使用 Subversion 进行版本控制

针对 Subversion 1.4（根据 r2866 编译）

Ben Collins-Sussman
Brian W. Fitzpatrick
C. Michael Pilato

(TBA)

目录

插图清单

表格清单

前言

Karl Fogel
芝加哥，2004 年 3 月 14 日

一个差劲的常见问题列表（FAQ）总是充斥着作者*渴望被问到的*问题，而不是人们真正想要了解的问题。也许你曾经见过下面这样的问题：

Q：怎样使用 Glorbosoft XYZ 最大程度的提高团队生产率？

A：许多客户希望知道怎样利用我们革命性的专利办公套件最大程度的提高生产率。答案非常简单：首先，点击“文件” 菜单，找到“提高生产率”菜单项，然后…

类似的问题完全不符合 FAQ 的精神。没人会打电话给技术支持中心，询问“怎样提高生产率？”相反，人们经常询问一些非常具体的问题，像“怎样让日程系统提前两天而不是一天提醒相关用户？”等等。但是想象比发现真正的问题更容易。构建一个真实的问题列表需要持之以恒的、有组织的辛勤工作：跨越整个软件生命周期，追踪新提出的问题，监控反馈信息，所有的问题要整理成一个统一的、可查询的整体，并且能够真实的反映所有用户的感受。这需要耐心，如自然学家一样严谨的态度，没有浮华的假设，没有虚幻的断言—相反的，需要开放的视野和精确的记录。

我很喜欢这本书，因为它正是按照这种精神建立起来的，这种精神体现在本书的每一页中。这是作者与用户直接交流的结果。而这一切是源于 Ben Collins-Sussman's 对于 Subversion 常见问题邮件列表的研究。他发现人们总是在邮件列表中重复询问一些基本问题：使用 subversion

的一般程序是怎样的？分支与标签同其它版本控制系统的工作方式一样吗？我怎样知道某一处修改是谁做的？

日复一日看到相同问题的烦闷，促使 Ben 在 2002 年的夏天努力工作了一个月，撰写了一本 *Subversion 手册*，一本六十页厚的、涵盖了所有 Subversion 使用基础知识的手册。这本手册没有说明最终定稿的时间，但它随着 Subversion 的每个版本一起发布，帮助许多用户跨过学习之初的艰难。当 O'Reilly 和 Associates 决定出版一本完备的 Subversion 图书的时候，一条捷径浮出水面：扩充 Subversion 手册。

新书的三位合著者因而面临着一个不寻常的机会。从职责上讲，他们的任务是从一个目录和一些草稿为基础，自上而下的写一部专著。但事实上，他们的灵感源泉则来自一些具体的内容，稳定却难以组织。Subversion 被数以千计的早期用户采用，这些用户提供了大量的反馈，不仅仅针对 Subversion，还包括业已存在的文档。

在写这本书的过程里，Ben，Mike 和 Brian 一直像鬼魂一样游荡在 Subversion 邮件列表和聊天室中，仔细的研究用户实际遇到的问题。监视这些反馈也是他们在 CollabNet 工作的一部分，这给他们撰写 Subversion 文档提供了巨大的便利。这本书建立在丰富的使用经验，而非在流沙般脆弱的想象之上，它结合了用户手册和 FAQ 的优点。初次阅读时，这种二元性的优势并不明显，按照顺序，从前到后，这本书只是简单的从头到尾描述了软件的细节。书中的内容包括一章概述，一章必不可少的快速指南，一章关于管理配置，一些高级主题，当然还包括

命令参考手册和故障排除指南。而当你过一段时间之后，再次翻开本书查找一些特定问题的解决方案时，这种二元性才得以显现：这些生动的细节一定来自不可预测的实际用例的提炼，大多是源于用户的需要和视点。

当然，没人可以承诺这本书可以回答所有问题。尽管有时候一些前人提问的惊人一致性让你感觉是心灵感应；你仍有可能在社区的知识库里摔跤，空手而归。如果有这种情况，最好的办法是写明问题发送 email 到 <users@subversion.tigris.org>，作者还在那里关注着社区，不仅仅封面提到的三位，还包括许多曾经作出修正与提供原始材料的人。从社区的视角，帮你解决问题只是逐步的调整这本书，进一步调整 Subversion 本身以更合理的适合用户使用这样一个大工程的一个有趣的额外效用。他们渴望你的信息，不仅仅可以帮助你，也因为可以帮助他们。与 Subversion 这样活跃的自由软件项目一起，*你并不孤单*。

让这本书将成为你的第一个伙伴。

序言

目录

> “即使你能确认什么是完美，也不要让完美成为好的敌人，更何况你不能确认。因为落入过去陷阱的不悦，你会在设计时因为担心自己的缺陷而无所作为。”
>
> --Greg Hudson

在开源软件世界，长久以来，并行版本系统（CVS）一直是版本控制工具的唯一选择。事实证明，这个选择不错。CVS 的自由软件身份，无约束的*处事态度*，和对网络化操作的支持（网络使众多身处不同地方的程序员可以共享他们的工作成果），正符合了开源世界协作的精神，CVS 和它半混乱状态的开发模式已成为开源文化的基石。

但是，CVS 也并不是没有缺陷，而修正这些缺陷必定要耗费很大的精力。而 Subversion 则是以 CVS 继任者的面目出现的新型版本控制系统。Subversion 的设计者们力图通过两方面的努力赢得 CVS 用户的青睐：保持开源系统的设计（以及“界面风格”）与 CVS 尽可能类似，同时尽力弥补 CVS 许多显著的缺陷。这些努力的结果使得从 CVS 迁移到 Subversion 不需要作出重大的变革，Subversion 确实是非常强大、非常有用和非常灵活的工具。并且很重要的一点，几乎新开的开源项目都选择了 Subversion 替代 CVS。

本书是为 Subversion 1.4 系列撰写的。在书中，我们尽力涵盖 Subversion 的所有内容。但是，Subversion 有一个兴盛和充满活力的开发社区，已有许多新的特性和改进措施计划在 Subversion 新版本中实现，本书中讲述的命令和特性可能会有所变化。

读者

本书是为了那些在计算机领域有丰富知识，并且希望使用 Subversion 管理数据的人士准备的。尽管 Subversion 可以在多种不同的操作系统上运行，但其基本用户操作界面是基于命令行的，也就是我们将要在本书中讲述和使用的命令行工具（**svn**）。

出于一致性的考虑，本书的例子假定读者使用的是类 Unix 的操作系统，并且熟悉 Unix 和命令行界面。当然，**svn** 程序也可以在入 Microsoft Windows 这样的非 Unix 平台上运行，除了一些微小的不同，如使用反

斜线（\）代替正斜线（/）作为路径分隔符，在 Windows 上运行 svn 程序的输入和输出与在 Unix 平台上运行完全一致。

大多数读者可能是那些需要跟踪代码变化的程序员或者系统管理员，这是 Subversion 最普遍的用途，因此这个场景贯穿于整本书的例子中。但是 Subversion 可以用来管理任何类型的数据：图像、音乐、数据库、文档等等。对于 Subversion，数据就是数据而已。

本书假定读者从来没有使用过任何版本控制工具，同时，我们也努力使 CVS 用户能够轻松的投入到 Subversion 使用当中，不时会出现一些涉及 CVS 的内容，此外，在附录的一个章节中总结了 Subversion 和 CVS 的区别。

需要说明的是，所有源代码示例仅仅是例子而已。这些例子需要通过正确编译参数进行编译，在这里列举它们只是为了说明特定的场景，并非为了展示优秀的编码风格。

怎样阅读本书

技术书籍经常要面对这样两难的困境：是迎合*自上至下*的初学者，还是*自下至上*的初学者。一个自上至下的学习者会喜欢略读文档，得到对系统工作原理的总体看法；然后她才会开始实际使用软件。而一个自下至上的学习者，是“通过实践学习”的人，他们希望快速的开始使用软件，自己领会软件的使用，只在必要时读取相关章节。大多数图书会倾向于针对某一类读者，而本书毫无疑问倾向于自上至下的方法。（如果你阅

读了本节，那你也一定是一个自上至下的学习者！）然而，如果你是自下至上的人，不要失望，本书以 Subversion 主题的广泛观察进行组织，每个章节都包含了大量可以尝试的详细实例。如果你希望马上开工，没有耐心等待，你可以看附录 A, *Subversion 快速入门指南*。

本书适用于具有不同背景知识的各个层次的读者—从未使用过版本控制的新手到经验丰富的系统管理员都能够从本书中获益。根据基础的不同，某些的章节可能对某些读者更有价值。下面的内容可以看作是为不同类型的读者提供的“推荐阅读清单”：

资深系统管理员

> 假定你从前使用过版本控制，并且迫切需要建立起 Subversion 服务器并尽快运行起来，第 5 章 *版本库管理*和第 6 章 *服务配置*将会告诉你如何建立起一个版本库，并将其在网络上发布。此后，依靠你的 CVS 使用经验，第 2 章 *基本使用*和附录 B, *CVS 用户的Subversion 指南*将向你展示怎样使用 Subversion 客户端软件。

新用户

> 如果管理员已经为你准备好了 Subversion 服务，你所需要的是学习如何使用客户端。如果你没有使用版本控制系统（像 CVS）的经验，那么第 1 章 *基本概念*和第 2 章 *基本使用*是重要的入门教程，其中介绍了版本控制的重要思想。

高级用户

无论是用户还是管理员，项目终将会壮大起来。那时，就需要学习更多 Subversion 的高级功能，像如何使用分支和执行合并（第 4 章 *分支与合并*）、怎样使用 Subversion 的属性（第 3 章 *高级主题*）、怎样配制运行参数（第 7 章 *定制你的 Subversion 体验*）等等。这两章在学习的初期并不重要，但熟悉了基本操作之后还是非常有必要了解一下。

开发者

你应该已经很熟悉 Subversion 了，并且想扩展它或使用它的 API 开发新软件。第 8 章 *嵌入 Subversion* 将最适合你。

本书以参考材料作为结束—第 9 章 *Subversion 完全参考*是一部 Subversion 全部命令的详细指南，此外，在附录中还有许多很有意义的主题。阅读完本书后，这些章节将会是你经常查阅的内容。

本书约定

本节描述了本书中使用的各种约定。

排版习惯

等宽字体

用于命令，命令输出和选项

等宽字体

用于代码和文本中的可替换部分

斜体

用于文件和路径名

图标

此图标表示旁边的文本内容需特别注意。

此图标表示旁边的文本描述了一个有用的小技巧。

此图标表示旁边的文本是警告信息。

本书的结构

以下是各个章节的内容介绍：

序言

回顾了 Subversion 的历史，描述了 Subversion 的特性、架构、组件。

第 1 章 *基本概念*

介绍了版本控制的基础知识及不同的版本模型，同时讲述了 Subversion 版本库，工作拷贝和修订版本的概念。

第 2 章 *基本使用*

引领你开始一个 Subversion 用户的工作。示范怎样使用 Subversion 获得、修改和提交数据。

第 3 章 *高级主题*

覆盖了许多普通用户最终要面对的复杂特性，例如版本化的元数据、文件锁定和 peg 修订版本。

第 4 章 *分支与合并*

讨论分支、合并与标签，包括最佳实践的介绍，常见用例的描述，怎样取消修改，以及怎样从一个分支转到另一个分支。

第 5 章 *版本库管理*

讲述 Subversion 版本库的基本概念，怎样建立、配置和维护版本库，以及哪些工具可以完成上述的工作。

第 6 章 *服务配置*

描述了如何配置 Subversion 服务器，以及三种访问版本库的方式，HTTP、svn 协议和本地磁盘访问。这里也介绍了认证，授权与匿名访问的细节。

第 7 章 *定制你的 Subversion 体验*

研究了 Subversion 的客户端配置文件，对国际化字符的处理，以及 Subversion 如何与外置工具交互。

第 8 章 *嵌入 Subversion*

介绍了 Subversion 的核心部件、Subversion 的文件系统，以及程序员眼中的工作拷贝管理区域，展示了如何使用公共 API 编写 Subversion 应用程序。最重要的内容是，如何为 Subversion 的开发贡献力量。

第 9 章 *Subversion 完全参考*

以大量的实例，详细描述了 **`svn`**、**`svnadmin`** 和 **`svnlook`** 的所有子命令。

附录 A, *Subversion 快速入门指南*

因为缺乏耐心，我们会立刻解释如何安装和使用 Subversion，我们已经告诉你了。

附录 B, *CVS 用户的 Subversion 指南*

详细比较了 Subversion 与 CVS 的异同，并针对如何消除多年使用 CVS 养成的坏习惯提出建议。内容包括 Subversion 修订版本号、版本化的目录、离线操作、**`update`** 与 **`status`** 的对比、分支、标签、元数据、冲突处理和认证。

附录 C, *WebDAV 和自动版本*

描述了 WebDAV 与 DeltaV 的细节，并介绍了如何将 Subversion 版本库作为可读/写的 DAV 共享装载。

附录 D, *第三方工具*

讨论一些支持和使用 Subversion 的工具，包括其它客户端工具，版本库浏览工具等。

本书是免费的

本书最初是作为 Subversion 项目的文档并由 Subversion 的开发者开始撰写的，后来成为一个独立的项目并进行了重写。与 Subversion 相同，它始终按免费许可证（见附录 E, *Copyright*）发布。事实上，本书是在公众的关注中写出来的，最初是 Subversion 项目的一部分，这有两种含义：

- 总可以在 Subversion 的版本库里找到本书的最新版本。
- 可以任意分发或修改本书—它在免费许可证的控制之下，你的唯一限制是必须保留正确的最初作者。当然，与其独自发布私有版本，不如向 Subversion 开发社区提供反馈和修正信息。

本书的在线主页在 http://svnbook.red-bean.com，有许多志愿的翻译工作。在网站上，你可以找到许多本书最新快照和标签版本的链接，也可以访问到本书的 Subversion 版本库（存放了 DocBook XML 源文件）。我们欢迎反馈—也愿意接受鼓励。请将所有的评论、抱怨和对本书源文件

的补丁发送到<svnbook-dev@red-bean.com>。本书的中文版主要是由 Subversion 中文站的志愿者翻译的，可以在 http://www.subversion.org.cn/看到本书的最新版本和其他资料，也要感谢 i18n-zh 的朋友的一些支持。

致谢

没有 Subversion 就不可能有（即使有也没什么价值）这本书。所以作者衷心感谢 Brian Behlendorf 和 CollabNet，他们独到的眼光开创了这个充满冒险但雄心勃勃的开源项目；Jim Blandy 贡献了 Subversion 最初的名字和设计—我们爱你，Jim。还有 Karl Fogel，一个好朋友和伟大的社区领袖。[1]

