列。

```
work/project> cvs -q update -d
work/project> #... resolve conflicts ...
work/project> #... run tests ...
work/project> cvs commit -m "check in message"
```

第一行命令将我们的本地 **workspace** 和文件库的当前状态进行同步。这很重要，尽管我们的代码可能与项目文件工作的很好，但是在我们自上一次 **update** 我们的 **workspace** 后，其他的人可能修改了一些可能会破坏我们的新的代码的文件。在 **update** 之后，我们可能不得不处理冲突的代码。

即使没有冲突，我们也应该重新编辑代码，并重新运行测试，有任何问题都应该即刻进行修改。这样才能保证当我们在 **check in** 的时候，我们所 **check in** 的是可以在一个大项目里实际运行的代码。

一旦我们检查一切都正确之后，就可以提交我们的修改。使用 **-m** 选项可以添加一个有意义的注释消息，如果你忽略了 **-m** 选项，CVS 会弹出一个编辑框，让你输入注释。注意后面的工具栏的关于日志消息的建议。

6.10 查看修改的历史信息

你可以使用 **cvs log** 命令查看你和你的团队输入的日志信息。

```
doc/StarterKit> cvs log pragprog.sty
RCS file: /home/CVSR00T/PP/doc/StarterKit/pragprog.sty,v
Working file: pragprog.sty
head: 1.5
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 5;    selected revisions: 5
description:
-----
revision 1.5
date: 2003/06/01 13:34:54;  author: dave;  state: Exp;  lines: +30 -5
Added support for colors in code and files.
-----
revision 1.4
date: 2003/05/30 18:30:44;  author: andy;  state: Exp;  lines: +17 -1
Fixed up spacing of callout macros.
-----
revision 1.3
date: 2003/05/30 16:21:44;  author: andy;  state: Exp;  lines: +2 -0
Added convenient \CF macro (we can rename this if you want)
-----
revision 1.2
date: 2003/05/30 13:25:58;  author: dave;  state: Exp;  lines: +32 -0
Fix problem with embedding files.
-----
revision 1.1
date: 2003/05/30 03:48:34;  author: dave;  state: Exp;
Initial load
```

有意义的日志消息

怎么样算一个好的日志消息？要回答这个问题就要假想你是另一个开发人员，几年后重新面对这些代码的情形。你被系统的一部分奇怪的代码搞的很迷惑，你尽力找出这样做的原因。你注意到在这个地方所做的修改，希望通过日志消息能够得到一些暗示：为什么要这样特别的设计，当初的动机是什么？

回到现实，怎么样编写今天的日志信息才能帮助那些几年后的后续开发人员？

部分答案来自对 CVS 的认识：CVS 已经保存了你对代码进行修改的细节。因此没有必要再写一个这样的日志：将 `timeout` 修改为 `42`，简单的 `cvs diff` 命令就能显示出 `setTimeout(10)` 已经变成了 `setTimeout(40)`，取而代之，应该用“为什么这样做”的答案作为日志消息。

```
If the round-robin DNS returns a machine that
is unavailable, the connect() method attempts
to retry for 30mS. In these circumstances our
timeout was too low.
```

如果某个修改是根据 bug 的报告而做的，那么日志消息里要包括该 bug 的跟踪编号，对于问题的描述已经保存在 bug 数据库里，因此这里不用再重复了。

最后，我们觉得使用完全空白的日志消息也是完全可以接受的，但是必须是没有没什么可说明的时候。例如，我输入了一篇文档，每隔 30 分钟就停下来进行一次格式化。如果成功了，为了安全起见我就将其 **check in**，因此我只需要做

```
work> cvs commit -m ""
```

但是，如果我修改某些会影响文档的创建的部分（例如共用的宏），我就要加一些解释性的日志消息。

为挑选个别的修订版本进行汇报，你可以多次使用 **-r** 选项。你也可以使用 **-d** 选项。一个 **-d** 选项汇报了在指定日期前的最后一个版本，两个 **-d** 选项则汇报了在该日期范围内的各版本。一个比较有用的方式是检查最后两天的活动情况。

```
work> cvs log -d "2 days ago" -d today
```

Line-by-Line 历史

cvs annotate 命令显示一个或多个文件的内容。对于文件的每一行都会显示出最后一个修改了该行的修订版本号，以及作者，日期。

```
doc/StarterKit> cvs ann OurColors.sty
Annotations for OurColors.sty
*****
:      :      :
1.3    (dave 05-Jun-03): %%
1.3    (dave 05-Jun-03): %% Colors for sections and chapter titles
1.3    (dave 05-Jun-03): %%
1.3    (dave 05-Jun-03): \definecolor{SECCOLOR}{rgb}{.2, .2, .2}
1.3    (dave 05-Jun-03): \definecolor{SUBSECCOLOR}{rgb}{.2, .2, .2}
1.3    (dave 05-Jun-03): \definecolor{SUBSUBSECCOLOR}{rgb}{.1, .1, .1}
1.3    (dave 05-Jun-03):
1.4    (andy 05-Jun-03): %%
1.4    (andy 05-Jun-03): %% The rule under captions
1.4    (andy 05-Jun-03): %%
1.4    (andy 05-Jun-03): \definecolor{CAPTIONRULECOLOR}{rgb}{.4, .4, .4}
1.4    (andy 05-Jun-03):
1.6    (dave 10-Jun-03): %%
1.6    (dave 10-Jun-03): %% The color of line numbers in code listings
1.6    (dave 10-Jun-03): %%
1.6    (dave 10-Jun-03): \definecolor{LINENUMBERCOLOR}{rgb}{.4, .4, .4}
1.3    (dave 05-Jun-03):
```

当你相对软件进行深究的时候，这是一个伟大的工具。你能迅速找出改变的模式，以及精确识别某个特定版本修改了哪些行。

cvs annotate 命令支持单个 **-r** 或 **-D** 选项，这些选项能够指定注释（**annotation**）的版本

号或日期。

6.11 取消修改

有时候我们宁愿没有对代码进行修改。

如果这是在本地的 `workspace` 的一系列修改，还没有 `check in`，那么我们可以从本地 `workspace` 删除这些文件，然后从文件库更新本地 `workspace` 即可。

如果这些修改已经提交了，`CVS` 可以帮助我们取消。要做到这点有多种方式，这里我们会提供我们认为最简单和最少出错的一系列步骤。这次我们假设我们工作的对象是联系管理系统。我们正在准备 `beta` 发布的前期准备工作，一切进展顺利，突然客户打来一个紧急电话，我们从客户联系地址列表中删除某一个客户的联系方式，结果这个客户的所有信息都从数据库里删除了。

第一步是确定我们的文件是最新的。

```
proj> cvs -q update -d
```

然后是识别我们需要删除或撤销的准确的修订版本。用 `cvs log` 命令比较有用。首先让我们看看和联系管理的主类的文件有关的日志。

```
proj> cvs log Contacts.java
RCS file: /Users/dave/sandbox/proj/Contacts.java,v
Working file: Contacts.java
head: 1.5
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 5;      selected revisions: 5
description:
Manage contact list
-----
revision 1.5
date: 2003/06/11 16:36:17;  author: dave;  state: Exp;  lines: +2 -0
Reformat PMB addresses
-----
revision 1.4
date: 2003/06/11 16:35:40;  author: fred;  state: Exp;  lines: +1 -0
Remove client from db too
-----
revision 1.3
date: 2003/06/11 16:35:11;  author: jane;  state: Exp;  lines: +2 -0
Sort clients into alpha order (CR:142)
-----
revision 1.2
:           :           :           :           :
```


修订版本 1.4 看起来很可疑,于是我们使用 `cvs annotate` 或 `cvs diff` 命令来精确查看 1.3 与 1.4 之间的变化。

```
proj> cvs diff -c -r1.3 -r1.4 Contacts.java
Index: Contacts.java
=====
RCS file: /Users/dave/sandbox/proj/Contacts.java,v
retrieving revision 1.3
retrieving revision 1.4
diff -c -r1.3 -r1.4
*** Contacts.java      2003/06/11 16:35:11      1.3
--- Contacts.java      2003/06/11 16:35:40      1.4
*****
*** 15,20 ****
--- 15,21 ----
        public void removeClient(Client c) {
            clientList.remove(c);
+       database.deleteAll(c);
        }
```

这看起来有点象这个问题。然而,在我们开始放手动别人的代码之前,先做一些调查研究。通过观察日志,我们发现 Fred 做了特别的修改,因此我们找到他并与之交谈。结果是一个简单的误解。Fred 不知道这个调用会删除客户的所有记录。撤销这个修改看来是对的。

现在我们要撤销 Contacts.java 文件版本 1.3 和 1.4 之间的修改。首先让我们看看命令。

```
proj> cvs update -j1.4 -j1.3 Contacts.java
RCS file: /Users/dave/sandbox/proj/Contacts.java,v
retrieving revision 1.4
retrieving revision 1.3
Merging differences between 1.4 and 1.3 into Contacts.java
```

-j 选项告诉 CVS 你要将修改合并到当前的工作的版本。当我们指定两个-j 选项的时候,则是让 CVS 合并这两个版本间的变化。然后有一些技巧性:我们颠倒了修订版本的顺序,先是 1.4,然后才是 1.3。这是让 CVS 将 1.4 的修改回滚到 1.3,然后再将这个修改应用到我们当前的工作的版本。

这个时候,我们可以回到正常的流程了。源代码已经被修改,我们需要先测试,然后提交到文件库。

```
cvs/proj> cvs commit -m "Revert deleteAll change from 1.4"
```

恢复大的修改

前面的例子我们只是处理了单个文件。如果涉及到多个文件,我们该怎么处理呢?答案要根据情况而定。

一种方式是依次对每个文件应用前面使用的对单个文件处理的方式。这样有必要得到每

Matrix-与 Java 共舞 (<http://www.Matrix.org.cn>)

个文件的独立的修订版本号。

一个好的方式是预测那些可能引起争议的修改，然后在文件库中给它们打上标签。这些标签对所有的文件有效，跟他们的各自的修订版本号没有关系。然后你可以使用象-r选项那样的技巧，使用标签名来取消这些修改。

```
proj> cvs update -j after_change -j before_change
```

第七章 使用标签和分支

CVS 的大部分都很简单：从文件库更新文件，编辑文件，测试通过后再保存文件。然而很多开发人员放弃了标签和分支。也许他们以前所属的团队滥用了分支，他们的文件库结构图看起来更象一碗绞成一团的意大利面条，而不是一个可控制的直线式的开发图。或者也许他们以前所属的团队合并分支间的修改一再被推迟，知道最后他们发现要解决冲突简直如同恶梦一样。或者也许正是因为分支功能提供的弹性，有如此多的选择，以至于让他们无从着手。

实际上，标签和分支是可以（也应该是）以简单的方式来使用的。而其中的技巧就是在正确的环境下使用。本章我们将展现两种场景，我们感觉在这些场景下应该使用分支，我们将在这些场景下生成发布，并提供开发人员一个试验的地方。

超出这些的范围，我们建议在你给文件库添加分支之前一定要好好想想。过分的进行分支很快就能导致任何文件库没法使用。

在讨论特定的诀窍之前，我们需要从整体上讨论一下标签和分支。

7.1 标签，分支和打标签

标签是一种象征性的符号。标签的名字以字母打头，可以包括字母，数字，连字符，下划线。例如：REL 1 0, rev-99 和 TRY DT 031215 都是有效的标签名，而 REL 1.0, Bug(123) 和 q&a 则都是非法的。

标签有两种类型。规则标签给某个时间点上某个模块里的文件命名。分支标签则用于给文件库的一个完整的分支命名。

规则标签

CVS 文件库的每个文件都拥有自己的版本序列号。典型的是一个文件一开始的版本是 1.1，然后随着每次新版本 check in，版本号会变大（1.2，1.3，依此类推）。然而每个文件都有自己的独立的序列号。如果你新项目开始的时候有三个文件，文件 a，文件 b，和文件 c，如果你对文件 b 编辑了两次，对文件 c 编辑了 1 次，每编辑完一次就做 check in，那么最后你将会得到：

File	Version
file_a	1.1
file_b	1.3
file_c	1.2

这三个文件反应了你的应用程序的当前的状态。如果我们现在就要发布应用，我们要发布文件 **a** 的 **1.1** 版本，文件 **b** 的 **1.3** 版本，文件 **c** 的 **1.2** 版本。没有一个单独的版本号能够表达当前发布的应用。这就是规则标签发挥作用的地方了。标签创建了一个内部列表（包括了每个文件的当前版本号）。例如我们可以将文件库的当前状态打上标签 **REL 1**，从今以后，我们就能够使用 **REL 1** 标签 **check out** 源文件，得到文件 **a** 的 **1.1** 版本，文件 **b** 的 **1.3** 版本和文件 **c** 的 **1.2** 版本。