感谢 O'Reilly 和我们的编辑 Linda Mui 和 Tatiana 对我们的耐心和支持。

最后，我们要感谢数不清的曾经为本书作出贡献的人们，他们进行了非正式的审阅，并给出了大量建议和修改意见。虽然无法列出一个完整的列表，但本书的完整和正确离不开他们：David Anderson, Jani Averbach, Ryan Barrett, Francois Beausoleil, Jennifer Bevan, Matt Blais, Zack Brown, Martin Buchholz, Brane Cibej, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Nick Duffek, Ben Elliston, Justin Erenkrantz, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Dave Gilbert, Eric Gillespie, David Glasser, Matthew Gregan, Art Haas, Eric Hanchrow, Greg Hudson, Alexis Huxley, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Andreas J. Koenig, Nuutti Kotivuori, Matt Kraai, Scott Lamb, Vincent Lefevre, Morten Ludvigsen, Paul Lussier, Bruce A. Mah, Philip Martin, Feliciano Matias, Patrick Mayweg, Gareth McCaughan, Jon Middleton, Tim Moloney, Christopher Ness, Mats Nilsson, Joe Orton, Amy Lyn Pilato, Kevin Pilch-Bisson, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen

Prohaska, Daniel Rall, Jack Repenning, Tobias Ringstrom, Garrett Rooney, Joel Rosdahl, Christian Sauer, Larry Shatzer, Russell Steicke, Sander Striker, Erik Sjoelund, Johan Sundstroem, John Szakmeister, Mason Thomas, Eric Wadsworth, Colin Watson, Alex Waugh, Chad Whitacre, Josef Wolf, Blair Zajac, 以及整个 Subversion 社区。

来自 Ben Collins-Sussman

感谢我的妻子 Frances，在好几个月里，我一直在对你说：“但是亲爱的，我还在为这本书工作”，此外还有，“但是亲爱的，我还在处理邮件”。我不知道她为什么会如此耐心！她是我完美的平衡点。

感谢我的家人对我的鼓励，无论他们是否真的对我的课题感兴趣。（你知道的，一个人说 “哇，你正在写一本书？”，然后当他知道你是写一本计算机书时，那种惊讶就变得没有那么多了。）

感谢我身边让我富有的朋友，不要那样看我—你们知道你们是谁。

感谢父母对我的低级格式化，和难以置信的角色典范 ，感谢儿子给我机会传承这些东西。

来自 Brian W. Fitzpatrick

非常非常感谢我的妻子 Marie 的理解，支持和最重要的耐心。感谢引导我学会 UNIX 编程的兄弟 Eric，感谢我的母亲和祖母的支持，对我在圣诞夜里埋头工作的理解。

Mike 和 Ben：与你们一起工作非常快乐，Heck，我们在一起工作很愉快！

感谢所有在 Subversion 和 Apache 软件基金会的人们给我机会与你们在一起，没有一天我不从你们那里学到知识。

最后，感谢我的祖父，他一直跟我说“自由等于责任”，我深信不疑。

来自 C. Michael Pilato

特别感谢 Amy，我最好的朋友和 9 年里不可思议的妻子，因为她的爱和耐心支持，因为她提供的深夜工作，因为她对我强加给她的版本控制过程的优雅忍受 。不要担心，甜心—你会立刻成为 TortoiseSVN 巫师！

Gavin，或许现在本书的很多词你还不能读出来，但是当你最终能够书写我们所说的疯狂语言时，希望你会为你的父亲感到骄傲，就像他对你一样。

Aidan, Daddy luffoo et ope Aiduh yike contootoo as much as Aiduh yike batetball, base-ball, et bootball. [2]

妈妈和爸爸，感谢你们的支持和热情，岳父岳母，以同样的理由感谢你们，还要感谢你们难以置信的女儿。

向你们致敬，Shep Kendall，为我打开了通向计算机世界的大门；Ben Collins Sussman，我在开源世界的导师；Karl Fogel—你*是*我的*.emacs*；

Greg Stain，让我在困境中知道怎样编程；Brain Fitzpatrick—同我分享他的写作经验。所有我曾经从你们那里获得知识的人—尽管又不断忘记。

最后，对所有为我展现完美卓越创造力的人们—感谢。

Subversion 是什么？

Subversion 是一个自由/开源的版本控制系统。也就是说，在 Subversion 管理下，文件和目录可以超越时空。也就是 Subversion 允许你数据恢复到早期版本，或者是检查数据修改的历史。正因为如此，许多人将版本控制系统当作一种神奇的“时间机器”。

Subversion 的版本库可以通过网络访问，从而使用户可以在不同的电脑上进行操作。从某种程度上来说，允许用户在各自的空间里修改和管理同一组数据可以促进团队协作。因为修改不再是单线进行，开发速度会更快。此外，由于所有的工作都已版本化，也就不必担心由于错误的更改而影响软件质量—如果出现不正确的更改，只要撤销那一次更改操作即可。

某些版本控制系统本身也是软件配置管理（SCM）系统，这种系统经过精巧的设计，专门用来管理源代码树，并且具备许多与软件开发有关的特性—比如，对编程语言的支持，或者提供程序构建工具。不过 Subversion 并不是这样的系统。它是一个通用系统，可以管理*任何*类型的文件集。对你来说，这些文件这可能是源程序—而对别人，则可能是一个货物清单或者是数字电影。

Subversion 的历史

早在 2000 年，CollabNet, Inc. (http://www.collab.net)就开始寻找 CVS 替代产品的开发人员。CollabNet 提供了一个名为 CollabNet 企业版（CEE）的协作软件套件。这个软件套件的一个组成部分就是版本控制系统。尽管 CEE 在最初采用了 CVS 作为其版本控制系统，但是 CVS 的局限性从一开始就很明显，CollabNet 知道，迟早要找到一个更好的替代品。遗憾的是，CVS 已经成为开源世界事实上的标准，很大程度上是因为*没有*更好的替代品，至少是没有可以自由使用的替代品。所以 CollabNet 决定从头编写一个新的版本控制系统，这个系统保留 CVS 的基本思想，但是要修正其中错误和不合理的特性。

2000 年 2 月，他们联系到 *Open Source Development with CVS*(Coriolis, 1999)的作者 Karl Fogel，并且询问他是否希望为这个新项目工作。巧合的是，当时 Karl 正在与朋友 Jim Blandy 讨论设计一个新的版本控制系统。1995 年时，他们两人曾经开办了一个提供 CVS 支持的公司 Cyclic Software，尽管他们最终卖掉了公司，但还是天天使用 CVS 进行日常工作。使用 CVS 时的挫折促使 Jim 认真的思考如何管理版本化的数据，并且他当时不仅使用了“Subversion”这个名字，并且已经完成了 Subversion 版本库的最初设计。所以当 CollabNet 提出邀请的时候，Karl 马上同意为这个项目工作，同时 Jim 也找到了他的雇主—Red Hat 软件公司—允许他到这个项目工作，并且没有限定最终的期限。CollabNet 雇佣了 Karl 和 Ben Collins Sussman，详细设计工作从

三月开始，在 Behlendorf 、CollabNet、Jason Robbins 和 Greg Stein（当时是一个独立开发者，活跃在 WebDAV/DeltaV 系统规范制订工作中）恰到好处的激励下，Subversion 很快吸引了许多活跃的开发者，结果是许多对 CVS 有过失望经历的人很乐于为这个项目做些事情。

最初的设计小组设定了简单的开发目标。他们不想在版本控制方法学中开垦处女地，他们只是希望修正 CVS。他们决定 Subversion 应符合 CVS 的特性，并保留相同的开发模型，但不再重复 CVS 的一些显著缺陷。尽管 Subversion 并不需要成为 CVS 的完全替代品，但它应该与 CVS 保持足够的相似性，以使 CVS 用户可以轻松的转移到 Subversion 上。

经过 14 个月的编码，2001 年 8 月 31 日，Subversion 能够“自己管理自己”了，开发者停止使用 CVS 保存 Subversion 的代码，而使用 Subversion 本身。

虽然 CollabNet 启动了这个项目，并且一直提供了大量的工作支持（它为一些全职的 Subversion 开发者提供薪水），但 Subversion 像其它许多开源项目一样，被松散的、透明的规则管理着，这样的规则激励着知识界的精英们。CollabNet 的版权许可证完全符合 Debian 的自由软件方针。也就是说，任何人都可以根据自己的意愿自由的下载、修改和重新发布 Subversion，不需要 CollabNet 或其他人的授权。

Subversion 的特性

在讲解 Subversion 为版本控制领域带来的特性时，我们会经常通过 Subversion 对 CVS 的改进进行说明。如果不熟悉 CVS，了解所有 Subversion 的特性会有一定的困难。而如果根本就不熟悉版本控制，你就只有干瞪眼的份儿了。因此，最好首先阅读一下第 1 章 *基本概念*，这一章简单介绍了一些版本控制的基本思想和概念。

Subversion 支持：

版本化的目录

> CVS 只能跟踪单个文件的变更历史，但是 Subversion 实现的“虚拟”版本化文件系统则可以跟踪目录树的变更。在 Subversion 中，文件*和*目录都是版本化的。

真实的版本历史

> 由于只能跟踪单个文件的变更，CVS 无法支持如文件拷贝和改名这些常见的操作—这些操作改变了目录的内容。同样，在 CVS 中，一个目录下的文件只要名字相同即拥有相同的历史，即使这些同名文件在历史上毫无关系。而在 Subversion 中，可以对文件或目录进行增加、拷贝和改名操作，也解决了同名而无关的文件之间的历史联系问题。

原子提交

一系列相关的更改，要么全部提交到版本库，要么一个也不提交。这样用户就可以将相关的更改组成一个逻辑整体，防止出现只有部分修改提交到版本库的情况。

版本化的元数据

每一个文件和目录都有自己的一组属性—键和它们的值。可以根据需要建立并存储任何键/值对。和文件本身的内容一样，属性也在版本控制之下。

可选的网络层

Subversion 在版本库访问的实现上具有较高的抽象程度，利于人们实现新的网络访问机制。Subversion 可以作为一个扩展模块嵌入到 Apache 之中。这种方式在稳定性和交互性方面有很大的优势，可以直接使用服务器的成熟技术—认证、授权和传输压缩等。此外，Subversion 自身也实现了一个轻型的，可独立运行的服务器软件。这个服务器使用了一个自定义协议，可以轻松的用 SSH 封装。

一致的数据操作

Subversion 用一个二进制差异算法描述文件的变化，对于文本（可读）和二进制（不可读）文件其操作方式是一致的。这两种

类型的文件压缩存储在版本库中，而差异信息则在网络上双向传递。

高效的分支和标签操作

在 Subversion 中，分支与标签操作的开销与工程的大小无关。Subversion 的分支和标签操作用只是一种类似于硬链接的机制拷贝整个工程。因而这些操作通常只会花费很少且相对固定的时间。

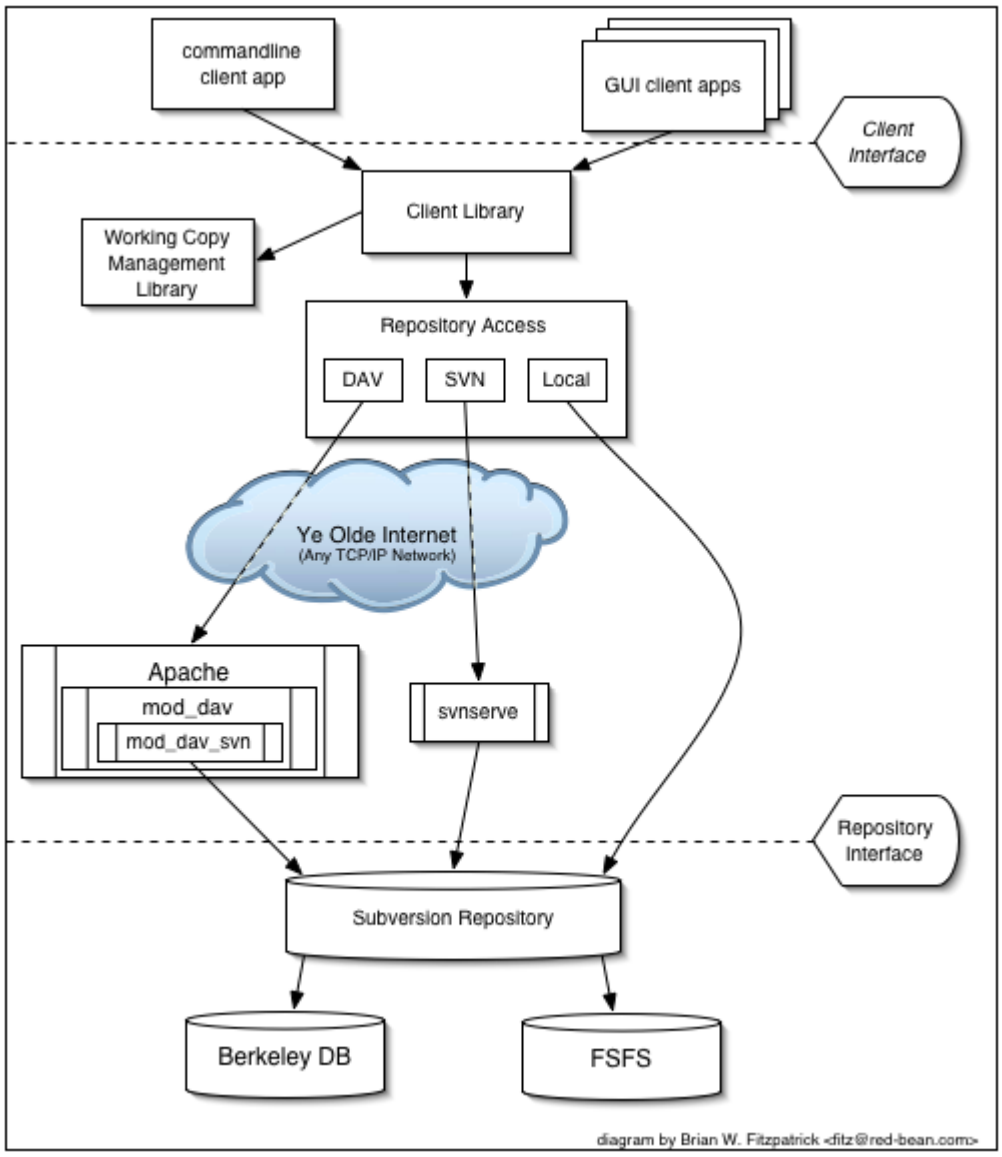
可修改性

Subversion 没有历史负担，它以一系列优质的共享 C 程序库的方式实现，具有定义良好的 API。这使得 Subversion 非常容易维护，和其它语言的互操作性很强。

Subversion 的架构

图 1 “Subversion 的架构”给出了 Subversion 设计总体上的“俯视图”。

图 1. Subversion 的架构



图中的一端是保存所有版本数据的 Subversion 版本库，另一端是 Subvesion 的客户程序，管理着所有版本数据的本地影射（称为“工作拷贝”），在这两极之间是各种各样的版本库访问（RA）层，某些使用电脑网络通过网络服务器访问版本库，某些则绕过网络服务器直接访问版本库。

Subversion 的组件

安装好的 Subversion 由几个部分组成，下面将简单的介绍一下这些组件。下文的描述或许过于简略，不易理解，但不用担心—本书后面的章节中会用*更多的*内容来详细阐述这些组件。

svn

命令行客户端程序。

svnversion

此工具用来显示工作拷贝的状态（用术语来说，就是当前项目的修订版本）。

svnlook

直接查看 Subversion 版本库的工具。

svnadmin

建立、调整和修复 Subversion 版本库的工具。

svndumpfilter

过滤 Subversion 版本库转储数据流的工具。

mod_dav_svn

Apache HTTP 服务器的一个插件，使版本库可以通过网络访问。

svnserve

一个单独运行的服务器程序，可以作为守护进程或由 SSH 调用。这是另一种使版本库可以通过网络访问的方式。

如果已经正确完成了 Subversion 的安装，我们就可以开始我们的学习之旅了。在后面的两章中，我们将讲解如何使用 Subversion 的客户端程序 **svn**。

---

[1] 噢，还要感谢 Karl 为了本书所付出的辛勤工作。

[2] 翻译：爸爸希望你会像喜欢篮球、棒球和足球一样喜欢计算机。（不是很明显吗？）

基本概念

目录

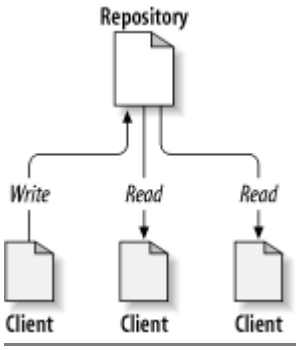
本章主要为那些不熟悉版本控制技术的入门者提供一个简单扼要的、非系统的介绍。我们将从版本控制的基本概念开始，随后阐述 Subversion 的独特理念，并演示一些使用 Subversion 的例子。

虽然我们在本章中以分享程序源代码作为例子，但是记住 Subversion 可以管理任何类型的文件集—它并非是程序员专用的。

版本库

Subversion 是一个“集中式”的信息共享系统。版本库是 Subversion 的核心部分，是数据的中央仓库。版本库以典型的文件和目录结构形式*文件系统树*来保存信息。任意数量的*客户端*连接到 Subversion 版本库，读取、修改这些文件。客户端通过写数据将信息分享给其他人，通过读取数据获取别人共享的信息。图 1.1 “一个典型的客户/服务器系统”展示了这种系统：

图 1.1. 一个典型的客户/服务器系统



这有什么意义吗？说了这么多，Subversion 听起来和一般的文件服务器没什么不同。事实上，Subversion 的版本库的确*是*一种文件服务器，但不是“一般”的文件服务器。Subversion 版本库的特别之处在于，*它会记录每一次改变*：每个文件的改变，甚至是目录树本身的改变，例如文件和目录的添加、删除和重新组织。

一般情况下，客户端从版本库中获取的数据是文件系统树中的最新数据。但是客户端也具备查看文件系统树*以前*任何一个状态的能力。举个例子，客户端有时会对一些历史性问题感兴趣，比如“上星期三时的目录结构是什么样的？”或者“谁最后一个修改了这个文件，都修改了什么？”这些都是*版本控制系统*的核心问题：设计用来记录和跟踪数据变化的系统。

版本模型

版本控制系统的核心任务是实现协作编辑和数据共享，但是不同的系统使用不同的策略实现这个目的。我们有许多理由要去理解这些策略的区别，首先，如果你遇到了其他类似 Subversion 的系统，可以帮助你比

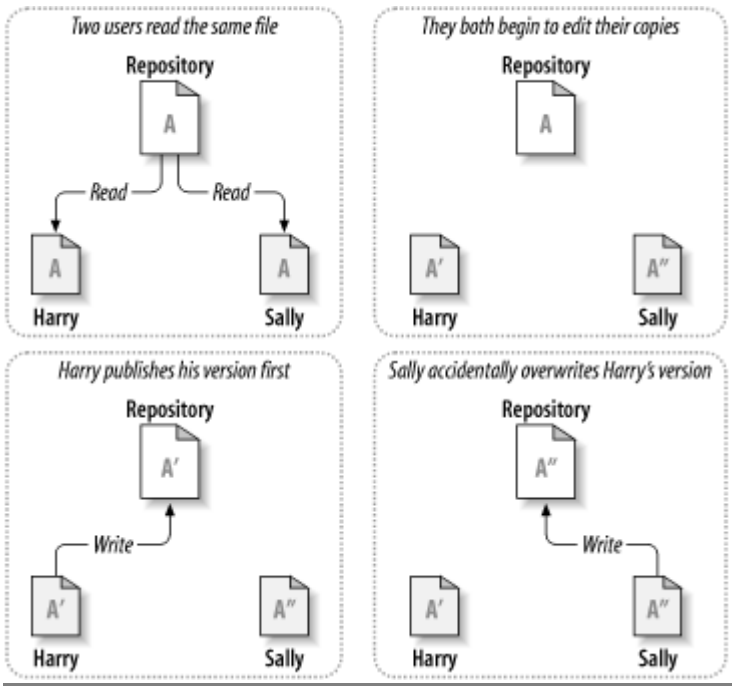
较现有的版本控制系统。此外，可以帮助你更有效的使用 Subversion，因为 Subversion 本身支持不同的工作方式。

文件共享的问题

所有的版本控制系统都需要解决这样一个基础问题：怎样让系统允许用户共享信息，而不会让他们因意外而互相干扰？版本库里意外覆盖别人的更改非常的容易。

考虑图 1.2 “需要避免的问题”的情景，我们有两个共同工作者，Harry 和 Sally，他们想同时编辑版本库里的同一个文件，如果首先 Harry 保存它的修改，过了一会，Sally 可能凑巧用自己的版本覆盖了这些文件，Harry 的更改不会永远消失（因为系统记录了每次修改），但 Harry 所有的修改*不会*出现在 Sally 新版本的文件中，所以 Harry 的工作还是丢失了—至少是从最新的版本中丢失了—而且可能是意外的，这就是我们要明确避免的情况！

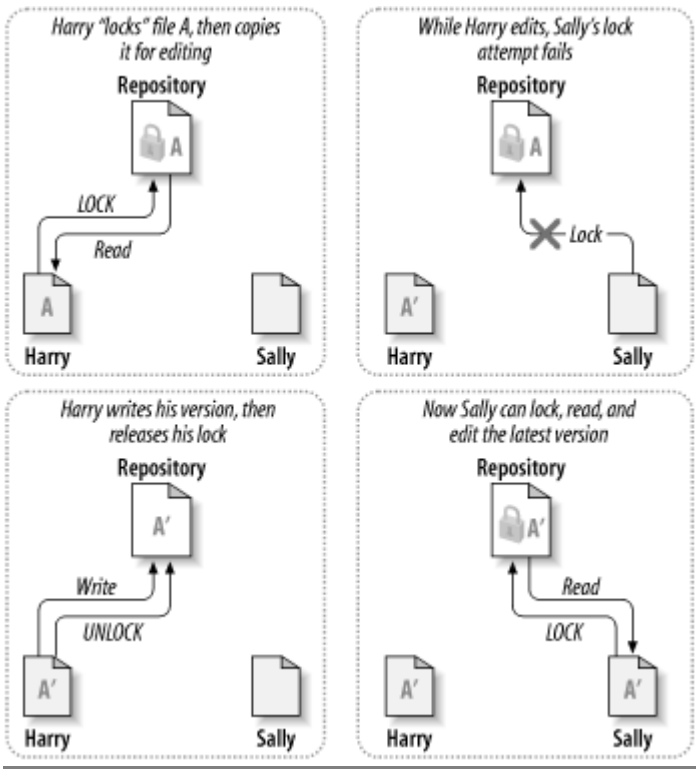
图 1.2. 需要避免的问题



锁定-修改-解锁 方案

许多版本控制系统使用*锁定-修改-解锁*机制解决这种问题，在这样的模型里，在一个时间段里版本库的一个文件只允许被一个人修改。首先在修改之前，Harry 要“锁定”住这个文件，锁定很像是从图书馆借一本书，如果 Harry 锁住这个文件，Sally 不能做任何修改，如果 Sally 想请求得到一个锁，版本库会拒绝这个请求。在 Harry 结束编辑并且放开这个锁之前，她只可以阅读文件。Harry 解锁后，就要换班了，Sally 得到自己的轮换位置，锁定并且开始编辑这个文件。图 1.3 “锁定-修改-解锁 方案”描述了这样的解决方案。

图 1.3. 锁定-修改-解锁 方案



锁定-修改-解锁模型有一点问题就是限制太多，经常会成为用户的障碍：

- *锁定可能导致管理问题。*有时候 Harry 会锁住文件然后忘了此事，这就是说 Sally 一直等待解锁来编辑这些文件，她在这里僵住了。然后 Harry 去旅行了，现在 Sally 只好去找管理员放开锁，这种情况会导致不必要的耽搁和时间浪费。
- *锁定可能导致不必要的线性化开发。*如果 Harry 编辑一个文件的开始，Sally 想编辑同一个文件的结尾，这种修改不会冲突，设想修改可以正确的合并到一起，他们可以轻松的并行工作而没有太多的坏处，没有必要让他们轮流工作。
- *锁定可能导致错误的安全状态。*假设 Harry 锁定和编辑一个文件 A，同时 Sally 锁定并编辑文件 B，如果 A 和 B 互相依赖，这种变

化是必须同时作的，这样 A 和 B 不能正确的工作了，锁定机制对防止此类问题将无能为力—从而产生了一种处于安全状态的假相。很容易想象 Harry 和 Sally 都以为自己锁住了文件，而且从一个安全，孤立的情况开始工作，因而没有尽早发现他们不匹配的修改。锁定经常成为真正交流的替代品

拷贝-修改-合并 方案

Subversion，CVS 和一些版本控制系统使用*拷贝-修改-合并*模型，在这种模型里，每一个客户联系项目版本库建立一个个人*工作拷贝*—版本库中文件和目录的本地映射。用户并行工作，修改各自的工作拷贝，最终，各个私有的拷贝合并在一起，成为最终的版本，这种系统通常可以辅助合并操作，但是最终要靠人工去确定正误。

这是一个例子，Harry 和 Sally 为同一个项目各自建立了一个工作拷贝，工作是并行的，修改了同一个文件 A，Sally 首先保存修改到版本库，当 Harry 想去提交修改的时候，版本库提示文件 A 已经*过期*，换句话说，A 在他上次更新之后已经更改了，所以当他通过客户端请求*合并*版本库和他的工作拷贝之后，碰巧 Sally 的修改和他的不冲突，所以一旦他把所有的修改集成到一起，他可以将工作拷贝保存到版本库，图 1.4 “拷贝-修改-合并 方案”和图 1.5 “拷贝-修改-合并 方案（续）”展示了这一过程。

图 1.4. 拷贝-修改-合并 方案

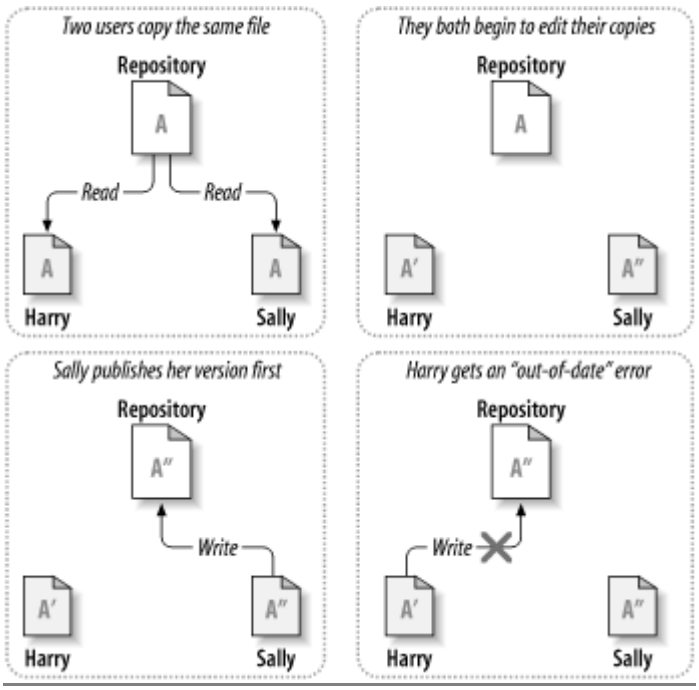
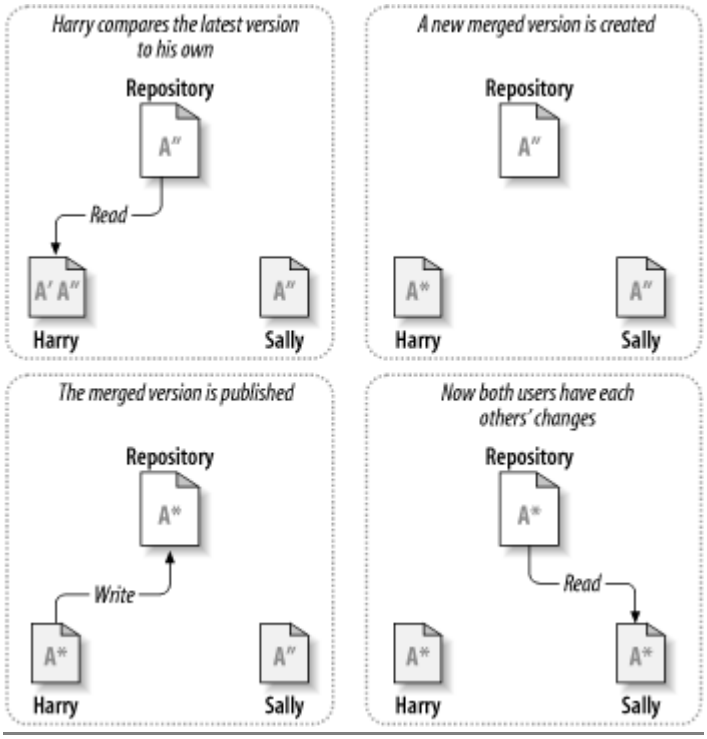