文件库里的标签你想要多少就有多少，但是标签名必须唯一。对于某个文件，多个标签可以同时指向同样的版本号（如果文件 **b** 在这一次以及下一次的发布之间没有任何修改，那么 **REL_1** 和 **REL_2** 都指向其版本 **1.3**）。

分支标签

我们第一次谈到分支是在第 **16** 页，那个时候我们还在讨论在版本控制系统里面怎么样使用分支来处理发布问题。分支代表了文件库历史状态的一个分叉，自这个分叉以后，同样的文件就拥有两个或者更多的独立的修改集，每个修改集都存在于一个独立的分支里。当年你在 **CVS** 里创建分支的时候，要给分支一个分支标签。这个标签代表了分支从其基础的代码线分叉的点。以后你使用到这个分支标签的时候，就好像你在访问分支被创建的那个点的文件状态。

实践

用标签和分支有很多种可能。但是过分地使用标签和分支会导致很明显的混乱。为了尽量简单化，我们建议开始的时候你只为以下四种目的才使用标签：

发布分支 我们建议将项目的每次发布都放在一个独立的分支里面。发布分支标签用于命名该分支。

发布 发布分支将包含一个（如果可能则是多个）发布：发布标签用来表示项目打包的点。

缺陷修复 正式报告的缺陷要在发布分支上修复，然后将修复部分合并到其他分支和主线。

两个缺陷修复标签用于标示出缺陷修复前后的时间点。

开发试验 有时候某个子团队要对项目的基础代码进行深入的修改。在这些修改的过程中，这些代码势必跟系统的其他部分不兼容，并且会破坏主创建。开发人员可以选择创建一个名

Matrix-与 Java 共舞 (<http://www.Matrix.org.cn>)

为"developer experiment"的分支，在这个分支上进行修改。

Thing To name	Tag Style	Examples
Release branch	RB_ <i>rel</i>	RB_1.0 RB_1.0.1a
Releases	REL_ <i>rel</i>	REL_1.0 REL_1.0.1a
Pre bug fix	PRE_ <i>track</i>	PRE_13145 PRE_4129
Post bug fix	POST_ <i>track</i>	POST_13145 POST_4129
Developer experiments	TRY_ <i>initials_yymmdd</i>	TRY_DT_030631 TRY_AH_021225

Table 7.1: Possible Tag Naming Conventions

在你的团队里使用一致的标签命名约定是一个好主意。表 7.1 展示了一些简单的模式，也是我们在本书里使用的。在表 7.1 里，**rel** 用于发布（使用下划线），而 **trach** 用于缺陷跟踪。

现在让我们看看在使用这些不同的标签的时候，怎么样进行分支和打标签的一些诀窍。

7.2 创建发布分支

在你的软件生涯中总会想生成发布应用。每次发布的日子临近的时候，注意力就从添加新功能上转移开来，取而代之的是紧紧地关注发布的细节。尽管刚开始的时候整个团队会参与这个过程，但始 **there'll come a time when the law of diminishing returns takes effect**，拥有一个专注于整理发布代码的发布子团队会有更高的效率。如果子团队在主干上工作，那么其他人就要停下来等他们完成。

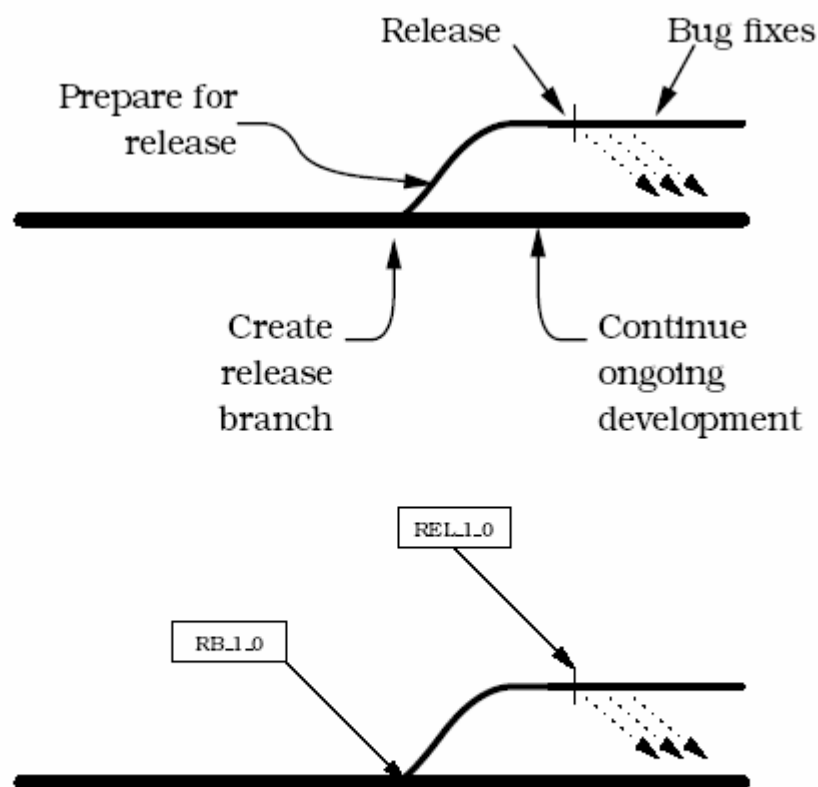


Figure 7.1: Tagging a Release Branch

取而代之，在这个时候，将要发布的代码移动到他们各自的分支。然后在发布团队在那个分支上工作的时候，项目的其他人还可以继续在主线上工作。在发布的时候，在发布分支的那个点上将打上发布号的标签。发布团队在发布分支上的修改可以合并到主线上，如图 7.1。我们在 18 页的时候描述抽象团队的分支的时候已经看到过这个图的上半部。图的下半部也是指的同一个文件库，但是要说明的是我们即将应用的标签。

使用 `cvs rtag` 来软件发布分支。在将该标签应用到文件库的当前版本的时候，要确定所有人都已经 `check in`，这样文件库才是最新的。接下来的例子里面，我们将运行 `cvs commit` 命令，以确定我们的 `workspace` 已经 `check in`，然后再使用 `cvs rtag` 命令创建发布分支（为 CVS 模块项目的发布 R1.0 而建）。

```
work/project> cvs commit -m ""
cvs commit: Examining .
work/project> cvs rtag -b RB_1_0 project
cvs rtag: Tagging project
```

Rtag 命令的 -b 选项让 CVS 创建一个分支并打上我们提供的分支标签 (RB_1_0)。然而我们并没有做任何影响本地 workspace 的事情，主线仍然继续工作。下一个技巧是怎么样 check out 发布分支。

7.3 在发布分支上工作

要访问发布分支，需要 check out 指定分支标签的项目。你可以在你当前的项目目录下进行（这样你将用发布分支的内容代替当前 workspace 下的文件），也可以在一个独立的本地目录下进行。我们推荐后者，因为后者更少引起混淆，而且简化了同时在两个分支上的工作。我们更改到 work 目录下，然后 check out，指定发布分支并修改默认的目录名，这样发布分支的代码将被 check out 到目录 rb1.0 下。当你 check out 一个分支的时候，你实际上 check out 了该分支下最近的文件，这和 check out 主线的最新开发的文件版本一样的道理。

```
work/project> cd ..
work> cvs co -r RB_1_0 -d rb1.0 project
cvs checkout: Updating rb1.0
U rb1.0/file_a
U rb1.0/file_b
U rb1.0/file_c
```

现在如果我们编辑保存在 check out 出来的发布分支的文件的目录下某个文件，并提交修改，我们可以看到 CVS 将修改添加到该分支，而不是主线。你可以通过该文件上的版本号来区分，这是基于主线 1.3 版本分离出来的分支。

```
work> cd rb1.0
work/rb1.0> # ... edit file_b ...
work/rb1.0> cvs commit -m "Tidy error msg"
cvs commit: Examining .
Checking in file_b;
/Users/dave/sandbox/project/file_b,v <-- file_b
new revision: 1.3.2.1; previous revision: 1.3
done
```

我们现在可以继续修改文件，为实际的发布准备了。

7.4 Generating a Release 生成发布

在所有的整理完成，验收测试也通过之后，团队决定生成一个发布版本。最重要考虑的是要确定我们在正确的分支上将正确的文件集打上标签，这样我们才能准确地知道发布的内

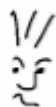
容。

我们可以使用 `cvss rtag` 命令，但是那样我们就不得不和团队的其他人的代码进行同步，这样在执行这个命令的时候，我们才能保证文件库的文件是稳定的。一个更好的方法是使用一个巧妙的别的命令 `cvss tag` 来给文件打标签，除了要使用我们 `check out` 到本地 `workspace` 的版本号以外，来决定怎么样引用这个标签，其他都和 `cvss rtag` 命令一样。下面的命令是：本地的文件测试通过之后，将所有的文件提交文件库，并为发布创建了标签 `REL_1_0`。

```
work/rb1.0> cvss update
work/rb1.0> # ... run tests ...
work/rb1.0> cvss commit -m "..." # if needed
work/rb1.0> cvss tag REL_1_0
cvss tag: Tagging .
T file_a
T file_b
T file_c
```

- 现在开始，开发人员可以指定发布标签，将代码 `check out` 出来用于创建发布。

```
work> cvss co -r REL_1_0 -d rel1.0 project
cvss checkout: Updating rel1.0
U rel1.0/file_a
U rel1.0/file_b
U rel1.0/file_c
```



Joe的问题。。。tag? rtag?怎么回事?

CVS 为给文件和模块打标签这个功能提供了两个命令。它们之间有什么区别？什么时候使用它们？

名字中就隐藏了线索：`cvss rtag` 命令给文件库中的一个模块打标签，而 `cvss tag` 则给本地 `workspace` 中的文件打标签。

因为 `rtag` 要使用文件库，我们不用在本地 `workspace` 目录下使用这个命令（即使我们在本地 `workspace` 目录下，整个命令仍然会自动使用文件库的 `ROOT`）。要使用 `cvss rtag` 命令，你只需要提供一个标签，名和一个模块名。默认的操作是将该标签应用于文件库中该模块当前的 `HEAD` 版本之上（我们已经说过你也可以做更多，这些命令的详细描述请参考 140 页）。

`cvss tag` 命令则不一样，你只有在位于本地 `workspace` 目录下时候才能使用。该命令给最近一次 `check out` 或者 `update` 的文件打上指定的标签名。这有很重要的意义，如果你

变价一个本地文件，最后保存的时候肯定会有一个新的版本号。然而如果你在提交前执行 `cvcs tag` 命令，那么这个标签和上一个版本关联（如果你使用了 `-c` 标志，`cvcs tag` 命令在应用该标签前会检查本地文件是不是没有被修改过。）

因此在给文件库端的状态打标签的时候应该使用 `cvcs rtag` 命令（例如你要生成发布版本），而 `cvcs tag` 试用于本地的情况（例如你要修改 bug）。

7.5 在发布分支修改 bug

发现 bug。处理它们的技巧是使用一套约束的习惯。在发布分支，这意味着我们需要跟踪修复 bug 所做的修改，并确定我们将修复代码应用到所有包括同样错误的分支。后面的问题特别重要。分支天生就是代码的重复。也就是说如果你再一个分支中发现了一个问题，那么很可能这个问题也存在于另一个分支（毕竟因为源码一开始就是相同的，错误自然也一样）。我们需要将在发布分支的修复代码应用到主线上，也许还需要应用到其他的发布分支（如果它们也包括了同样的错误代码）。

没有版本控制，这将是一个棘手的问题。有了版本控制我们就能更好的控制这个过程。在修改错误的时候我们让版本控制系统跟踪修改的源码，然而合并到其他受影响的分支。理论上步骤如下。首先修改错误，使用标签，孤立我们修改的部分。

- 将包含错误的分支 `check out`，保存到本地 `workspace`。
- 将文件库打上 `pre-fix` 标签。
- 用测试重现错误，修复代码，并验证创建。
- 向文件库提交修改。
- 将文件库打上 `post-fix` 标签。

现在我们深入所有其他被影响的分支（包括主线），并将我们做所的修改合并到这些分支。修改部分能够通过比较 `pre-fix` 和 `post-fix` 标签之间的区别而得到。

在 CVS 下我们使用下面的诀窍。假设这个错误在 1.0 分支里被报告出来，其跟踪号为 1234。

```
work> cvs co -r RB_1_0 -d rb1.0 project
cvs checkout: Updating rb1.0
U rb1.0/file_a
U rb1.0/file_b
U rb1.0/file_c
work> cd rb1.0
work/rb1.0> cvs tag PRE_1234
cvs tag: Tagging .
T file_a
T file_b
T file_c
work/rb1.0> # create test, fix problem, validate
work/rb1.0> cvs commit -m "Fix PR1234"
cvs commit: Examining .
Checking in file_c;
/Users/dave/sandbox/project/file_c,v <-- file_c
new revision: 1.2.2.1; previous revision: 1.2
done
work/rb1.0> cvs tag POST_1234
cvs tag: Tagging .
T file_a
T file_b
T file_c
```

现在我们需要将修复结果应用到主线上。首先要到主线的 `workspace` 的目录下，并确定是最新版本，然后从发布分支合并修复结果。最后，运行测试，如果测试通过就可以向主线提交修改。

```
work/rb1.0> cd ../project
project> cvs update
cvs update: Updating .
project> cvs update -j PRE_1234 -j POST_1234
cvs update: Updating .
RCS file: /Users/dave/sandbox/project/file_c,v
retrieving revision 1.2
retrieving revision 1.2.2.1
Merging differences between 1.2 and 1.2.2.1 into file_c
project> # ... test ...
project> cvs commit -m "Apply fix for PR1234 from RB1.0"
cvs commit: Examining .
Checking in file_c;
/Users/dave/sandbox/project/file_c,v <-- file_c
new revision: 1.3; previous revision: 1.2
done
```

7.6 开发人员的试验分支

有时候开发人员需要对项目进行大范围修改(例如修改持久层部分或者引入新的安全机制)。象这种对代码的修改最少都要花好几天。这样的修改也不能立刻通过不断增加的方式引入，因为它们影响到太多的代码。而且这样的修改往往是在应用的底层进行的，对系统的

其他部分有深远的影响。

如果单个开发人员想对源码进行大范围修改，他可以在本地 `workspace` 进行。然而这两个潜在的问题。首先，开发人员在修改的时候失去了版本控制的优越性，他们不能恢复以前的工作，市区了修订版本的历史等等。他们没有在中心文件库上工作，这样失去了备份的机会。

如果多个开发人员要进行大范围的修改，则有更大的问题，他们需要共享修改部分，而且修改要基于相同的代码。

答案是将这些试验性质的代码放到版本控制系统的分支下面。然后开发人员在他们各自的 `workspace` 使用那个分支进行修改。完成工作时候，他们可以决定将他们的工作成果集成到主线上。如果他们认定试验是失败的，他们可以放弃这个分支。否则他们只需要简单地将在这个分支上的变化合并到主线上。无论他们怎么决定，将来的工作仍然在主线上继续，分支只是历史。

创建开发分支和创建发布分支完全一致。我们需要将该分支打上 `experimental tag`。

```
work/project> cvs commit -m ""  
cvs commit: Examining .  
work/project> cvs rtag -b TRY_DT_030925 project  
cvs rtag: Tagging project
```

这不会改变当前工作目录下的标签，为了切换到新的创建的分支，你需要指定分支标签先做 `update`。

```
work/project> cvs update -r TRY_DT_030925  
cvs update: Updating .
```

使用带 `-A` 参数的 `cvs update` 命令可以将你的工作文件返还主线。

```
work/project> cvs update -A  
cvs update: Updating .
```

7.7 在试验分支上编码

当你在试验代码分支上工作的时候，你既可以替换当前 `workspace` 的代码（使用带 `-r` 选项的 `cvs update` 命令），也可以 `check out` 到一个独立的目录下。如果你使用 `update` 方法，为可靠起见，你应该确定你在目录的最上一级上运行的该命令。

```
work/project> cvs update -r TRY_DT_092503
cvs rtag: Updating project
```

如果你决定 `check out` 到一个独立的目录（这也是我们推荐使用的），记得使用 `-d` 选项来指定目录名。

```
work/project> cd ..
work> cvs co -r TRY_DT_092503 -d proj_exp project
cvs checkout: Updating proj_exp
U proj_exp/file_a
U proj_exp/file_b
U proj_exp/file_c
work> cd proj_exp
```

7.8 合并试验分支

使用单个 `-j` 标签将试验分支合并到主线上，该标签表示 CVS 应该合并分支的所有变化。在执行该命令之前，你需要确定试验分支上的所有修改都已经 `check in`，而且你要移动到对应主线的 `workspace` 的目录下（这就是我们推荐将试验分支 `check out` 到一个独立的目录下的原因）。

后面的例子里面，我么假设 `proj_exp` 目录下包括了试验的代码，而 `project` 目录下包括了主线的代码。合并过程如下：

```
work> cd proj_exp
work/proj_exp> cvs commit -m "Finalize changes"
work/proj_exp> cd ..
work> cd project
work/project> cvs update -j TRY_DT_092503
cvs update: Updating .
RCS file: /Users/dave/sandbox/project/file_a,v
retrieving revision 1.1
retrieving revision 1.1.4.1
Merging differences between 1.1 and 1.1.4.1 into file_a
RCS file: /Users/dave/sandbox/project/file_c,v
...
```

...

第八章 创建项目

项目这个单词定力非常随意。一个项目可以是一个人一周完成一个 **web form**，也可以是数以百计的人花费数年。但是大部分项目都有一些共同的特征：

- 每个项目都有一个名字。这听起来有点微不足道，但是我们往往会给我们想标示为独立的实体的事物命名。名字不用是一个外部的品牌, **approved by marketing and subject to field tests in major metropolitan areas**.项目的名字属于你的内部组织。
- 每个项目都具备内聚性，项目内部事物一起作用以达成某些商业目标。
- 项目内部组件往往以单元的形式进行维护，而发布的时候则是完整的项目。
- 项目素材共享一套通用的工程学标准和指导方针，使用通用的体系架构。

在把项目置于版本控制系统之下考虑这些方面是非常重要的，因为划分不同项目之间的边界通常很苦难。使用版本控制系统的时候错误的项目结构是失败的主要原因，随着时间的流失，会白白浪费大量的努力。

因此在你的文件库里面创建项目之前，花点时间进行计划是值得的。比方说，你的项目是否是要实现一个用于公司将来的开发的框架结构？如果是，也许这个框架应该在一个独立自主的项目里面，而你的项目以及其他的项目共享该项目。你的项目是否要开发多个独立的组件？如果是那么每个都应该有自己独立的项目。或者你的项目是否是现有的很大的代码的扩展？如果是那么应该是原来的项目的子项目。

8.1 创建初始项目

有四种方式可以在 **CVS** 下创建项目。

- 将现有的源码导入文件库，作为一个新的 **CVS** 项目
- 将你的所有项目都作为公司级的虚拟项目的子项目。
- 先在文件库中创建一个空的 **CVS** 项目，然后将其 **check out** 到客户端，然后在上面添加文件。
- 从现有的 **RCS** 文件库里面拷贝。**RCS** 是一种早期版本控制系统，也是 **CVS** 的前身。

最后两个方法在实践中很少使用。特别是 **cleanslate** 这种方式需要存取文件库，很大可

能将事情搞的一团糟。RCS 拷贝方法非常丑陋，而且也不特别适合（除非你碰巧在 RCS 里面有大量的代码）。因此留给我们的有两个选择：**import** 和

Import 方法

如果你已经有一些源文件（甚至只有 README 文件），你可以使用 CVS 的 **import** 功能来创建一个新项目。后面的例子我们假设你在一个名为 **Wibble** 的项目上工作（全称 **Wickedly Integrated Business to Business Lease Exchange**）。

你需要一个容纳你用于 **import** 的文件的目录（只包括你需要 **import** 的文件，确信在采取下一步动作之前，将所有的各种备份文件和其他垃圾文件都清除干净），并确定自己处于该目录的最高级（在我们这个例子是 **wibble** 目录），然后执行 **cv**s **import** 命令。

```
wibble> cvs import -m "Initial import" wibble wibble initial
```

关于这个命令还有很多要讲，我们一个一个地进行。

-m 选项指明在 **import** 的时候需要记录的日志信息。

跟着是我们放源码的项目的名字。这里我们使用 **wibble** 作为项目名。这个项目添加到文件库的最上层。

后面两个参数可以说既需要又多余。说需要是因为这个命令要求你必须提供，说多余是因为他们确实没有什么太大的意义。118 页我们谈论第三方代码的时候还会看到他们真正的用途，现在只需要提供字符串 **wibble** 和 **initial**，然后继续。

现在你的项目已经 **check in** 了。你应该使用常用的 **cv**s **checkout** 命令将其 **check out**，如果一切正常，你可以将原来用于 **import** 的目录删除。

Uber-project 方法

第二种创建项目的方法某种程度上说是最简单的。但是为简单付出的代价就是：使用的时候必须严格遵守纪律。

这种方法需要你的文件库管理员一开始就创建一个空的文件库。然后要添加一个新项目你需要按照以下步骤：

- 将最上一级的文件库 **Check out**。使用 **cv**s **checkout** 命令，并添加 **-d** 和 **-l** 选项，指定特定的项目名 **"."**。**-d** 选项用于指定 **workspace** 目录名，**-l** 选项告诉 CVS 不要 **check out** 整个文件库的内容，只在 **workspace** 创建最上一级的目录。
- 在此 **check out** 的 **workspace** 里创建你的新项目的目录，并使用 **cv**s **add** 命令将其添加到文件库。

- 释放整个 check out 的 workspace（因为你并不需要整个文件库）。
- 再次 check out，只不过这次要指定你刚才创建的项目的名字。

```
work> cvs checkout -l -d tmp .
cvs checkout: Updating tmp
work> cd tmp
work/tmp> mkdir new_project
work/tmp> cvs add new_project
Directory /Users/.../new_project added to the repository
work/tmp> cd ..
work> cvs release -d tmp
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'tmp': yes
work> cvs checkout new_project
cvs checkout: Updating new_project
work> cd new_project
work/new_project> # edit ...
work/new_project> cvs add ...
```

如果你决定使用这种方法，你需要遵守纪律，保持项目的清晰。将公司的所有项目视为单个大项目，一开始就使用或者修改并不属于自己项目的代码，既具诱惑性又很容易上手。

8.2 项目结构

你的公司也许已经有关于怎么样组织项目源码和目录的标准。如果你使用 Java 来开发，你可以使用 Jakarta 在目录布局方面的协定。如果你目前还没有使用标准，下面有一些基本的建议。

根目录文件

README 尽管看起来难以置信，两年后即使是最热火朝天的项目也会没落下来，到时候恐怕你很难准确记得 Wibble 这个项目到底是干啥的。因此在目录的最高级创建一个名为 README 的文件很有必要。用一小段文字描述一下项目是干什么的，项目要解决的商业问题，使用到的基本技术等等。不是说要多么完整的描述，这只是一个备忘录，当你很久以后重返这个项目的时候还能通过它泛起陈旧的记忆。

BUILDING 另一个文件叫 BUILDING。这个文件给那些将来的代码考古学教提供了一些线索：怎么样从代码重建项目。因为要自动进行创建，这个文档相对短小。105 页的图 8.2 提供了一个例子。

GLOSSARY 再创建一个名叫 GLOSSARY 的文件。形成一个习惯，那就是在这个文件里记录项目特定的术语。这可以帮助将来的开发人员找出 wibble channel 的意义，

也可以在类，方法和变量的命名方面指导项目团队。

第一级目录

大部分项目至少拥有一下的 **top-level** 目录。

doc/ 将所有的项目文档放在 **doc/**及其子目录下，不要忘了添加那些描述决定的备忘录和邮件，通常在 **doc/**下面还会包括不同的文档类型或者项目不同阶段的文档。