图 1.5. 拷贝-修改-合并 方案（续）



但是如果 Sally 和 Harry 的修改*交迭*了该怎么办？这种情况叫做*冲突*，这通常不是个大问题，当 Harry 告诉他的客户端去合并版本库的最新修

改到自己的工作拷贝时，他的文件 A 就会处于冲突状态：他可以看到一对冲突的修改集，并手工的选择保留一组修改。需要注意的是软件不能自动的解决冲突，只有人可以理解并作出智能的选择，一旦 Harry 手工的解决了冲突—也许需要与 Sally 讨论—它可以安全的把合并的文件保存到版本库。

拷贝-修改-合并模型感觉有一点混乱，但在实践中，通常运行的很平稳，用户可以并行的工作，不必等待别人，当工作在同一个文件上时，也很少会有交迭发生，冲突并不频繁，处理冲突的时间远比等待解锁花费的时间少。

最后，一切都要归结到一条重要的因素：用户交流。当用户交流贫乏，语法和语义的冲突就会增加，没有系统可以强制用户完美的交流，没有系统可以检测语义上的冲突，所以没有任何证据能够承诺锁定系统可以防止冲突，实践中，锁定除了约束了生产力，并没有做什么事。

什么时候锁定是必需的

锁定-修改-解锁模型被认为不利于协作，但有时候锁定会更好。

拷贝-修改-合并模型假定文件是可以根据上下文合并的：就是版本库的文件主要是以行为基础的文本文件（例如程序源代码）。但对于二进制格式，例如艺术品或声音，在这种情况下，十分有必要让用户轮流修改文件，如果没有线性的访问，有些人的许多工作就最终要被放弃。

尽管 Subversion 一直主要是一个拷贝-修改-合并系统，但是它也意识到了需要锁定一些文件，并且提供这种锁定机制，这个特性的讨论可以见“锁定”一节。

Subversion 实践

是时候从抽象转到具体了，在本小节，我们会展示一个 Subversion 真实使用的例子。

Subversion 版本库 URL

正如我们在整本书里描述的，Subversion 使用 URL 来识别 Subversion 版本库中的版本化资源，通常情况下，这些 URL 使用标准的语法，允许服务器名称和端口作为 URL 的一部分：

```
$ svn checkout http://svn.example.com:9834/repos
…
```

但是 Subversion 处理 URL 的一些细微的不同之处需要注意，例如，使用 file:访问方法的 URL（用来访问本地版本库）必须与习惯一致，可以包括一个 localhost 服务器名或者没有服务器名：

```
$ svn checkout file:///path/to/repos
…
$ svn checkout file://localhost/path/to/repos
…
```

同样，在 Windows 平台下使用 file:// 模式时需要使用一个非正式的“标准”语法来访问本机上不在同一个磁盘分区中的版本库。下面的任意一个 URL 路径语法都可以工作，其中的 X 表示版本库所在的磁盘分区：

```
C:\> svn checkout file:///X:/path/to/repos

…

C:\> svn checkout "file:///X|/path/to/repos"

…
```

在第二个语法里，你需要使用引号包含整个 URL，这样竖线字符才不会被解释为管道。当然，也要注意 URL 使用普通的斜线而不是 Windows 本地（不是 URL）的反斜线。

也必须意识到 Subversion 的 file: URL 不能在普通的 web 服务器中工作。当你尝试在 web 服务器查看一个 file: 的 URL 时，它会通过直接检测文件系统读取和显示那个位置的文件内容，但是 Subversion 的资源存在于虚拟文件系统（见“版本库层”一节）中，你的浏览器不会理解怎样读取这个文件系统。

最后，必须注意 Subversion 的客户端会根据需要自动编码 URL，这一点和一般的 web 浏览器一样，举个例子，如果一个 URL 包含了空格或是一个字符编码大于 128 的 ASCII 字符：

```
$ svn checkout "http://host/path with space/project/españa"
```

…Subversion 会回避这些不安全字符，并且会像你输入了这些字符一样工作：

```
$ svn checkout
http://host/path%20with%20space/project/espa%C3%B1a
```

如果 URL 包含空格，一定要使用引号，这样你的脚本才会把它做一个单独的 **svn** 参数。

工作拷贝

你已经阅读过了关于工作拷贝的内容；现在我们要讲一讲客户端怎样建立和使用它。

一个 Subversion 工作拷贝是你本地机器上的一个普通目录，保存着一些文件，你可以任意的编辑文件，而且如果是源代码文件，你可以像平常一样编译，你的工作拷贝是你的私有工作区，在你明确的做了特定操作之前，Subversion 不会把你的修改与其他人的合并，也不会把你的修改展示给别人，你甚至可以拥有同一个项目的多个工作拷贝。

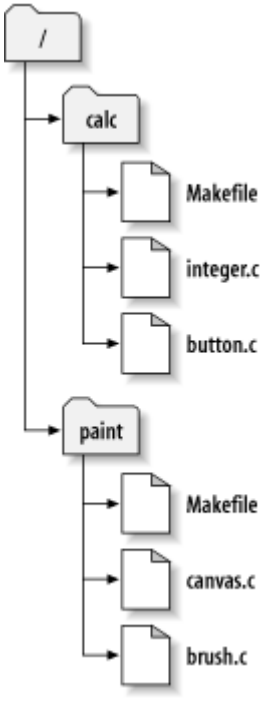
当你在工作拷贝作了一些修改并且确认它们工作正常之后，Subversion 提供了一个命令可以“发布”你的修改给项目中的其他人（通过写到版本库），如果别人发布了各自的修改，Subversion 提供了手段可以把这些修改与你的工作目录进行合并（通过读取版本库）。

工作副本也包括一些由 Subversion 创建并维护的额外文件，用来协助执行命令。通常情况下，你的工作副本的每个文件夹都有一个以 *.svn* 为名的文件夹，也被叫做工作副本的*管理目录*，这个目录里的文件能够帮助 Subversion 识别哪些文件做过修改，哪些文件相对于别人的工作已经过期。

一个典型的 Subversion 的版本库经常包含许多项目的文件（或者说源代码），通常每一个项目都是版本库的子目录，在这种布局下，一个用户的工作拷贝往往对应版本库的的一个子目录。

举一个例子，你的版本库包含两个软件项目，paint 和 calc。每个项目在它们各自的顶级子目录下，见图 1.6 “版本库的文件系统”。

图 1.6. 版本库的文件系统



为了得到一个工作拷贝，你必须*检出*（*check out*）版本库的一个子树，（术语“check out”听起来像是锁定或者保留资源，实际上不是，只是简单的得到一个项目的私有拷贝），举个例子，你检出 */calc*，你可以得到这样的工作拷贝：

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
A    calc/button.c
Checked out revision 56.

$ ls -A calc
Makefile  integer.c  button.c  .svn/
```

列表中的 A 表示 Subversion 增加了一些条目到工作拷贝，你现在有了一个*/calc* 的个人拷贝，有一个附加的目录—*.svn*—保存着前面提及的 Subversion 需要的额外信息。

假定你修改了 *button.c*，因为*.svn* 目录记录着文件的修改日期和原始内容，Subversion 可以告诉你已经修改了文件，然而，在你明确告诉它之前，Subversion 不会将你的改变公开，将改变公开的操作被叫做提交（*committing*，或者是 *checking in*）修改到版本库。

将你的修改发布给别人，你可以使用 Subversion 的提交（**`commit`**）命令。

```
$ svn commit button.c -m "Fixed a typo in button.c."

Sending       button.c

Transmitting file data .

Committed revision 57.
```

这时你对 *button.c* 的修改已经提交到了版本库，其中包含了关于此次提交的日志信息（例如是修改了拼写错误）。如果其他人取出了*/calc* 的一个工作拷贝，他们会看到这个文件最新的版本。

假设你有个合作者，Sally，她和你同时取出了*/calc* 的一个工作拷贝，你提交了你对 *button.c* 的修改，Sally 的工作拷贝并没有改变，Subversion 只在用户要求的时候才改变工作拷贝。

要使项目最新，Sally 可以要求 Subversion *更新*她的工作备份，通过使用更新（**update**）命令，将结合你和所有其他人在她上次更新之后的改变到她的工作拷贝。

```
$ pwd

/home/sally/calc



$ ls -A

.svn/ Makefile integer.c button.c



$ svn update

U   button.c
```

```
Updated to revision 57.
```

svn update 命令的输出表明 Subversion 更新了 *button.c* 的内容，注意，Sally 不必指定要更新的文件，subversion 利用*.svn* 以及版本库的进一步信息决定哪些文件需要更新。

版本库的 URL

Subversion 可以通过多种方式访问—本地磁盘访问，或各种各样不同的网络协议，这要看你的管理员是如何设置，但一个版本库地址永远都只是一个 URL，表 1.1 “”描述了不同的 URL 模式对应的访问方法。

表 1.1.

模式	访问方法
file:///	直接版本库访问（本地磁盘）
http://	通过配置 Subversion 的 Apache 服务器的 WebDAV 协议
https://	与 http://相似，但是包括 SSL 加密。
svn://	通过 svnserve 服务自定义的协议
svn+ssh://	与 svn://相似，但通过 SSH 封装。

关于 Subversion 解析 URL 的更多信息，见“Subversion 版本库 URL”一节。关于不同的网络服务器类型，见第 6 章 *服务配置*。

修订版本

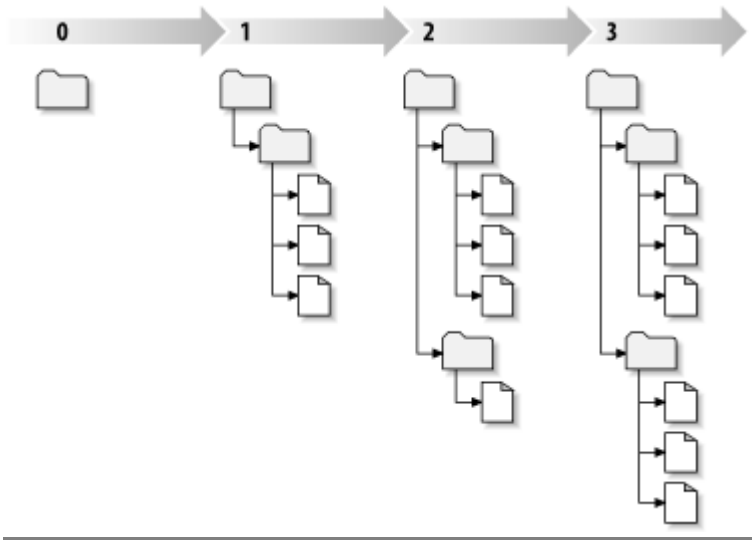
一个 **`svn commit`** 操作可以作为一个原子事务操作发布任意数量文件和目录的修改，在你的工作拷贝里，你可以改变文件内容、删除、改名以及拷贝文件和目录，然后作为一个原子事务一起提交。

“原子事务”的意思是：要么所有的改变发生，要么都不发生，Subversion 努力保持原子性以应对程序错误、系统错误、网络问题和其他用户行为。

每当版本库接受了一个提交，文件系统进入了一个新的状态，叫做一次修订（*revision*），每一个修订版本被赋予一个独一无二的自然数，一个比一个大，初始修订号是 0，只创建了一个空目录，没有任何内容。

图 1.7 “版本库”可以更形象的描述版本库，想象有一组修订号，从 0 开始，从左到右，每一个修订号有一个目录树挂在它下面，每一个树好像是一次提交后的版本库“快照”。

图 1.7. 版本库



全局版本号

不像其他版本控制系统，Subversion 的修订号是针对整个*目录树*的，而不是单个文件。每一个修订号代表了一次提交后版本库整个目录树的特定状态，另一种理解是修订号 N 代表版本库已经经过了 N 次提交。当 Subversion 用户讨论“*foo.c* 的修订号 5”时，他们的实际意思是“在修订号 5 时的 *foo.c*”。需要注意的是，一个文件的修订版本 N 和 M *并不必*有所不同。许多其它版本控制系统使用每文件一个修订号的策略，所以会感觉这些概念有点不一样。（以前的 CVS 用户可能希望察看附录 B, *CVS 用户的 Subversion 指南*来得到更多细节。）

需要特别注意的是，工作拷贝并不一定对应版本库中的单个修订版本，他们可能包含多个修订版本的文件。举个例子，你从版本库检出一个工作拷贝，最近的修订号是 4：

```
calc/Makefile:4
    integer.c:4
    button.c:4
```

此刻，工作目录与版本库的修订版本 4 完全对应，然而，你修改了 *button.c* 并且提交之后，假设没有别的提交出现，你的提交会在版本库建立修订版本 5，你的工作拷贝会是这个样子的：

```
calc/Makefile:4
    integer.c:4
```

```
    button.c:5
```

假设此刻，Sally 提交了对 *integer.c* 的修改，建立修订版本 6，如果你使用 **svn update** 来更新你的工作拷贝，你会看到：