如果你的项目依赖外部文档（例如在第三方的 **web** 站点上保存的那些关于算法和文件格式的描述文档），可以考虑将它们拷贝并保存在 **doc/**目录下（当然要在版权允许的情况下）。如果将来这些外部站点不存在了，这让将来的维护人员更轻松一点。如果你不能将这些材料拷贝到你的项目，可以在 **doc/**目录想传见一个 **BIBLIOGRAPHY** 文件，并在里面添加链接和说明性文字。

data/ 许多项目都带有数据（如用来初始化数据库的一些用于查询的表的数据），将这些数据保单独保存在一个位置（if for no other reason that someone, at sometime, will urgently need to find out why we're charging 127% sales tax in Guam）。

db/ 如果你的项目使用了数据库，将跟数据库模式有关的资料都保存在这里。尽量不要养成联机修改模式的习惯。对于每次 **udpate** 都应该让数据库管理员创建 **SQL** 脚本，脚本包括对模式的修改以及数据移植的信息。将这些脚本都保存在文件库，将来可以到在任何版本的数据库间移植。

src/ 在这个目录下保存项目的源码。

util/ 该目录下保存了各种项目特定的有用的程序，工具和脚本。有些团队将该目录命名为 **tools/**。

vendor/ 如果你的项目使用了第三方的库或者头文件，而且你想将其和你自己的代码一起存档，可以在 **vendor/**目录下进行。

vendorsrc/ 有时候一个项目需要导入或者包括第三方的代码（例如可能使用开源的库，需要在应用的生命周期里存取某个特定的源文件）。你可以在 **vendor** 目录下放二进制库文件（也可能是头文件），但也可能想将源文件和库文件分开放。那么可以将源文件放在 **vendorsrc/**下面。118 页关于第三方代码那一章我们会更详细地探讨 **vendor** 源代码。

下一页的图 8.1 展示了 **Wibble** 项目的文件的一种可能的布局。在这个项目里面我们将我们自己的源码（分为客户端和服务端组件）和一些导入的卡员代码放一起（**Junit** 和

Xerces 框架)。

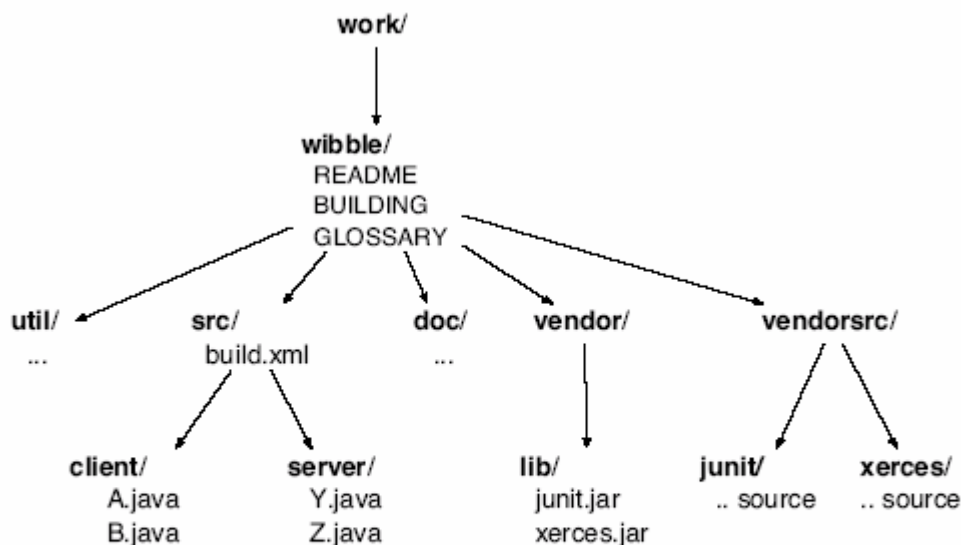


Figure 8.1: Possible Workspace Layout

另外很多项目有自己的一套标准的目录结构用于创建和发布项目。这些目录下并不包括要保存在文件库的文件（因为这些目录下的内容是中间生成的），但是有些团队发现在每个开发人员的 **workspace** 里面保留这样的目录很方便。要做到这样，可以将这些空目录添加到文件库，那样当开发人员 **check out** 的时候这些目录就能出现在他们的 **workspace**（还有等价的选择是不将这些目录保存到 **CVS**，而是用 **build** 脚本根据需要来创建，做完之后就将其整理。如果你使用这个模式，记得将这些目录名添加到你的 **cvsignore** 文件，这样在每次 **check in** 或者 **check out** 的时候 **CVS** 才不会抱怨）。你可能也会想把测试代码放在某个地方，但是有很多存放位置的选择。有些团度喜欢将其保存在和源文件并行的目录，还有一些将其放在源文件目录下用于测试的子目录下。从某种程度上讲正确的答案取决于你使用的语言。例如，**Java** 包命名规则意味着如果你想测试保护型的方法你需要使用并行的目录（或者将你的测试放在被测试的源文件的同一个目录下）。在本系列书之《实效单元测试》这本书里面我们会讲述更多的细节。

在项目里面没有什么关于目录结构的必须遵守的规则。然而在项目间使用一致的约定将很大程度上帮助那些将来加入的开发人员，也会给你在项目间调动的弹性，即使对此没有经验。我现在已经快麻木了（译者：作者经常被各个项目借来借去）。

第九章 使用模块

小的项目在整体上更易于管理，典型地，开发人员在本地 **workspace** 拥有项目的所有源文件，可以对整个项目的源码进行完整测试。然而，当项目（或团队）达到某个大小，我们发现将其划分成子项目是非常有用的。子项目的规则跟所有项目的类似：需要名字，内聚性，维护单元，内部一致性。子项目相对互相独立，也可以一起写作以达成整体需求（如果他们并不需要协作，那么就不应该是子项目，而是 **top-level** 项目）。

一种对大项目的比较有代表性的划分是将其划分为客户端和服务端组件。客户端团队在客户端子项目上工作，服务器团队在服务器子项目上工作。划分子项目是一种在软件中产生干净接口的强制性纪律。可以帮助团队生产独立测试的软件（例如服务器端代码可以不需要客户端代码就能独立测试）。

其他项目也可以水平划分，后台应用可以划分为数据库存取代码，计算模块，外部应用接口等等。

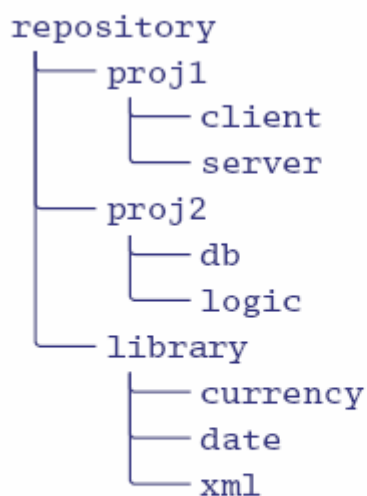


Figure 9.1: Projects and Subprojects

大部分大项目都从划分成小项目中获益，实际上大部分划分包括了水平和垂直方向的划分。在创建一个大项目前花点时间好好想想文件库的代码应该怎么样和项目的架构映射起来。别担心这会超前，因为在进行的时候你是可以调整的。但是在开始的时候适当的规划可以让以后的道路走得更容易。

9.1 组织子项目变的很容易

当你组织源码的目录结构的时候，通常都会将子项目分到各自独立的目录下。例如，图 9.1 展示了一个包括两个项目和一套通用库文件的文件库（这个图只是为了表示文件库的逻辑结构，一个真实的文件库可能有更多的层次结构）。在项目 **proj1** 上工作的团队决定将他们的代码划分为客户端和服务端组件。而 **proj2** 这个团队则决定将他们的代码划分为：一个目录下是数据库访问代码和其他是程序逻辑代码。另外，两个团队在库文件目录下共享一些共用代码。

在这个环境下，你可以手工控制文件库在本地 **workspace** 的大小。例如你可能是 **proj1** 的客户端开发人员。你需要 **xml** 和 **data** 库用于 **build** 应用程序。



Figure 9.2: A Developer's Workspace

你可以使用如下命令 **check out** 文件库的三个部分。

```
work> cvs co proj1/client
work> cvs co library/xml
work> cvs co library/date
```

默认地，CVS **check out** 文件库的这些部分，并放在一个映射文件库的目录结构下。开发人员的 **workspace** 看起来可能很象图 9.2。

也可能覆盖 CVS 的默认行为，**cvs checkout** 命令可以接受一个可选择的参数，该选项指定存放 **check out** 的文件的位置。但是这个是一个要小心使用的功能。想知道为什么，我们需要看看我们是怎么样 **build** 项目代码的。

首先我们假设你使用第三个我们推荐的项目，而且并为之实现了自动 **build** 系统。找个系统将自动从文件库拿项目源文件，并运行 **build** 脚本进行编译，链接项目源码。要做到这样，**build** 脚本需要知道所有源码的位置。

如果我们正好在一个项目上工作，所有的源文件都保存在单个目录下，这样组件之间的

相对路径是一致的。但是，一旦我们在 **build** 过程中包括了第二个模块，一切就变得有趣起来。现在我们的代码必须能够访问其他模块的代码。怎么样做到这点呢？不同的团队有不同的解决方案：

- **定制开发环境。** 每个开发这都必须设置他们使用的 **IDE** 参数，将其指向恰当的目录。自动 **build** 过程也有。尽管很常见，这是一个很容易出错的过程。在同一台机器将不同版本的库文件多次 **check out** 是很常见的（例如你可能在主线上开发的同时还要修复上一个版本的错误）。如果你忘记更新所有的 **build** 参数，你就会用旧的库来 **build** 新的代码，反之亦然。如果你的项目在 **build** 的时候使用不同的操作系统，这种方式特别不方便。象这样依赖手工的过程并不是个好主意。
- **使用环境变量。** 不同的 **build** 脚本使用环境变量来参照不同的模块目录，目录的内容则使用相对参照。这比在每台机器上都手工配置 **build** 脚本要好一点。一次性在各个机器上配置好环境变量。也意味着 **build** 要 **check in** 文件库，脚本对于每台机器上都是一样的，只需要改变环境变量就可以了。但是这也会带来多版本问题，在相同代码的不同版本间切换很容易出错。象 **Eclipse** 这样的 **IDE** 对这种操作方式提供了支持，你可以设置机器级的配置参数，然后在每个项目的 **build** 指令里参考这些参数。
- **使用相对路径。** 除了让每个用户针对自己的机器配置 **build** 之外，还可以在每台机器上自包含和自识别 **build** 环境。我们通过指定规则来完成这个功能，那就是在我们 **check out** 的时候，在文件库的任何两样东西之间的相对路径完全一致。换个方式说，也就是在 **workspace** 里的 **check out** 出来的源文件目录结构总是能够映射到文件库的目录结构（尽管 **workspace** 里面只包括了文件库里的资料的一个子集）。这是组织的很好的方式：所有的 **build** 脚本都知道其他的所有东西的位置，因为相对路径总是相同的。在不同机器间切换的开发人员发现每个 **check out** 的目录结构都是一致的。

因此我们强烈向你推荐：

- 选择一个可以有效工作的目录结构
- 在文件库里使用该结构
- 坚持保证所有的开发人员使用该结构进行 **check out**

一旦你适当地使用这样的结构，**build** 工具知道去哪里查找库文件，以及其他项目组件，

开发人员可以从一台机器换到另一台机器,但是不用花费时间去搞清楚每个机器是怎样配置的。

多文件库项目

CVS1.11 以及后续版本支持从多个文件库 **check out** (或者从同一文件库的多个位置) 代码,并将结构存放在已知的 **check out** 目录的子目录下。如果你在根目录级执行 **update** 或者 **commit** 命令, CVS 能够在不同子目录间自动切换文件库。

在一些特别的情况下,这是一种非常游泳的特征。然而,通常这回带来同样的 **build** 问题,在从一个文件库中 **check out** 多个项目,并保存在你的 **workspace** 下的随机目录下。因此我们再次忠告大家,除非你的团队已经建立了一套关于指示不同的子目录的位置的一致标准,否则不要使用这个功能。再说一次,这个的目的是为了所有项目成员间的一致性。

9.2 CVS 模块

CVS 将文件库保存在一个标准的文件系统目录树下。直到现在我们还依赖于 **cv** **co** **name** 命令, CVS 查造文件库的名为 **name** 的根目录,将其 **check out**,并保存在本地 **workspace**。然而, CVS 也可以让你将文件库分割为模块。许多情况下,模块能够帮助你找出所有的问题:将文件库的源码组织成有用的块。

CVS 模块基本上就是给文件库下的一个或多个目录命名的一种方式。然而,事情还没有那么简单。实际上 CVS 支持三种模块: **alias modules**, **regular modules**, 和优雅的 **ampersand modules**。在学习这三个模块前,我们首先需要知道怎么样配置 CVS 的模块。

CVS 配置

你也许已经注意到一个特殊的目录: **CVSROOT**,在你创建文件库的时候这个目录会自动出现在文件库下。该目录包括了很多 CVS 配置文件和一些选项文件。因为这个目录存在于文件库下,你可以象其他目录一样将其 **check out**。


```
work> cvs -d ~/sandbox co CVSROOT
cvs checkout: Updating CVSROOT
U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
U CVSROOT/cvswrappers
U CVSROOT/editinfo
U CVSROOT/logininfo
U CVSROOT/modules
U CVSROOT/notify
U CVSROOT/rcsinfo
U CVSROOT/taginfo
U CVSROOT/verifymsg
```

因为我们关心在文件库里模块的配置，我们需要在 `CVSROOT/modules` 文件工作。如果你在编辑器中打开该文件，你会发现文件里完全是大量的注释行（以#开头）。那是因为我们的 `sandbox` 文件库还没有定义任何模块（典型的空的模块文件如图 9.3）。

```
# Three different line formats are valid:
#   key      -a    aliases...
#   key [options] directory
#   key [options] directory files...
#
# Where "options" are composed of:
#   -i prog  Run "prog" on "cvs commit" from top-level of module.
#   -o prog  Run "prog" on "cvs checkout" of module.
#   -e prog  Run "prog" on "cvs export" of module.
#   -t prog  Run "prog" on "cvs rtag" of module.
#   -u prog  Run "prog" on "cvs update" of module.
#   -d dir   Place module in directory "dir" instead of module name.
#   -l       Top-level directory only -- do not recurse.
#
# NOTE: If you change any of the "Run" options above, you'll have
# to release and re-checkout any working directories of these modules.
#
# And "directory" is a path to a directory relative to $CVSROOT.
#
# The "-a" option specifies an alias. An alias is interpreted as if
# everything on the right of the "-a" had been typed on the command line.
#
# You can encode a module within a module by using the special '&'
# character to interpose another module into the current module.
# This can be useful for creating a module that consists of many
# directories spread out over the entire source repository.
```

Figure 9.3: A Typical Empty modules File

让我们定义一个模块来作为试验。如果你在我们初始的 `sandbox` 文件库上工作，你应该有一个已经定义好的名叫 `sesame` 的 top-level 目录。让我们定义一个指向 `sesame` 项目的模块 `projectX`。将 `modules` 文件 check out，打开编辑器，在该文件底部添加行：

```
projectX    sesame
```

修改之后，我们现在需要将其返还文件库。很简单，只需要 `commit` 就可以了。

```
work/CVSR00T> cvs commit -m "Add module projectX"
cvs commit: Examining .
Checking in modules;
/Users/dave/sandbox/CVSR00T/modules,v <-- modules
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
```

注意这里有些另外的逻辑：CVS 识别出我们修改的配置信息，并相应地进行自我更新。

现在我们来测试一下模块名 `projectX`，回到你的工作目录下，然后尝试将其 `check out`。

```
work> cvs -d /Users/dave/sandbox co projectX
cvs checkout: Updating projectX
U projectX/Color.txt
U projectX/Number.txt
```

现在你应该有乐意个名为 `projectX` 的子目录，下面包括了 `sesame` 项目的所有内容。

在我们深入探讨之前，先整理一下。我们已经 `check out` 两个我们不需要的模块（`CVSR00T` 和 `projectX`），因此先将其释放。这样会将其从我们的 `workspace` 里删除这两个模块（但是并不会从文件库里删除它们）。通过 `cvs release` 命令可以做到这点。指定 `-d` 选项，CVS 删除我们的本地文件。

```
work> cvs -d /Users/dave/sandbox release -d CVSR00T
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'CVSR00T': yes
work> cvs -d /Users/dave/sandbox release -d projectX
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'projectX': yes
```

了解了定义模块的机制后，我们来看看可以创建的模块的类型。

Alias Modules

`alias modules` 是一种简单的快捷方式：when I say X convert it to Y/Z。在你想将文件库分解成子项目的时候，你想让开发人员使用一致的目录结构的时候，可以使用 `alias modules`。

记住我们的开发人员只想 `check out` 项目的子集，如 107 页的图 9.1。使用基本的 CVS 命令，开发人员不得不使用以下的命令才能将每个独立的子项目 `check out`。

```
work> cvs co proj1/client
work> cvs co library/xml
work> cvs co library/date
```

对于一个小的目录树，如本例，这还不是大问题。但是随着项目的增长，这将变得越来

越繁重。那就是我们为什么要使用别名。对于我们的简单项目，可以给我们的 `modules` 文件添加：

```
client -a proj1/client
xml    -a library/xml
date   -a library/date
```

现在开发人员只需要输入：

```
work> cvs co client
work> cvs co xml
work> cvs co date
```

CVS 在 `modules` 文件中查找这些名字，并转化为相应的路径。然后将这些路径 `check out`。这就是说，即使你使用 `cvs co xml`，CVS 会将 `check out` 的文件放在 `workspace` 的目录 `library/xml` 下。这样我们总能保持 `check out` 的代码在一致的位置。

然而我们可以更进一步。也许目录树的这三个部分形成了具有一定意义的组。我们可以将其定义为模块，这样我们可以使用一致的名字来引用它们。在 `modules` 文件后面继续添加：

```
clientall -a proj1/client library/xml library/date
```

现在开发人员可以只需要一个命令就可以将三个子目录都 `check out`。

```
work> cvs co clientall
cvs checkout: Updating proj1/client
cvs checkout: Updating library/xml
cvs checkout: Updating library/date
```

最后，我们把这些整理一遍。在 `modules` 文件里面，还可以将单独的别名组成组合别名，而不是重复使用各个子路径，如下四行（以及下一页的概略图 9.4）。

```
client    -a proj1/client
xml       -a library/xml
date      -a library/date
clientall -a client xml date
```

Regular Modules

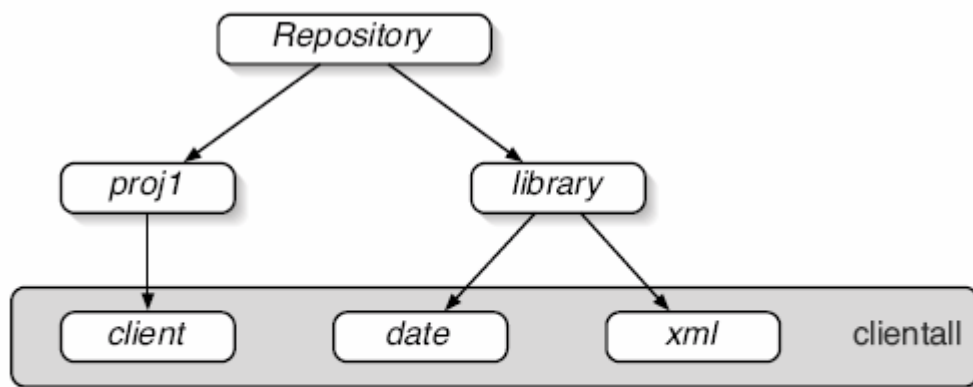


Figure 9.4: Modules and Aliases

`alias modules` 提供了对文件库中已知的子目录定义快捷方式的功能，`regular modules` 则可以在 `check out` 的时候对目录树的部分进行更名。例如，你在 `modules` 文件中定义了如下的模块：

```
clnt proj1/client
```

然后使用该模块名进行 `check out`，CVS 将忽略问文件在文件库的路径，而是将其放在目录 `clnt` 下。例如文件 `proj1/client/README` 被 `check out` 之后变为 `clnt/README`

在此之前我们已经知道，当文件库中的某个项目来自于多个位置，我们要将其 `check out`，必定要在目录间移动，这太糟糕了，文件移动也一样，`build` 变得如同管弦乐一样难缠。

但是如果子项目并不相互依赖，这种更名就很方便。例如，在实效系列这套丛书的工作环境下，在我们将所有的工作内容都保存在一个中心 CVS 文件库。包括项目的代码，文档，`web` 站点，本书的源码。很多都是独立的，"Introduction to Ruby"的教程资料就不依赖文件库里的其他任何材料，因此 将其独立地 `check out` 是有意义的。在我们的 `modules` 文件里，你可以发现如下的行：

```
courses PP/doc/Courses
halfruby PP/doc/Courses/HalfDayRuby
```

这意味着我们可以 `check out` 模块 `halfruby`，这些课程材料将出现同样的目录下，没有必要创建一个完整目录，而目的只是为了得到单独一套文件。同时也要注意别名不是一定要指向一个目录叶（也就是没有子目录的目录），`check out` 模块 `courses` 将会获得所有的课程材料。

Alias modules 可以将多个目录以及其他别名作为模块的定义。**regular modules** 就不具备这样的功能，每个定义必须准确地参考一个目录（*but see the next section on ampersand modules for a way around this*）。然而，你可以对 **regular module** 执行一些额外的，可以在住目录路径后包括一系列特定的文件和目录。如果这样，只有这些文件和目录才会被 **check out**。

例如 **Half-Day Ruby** 课程的文件相当大。如果我在酒店使用拨号网络，而且只需要课程的简单代码，我不想浪费时间来下载数兆的 **Powerpoint** 文件。我可以用 **CVS** 命令只 **check out** 子目录 **samples**，**CVS** 具有相当智能，你可以给模块添加路径，就像在 **top-level** 目录后面添加子目录一样。

```
work> cvs co halfruby/samples
```

然而，我们可以做多一点，可以在 **modules** 文件后面添加：

```
halfrubysamp PP/doc/Courses/HalfDayRuby/samples
```

将 **halfrubysamp** 模块 **check out** 将会创建一个名为 **halfrubysamp** 的目录，并在下面放文件库中的 **samples** 的内容。

Ampersand Modules

前面我们说过通常模块只能定义在单个目录下。那只不过是小小的谎言。还有另外的语法可以对付这样的限制。一个模块可以根据其他模块而定义，只需要在其他的模块名前面添加一个**&**符号。例如，**Andy** 和 **Dave** 可以设置他们所有课程和讲演的模块。

```
halfruby PP/doc/Courses/HalfDayRuby
fullruby PP/doc/Courses/OneDayRuby
introruby PP/doc/Talks/IntroRuby
vacation PP/doc/Talks/SummerVacation
:      :
```

可以使用一个命令就将所有模块都 **check out**，这很方便。要这样需要添加一个新的模块：

```
allruby &halfruby &fullruby &introruby &vacation . . .
```

然后我们可以通过一个命令 **check out** 所有的 **Ruby** 的材料。

```
work> cvs co allruby
```

该命令将 Ruby 的材料 check out，并存放在图 9.5 所示的目录结构下。

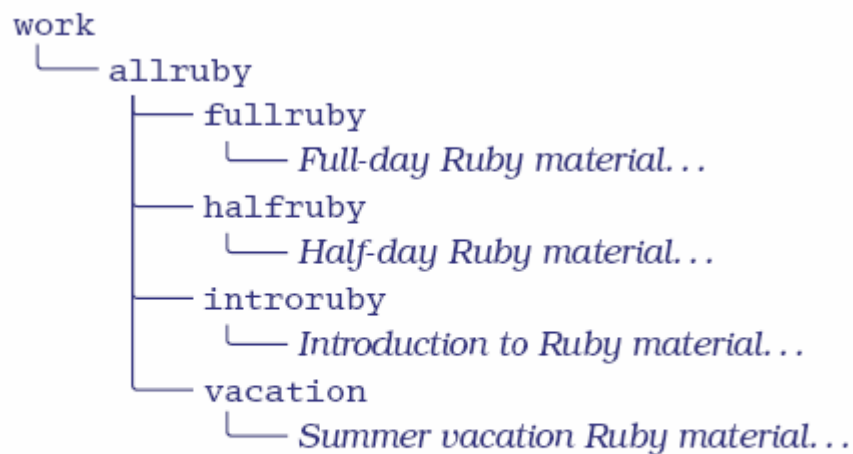


Figure 9.5: Checking-out an Ampersand Module

9.3 总结

在你想对你的项目创建一个逻辑结构的时候，可以使用 CVS 模块，它将文件库中分散的部分整合在一起。通过提供一个精确定义一个特定的项目或者子项目的含义的机制，减少了开发人员的错误。同时也为你所有的项目提供了一个一致的外部结构。

第十章 第三方代码

所有的项目在某种程度上都要依赖外部的库文件：**C** 程序要 **libc** 库，**Java** 程序需要 **rt.jar** 等等。这些库应该形成你的个人 **workspace** 的一部分吗？