```
calc/Makefile:6
    integer.c:6
    button.c:6
```

Sally 对 *integer.c* 的改变会出现在你的工作拷贝，你对 *button.c* 的改变还在，在这个例子里，*Makefile* 在 4、5、6 修订版本都是一样的，但是 Subversion 会把他的 *Makefile* 的修订号设为 6 来表明它是最新的，所以你在工作拷贝顶级目录作一次干净的更新，会使得所有内容对应版本库的同一修订版本。

工作拷贝怎样跟踪版本库

对于工作拷贝的每一个文件，Subversion 在管理区域*.svn/*记录两项关键的信息：

- 工作文件所作为基准的修订版本（叫做文件的*工作修订版本*）和
- 一个本地拷贝最后更新的时间戳。

给定这些信息，通过与版本库通讯，Subversion 可以告诉我们工作文件是处于如下四种状态的那一种：

未修改且是当前的

文件在工作目录里没有修改，在工作修订版本之后没有修改提交到版本库。**svn commit** 操作不做任何事情，**svn update** 不做任何事情。

本地已修改且是当前的

在工作目录已经修改，从基本修订版本之后没有修改提交到版本库。本地修改没有提交，因此 **svn commit** 会成功提交，**svn update** 不做任何事情。

未修改且不是当前的了

这个文件在工作目录没有修改，但在版本库中已经修改了。这个文件最终将更新到最新版本，成为当时的公共修订版本。**svn commit** 不做任何事情，**svn update** 将会取得最新的版本到工作拷贝。

本地已修改且不是最新的

这个文件在工作目录和版本库都得到修改。一个 **svn commit** 将会失败，这个文件必须首先更新，**svn update** 命令会合并公共和本地修改，如果 Subversion 不可以自动完成，将会让用户解决冲突。

这看起来需要记录很多事情，但是 **svn status** 命令可以告诉你工作拷贝中文件的状态，关于此命令更多的信息，请看“查看你的修改概况”一节。

混合修订版本的工作拷贝

作为一个普遍原理，Subversion 努力做到尽可能的灵活，一个特殊的灵活特性就是让工作拷贝包含不同工作修订版本的文件和目录，不幸的是，这个灵活性会让许多新用户感到迷惑。如果上一个混合修订版本的例子让你感到困惑，这里是一个为何有这种特性和如何利用这个特性的基础介绍。

更新和提交是分开的

Subversion 有一个基本原则就是一个“推”动作不会导致“拉”，反之亦然，因为你准备好了提交你的修改并不意味着你已经准备好了从其他人那里接受修改。如果你的新的修改还在进行，**svn update** 将会优雅的合并版本库的修改到你的工作拷贝，而不会强迫将修改发布。

这个规则的主要副作用就是工作拷贝需要记录额外的信息来追踪混合修订版本，并且也需要能容忍这种混合，当目录本身也是版本化的时候情况更加复杂。

举个例子，假定你有一个工作拷贝，修订版本号是 10。你修改了 *foo.html*，然后执行 **svn commit**，在版本库里创建了修订版本 15。当成功提交之后，许多用户希望工作拷贝完全变成修订版本 15，但是事实并非如

此。修订版本从 10 到 15 会发生任何修改，可是客户端在运行 **svn update** 之前不知道版本库发生了怎样的改变，**svn commit** 不会拖出任何新的修改。另一方面，如果 **svn commit** 会自动下载最新的修改，可以使得整个工作拷贝成为修订版本 15—但是，那样我们会打破“push”和“pull”完全分开的原则。因此，Subversion 客户端最安全的方式是标记一个文件—*foo.html*—为修订版本 15，工作拷贝余下的部分还是修订版本 10。只有运行 **svn update** 才会下载最新的修改，整个工作拷贝被标记为修订版本 15。

混合修订版本很常见

事实上，*每次*运行 **svn commit**，你的工作拷贝都会进入混合多个修订版本的状态，刚刚提交的文件会比其他文件有更高的修订版本号。经过多次提交（之间没有更新），你的工作拷贝会完全是混合的修订版本。即使只有你一个人使用版本库，你依然会见到这个现象。为了检验混合工作修订版本，可以使用 **svn status --verbose** 命令（详细信息见“查看你的修改概况”一节）。

通常，新用户对于工作拷贝的混合修订版本一无所知，这会让人糊涂，因为许多客户端命令对于所检验条目的修订版本很敏感。例如 **svn log** 命令显示一个文件或目录的历史修改信息（见“产生历史修改列表”一节），当用户对一个工作拷贝对象调用这个命令，他们希望看到这个对象的整个历史信息。但是如果这个对象的修订版本已经相当老了（通常

因为很长时间没有运行 **svn update**），此时会显示比这个对象*更老的*历史。

混合版本很有用

如果你的项目十分复杂，有时候你会发现强制工作拷贝的一部分“回溯”到过去非常有用（或者更新到过去的某个修订版本），你将在第 2 章 *基本使用*学习到如何这样做。或许你很希望测试某一子目录下某一子模块的早期版本，又或是要测试一个 bug 什么时候发生，这是版本控制系统像“时间机器”的一个方面—这个特性允许工作拷贝的任何一个部分在历史中前进或后退。

混合版本有限制

无论你如何在工作拷贝中利用混合修订版本，这种灵活性还是有限制的。

首先，你不可以提交一个不是完全最新的文件或目录，如果有个新的版本存在于版本库，你的删除操作会被拒绝，这防止你不小心破坏你没有见到的东西。

第二，如果目录已经不是最新的了，你不能提交一个目录的元数据更改。你将会在第 3 章 *高级主题*学习附加“属性”，一个目录的工作修订版本定义了许多条目和属性，因而对一个过期的版本提交属性会破坏一些你没有见到的属性。

总结

我们在这一章里学习了许多 Subversion 的基本概念：

- 我们介绍了中央版本库、客户工作拷贝和版本修订树的概念。
- 我们介绍了两个协作者如何使用 Subversion 通过“拷贝-修改-合并”模型发布和获得对方的修改。
- 我们讨论了一些 Subversion 跟踪和管理工作拷贝信息的方式。

现在，你一定对 Subversion 在多数情形下的工作方式有了很好的认识，有了这些知识的武装，你一定已经准备好跳到下一章去了，一个关于 Subversion 命令与特性的详细教程。

基本使用

目录

现在，我们将要深入到 Subversion 的使用细节当中，完成本章时，你将学会所有 Subversion 日常使用的命令，你将从把数据导入到 Subversion 开始，接着是初始化的检出（check out），然后是做出修改并检查，你也将会学到如何在工作拷贝中获取别人的修改，检查他们，并解决所有可能发生的冲突。

这一章并不是 Subversion 命令的完全列表—而是你将会遇到的最常用任务的介绍，这一章假定你已经读过并且理解了第 1 章 *基本概念*，而且熟悉 Subversion 的模型，如果想查看所有命令的参考，见第 9 章 *Subversion 完全参考*。

求助！

在继续阅读之前，需要知道 Subversion 使用中最重要的命令：**svn help**，Subversion 命令行工具是一个自文档的工具—在任何时候你可以运行 **svn help***SUBCOMMAND* 来查看子命令的语法、参数以及行为方式。

```
$ svn help import
import: Commit an unversioned file or tree into the repository.
usage: import [PATH] URL

  Recursively commit a copy of PATH to URL.
  If PATH is omitted '.' is assumed.
  Parent directories are created as necessary in the repository.
  If PATH is a directory, the contents of the directory are added
  directly under URL.

Valid options:
  -q [--quiet]            : print as little as possible
  -N [--non-recursive]    : operate on single directory only
…
```

导入数据到你的版本库

有两种方法可以将新文件引入 Subversion 版本库：**svn import** 和 **svn add**，我们将在本章讨论 **svn import**，而会在回顾 Subversion 的典型一天时讨论 **svn add**。

svn import

svn import 是将未版本化文件导入版本库的最快方法，会根据需要创建中介目录。**svn import** 不需要一个工作拷贝，你的文件会直接提交到版本库，这通常用在你希望将一组文件加入到Subversion版本库时，例如：

```
$ svnadmin create /usr/local/svn/newrepos

$ svn import mytree
file:///usr/local/svn/newrepos/some/project \

            -m "Initial import"

Adding         mytree/foo.c

Adding         mytree/bar.c

Adding         mytree/subdir

Adding         mytree/subdir/quux.h



Committed revision 1.
```

在上一个例子里，将会拷贝目录 *mytree* 到版本库的 *some/project* 下：

```
$ svn list file:///usr/local/svn/newrepos/some/project

bar.c

foo.c

subdir/
```

注意，在导入之后，原来的目录树并*没有*转化成工作拷贝，为了开始工作，你还是需要运行 **svn checkout** 导出一个工作拷贝。

推荐的版本库布局

尽管 Subversion 的灵活性允许你自由布局版本库，但我们有一套推荐的方式，创建一个 *trunk* 目录来保存开发的“主线”，一个 *branches* 目录存放分支拷贝，*tags* 目录保存标签拷贝，例如：

```
$ svn list file:///usr/local/svn/repos
/trunk
/branches
/tags
```

你将会在第 4 章 *分支与合并*看到标签和分支的详细内容，关于设置多个项目的信息，可以看“版本库布局”一节和“规划你的版本库结构”一节中关于“项目根目录”的内容。

初始化检出

大多数时候，你会使用 *checkout* 从版本库取出一个新拷贝开始使用 Subversion，这样会在本机创建一个项目的“本地拷贝”，这个拷贝包括了命令行指定版本库中的 HEAD（最新的）版本：

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A    trunk/Makefile.in
A    trunk/ac-helpers
A    trunk/ac-helpers/install.sh
A    trunk/ac-helpers/install-sh
A    trunk/build.conf
…
```

```
Checked out revision 8810.
```

名称中有什么？

Subversion 努力控制版本控制下数据的类型，文件的内容和属性值都是按照二进制数据存储和传递，并且“文件内容类型”一节给 Subversion 提示以说明对于特定文件“文本化的”操作是没有意义的，也有一些地方，Subversion 对存放的信息有限制。

Subversion 内部使用二进制处理数据—例如，属性名称，路径名和日志信息—UTF-8 编码的 Unicode，这并不意味着与 Subversion 的交互必须完全使用 UTF-8。作为一个惯例，Subversion 的客户端能够透明的转化 UTF-8 和你所使用系统的编码，前提是可以进行有意义的转换（当然是大多数目前常见的编码）。

此外，路径名称在 WebDAV 交换中会作为 XML 属性值，就像 Subversion 的管理文件。这意味着路径名称只能包含合法的 XML（1.0）字符，Subversion 也会禁止路径名称中出现 TAB、CR 或 LF 字符，所以它们才不会在区别程序或如 svn log 和 svn status 的输出命令中断掉。

虽然看起来要记住很多事情，但在实践中这些限制很少会成为问题。只要你的本地设置兼容 UTF-8，也不在路径名称中使用控制字符，与 Subversion 的通讯就不会有问题。命令行客户端会添加一些额外的帮助字节—自动将你输入的 URL 路径字符转化为“合法正确的”内部用版本。

尽管上面的例子取出了 trunk 目录，你也完全可以通过输入特定 URL 取出任意深度的子目录：

```
$ svn checkout \

  http://svn.collab.net/repos/svn/trunk/subversion/tests/cmdline/

A    cmdline/revert_tests.py

A    cmdline/diff_tests.py

A    cmdline/autoprop_tests.py

A    cmdline/xmltests

A    cmdline/xmltests/svn-test.sh

…

Checked out revision 8810.
```

因为 Subversion 使用“拷贝-修改-合并”模型而不是“锁定-修改-解锁”模型（见“版本模型”一节），你可以在工作拷贝中开始修改的目录和文件，你的工作拷贝和你的系统中的其它文件和目录完全一样，你可以编辑并改变它，移动它，也可以完全的删掉它，把它忘了。

因为你的工作拷贝“同你系统上的文件和目录没有任何区别”，你可以随意修改文件，但是你必须告诉 Subversion 你做的*其他任何事*。例如，你希望拷贝或移动工作拷贝的一个文件，你应该使用 **svn copy** 或者 **svn move** 而不要使用操作系统的拷贝移动命令，我们会在本章后面详细介绍。

除非你准备好了提交一个新文件或目录，或改变了已存在的，否则没有必要通知 Subversion 你做了什么。

.svn 目录包含什么？

工作拷贝中的任何一个目录包括一个名为*.svn* 管理区域，通常列表操作不显示这个目录，但它仍然是一个非常重要的目录，无论你做什么？不要删除或是更改这个管理区域的任何东西，Subversion 使用它来管理工作拷贝。

如果你不小心删除了子目录*.svn*，最简单的解决办法是删除包含的目录（普通的文件系统删除，而不是 **`svn delete`**），然后在父目录运行 **`svn update`**，Subversion 客户端会重新下载你删除的目录，并包含新的*.svn*。

因为你可以使用版本库的 URL 作为唯一参数取出一个工作拷贝，你也可以在版本库 URL 之后指定一个目录，这样会将你的工作目录放到你的新目录，举个例子：

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subv
A    subv/Makefile.in
A    subv/ac-helpers
A    subv/ac-helpers/install.sh
A    subv/ac-helpers/install-sh
A    subv/build.conf
…
```

```
Checked out revision 8810.
```

这样将把你的工作拷贝放到 subv 而不是和前面那样放到 trunk，如果 subv 不存在，将会自动创建。

禁用密码缓存

当你执行的 Subversion 命令需要认证时，缺省情况下 Subversion 会在磁盘缓存认证信息，这样做出于便利，在接下来的操作中你就可以不必输入密码，但如果你很在乎密码缓存，[3]你可以永久关闭缓存或每次执行命令时说明。

在某次命令关闭密码缓存可以在命令中使用--no-auth-cache 选项，如果希望永久关闭缓存，可以在本机的 Subversion 配置文件中添加 store-passwords = no 这一行，详情请见“客户端凭证缓存”一节。

用其它身份认证

因为 Subversion 认证缓存是缺省设置（包含用户名和密码），用来记住上一次修改工作拷贝的人非常方便。但是有时候会不好用—特别是如果你使用的是共享工作拷贝，在这种情况下，你只需要为命令行传递 --username 选项，Subversion 就会尝试使用该用户认证，如果需要也提示你输入密码。

基本的工作周期

Subversion 有许多特性、选项和华而不实的高级功能，但日常的工作中你只使用其中的一小部分，在这一节里，我们会介绍许多你在日常工作中常用的命令。

典型的工作周期是这样的：

- 更新你的工作拷贝
  - **svn update**
- 做出修改
  - **svn add**
  - **svn delete**
  - **svn copy**
  - **svn move**
- 检验修改
  - **svn status**
  - **svn diff**
- 可能会取消一些修改
  - **svn revert**
- 解决冲突（合并别人的修改）
  - **svn update**
  - **svn resolved**
- 提交你的修改
  - **svn commit**

更新你的工作拷贝

当你在一个团队的项目里工作时，你希望更新你的工作拷贝得到所有其他人这段时间作出的修改，使用 **svn update** 让你的工作拷贝与最新的版本同步。