要回答这个问题，需要先问你另一个问题。在将来的任意的时间你需要有能力随时重新 **build** 一个可以工作的程序。到时候你能够使用这些库文件的有效版本吗？

如果你觉得你代码使用到的库文件在你的应用开发的整个生命周期都是有效的，那么没有必要专门为他们做什么特别的事情，只需要将其安装在你机器上即可。

除了标准的编程语言工具，许多项目还在其项目中使用了其他的，没有那么稳定的库文件。例如，许多 **Java** 开发人员就使用了 **Junit** 框架来测试他们的代码。相比标准的库文件，这些框架相对来说更容易变化（2003 年六月 **Junit** 已经发展到 3.8 版本）。尽管它们的版本间的变化几乎兼容，但是总有些变化会影响你的应用程序。因此我们推荐将这些库放到你的 **workspace** 以及项目的文件库中。

在决定将第三方的库置入你的 **workspace** 以及文件库之后，现在可以考虑怎么样存放以及存放的位置。

首先是怎么将第三方库文件包括进来。这相对来说比较简单。如果你直接使用改库文件的发布形式，并对该库文件在应用的生命周期始终能够起作用而且不会改变非常自信，那么直接用二进制文件的形式保存就完全够了。建议将所有的库文件放在根目录 **vendor/** 的子目录下。如果库文件是架构独立的（例如 **Java** 的 **rt.jar** 文件），可以捡起放在子目录 **lib/** 下面。如果你有依赖于架构的库文件（假设你的应用针对多个架构），你需要针对每个架构和操作系统的组合在 **vendor/** 目录下创建相应的子目录。这些子目录的常用命名模式是使用 **arch-os**，其中 **arch** 指的是目标架构（**Intel Pentium** 用 **i586**，**PowerPC** 用 **ppc** 等等），而 **os** 指操作系统（**linux**，**win2k** 等等）。在 **import** 或者 **add** 库文件的时候一定要记得使用 **-kb** 标志（如 **DLL** 动态链接库或者其他库文件）。

象 **C** 和 **C++** 这样的语言需要在使用特定库文件的应用程序的代码里面包括相应的头文件。库文件提供了这些头文件，这些头文件也应该保存在 **workspace** 和文件库。建议将其保存在 **vendor** 下的 **include/** 子目录下。通过 **vendor/include/** 这样的子目录结构编译器自然能够找到库的头文件。例如，名为 **datetime** 的 **C** 库，作用是执行日期和时间的计算，带一

个二进制的库文件 `libdatetime.a` 和两个头文件 `datetime.h` 和 `extras.h`，其中 `datetime.h` 头文件库适合安装在 `include/` 根目录，而 `extras.h` 需要在子目录 `dt/`。使用到两个头文件的程序通常以如下代码作为开始：

```
#include <datetime>
#include <dt/extras>
// . . .
```

本例我们的文件库以及 `workspace` 组织结构如图 10.1。



Figure 10.1: Sample Repository With Third Party Library

和创建的环境集成

如果在你的 `workspace` 里包括了第三方的库和头文件，你需要保证编辑器，链接器，和 IDE 能够访问它们。还有一个小问题：你需要保证没有将那些包含绝对路径的东西 `check in` 文件库（绝对路径可能会在其他开发人员的机器上引发问题）。有两个选择：

- 调整你的 `build` 工具，让所有的路径都是项目根目录的相对路径。如果你使用的外部 `build` 工具，例如 `make`，这样是可以工作的。
- 设置指向项目根目录的外部环境变量，让 `build` 内部所有的都指向这些变量。这样可以让每个开发人员通过外部变量定义不同的值，但是却能共享一个通用的 `build` 环境结构。

环境变量不必一定要是真正的操作系统环境变量。例如 `Eclipse` 就支持每个用户的内部变量然后可以共享一个通用的 `build` 结构。这样所有的开发人员可以共享一个通用的 `Eclipse` 的 `build` 定义，开发人员仍然可以将源码安装在不同位置。

我们推荐第二种方式。

10.1 包括源码的库

有时候库与源码是同时发布的（有时只通过源码的形式发布）。如果你同时拥有库的源码和二进制版本，那么你应该向文件库保存哪一个呢，又怎么样设置 **workspace** 呢？

答案是这是风险管理的一个练习。使用源码意味着你总是处于修改 **bug** 和添加新功能，而使用二进制库文件就没有这些事情。很明显这是好事。但同时，将项目用到的所有库的源码包括进来会降低 **build** 速度，同时增加项目结构的复杂度。将来的维护人员也会头痛。如果出现 **bug**，他们是考虑库源码的潜在的变化呢？还是只集中注意力到你们组织的代码上？

我们推荐将第三方代码添加到文件库，但是对其要特殊处理，特殊对待。对此要进行一点角色扮演。

假设你是某个特殊的库的作者，你不时地向你的用户群发布新版本。作为你个高质量的库作者，很自然地你就将所有源码置于版本控制系统的管理之下，并实践所有必要的发布控制过程。

现在从角色扮演中回来（记住：吸气，吐气，吸气，吐气）。在理想的世界里，我们能直接进入第三方文件库，并直接抽取发布。但是不能，我们必须亲自动手。在我们收到代码，已修复的错误，第三方的新发布版本，我们必须假装我们生成了这些代码，在我们的版本控制系统中处理它们，就好象我们就是第三方的供应商，我们正在用供应商的角色来处理。实际上这比听起来还简单。

导入初始代码

在我们第一次收到第三方库源码的时候，需要将其 **import** 到文件库。我们推荐将这些代码和项目代码分开处理。如果你预计要 **import** 多个代码源，将它们都保存在一个公共的根目录下是有意义的，假设该目录叫 **vendormsg/**（主要是和 **vendor/**区别开，**vendor/**目录下包括了库文件和头文件）。

更具体一点，假设我们在项目里面使用 **GNU** 的 **readline** 库的 4.3 版本。

现在要从 **GNU** 的 **ftp** 站点下载最新的源码，并将其保存在一个临时目录下。

```
> cd tmp
tmp> ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:dave): ftp
331 Please specify the password.
Password:
230 Login successful. Have fun.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/gnu/readline
250 Directory successfully changed.
ftp> get readline-4.3.tar.gz
local: readline-4.3.tar.gz remote: readline-4.3.tar.gz
961662 bytes received in 00:06 (136.48 KB/s)
ftp> bye
221 Goodbye.
```

然后对包进行解压，会在子目录下创建源文件的目录树（根据我们的经验是 readline-4.3）。我们将该目录作为当前工作目录。

```
tmp> tar xzf readline-4.3.tar.gz
tmp> cd readline-4.3
```

现在我们要将当前目录下的源码 import 到文件库中的 vendorsrc/fsf/readline 目录下（记住所有的第三方代码都保存在 vendorsrc/下面。这个例子的供应商是 Free Software Foundation，产品是 readline）。

```
tmp/readline-4.3> cvs import -ko -m "load 4.3" \
                    vendorsrc/fsf/readline FSF_RL RL_4_3
N import/aclocal.m4
N import/ansi_stdlib.h
N import/bind.c
N import/callback.c
N import/chardefs.h
N import/compat.c
:
N import/support/shobj-conf
N import/support/wcwidth.c
No conflicts created by this import
```

这可真真是个命令（译者：这个命令好长啊），我们将在下一页的图 10.2 一一对其分析。
-ko 标志很重要，但也很微妙。通常 CVS 将扩展其管理之下的文件里面的特殊的关键字（例如 \$Author\$），并让你给文件添加注释（我们并不鼓励这个，因此本书没有过多地讲述）。问题是文件每次被 check out 其中的关键字都会被扩展。如果供应商正好也使用 CVS，而且正好也使用了同样的关键，那么你得到的源码里面将会在这些关键字的位置出现供应商的信息。但是如果你将这些文件就这样 import 到文件库，然后又将其 check out，你会发现你

的名字出现在 **author** 的位置上。现在这还没什么，勉强能够接受，但是将来你合并其下一个版本的时候就会出问题了。**CVS** 注意到这些标志所在的行已经改变了，那么你就要处理合并冲突。如果指定了 **-ko** 选项，在 **import** 的时候 **CVS** 将关闭这个扩展功能，这样你就不会看到这个问题了。

vendor tag 提供了引用整个代码集的一种方式。在本例，我们可以使用 **FSF RL** 标签来引用 **readline** 的代码。所有的 **readline** 代码都会共享这个标签。

release tag 指定了该版本的代码的标签。如果 **FSF** 发布 **readline** 的 4.4 版本，我们 **check in** 的时候将用另一个不同的发布标签。遮掩我们总能通过使用原来的 **RL 4 3** 标签得到 **readline** 的 4.3 版本。

将这些代码 **import** 到文件库之后，就可以将刚才用的临时目录删除了。

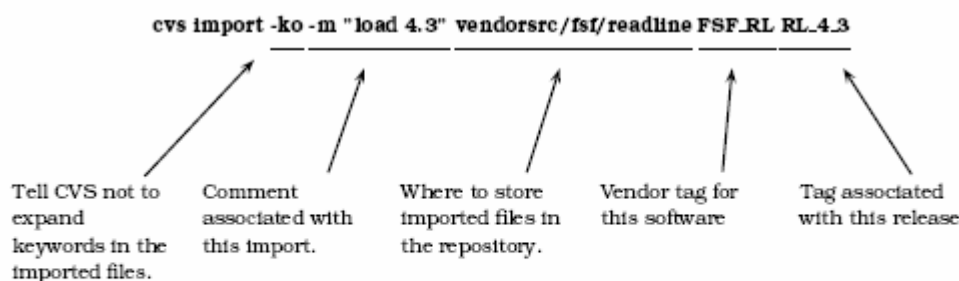


Figure 10.2: A CVS "import" command

导入新发布的版本

当供应商发布软件新版本的时候，你可能想将新的版本和你的文件库的版本进行合并。假设你没有修改原来的源码，这个工作就很容易，只需要将其再次 **import** 就可以了，如下步骤：

- 下载最新的源文件，解压到临时目录。
- 执行 **cvs import** 命令，使用和上次一样的目录和 **vendor tag**，但是 **release tag** 要用新的。

例如，如果 **FSF** 的 **readline** 新发布的版本为 4.4，可以这样做：

```
tmp> tar zxf readline-4.4.tar.gz
tmp> cd readline-4.4
tmp/readline-4.4> cvs import -m -ko "load 4.4" \
                    vendorsrc/fsf/readline FSF_RL RL_4_4
N import/aclocal.m4
N import/ansi_stdlib.h
N import/bind.c
N import/callback.c
N import/chardefs.h
N import/compat.c
:
N import/support/shobj-conf
N import/support/wcwidth.c
No conflicts created by this import
```

10.2 修改第三方代码

有时候出于某种原因，需要你的团队对第三方的源码进行修改。可能需要添加一些应用程序特别需要的功能，也可能需要修改一些 bug。

很明显理想的解决方案是将这些修改反馈给其供应商，让他们将其合并到他们的代码。然后他们将下一个版本发给你，这样他们的代码包括了你的修改，生活也会很美好。

然而这不是总是可行。这样我们需要维护我们的修改，并自动地跟进供应商的脚步，从一个版本到下一个版本。

还好很幸运，CVS 在这方面相对比较容易。在这样的背景下，import 实际上创建和管理了一个简单的发布树。如下：

你第一次将代码 import 到 CVS，会创建一个主线，跟着立即创建一个分支（1.1.1），然后将你 import 的代码放在该分支下（这样文件的第一个版本为 1.1.1.1）。尽管听起来很复杂，实际上跟 18 页所讲的那个简单的发布结构没什么区别。那并不是巧合，在后面隐藏的是 CVS 把这些 import 当成是发布。你提供给 import 命令的 vendor tag 就好比提供给发布分支的 tag，而提供给每个 import 的 release tag 用于识别每个单独的发布版本所在的分支。请看下一页的图 10.3。

如果你 check out vendor 代码，你就是在 check out 发布分支（该分支打上了 vendor tag）。这是可以验证的，只需要对文件运行 cvs status 命令，就会发现文件关联了一个带四个级别的修订版本号（1.1.1.1）。不过这里还有一点神奇。如果你编辑了 vendor 代码，并 check in，CVS 将把你的修改保存在主线，修订版本也会变为 1.2，而不是 1.1.1.2。CVS

将 vendor 代码保存在 vendor 分支。

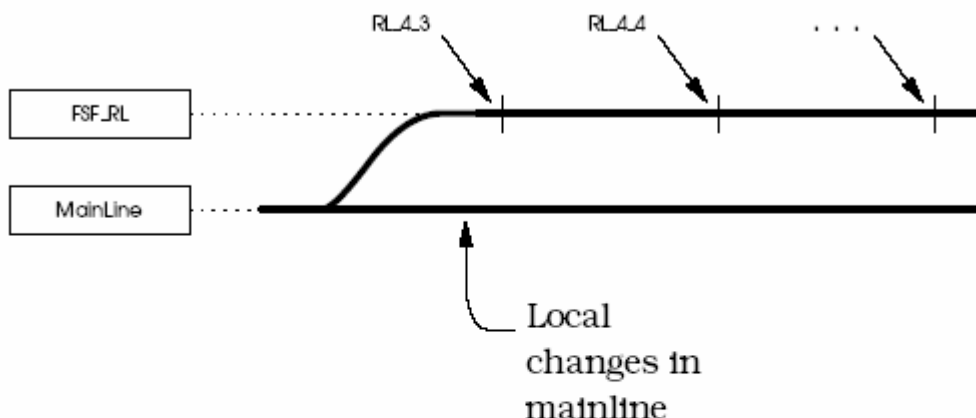


Figure 10.3: Imported Third Party Code. Code is imported in to a release branch, labeled by the vendor tag. Each import generates a new release tag in that branch. Local work automatically takes place in the mainline.