```
$ svn update
U  foo.c
U  bar.c
Updated to revision 2.
```

这种情况下，其他人在你上次更新之后提交了对 *foo.c* 和 *bar.c* 的修改，因此 Subversion 更新你的工作拷贝来引入这些更改。

当服务器通过 **svn update** 将修改传递到你的工作拷贝时，每一个项目之前会有一个字母，来让你知道 Subversion 为保持最新对你的工作拷贝作了哪些工作。关于这些字母的详细含义，可以看 svn update。

修改你的工作拷贝

现在你可以开始工作并且修改你的工作拷贝了，你很容易决定作出一个修改（或者是一组），像写一个新的特性，修正一个错误等等。这时可以使用的 Subversion 命令包括 **svn add**、**svn delete**、**svn copy** 和 **svn move**。如果你只是修改版本库中已经存在的文件，在你提交之前，不必使用上面的任何一个命令。

你可以对工作拷贝做出两种修改：文件修改和目录树修改。你不需要告诉 Subversion 你希望修改一个文件，只需要用你的编辑器、字处理器、图形程序或任何工具做出修改，Subversion 自动监测到文件的更改，此外，二进制文件的处理方式和文本文件一样—也有同样的效率。对于目录树更改，你可以告诉 Subversion 将文件和目录预定的删除、添加、

拷贝或移动标记，这些动作会在工作拷贝上立刻发生效果，但只有提交后才会在版本库里生效。

下面是 Subversion 用来修改目录树结构的五个子命令。

版本控制符号连接

在非 Windows 平台，Subversion 可以将特殊类型*符号链接*（或是“symlink”）版本化，一个符号链接是对文件系统中其他对象的透明引用，可以通过对符合链接操作实现对引用对象的读写操作。

当符号链提交到 Subversion 版本库，Subversion 会记住这个文件实际上是一个符号链，也会知道这个符号链指向的“对象”。当这个符号链检出到另一个支持符号链的操作系统上时，Subversion 会重新构建文件系统级的符号链接。当然这样不会影响在 Windows 这类不支持符号链的系统上，在此类系统上，Subversion 只会创建一个包含指向对象路径的文本文件，因为这个文件不能在 Windows 系统上作为符号链使用，所以它也会防止 Windows 用户作其他 Subversion 相关的操作。

`svn add foo`

预定将文件、目录或者符号链 *foo* 添加到版本库，当你下次提交后，*foo* 会成为其父目录的一个子对象。注意，如果 *foo* 是目录，所有 foo 中的内容也会预定添加进去，如果你只想添加 *foo* 本身，请使用`--non-recursive (-N)`参数。

`svn delete foo`

预定将文件、目录或者符号链 *foo* 从版本库中删除，如果 foo 是文件，它马上从工作拷贝中删除，如果是目录，不会被删除，但是 Subversion 准备好删除了，当你提交你的修改，*foo* 就会在你的工作拷贝和版本库中被删除。[4]

svn copy foo bar

建立一个新的项目 *bar* 作为 *foo* 的复制品，会自动预定将 *bar* 添加，当在下次提交时会将 *bar* 添加到版本库，这种拷贝历史会记录下来（按照来自 *foo* 的方式记录），**svn copy** 并不建立中介目录。

svn move foo bar

这个命令与与运行 **svn copy foo bar;svn delete foo** 完全相同，*bar* 作为 *foo* 的拷贝准备添加，*foo* 已经预定被删除，**svn move** 不建立中介的目录。

svn mkdir blort

这个命令同运行 **mkdir blort; svn add blort** 相同，也就是创建一个叫做 *blort* 的文件，并且预定添加到版本库。

不通过工作拷贝修改版本库

有一些情况下会立刻提交目录树的修改到版本库，这只发生在子命令直接操作 URL，而不是工作拷贝路径时。以特定的方式使用 **svn mkdir**、**svn copy**、**svn move** 和 **svn delete** 可以针对 URL 操作（并且不要忘记 **svn import** 只针对 URL 操作）。

指定URL的操作方式有一些区别，因为在使用工作拷贝的运作方式时，工作拷贝成为一个“集结地”，可以在提交之前整理组织所要做的修改，直接对 URL 操作就没有这种奢侈，所以当你直接操作 URL 的时候，所有以上的动作代表一个立即的提交。

检查你的修改

当你完成修改，你需要提交他们到版本库，但是在此之前，检查一下做过什么修改是个好主意，通过提交前的检查，你可以整理一份精确的日志信息，你也可以发现你不小心修改的文件，给了你一次恢复修改的机会。此外，这是一个审查和仔细察看修改的好机会，你可通过命令 **`svn status`** 浏览所做的修改，通过 **`svn diff`** 检查修改的详细信息。

看！没有网络！

这三个命令（**`svn status`**、**`svn diff`** 和 **`svn revert`**）都可以在没有网络的情况下工作（假定你的版本库是通过网络而不是本地访问的），这让你在没有网络连接时的管理修改过程更加容易，像在飞机上旅行，乘坐火车往返或是在海滩上奋力工作时。[5]

Subversion 通过在*.svn* 管理区域使用原始的版本缓存来做到这一点，这使得报告和恢复本地修改而*不必访问网络*，这个缓存（叫做“text-base”）也允许 Subversion 可以根据原始版本生成一个压缩的增量（“区别”）提交—即使你有个非常快的网络，有这样一个缓存有极大的好处，只向服务器提交修改的部分而不是整个文件。

Subversion 已经被优化来帮助你完成这个任务，可以在不与版本库通讯的情况下做许多事情，详细来说，对于每一个文件，你的的工作拷贝在.*svn* 包含了一个“原始的”拷贝，所以 Subversion 可以快速的告诉你那些文件修改了，甚至允许你在不与版本库通讯的情况下恢复修改。

查看你的修改概况

为了浏览修改的内容，你会使用这个 **svn status** 命令，在所有 Subversion 命令里，**svn status** 可能会是你用的最多的命令。

CVS 用户：控制另类的更新！

你也许使用 **cvs update** 来看你做了哪些修改，**svn status** 会给你所有你做的改变—而不需要访问版本库，并且不会在不知情的情况下与其他用户作的更改比较。

在 Subversion，**update** 只是做这件事—将工作拷贝更新到版本库的最新版本，你可以消除使用 **update** 察看本地修改的习惯。

如果你在工作拷贝的顶级目录运行不带参数的 **svn status** 命令，它会检测你做的所有的文件或目录的修改，以下的例子是来展示 **svn status** 可能返回的状态码(注意,#之后的不是 **svn status** 打印的)。

```
A       stuff/loot/bloo.h   # file is scheduled for addition
C       stuff/loot/lump.c   # file has textual conflicts from an
update
D       stuff/fish.c        # file is scheduled for deletion
```

```
M       bar.c             # the content in bar.c has local
modifications
```

在这种格式下，**svn status** 打印 6 列字符，紧跟一些空格，接着是文件或者目录名。第一列告诉一个文件或目录的状态或它的内容，返回代码如下：

A item

预定加入到版本库的文件、目录或符号链的 *item*。

C item

文件 *item* 发生冲突，在从服务器更新时与本地版本发生交迭，在你提交到版本库前，必须手工的解决冲突。

D item

文件、目录或是符号链 *item* 预定从版本库中删除。

M item

文件 *item* 的内容被修改了。

如果你传递一个路径给 **svn status**，它只给你这个项目的信息：

```
$ svn status stuff/fish.c

D      stuff/fish.c
```

svn status 也有一个--verbose (-v)选项，它可以显示工作拷贝中的*所有*项目，即使没有改变过的：

```
$ svn status -v
M             44        23    sally     README
              44        30    sally     INSTALL
M             44        20    harry     bar.c
              44        18    ira       stuff
              44        35    harry     stuff/trout.c
D             44        19    ira       stuff/fish.c
              44        21    sally     stuff/things
A              0         ?    ?         stuff/things/bloo.h
              44        36    harry     stuff/things/gloo.c
```

这是 **svn status** 的“加长形式”，第一列保持相同，第二列显示一个工作版本号，第三和第四列显示最后一次修改的版本号和修改人（这些列不会与我们刚才提到的字符混淆）。

上面所有的 **svn status** 调用并没有联系版本库，只是与*.svn* 中的原始数据进行比较的结果，最后，是--show-updates (-u)选项，它将会联系版本库为已经过时的数据添加新信息：

```
$ svn status -u -v
M      *      44        23    sally     README
M             44        20    harry     bar.c
       *      44        35    harry     stuff/trout.c
D             44        19    ira       stuff/fish.c
A              0         ?    ?         stuff/things/bloo.h
```

```
Status against revision:   46
```

注意这两个星号：如果你现在执行 **svn update**，你的 *README* 和 *trout.c* 会被更新，这告诉你许多有用的信息—你可以在提交之前，需要使用更新操作得到文件 *README* 的更新，或者说文件已经过时，版本库会拒绝了你的提交。（后面还有更多关于此主题）。

关于文件和目录，**svn status** 可以比我们的展示显示更多的内容，完整的描述可以看 svn status。

检查你的本地修改的详情

另一种检查修改的方式是 **svn diff** 命令，你可以通过不带参数的 **svn diff** *精确的*找出你所做的修改，这会输出*统一区别格式*的区别信息：

```
$ svn diff
Index: bar.c
===================================================================
--- bar.c   (revision 3)
+++ bar.c   (working copy)
@@ -1,7 +1,12 @@
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>
```

```
 int main(void) {
-  printf("Sixty-four slices of American Cheese...\n");
+  printf("Sixty-five slices of American Cheese...\n");
 return 0;
 }

Index: README
===================================================================
--- README  (revision 3)
+++ README  (working copy)
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: stuff/fish.c
===================================================================
--- stuff/fish.c   (revision 1)
+++ stuff/fish.c   (working copy)
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: stuff/things/bloo.h
===================================================================
```

```
--- stuff/things/bloo.h    (revision 8)
+++ stuff/things/bloo.h    (working copy)
+Here is a new file to describe
+things about bloo.
```

svn diff 命令通过比较你的文件和.*svn* 的“原始”文件来输出信息，预定要增加的文件会显示所有增加的文本，要删除的文件会显示所有要删除的文本。

输出的格式为统一区别格式（unified diff format），删除的行前面加一个-，添加的行前面有一个+，**svn diff** 命令也打印文件名和**打补丁**需要的信息，所以你可以通过重定向一个区别文件来生成“补丁”：

```
$ svn diff > patchfile
```

举个例子，你可以把补丁文件发送邮件到其他开发者，在提交之前审核和测试。

Subversion 使用内置区别引擎，缺省情况下输出为统一区别格式。如果你期望不同的输出格式，你可以使用--diff-cmd 指定外置的区别程序，并且通过--extensions 传递其他参数，举个例子，察看本地文件 *foo.c* 的区别，同时忽略大小写差异，你可以运行 **svn diff --diff-cmd /usr/bin/diff --extensions '-bc' foo.c**。

取消本地修改

假定我们在看 **svn diff** 的输出，你发现对某个文件的所有修改都是错误的，或许你根本不应该修改这个文件，或者是从开头重新修改会更加容易。

这是使用 **svn revert** 的好机会：

```
$ svn revert README

Reverted 'README'
```

Subversion 把文件恢复到未修改的状态，叫做*.svn* 目录的“原始”拷贝，应该知道 **svn revert** 可以恢复任何预定要做的操作，举个例子，你不再想添加一个文件：

```
$ svn status foo

?      foo


$ svn add foo

A         foo


$ svn revert foo

Reverted 'foo'


$ svn status foo

?      foo
```

svn revert*ITEM*的效果与删除 *ITEM*然后执行 **svn update -r BASE***ITEM*完全一样，但是，如果你使用 **svn revert** 它不必通知版本库就可以恢复文件。

或许你不小心删除了一个文件：

```
$ svn status README

      README



$ svn delete README

D         README



$ svn revert README

Reverted 'README'



$ svn status README

      README
```

解决冲突（合并别人的修改）

我们可以使用 **svn status -u** 来预测冲突，当你运行 **svn update** 一些有趣的事情发生了：

```
$ svn update

U  INSTALL

G  README

C  bar.c
```

```
Updated to revision 46.
```

U 和 G 没必要关心，文件干净的接受了版本库的变化，文件标示为 U 表明本地没有修改，文件已经根据版本库更新。G 标示合并，标示本地已经修改过，与版本库没有重迭的地方，已经合并。

但是 C 表示冲突，说明服务器上的改动同你的改动冲突了，你需要自己手工去解决。

当冲突发生了，有三件事可以帮助你注意到这种情况和解决问题：

- Subversion 在更新时打印 C 标记，并且标记这个文件已冲突。
- 如果 Subversion 认为这个文件是可合并的，它会置入*冲突标记*—特殊的横线分开冲突的“两面”—在文件里可视化的描述重叠的部分（Subversion 使用 svn:mime-type 属性来决定一个文件是否可以使用上下文的，以行为基础的合并，更多信息可以看“文件内容类型”一节。）
- 对于每一个冲突的文件，Subversion 放置三个额外的未版本化文件到你的工作拷贝：

  filename.mine

  你更新前的文件，没有冲突标志，只是你最新更改的内容。（如果 Subversion 认为这个文件不可以合并，*.mine* 文件不会创建，因为它和工作文件相同。）

  filename.rOLDREV

这是你的做更新操作以前的 BASE 版本文件，就是你在上次更新之后未作更改的版本。

filename.rNEWREV

这是你的 Subversion 客户端从服务器刚刚收到的版本，这个文件对应版本库的 HEAD 版本。

这里 OLDREV 是你的*.svn* 目录中的修订版本号，NEWREV 是版本库中 HEAD 的版本号。

举一个例子，Sally 修改了 *sandwich.txt*，Harry 刚刚改变了他的本地拷贝中的这个文件并且提交到服务器，Sally 在提交之前更新它的工作拷贝得到了冲突：

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls -1
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

在这种情况下，Subversion *不*会允许你提交 *sandwich.txt*，直到你的三个临时文件被删掉。

```
$ svn commit -m "Add a few more things"

svn: Commit failed (details follow):

svn: Aborting commit: '/home/sally/svn-work/sandwich.txt'
remains in conflict
```

如果你遇到冲突，三件事你可以选择：

- “手动”合并冲突文本（检查和修改文件中的冲突标志）。
- 用某一个临时文件覆盖你的工作文件。
- 运行 **svn revert <filename>**来放弃所有的本地修改。

一旦你解决了冲突，你需要通过命令 **svn resolved** 让 Subversion 知道，这样就会删除三个临时文件，Subversion 就不会认为这个文件是在冲突状态了。[6]

```
$ svn resolved sandwich.txt

Resolved conflicted state of 'sandwich.txt'
```

手工合并冲突

第一次尝试解决冲突让人感觉很害怕，但经过一点训练，它简单的像是骑着车子下坡。

这里一个简单的例子，由于不良的交流，你和同事 Sally，同时编辑了 *sandwich.txt*。Sally 提交了修改，当你准备更新你的工作拷贝，冲突发生了，我们不得不去修改 *sandwich.txt* 来解决这个问题。首先，看一下这个文件：

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

小于号、等于号和大于号串是冲突标记，并不是冲突的数据，你一定要确定这些内容在下次提交之前得到删除，前两组标志中间的内容是你在冲突区所做的修改：

```
<<<<<<< .mine
Salami
Mortadella
Prosciutto
```

```
=======
```

后两组之间的是 Sally 提交的修改冲突：

```
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
```

通常你并不希望只是删除冲突标志和 Sally 的修改—当她收到三明治时，会非常的吃惊。所以你应该走到她的办公室或是拿起电话告诉 Sally，你没办法从从意大利熟食店得到想要的泡菜。[7]一旦你们确认了提交内容后，修改文件并且删除冲突标志。

```
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
Salami
Mortadella
Prosciutto
Creole Mustard
Bottom piece of bread
```

现在运行 **`svn resolved`**，你已经准备好提交了：

```
$ svn resolved sandwich.txt

$ svn commit -m "Go ahead and use my sandwich, discarding Sally's
edits."
```

现在我们准备好提交修改了，注意 **svn resolved** 不像我们本章学过的其他命令一样需要参数，在任何你认为解决了冲突的时候，只需要小心运行 **svn resolved**，一一旦删除了临时文件，Subversion 会让你提交这文件，即使文件中还存在冲突标记。

记住，如果你修改冲突时感到混乱，你可以参考 subversion 生成的三个文件—包括你未作更新的文件。你也可以使用三方交互合并工具检验这三个文件。

复制文件到你的工作文件

如果你只是希望取消你的修改，你可以仅仅拷贝 Subversion 为你生成的文件替换你的工作拷贝：

```
$ svn update

C  sandwich.txt

Updated to revision 2.