如果你编辑了第三方代码之后这些代码又有新的发布版本,那么会发生什么事情呢? 让我们来找出答案。为此先设置一个虚拟的文件库。然后假设我们就是供应商(假设叫 **Acme**), 创建两个简单的文件, 回到原来的身份, 将这些文件 **import**, 然后 **check out** 到我们的 **workspace**, 跟着进行修改并 **check in**。

返回 **vendor** 目录, 开始准备进行新的发布。然后尝试着做 **import**, 并将 **vendor** 的修改合并到我们自己的上面。

这些角色扮演有点乱, 一旦我们开始, 就应该在每个命令前显示出当前目录的全名。在提示符里只显示目录名。通常在我们扮演 **vendor** 的时候, 我们身处目录:

```
tmp/3rdparty/Acme
```

当我们身为处理 **check out** 的 **vendor** 文件的客户端的时候, 我们应该身处目录:

```
tmp/3rdparty/work/vendorsrc/Acme
```

Step 1: 设置 Repository

所有的工作都在目录下 **3rdparty** 进行, 结束的时候可以清除该目录下的所有信息。从文件库的信息都保存在 **repository** 子目录下。

```
# In directory tmp
tmp> mkdir 3rdparty
tmp> cd 3rdparty
tmp/3rdparty> export CVSR00T="/tmp/3rdparty/repository
tmp/3rdparty> cvs init
tmp/3rdparty> ls      # use 'dir' under Windows
repository
```

Step 2: 创建第三方代码

创建名为 Acme 的目录，下面用来放 `thirdparty` 的代码。该目录将被 import 到 CVS。

在这个目录下，打开编辑器，创建两个文件 `Color.txt` 和 `Number.txt`。

```
# in directory tmp/3rdparty
tmp/3rdparty> mkdir Acme
tmp/3rdparty> cd Acme
```

编辑文件

文件 `Color.txt`:

```
black
brown
red
orange
yellow
green
```

文件 `Number.txt`:

```
zero
one
two
three
four
```

Step 3: 导入第三方代码

扮演 `vendor` 结束，现在假装我们从 `vendor` 那里获得这些代码，并 import 到文件库，保存在 `vendormsg/Acme` 下面。

```
# In directory tmp/3rdparty/Acme
Acme> cvs -ko import -m "load" vendormsg/Acme Acme REL_1_0
N vendormsg/Acme/Color.txt
N vendormsg/Acme/Number.txt
No conflicts created by this import
```

Step 4: 设置 workspace

现在创建一个 workspace，然后在下面将 `vendor` 的代码 check out。


```
# In directory tmp/3rdparty/Acme
Acme> cd ..
tmp/3rdparty> mkdir work
tmp/3rdparty> cd work
tmp/3rdparty/work> cvs co vendorsrc/Acme
cvs checkout: Updating vendorsrc/Acme
U vendorsrc/Acme/Color.txt
U vendorsrc/Acme/Number.txt
```

Step 5: 修改第三方代码

项目进行的过程中，我们发现了 vendor 代码的一个问题，他们的 numbers 文件使用了 zero，而我们的项目使用的是 naught。Vendor 忽视了我们要求修改的请求，并声称我们是唯一使用中世纪英语中的数字名称的客户。我们至少亲自上阵了。我们在 workspace 里编辑了该文件，然后 check in。

```
# In directory tmp/3rdparty/work
work> cd vendorsrc/Acme
Acme> # ... edit file ...
Acme> cvs commit -m "Zero becomes naught"
cvs commit: Examining .
Checking in Number.txt;
.../repository/vendorsrc/Acme/Number.txt,v <-- Number.txt
new revision: 1.2; previous revision: 1.1
done
```

Step 6: 第三方代码发生了变化

同时，回到 Acme 的公司，他们决定开发产品的 V1.1，作为新版本的新增功能的一部分，他们给 numbers 文件添加了三个新数字。我们回到 Acme 目录并编辑文件来进行模拟。

```
# In directory tmp/3rdparty/work/vendorsrc/Acme
Acme> cd ../../../../Acme
tmp/3rdparty/Acme> # ... edit file ...
```

编辑后，新的 numbers 文件包括一下内容：

文件 Number.txt:

```
zero
one
two
three
four
five
six
seven
```

文件里仍然包括 zero，记住 Acme 并没有将其修改为 naught。只有我们在本地进行了修改。

Step 7: 导入新版本

Acme 给我们发送了新版本，我们要以客户的角色把新版本 import 到 CVS。

```
# In directory tmp/3rdparty/Acme
Acme> cvs import -ko -m "update" vendorsrc/Acme Acme REL_1_1
U vendorsrc/Acme/Color.txt
C vendorsrc/Acme/Number.txt
1 conflicts created by this import.
Use the following command to help the merge:
    cvs checkout -jAcme:yesterday -jAcme vendorsrc/Acme
```

CVS 具备智能，足以识别出这次 import 实际上是在 update 已经存在的文件。Color.txt 文件成功的 update（实际上这个文件没有变化），但是 Nubmer.txt 文件存在潜在的冲突，因为这个文件同时被我们和 vendor 修改了。CVS 还能建议我们使用什么样的命令来解决这个问题（译者：输出的最后一航）。一般来说这个命名都运行的很好，不行的是不适合我们。为了找出原因，让我们仔细看看这个命名：

正如在 7.2 章所看到的，-j 选项用于合并变化。在这个例子，我们讲使用两个 -j 选项。第一个 -jAcme:yesterday，也就是 Acme 分支的 yeaterday 的状态，第二个 -jAcme，也就是在 import 前的最新状态。两个放一起就是让 CVS 比较它们之间的区别，比较的结果是为了得到 vendor 所做的修改。然后将这些修改应用到主线的 head 上。最终的结果是要将 vendor 的修改更新到我们的本地文件上。

尽管这个咒语通常都有效（很少有 vendor 一天发布多个版本），但是在我们的例子里却没什么用，因为我们并没有 vendor 头天的代码。我们用 -j 选项的另一种形式来代替，这样可以基于 release tag 进行 merges

```
# In directory tmp/3rdparty/Acme
Acme> cd ../work
work> cvs co -jREL_1_0 -jREL_1_1 vendorsrc/Acme
cvs checkout: Updating vendorsrc/Acme
RCS file: /Users/dave/tmp/3rdparty/repository/vendorsrc/Acme/Number.txt,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.2
Merging differences between 1.1.1.1 and 1.1.1.2 into Number.txt
```

记住我们第一次 import 的标签是 REL_1_0，第二次是 REL_1_1。可以让 CVS 将这两个不同发布之间的区别应用到主线上。命令的结果：CVS 将 vendor 的修改合并到我们的本地文件。让我们查看并确认一下，现在我们得到的 vendor 添加的三个数字，还有就是我们修改的 naught 没有被修改。

```
# In directory tmp/3rdparty/work
work> cd vendorsrc/Acme
Acme> cvs status Number.txt
=====
File: Number.txt      Status: Locally Modified
Working revision:     1.2      Result of merge
Repository revision: 1.2      /Users/dave/tmp/3rdparty/repos...
Sticky Tag:           (none)
Sticky Date:          (none)
Sticky Options:       (none)
```

也可以查看一下文件内容

文件 Number.txt:

```
naught
one
two
three
four
five
six
seven
```

可以看到在 vendor 和我们的修改之间的冲突，以及文件里常见的冲突的标志。

Step 8: 保存合并后的文件

现在我们已经合并了变化的部分（之前要进行测试，保证系统仍然可以工作），并将所有的内容 check in。

```
# In directory tmp/3rdparty/work/vendorsrc/Acme
Acme> cvs commit -m "Merged 1.1 changes"
cvs commit: Examining .
Checking in Number.txt;
/Users/.../vendorsrc/Acme/Number.txt,v <-- Number.txt
new revision: 1.3; previous revision: 1.2
done
```

概括:修改第三方代码

使用这些简单的步骤来管理 vendor 的发布是直接而非常有效的。CVS 自动维护一个发布分支，该分支包括了来自 vendor 的未修改的代码，这些代码都打上了每次发布的标签。文件库的主线也包括了同样的代码，但是里面的变化是我们本地修改的。使用-j 选项可以将 vendor 的每次发布的变化合并到我们的本地版本。

总结如下：

* Import the vendor code:

```
cvs import -ko -m "load" vendor module n
```

```
vendor release
```

* Check out vendor code into a local workspace:

```
cd work
```

```
cvs co vendor module
```

* Make local changes to vendor code and check back in:

```
cvs commit -m "summary of changes"
```

* If the vendor issues a new release, import it into the vendor

branch:

```
cvs import -ko -m "update" vendor module n
```

```
vendor release tag
```

* Fix conflicts between vendor changes and our changes:

```
cvs co -j release 1 -j release 2 vendor module
```

* Save the changes back:

```
cvs commit -m "summary of changes"
```

附录 A

CVS概要及诀窍

This section is a brief summary of CVS commands, and the particular recipes used in this book.

A.1 CVS Command Format CVS 命令格式

cv<global options..> command <options and arguments. . .>

Global Options

- H Displays usage information for command.
- Q Cause CVS to be really quiet.
- q Cause CVS to be somewhat quiet.
- r Make checked-out files read-only.
- w Make checked-out files read-write (default).
- l Turn history logging off.
- n Do not execute anything that will change the disk.
- t Show trace of program execution . try with -n.
- v CVS version and copyright.
- b bindir Find RCS programs in .bindir..
- T tmpdir Use .tmpdir. for temporary files.
- e editor Use .editor. for editing log information.
- d CVSROOT Overrides CVSROOT environment variable as the root of the CVS tree.
- f Do not use the /.cvsrc file.
- z # Use compression level .#. for net traffic.
- a Authenticate all net traffic.
- s VAR=VAL Set CVS user variable.

Flag Characters

During update operations, CVS will display a list of file names preceded by flag characters. The following table lists the meanings of these characters.

A file has been added locally and is not yet in the repository.

C file A conflict was detected when trying to update file (that is, local changes conflicted with changes made in the repository version). Your local copy of the file contains conflict markers, and the original version of the file is stored in a new file called `.# file.version`.

M file has been modified in your workspace and needs to be stored back to bring the repository up-to-date.

P file Equivalent to `.U,` documented below. The `.P` flag signifies that the server used a patch to bring the file up to date.

R file has been removed from your working copy of the repository (using `cv remove`). The repository version will be removed when you run `cv commit`.

U file The local copy of file has been updated to bring it up-to-date with the repository. This happens both when the repository version is later than the local version and when a new file is in the repository but not (yet) available locally.

? file exists in your workspace but nothing is known about it in the repository. You can use `cv add` to add it, or possibly update `.cvsignore` to tell CVS to ignore it.

CVS Environment

The following environment variables are commonly used with CVS.

They are described in more detail in [Connecting to CVS](#) on page 49.

CVSROOT

Specifies the default repository location and access method.

Setting this variable means you don't need to use the global

CVS -d option.

CVS RSH

Specifies the program to be used to access the remote repository.

We recommend using ssh for this purpose.

CVS Commands

CVS supports a rich set of commands, listed in the table that follows.

In this book we use only a subset of these (marked y in the following table). In the sections that follow, we'll show the specific options for these commands. These descriptions are based on the CVS help information: use `cvcs --help` for details.