$ ls sandwich.*

sandwich.txt  sandwich.txt.mine  sandwich.txt.r2
sandwich.txt.r1

$ cp sandwich.txt.r2 sandwich.txt

$ svn resolved sandwich.txt
```

脚注：使用 svn revert

如果你得到冲突，经过检查你决定取消自己的修改并且重新编辑，你可以恢复你的修改：

```
$ svn revert sandwich.txt

Reverted 'sandwich.txt'

$ ls sandwich.*

sandwich.txt
```

注意，当你恢复一个冲突的文件时，不需要再运行 **svn resolved**。

提交你的修改

最后！你的修改结束了，你合并了服务器上所有的修改，你准备好提交修改到版本库。

svn commit 命令发送所有的修改到版本库，当你提交修改时，你需要提供一些描述修改的*日志信息*，你的信息会附到这个修订版本上，如果信息很简短，你可以在命令行中使用--message（或-m）选项：

```
$ svn commit -m "Corrected number of cheese slices."

Sending        sandwich.txt

Transmitting file data .

Committed revision 3.
```

然而，如果你把写日志信息当作工作的一部分，你也许会希望告诉 Subversion 通过一个文件名得到日志信息，使用--file 选项：

```
$ svn commit -F logmsg

Sending        sandwich.txt

Transmitting file data .

Committed revision 4.
```

如果你没有指定--message 或者--file 选项，Subversion 会自动地启动你最喜欢的编辑器（见“配置”一节的 editor-cmd 部分）来编辑日志信息。

如果你使用编辑器撰写日志信息时希望取消提交，你可以直接关掉编辑器，不要保存，如果你已经做过保存，只要简单的删掉所有的文本并再次保存，然后退出。

```
$ svn commit

Waiting for Emacs...Done



Log message unchanged or not specified

a)bort, c)ontinue, e)dit

a

$
```

版本库不知道也不关心你的修改作为一个整体是否有意义，它只检查是否有其他人修改了同一个文件，如果别人*已经*这样做了，你的整个提交会失败，并且提示你一个或多个文件已经过时了：

```
$ svn commit -m "Add another rule"

Sending        rules.txt
```

```
svn: Commit failed (details follow):

svn: Your file or directory 'sandwich.txt' is probably
out-of-date

…
```

（错误信息的精确措辞依赖于网络协议和你使用的服务器，但对于所有的情况，其思想完全一样。）

此刻，你需要运行 **svn update** 来处理所有的合并和冲突，然后再尝试提交。

我们已经覆盖了 Subversion 基本的工作周期，还有许多其它特性可以管理你得版本库和工作拷贝，但是只使用前面介绍的命令你就可以很进行日常工作了，我们还会覆盖更多用的还算频繁的命令。

检验历史

你的版本库就像是一台时间机器，它记录了所有提交的修改，允许你检查文件或目录以及相关元数据的历史。通过一个 Subversion 命令你可以根据时间或修订号取出一个过去的版本（或者恢复现在的工作拷贝），然而，有时候我们只是想*看看*历史而不想*回到*历史。

有许多命令可以为你提供版本库历史：

svn log

展示给你主要信息：每个版本附加在版本上的作者与日期信息和所有路径修改。

svn diff

显示特定修改的行级详细信息。

svn cat

取得在特定版本的某一个文件显示在当前屏幕。

svn list

显示一个目录在某一版本存在的文件。

产生历史修改列表

找出一个文件或目录的历史信息，使用 **svn log** 命令，**svn log** 将会提供你一条记录，包括：谁对文件或目录作了修改、哪个修订版本作了修改、修订版本的日期和时间、还有如果你当时提供了日志信息，也会显示。

```
$ svn log
------------------------------------------------------------------------
r3 | sally | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line

Added include lines and corrected # of cheese slices.
------------------------------------------------------------------------
r2 | harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line

Added main() methods.
```

```
------------------------------------------------------------------------
r1 | sally | Mon, 15 Jul 2002 17:40:08 -0500 | 1 line


Initial import

------------------------------------------------------------------------
```

注意日志信息缺省根据*时间逆序排列*，如果希望察看特定顺序的一段修订版本或者单一版本，使用--revision(-r) 选项：

```
$ svn log -r 5:19    # shows logs 5 through 19 in chronological order


$ svn log -r 19:5    # shows logs 5 through 19 in reverse order


$ svn log -r 8       # shows log for revision 8
```

你也可以检查单个文件或目录的日志历史，举个例子：

```
$ svn log foo.c

…
$ svn log http://foo.com/svn/trunk/code/foo.c

…
```

这样*只会*显示这个工作文件（或者 URL）做过修订的版本的日志信息。

如果你希望得到目录和文件更多的信息，你可以对 **svn log** 命令使用--verbose（-v)开关，因为 Subversion 允许移动和复制文件和目录，所以跟踪路径修改非常重要，在详细模式下，**svn log** 输出中会包括一个路径修改的历史：

```
$ svn log -r 8 -v
------------------------------------------------------------------------
r8 | sally | 2002-07-14 08:15:29 -0500 | 1 line
Changed paths:
M /trunk/code/foo.c
M /trunk/code/bar.h
A /trunk/code/doc/README

Frozzled the sub-space winch.

------------------------------------------------------------------------
```

svn log 也有一个--quiet (-q)选项，会禁止日志信息的主要部分，当与--verbose 结合使用，仅会显示修改的文件名。

为什么 svn log 给我一个空的回应？

当使用 Subversion 一些时间后，许多用户会遇到这种情况：

```
$ svn log -r 2
```

```
------------------------------------------------------------------
$
```

乍一看，好像是一个错误，但是想一下修订版本号是作用在版本库整体之上的，如果你没有提供路径，**svn log** 会使用当前目录作为默认的目标，所以，作为结果，如果你对一个本身和子目录在指定版本到现在没有做过修改的目录运行这个命令，你会得到空的日志。如果你希望察看某个版本做的修改的日志，只需要直接告诉 **svn log** 使用版本库顶级的目录作为参数，例如 **svn log -r 2 http://svn.collab.net/repos/svn**。

检查历史修改的详情

我们已经看过 **svn diff**—使用标准区别文件格式显示区别，它在提交前用来显示本地工作拷贝与版本库的区别。

事实上，**svn diff** 有三种不同的用法：

- 检查本地修改
- 比较工作拷贝与版本库
- 比较版本库与版本库

比较本地修改

像我们看到的，不使用任何参数调用时，**svn diff** 将会比较你的工作文件与缓存在.*svn* 的“原始”拷贝：

```
$ svn diff

Index: rules.txt

===================================================================
=======

--- rules.txt       (revision 3)

+++ rules.txt       (working copy)

@@ -1,4 +1,5 @@

 Be kind to others

 Freedom = Responsibility

 Everything in moderation

-Chew with your mouth open

+Chew with your mouth closed

+Listen when others are speaking

$
```

比较工作拷贝和版本库

如果传递一个--revision（-r）参数，你的工作拷贝会与指定的版本比较。

```
$ svn diff -r 3 rules.txt

Index: rules.txt

===================================================================
=======

--- rules.txt       (revision 3)

+++ rules.txt       (working copy)

@@ -1,4 +1,5 @@
```

```
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

比较版本库与版本库

如果通过--revision （-r）传递两个通过冒号分开的版本号，这两个版本会进行比较。

```
$ svn diff -r 2:3 rules.txt
Index: rules.txt
===================================================================
=======
--- rules.txt       (revision 2)
+++ rules.txt       (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

与前一个修订版本比较更方便的办法是使用--change (-c)：

```
$ svn diff -c 3 rules.txt

Index: rules.txt

===================================================================
=======

--- rules.txt       (revision 2)

+++ rules.txt       (revision 3)

@@ -1,4 +1,4 @@

 Be kind to others

-Freedom = Chocolate Ice Cream

+Freedom = Responsibility

 Everything in moderation

 Chew with your mouth open

$
```

最后，即使你在本机没有工作拷贝，还是可以比较版本库的修订版本，只需要在命令行中输入合适的 URL：

```
$ svn diff -c 5
http://svn.example.com/repos/example/trunk/text/rules.txt

…

$
```

浏览版本库

通过 **svn cat** 和 **svn list**，你可以在未修改工作修订版本的情况下查看文件和目录的内容，实际上，你甚至也不需要有一个工作拷贝。

svn cat

如果你只是希望检查一个过去的版本而不希望察看它们的区别，使用 **svn cat:**

```
$ svn cat -r 2 rules.txt

Be kind to others

Freedom = Chocolate Ice Cream

Everything in moderation

Chew with your mouth open

$
```

你可以重定向输出到一个文件：

```
$ svn cat -r 2 rules.txt > rules.txt.v2

$
```

svn list

svn list 可以在不下载文件到本地目录的情况下来察看目录中的文件：

```
$ svn list http://svn.collab.net/repos/svn

README

branches/

clients/

tags/
```

```
trunk/
```

如果你希望察看详细信息，你可以使用--verbose(-v) 参数：

```
$ svn list -v http://svn.collab.net/repos/svn
  20620 harry          1084 Jul 13  2006 README
  23339 harry               Feb 04 01:40 branches/
  21282 sally               Aug 27 09:41 developer-resources/
  23198 harry               Jan 23 17:17 tags/
  23351 sally               Feb 05 13:26 trunk/
```

这些列告诉你文件和目录最后修改的修订版本、做出修改的用户、如果是文件还会有文件的大小，最后是修改日期和项目的名字。

没有任何参数的 **svn list** 命令缺省使用当前工作拷贝的*版本库 URL*，而*不是*本地工作拷贝的目录。毕竟，如果你希望列出本地目录，你只需要使用 **ls**（或任何合理的非 UNIX 等价物）。

获得旧的版本库快照

除了以上的命令，你可以使用带参数--revision 的 **svn update** 和 **svn checkout** 来使整个工作拷贝“回到过去”[8]：

```
$ svn checkout -r 1729 # Checks out a new working copy at r1729
…
$ svn update -r 1729 # Updates an existing working copy to r1729
…
```

许多 Subversion 新手使用前面的 **svn update** 实例来“回退”修改，但是你不能提交修改，你获得有新修订版本的过时工作拷贝也是没有用的。关于如何“回退”，我们可以看“找回删除的项目”一节。

最后，如果你构建了一个版本，并且希望从 Subversion 打包文件，但是你不希望有讨厌的.svn 目录，这时你可以导出版本库的一部分文件而没有.svn 目录。就像 **svn update** 和 **svn checkout**，你也可以传递 --revision 选项给 **svn export**：

```
$ svn export http://svn.example.com/svn/repos1 # Exports latest revision

…

$ svn export http://svn.example.com/svn/repos1 -r 1729

# Exports revision r1729

…
```

有时你只需要清理

当 Subversion 改变你的工作拷贝（或是*.svn* 中的任何信息），它会尽可能的小心，在修改任何事情之前，它把意图写到日志文件中去，然后执行 log 文件中的命令，并且执行过程中在工作拷贝的相关部分保存一个锁— 防止 Subversion 客户端在变更过程中访问工作拷贝。然后删掉日志文件，这与记帐试的文件系统架构类似。如果 Subversion 的操作中断了（举个例子：进程被杀死了，机器死掉了），日志文件会保存在

硬盘上，通过重新执行日志文件，Subversion 可以完成上一次开始的操作，你的工作拷贝可以回到一致的状态。

这就是 **svn cleanup** 所作的：它查找工作拷贝中的所有遗留的日志文件，删除进程中工作拷贝的锁。如果 Subversion 告诉你工作拷贝中的一部分已经“锁定”了，你就需要运行这个命令了。同样，**svn status** 将会使用 L 标示锁定的项目：

```
$ svn status
  L    somedir
M      somedir/foo.c


$ svn cleanup
$ svn status
M      somedir/foo.c
```

不要将工作拷贝锁与 Subversion 用户使用并发版本控制的“锁定-修改-解锁”模型创建的锁混淆；更多细节见“锁定”的三种含义。

总结

我们已经覆盖了大多数 Subversion 的客户端命令，引人注目的例外是处理分支与合并（见第 4 章 *分支与合并*）以及属性（见“属性”一节）的命令，然而你也许会希望跳到第 9 章 *Subversion 完全参考*来察看所有不同的命令—怎样利用它们使你的工作更容易。

[3] 当然，你不必太过担心—首先你要知道你不会从 Subversion *真的*删除文件，第二，Subversion 密码不是和你的三百万个密码的任何一个相同，对吧？对吧？

[4] 当然没有任何东西是在版本库里被删除了—只是在版本库的 HEAD 里消失了，你可以通过检出（或者更新你的工作拷贝）你做出删除操作的前一个修订版本来找回所有的东西，详细请见“找回删除的项目”一节。

[5] 而且你也没有 WAN 卡，考虑到你得到我们，哈!

[6] 你也可以手工的删除这三个临时文件，但是当 Subversion 会给你做时你会自己去做吗？我们是这样想的。

[7] 如果你向他们询问，他们非常有理由把你带到城外的铁轨上。

[8] 看到了吧？我们说过 Subversion 是一个时间机器。

高级主题

目录

如果你是从头到尾按章节阅读本书，你一定已经具备了使用 Subversion 客户端执行大多数不同的版本控制操作足够的知识，你理解了怎样从

Subversion 版本库取出一个工作拷贝，你已经熟悉了通过 **svn commit** 和 **svn update** 来提交和接收修改，你甚至也经常下意识的使用 **svn status**，无论目的是什么，你已经可以正常使用 Subversion 了。

但是 Subversion 的特性并没有止于“普通的版本控制操作”，它也有一些超越了与版本库传递文件和目录修改以外的功能。

本章重点介绍了一些很重要但不是经常使用的 Subversion 特性，本章假定你熟悉 Subversion 对文件和目录的基本版本操作能力，如果你还没有阅读这些内容，或者是需要一个复习，我们建议你重读第 1 章 *基本概念*和第 2 章 *基本使用*，一旦你已经掌握了基础知识和本章的内容，你会变成 Subversion 的超级用户！

版本清单

就像你在“修订版本”一节见到的，Subversion 的修订版本号码非常直接—就是随提交增大的整数。尽管如此，不会花很长时间你就会忘记每个修订版本的修改，但幸运的是，典型的 Subvesion 工作流程中一般不会要求你提供任意的修订版本号。在需要输入修订版本号时，通常或者是你在一个提交邮件中看到了一个修订版本，或者是在其他 Subversion 命令的输出结果中，或者是任何上下文环境得到某个版本号码的情况下。

但是有时候，你需要精确指定一个时间，而无法记住或者记录了某个版本，这时除了使用修订版本号码，**svn** 允许使用其他形式来指定修订版本—修订版本关键字和修订版本日期。

当用来指定修订版本范围时，不同形式的 Subversion 修订版本可以混合匹配。例如，你可以 *REV1* 是修订版本关键字，*REV2* 是修订版本号，或者是 *REV1* 是日期，而 *REV2* 是修订版本关键字，等等。不同的修订版本指定符是等价的，所以你可以在冒号两边任意使用。

修订版本关键字

Subversion 客户端可以理解一些*修订版本关键字*，这些关键字可以用来代替--revision (r)的数字参数，这会被 Subversion 解释到特定修订版本号：

HEAD

版本库中最新的（或者是“最年轻的”）版本。

BASE

工作拷贝中一个条目的修订版本号，如果这个版本在本地修改了，则“BASE 版本”就是这个条目在本地未修改的版本。

COMMITTED

项目最近修改的修订版本，与 BASE 相同或更早。

PREV

一个项目最后修改版本*之前*的那个版本，技术上可以认为是 COMMITTED -1。

因为可以从描述中得到，关键字 PREV，BASE 和 COMMITTED 只在引用工作拷贝路径时使用，而不能用于版本库 URL，而关键字 HEAD 则可以用于两种路径类型。

下面是一些修订版本关键字的例子：