`add` Add a new file/directory to the repository

`admin` Administration front end for rcs

`annotate` Show last revision where each line was modified

`checkout` Checkout sources for editing

`commit` Check files into the repository

`diff` Show differences between revisions

`edit` Get ready to edit a watched file

`editors` See who is editing a watched file

`export` Export sources from CVS, similar to checkout

`history` Show repository access history

`import` Import sources into CVS, using vendor branches

`init` Create a CVS repository if it doesn't exist

`log` Print out history information for files

`login` Prompt for password for authenticating server

`logout` Removes entry in `.cvspass` for remote repository

`rdiff` Create `.patch` format diffs between releases

`release` Indicate that a Module is no longer in use

`remove` Remove an entry from the repository

`rtag` Add a symbolic tag to a module

statusy Display status information on checked out files

tagy Add a symbolic tag to checked out version of files

unedit Undo an edit command

updatey Bring work tree in sync with repository

watch Set watches

watchers See who is watching a file

Add New File or Directory add

cvcs add [-k rcs-kflag] [-m message] files...

-k Use rcs-kfiag to add the file with the specified kfiag. Commonly used as .-kb. to add binary files to the repository.

-m Use message for the creation log.

Administer Underlying Repository admin

cvcs admin rcsoptions...

-k Use .rcs-kfiag. to change the fiags associated with a file. Sometimes used to change the status of a file to binary (using .-kb.).

Show Revisions for Lines in Files annotate

cvcs annotate [-lRf] [-r rev|-D date] [files...]

-l Local directory only, no recursion.

-R Process directories recursively.

-f Use head revision if tag/date not found.

-r rev Annotate file as of specified revision/tag.

-D date Annotate file as of specified date.

Check Out Sources For Editing checkout

cvcs checkout [-ANPRcflnps] [-r rev] -D date]

[-d dir] [-j rev1] [-j rev2] [-k kopt]

modules...

-A Reset any sticky tags/date/kopts.

-N Don't shorten module paths if -d specified.

-P Prune empty directories.

- R Process directories recursively.
- c Show contents of the module database.
- f Force a head revision match if tag/date not found.
- l Local directory only, not recursive.
- n Do not run module program (if any).
- p Check out files to standard output (avoids stickiness).
- s Like -c, but include module status.
- r rev Check out revision or tag (implies -P; is sticky).
- D date Check out revisions as of date (implies -P; is sticky).
- d dir Check out into dir instead of module name.
- k kopt Use RCS kopt -k option on checkout.
- j rev Merge in changes made between current revision and rev.

Check Files in to the Repository commit

cvs commit [-nRlf] [-m msg | -F logfile] [-r rev]
files...

- n Do not run the module program (if any).
- R Process directories recursively.
- l Local directory only (not recursive).
- f Force the file to be committed; disables recursion.
- F file Read the log message from file.
- m msg Log message.
- r rev Commit to this branch or trunk revision.

Show Differences Between Revisions diff

cvs diff [-lNR] [rcsdiff-options]

[[-r rev1 | -D date1] [-r rev2 | -D date2]]

[files...]

- l Local directory only, not recursive.
- R Process directories recursively.

-D date1 Diff revision for date against working file.

-D date2 Diff rev1/date1 against date2.

-N Include diffs for added and removed files.

-r rev1 Diff revision for rev1 against working file.

-r rev2 Diff rev1/date1 against rev2.

.ifdef=arg Output diffs in ifdef format.

rcsdiff Common options include -c for context diffs, -u for unified diffs, and --side-by-side.

Import Sources Into CVS import

cvcs import [-d] [-k subst] [-l ign] [-m msg]

[-b branch] [-W spec] repository

vendor-tag release-tags...

-d Use the file's modification time as the time of import.

-k sub Set default RCS keyword substitution mode.

-l ign Files to ignore (! to reset).

-b bra Vendor branch id.

-m msg Log message.

-W spec Wrappers specification line.

Create a CVS Repository init

cvcs init

CVS COMMAND FORMAT 139

Print File History log

cvcs log [-lRhtNb] [-r[revisions]] [-d dates]

[-s states] [-w[logins]] [files...]

-l Local directory only, no recursion.

-R Only print name of RCS file.

-h Only print header.

-t Only print header and descriptive text.

- N Do not list tags.
- b Only list revisions on the default branch.
- r[revisions] Specify revision(s) to list.
- d dates Specify dates (D1<D2 for range, D for latest before).
- s states Only list revisions with specified states.
- w[logins] Only list revisions checked in by specified logins.

Log In to PServer login

cvs login

Stop Using a Module release

cvs release [-d] directories...

- d Delete the local copy of the given directories.

Remove Entry from Repository remove

cvs remove [-fIR] [files...]

- f Delete the file before removing it.
- I Process this directory only (not recursive).
- R Process directories recursively.

Tag Module in Repository rtag

cvs rtag [-aIRnF] [-b] [-d] [-r tag|-D date] tag

modules...

- a Clear tag from removed files that would not otherwise be tagged.
- f Force a head revision match if tag/date not found.
- I Local directory only, not recursive.
- R Process directories recursively.
- n No execution of .tag program..
- d Delete the given Tag.
- b Make the tag a .branch. tag, allowing concurrent development.
- r rev Existing revision/tag.
- D Existing date.

-F Move tag if it already exists.

Display Status of Files status

cvs status [-vIR] [files...]

-v Verbose format; includes tag information for the file.

-I Process this directory only (not recursive).

-R Process directories recursively.

Tag Local Files tag

cvs tag [-IRF] [-b] [-d] [-c] [-r tag |-D date] tag
[files...]

-I Local directory only, not recursive.

-R Process directories recursively.

-d Delete the given tag.

-r rev Existing revision/tag.

-D date Existing date.

-f Force a head revision if specified tag not found.

-b Make the tag a .branch. tag, allowing concurrent development.

-F Move tag if it already exists.

-c Check that working files are unmodified.

Bring Local Files Up To Date update

cvs update [-APdfIRp] [-k kopt] [-r rev |-D date]
[-j rev] [-I ign] [-W spec] [files...]

-A Reset any sticky tags/date/kopts.

-P Prune empty directories.

-d Build directories, like checkout does.

-f Force a head revision match if tag/date not found.

-I Local directory only, no recursion.

-R Process directories recursively.

-p Send updates to standard output (avoids stickiness).

-k kopt Use RCS kopt -k option on checkout.

- r rev Update using specified revision/tag (is sticky).
- D date Set date to update from (is sticky).
- j rev Merge in changes made between current revision and rev.
- I ign Files to ignore (! to reset).
- W spec Wrappers specification line.

A.2 Recipes

Connecting via SSH.	Page 54
set CVSROOT to :ext: user@ address: repository	
set CVSfiRSH to ssh	
Issue normal CVS commands.	
Connecting via pserver	Page 55
set CVSROOT to :pserver: user@ address: repository	
cvs login	
Checking things out	Page 57
cvs co module...	
Checking out a particular revision.	Page 57
cvs co -r tag module	
Updating a Workspace	Page 59
cvs -q update -d	
Updating Specific Files	Page 59
cvs -q update file...	
Adding Files and Directories.	Page 62
cvs add name...	
Adding Binary Files	Page 62
cvs add -kb name...	
Ignoring certain files	Page 67
Add names to .cvsignore file	
(remember to cvs add .cvsignore)	

Renaming files.	Page 68
cvs -q update -d	
rename old name to new name	
cvs remove old name	
cvs add new name	
cvs commit -m "Rename old name to new name"	
Renaming a Directory	Page 70
mkdir new dir	
cvs add new dir	
move files from old dir/ new dir...	
cvs remove old dir/file...	
cvs add new dir/file...	
cvs commit -m "Rename old dir/ to new dir/"	
cvs update -P	
Seeing what's changed since checkout.	Page 71
cvs diff file or dir	
Seeing what's changed between versions	Page 72
cvs diff -r r1 [-r r2] file or dir	
Committing changes.	Page 79
cvs commit -m " message"	
Examining change history	Page 80
cvs log file or dir	
cvs annotate file or dir	
Undo change made between r1 and r2.	Page 84
cvs update -j r2 -j r1 file	
Creating a Release Branch	Page 90
cvs commit -m "..."	
cvs rtag -b RBfixfiy project	

Checking out a Release Branch	Page 91
cd work	
cvcs co -r RBfixfiy -d rbx.y project	
Generating a Release	Page 92
cvcs update	
# ... run tests ...	
cvcs commit -m "... " # if needed	
cvcs tag RELfixfiy	
Checking Out a Release	Page 92
cd work	
cvcs co -r RELfixfiy -d relx.y project	
Fixing Bugs in a Release Branch	Page 94
cd work	
cvcs co -r RBfixfiy -d rbx.y project	
cd rbx.y	
cvcs tag PREfi bugno	
# create test, fix problem, validate	
cvcs commit -m "Fix PR bugno"	
cvcs tag POSTfi bugno	
Apply Bug Fix to Another Branch	Page 95
cd workingdir	
cvcs update	
cvcs -j PREfi bugno -j POSTfi bugno update	
# test...	
cvcs commit -m "Apply fix for PR bugno from RBx.y"	
Creating Experimental Branches.	Page 95
cvcs commit -m ""	
cvcs rtag -b TRYfi initialsfi yymmdd project	
Using an Experimental Branch	Page 95

cvs update -r TRYfi initialsfi yymmdd

Returning to the Head of the Mainline Page 95

cvs update -A

Merging An Experimental Branch Page 97

In experimental workspace:

cvs commit -m "Finalize changes"

cd mainline

cvs update -j TRYfi initialsfi yymmdd

Creating Sub-modules Page 110

cvs co CVSROOT

cd CVSROOT

edit file: modules

cvs commit -m "Add module name"

cd ..

cvs release -d CVSROOT

Importing Third Party Code. Page 122

cvs import -ko -m " msg" rep locn vendor tag

附录 B

其他资源

CVS 是目前世界上最常用的版本控制系统, 查询帮助的通常方法就是用 Google 进行查询。但是我们页发现了一些特别有用的资源, 如下:

B.1 CVS 在线资源

CVS 主页

=>www.cvshome.org

The canonical site for CVS. Here you'll find information on new releases, downloads, the list of Frequently Asked Questions, and documentation.

Clicking the Hosted Projects link in the side panel takes you to a list of auxiliary CVS projects. Under the Integration link you'll find tools that might help you integrate CVS with your particular environment.

CVS 手册

=>www.cvshome.org/docs/manual/

Per Cederqvist's manual for CVS is available in HTML, PDF, and Postscript formats.

B.2 其他 CVS 书籍

This book takes a recipe-based approach to CVS. We believe that the commands documented here represent the essential subset of CVS for the vast majority of project teams. The bibliography contains references to three other CVS books ([Fog99], [Pur00], and [Ves03]) which contain more detailed information on the nitty-gritty of CVS.

It also contains a reference to a higher-level book, Software Configuration Management Patterns [BA03], which explains some of the theory behind the things we do with version control systems.

B.3 其他版本控制系统

The list below is a set of pointers to some well-known version control systems. We've tried hard to be non-judgemental; different folks are looking for different capabilities in their tools. Before investing in any of the commercial products, we strongly recommend searching for other users and getting their opinions. You'll find some surprisingly strong reactions.

...

BitKeeper

=>www.bitkeeper.com

BitKeeper uses an interesting approach to version control; it operates largely without a central server or repository.

ClearCase

=>www.rational.com/products/clearcase/index.jsp

Originally a Rational product, now owned by IBM.

Forte Code Management Software

=>www.sun.com/software/sundev/previous/teamware

Formerly by Forte TeamWare, now owned by Sun, and the inspiration for BitKeeper.

PVCS

=>www.merant.com

The professional offering includes version control, change management, bug tracking, and build automation.

Perforce

=>www.perforce.com

Powerful version control system. Many developers feel it has the simplest-to-use branching model.

Subversion

=>subversion.tigris.org

An open-source project intended to produce an eventual replacement for CVS. Usable, but as of September 2003 is still alpha quality.

Visual SourceSafe

=>msdn.microsoft.com/ssafe/

Microsoft's version control offering, so expect good integration with Microsoft tools.

B.4 参考书目

- [BA03] Stephen P. Berczuk and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2003.
- [Fog99] Karl Franz Fogel. *Open Source Development with CVS: Learn How to Work With Open Source Software*. The Coriolis Group, third edition, 1999.
- [HT03] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing with JUnit*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.
- [Pro04] Pragmatic Programmers. *Pragmatic Automation*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, (planned for) 2004.
- [Pur00] Gregor N. Purdy. *CVS Pocket Reference*. O'Reilly & Associates, Inc, Sebastopol, CA, 2000.
- [Ves03] Jennifer Vesperman. *Essential CVS*. O'Reilly & Associates, Inc, Sebastopol, CA, 2003.