```
$ svn diff -r PREV:COMMITTED foo.c
# shows the last change committed to foo.c

$ svn log -r HEAD
# shows log message for the latest repository commit

$ svn diff -r HEAD
# compares your working copy (with all of its local changes) to
the
# latest version of that tree in the repository

$ svn diff -r BASE:HEAD foo.c
# compares the unmodified version of foo.c with the latest version
of
# foo.c in the repository

$ svn log -r BASE:HEAD
# shows all commit logs for the current versioned directory since
you
# last updated
```

```
$ svn update -r PREV foo.c

# rewinds the last change on foo.c, decreasing foo.c's working
revision



$ svn diff -r BASE:14 foo.c

# compares the unmodified version of foo.c with the way foo.c
looked

# in revision 14
```

版本日期

在版本控制系统以外，修订版本号码是没有意义的，但是有时候你需要将时间和历史修订版本号关联。为此，--revision (-r)选项接受使用花括号（{和}）包裹的日期输入，Subversion 支持标准 ISO-8601 日期和时间格式，也支持一些其他的。下面是一些例子。（记住使用引号括起所有包含空格的日期。）

```
$ svn checkout -r {2006-02-17}

$ svn checkout -r {15:30}

$ svn checkout -r {15:30:00.200000}

$ svn checkout -r {"2006-02-17 15:30"}

$ svn checkout -r {"2006-02-17 15:30 +0230"}

$ svn checkout -r {2006-02-17T15:30}

$ svn checkout -r {2006-02-17T15:30Z}

$ svn checkout -r {2006-02-17T15:30-04:00}

$ svn checkout -r {20060217T1530}

$ svn checkout -r {20060217T1530Z}
```

```
$ svn checkout -r {20060217T1530-0500}

…
```

当你指定一个日期，Subversion 会在版本库找到接近这个日期的最近版本，并且对这个版本继续操作：

```
$ svn log -r {2006-11-28}

------------------------------------------------------------------------
r12 | ira | 2006-11-27 12:31:51 -0600 (Mon, 27 Nov 2006) | 6 lines

…
```

Subversion 会早一天吗？

如果你只是指定了日期而没有时间（举个例子 2006-11-27），你也许会以为 Subversion 会给你 11-27 号最后的版本，相反，你会得到一个 26 号版本，甚至更早。记住 Subversion 会根据你的日期找到*最新的*版本，如果你给一个日期，而没有给时间，像 2006-11-27，Subversion 会假定时间是 00:00:00，所以在 27 号找不到任何版本。

如果你希望查询包括 27 号，你既可以使用（{"2006-11-27 23:59"}），或是直接使用第二天（{2006-11-28}）。

你可以使用时间段，Subversion 会找到这段时间的所有版本：

```
$ svn log -r {2006-11-20}:{2006-11-29}

…
```

因为一个版本的时间戳是作为一个属性存储的—不是版本化的，而是可以编辑的属性（见“属性”一节）—版本号的时间戳可以被修改，从而建立一个虚假的年代表，也可以被完全删除。Subversion 正确转化修订版本日期到修订版本的能力依赖于修订版本时间戳顺序排列—修订版本越年轻，则时间戳越年轻。如果顺序没有被维护，你会发现使用日期指定修订版本不会返回你期望的数据。

属性

我们已经详细讲述了 Subversion 存储和检索版本库中不同版本的文件和目录的细节，并且用了好几个章节来论述这个工具的基本功能。如果对于版本化的支持到此为止，从版本控制的角度来看 Subversion 已经完整了。

但不仅仅如此。

作为目录和文件版本化的补充，Subversion 提供了对每一个版本化的目录和文件添加、修改和删除版本化的元数据的接口，我们用*属性*来表示这些元数据。我们可以认为它们是一个两列的表，附加到你的工作拷贝的每个条目上，映射属性名到任意的值。一般来说，属性的名称和值可以是你希望的任何值，限制就是名称必须是可读的文本，并且最好的一点是这些属性也是版本化的，就像你的文本文件内容，你可以像提交文本修改一样修改、提交和恢复属性修改，当你更新时也会接收到别人的属性修改—你不必为适应属性改变你的工作流程。

Subversion 自己保留了一组名称以 svn:开头的属性，现在已经有了一些在用的属性，所以在你根据需要创建自定义属性时，需要避免这些前缀开头的名称，否则，Subversion 的新版本可能会采用同名的属性来满足新的特性，而其含义可能会完全不同。

Subversion 的属性也可以在别的地方出现，就像文件和目录可能附加有任意的属性名和值，每个修订版本作为一个整体也可以附加任意的属性，也有同样的限制—可读的文本名称和任何你希望的二进制值，主要的区别是修订版本属性不是版本化的，换句话说，如果你修改，删除一个修订版本属性，在 Subversion 领域内没有办法恢复到以前的值。

Subversion 不关心如何使用属性，但是要求你不要使用 svn:为前缀的属性名，这是 Subversion 自己使用的命名空间，Subversion 使用了版本化的和未版本化的属性。文件和目录上的特定版本化属性都有特别的意义或效果，或者是提供了修订版本的一些信息。一些修订版本属性会在提交时自动附加到修订版本上，包含了修订版本的信息。大多数这些属性会作为普通的主题在后面提及，关于 Subversion 预定义的属性的详细列表可以看“Subversion 属性”一节。

在本小节，我们将会检验这个工具—不仅是对 Subversion 的用户，也对 Subversion 本身—对于属性的支持。你会学到与属性相关的 **svn** 子命令，和属性怎样影响你的普通 Subversion 工作流，希望你会感到 Subversion 的属性可以提高你的版本控制体验。

为什么需要属性？

就像 Subversion 使用属性保存其包含的文件、目录和修订版本的附加信息，你也会发现属性有一些类似的使用，你会发现如果在数据附近有个地方保存自定义元数据会非常有用。

假设你希望设计一个存放许多数码照片的网站，会显示标题和缩略图。现在你的图片会经常修改，所以你希望能够让这个站点尽量自动处理这些事情，这些照片会很大，所以作为网站，你希望为访问者提供相似的缩略图。

现在，你可以利用这些功能使用传统文件。你可以有一个 *image123.jpg* 和一个对应的 *image123-thumbnail.jpg* 在同一个目录里，有时候你希望保持文件名相同，你可以使用不同的目录，如 *thumbnails/image123.jpg*。你可以用一种相似的样式来保存你的标题和时间戳，同原始图像文件分开。每个新图片的添加都会成倍的增加混乱，很快你的目录树会是一团糟。

现在考虑使用 Subversion 文件的属性的方式来管理这个站点，想象我们有一个单独的图像文件 *image123.jpg*，然后这个文件的属性集包括 caption、datestamp 甚至 thumbnail。现在你的工作拷贝目录看起来更容易管理—实际上，它看起来只有图像文件，但是你的自动化脚本知道得更多，它们知道可以用 **`svn`**（更好的选择是使用 Subversion 的语言绑定—见“使用 API”一节）来挖掘更多的站点显示需要的额外信息，而不必去阅读一个索引文件或者是玩一个路径处理的游戏。

自定义修订版本属性也经常被使用，一个常见的用法是一个包含问题跟踪 ID 的属性，可能是因为这个修改修正了这个 ID 的问题。另外一些人用属性来存放更容易记的修订版本名称—记住修订版本 1935 是一个完全测试的版本是很困难的，但是如果在修订版本上设置一个值为 all passing 的 test-results 属性，这就有了一个有用的信息。

可搜索性(或者，为什么*不*使用属性)

对于 Subversion 属性的所有功能—或者更准确的讲，对于属性的所有接口—都有一些主要的应用会削弱他们的应用。设置一个自定义属性后，很容易发现属性完全变成另外一会儿事。

为了定位一个自定义属性通常要线形访问版本库的所有修订版本，向每个修订版本询问，“你们有我找的属性吗？”尝试查找自定义版本化属性也是同样的痛苦，通常需要在整个工作拷贝递归调用 **svn propget**。在你的情况下，可能不会比遍历所有修订版本差。但也在性能和成功可能性里留下了许多悬念，特别是当你需要从版本库的根开始搜索时。

因为这个原因，你会选择—特别是在修订版本属性用例—简单的添加你的元数据到修订版本日志信息，使用一些政策驱动（并且是编程强制的）且可以通过 **svn log** 快速解析的格式。如下的 Subversion 日志信息会很常见：

```
Issue(s): IZ2376, IZ1919

Reviewed by:  sally
```

```
This fixes a nasty segfault in the wort frabbing process
…
```

但是现在依然有一些不幸，Subversion 不支持日志信息模版机制，虽然这样对用户与日志嵌入的修订版本元数据保持一致有很大帮助。

操作属性

svn 命令提供一些方法来添加和修改文件或目录的属性，对于短的，可读的属性，最简单的添加方法是在 **propset** 子命令里指定正确的名称和值。

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/button.c
property 'copyright' set on 'calc/button.c'
$
```

但是我们已经吹捧了 Subversion 提供的属性功能的灵活性，如果你计划使用多行文本，或者是二进制属性值，你可能不会希望通过命令行提供这些值，所以 **propset** 子命令提供的`--file (-F)`选项可以指定包含属性值的文件。

```
$ svn propset license -F /path/to/LICENSE calc/button.c
property 'license' set on 'calc/button.c'
$
```

对于属性名称也有一些限制，属性名必须以一个字符、一个冒号(:)或下划线(_)开始，之后你可以使用数字，横线(-)和句号(.)。[9]

作为 **propset** 命令的补充，**svn** 提供了一个 **propedit** 命令，这个命令使用定制的编辑器程序（见“配置”一节）来添加和修改属性。当你运行这个命令，**svn** 调用你的编辑器程序打开一个临时文件，文件中保存当前的属性值（或者是空文件，如果你正在添加新的属性）。然后你只需要修改为你想要的值，保存临时文件，然后离开编辑器程序。如果 Subversion 发现你已经修改了属性值，就会接受新值，如果你未作任何修改而离开，不会产生属性修改操作：

```
$ svn propedit copyright calc/button.c  ### exit the editor
without changes

No changes to property 'copyright' on 'calc/button.c'

$
```

我们也应该注意到，像其它 **svn** 子命令一样，这些关联的属性可以一次添加到多个路径上，这样就可以通过一个命令修改一组文件的属性。例如，我们可以：

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/*

property 'copyright' set on 'calc/Makefile'

property 'copyright' set on 'calc/button.c'

property 'copyright' set on 'calc/integer.c'

…

$
```

如果不能方便的得到存储的属性值，那么属性的添加和编辑操作也不会很容易，所以 **svn** 提供了两个子命令来显示文件和目录存储的属性名和值。**svn proplist** 命令会列出路径上存在的所有属性名称，一旦你知道了某个节点的属性名称，你可以用 **svn propget** 获取它的值，这个命令获取给定的路径（或者是一组路径）和属性名称，打印这个属性的值到标准输出。

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
  copyright
  license
$ svn propget copyright calc/button.c
(c) 2006 Red-Bean Software
```

还有一个 **proplist** 变种命令会列出所有属性的名称和值，只需要设置--verbose（-v）选项。

```
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2006 Red-Bean Software
  license :
================================================================
====
Copyright (c) 2006 Red-Bean Software.  All rights reserved.


Redistribution and use in source and binary forms, with or without
```

```
modification, are permitted provided that the following
conditions

are met:


1. Redistributions of source code must retain the above copyright

notice, this list of conditions, and the recipe for Fitz's famous

red-beans-and-rice.

…
```

最后一个与属性相关的子命令是 **propdel**，因为 Subversion 允许属性值为空，所有不能用 **propedit** 或者 **propset** 命令删除一个属性。例如，这个命令*不会*产生预期的效果：

```
$ svn propset license '' calc/button.c

property 'license' set on 'calc/button.c'

$ svn proplist -v calc/button.c

Properties on 'calc/button.c':

  copyright : (c) 2006 Red-Bean Software

  license :

$
```

你需要用 **propdel** 来删除属性，语法与其它与属性命令相似：

```
$ svn propdel license calc/button.c

property 'license' deleted from 'calc/button.c'.

$ svn proplist -v calc/button.c

Properties on 'calc/button.c':
```

```
  copyright : (c) 2006 Red-Bean Software

$
```

还记的这些未版本化的修订版本属性？你也可以使用 **svn** 子命令修改这些属性。只需要添加--revprop 命令参数，说明希望修改属性的修订版本。因为修订版本是全局的，你不需要指定一个路径，只要你已经位于你希望修改属性的工作拷贝路径，或者，你也可以提供版本库的 URL 的任何路径（也包括版本库的根 URL）。例如，[10]如果你当前的工作路径是一个版本库工作拷贝的一部分，你可以简单的运行没有目标路径的 **svn propset** 命令：

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11
--revprop

property 'svn:log' set on repository revision '11'

$
```

但是即使你没有从版本库检出一个工作拷贝，你仍然可以通过提供版本库根 URL 来影响属性修改。

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11
--revprop \

            http://svn.example.com/repos/project

property 'svn:log' set on repository revision '11'

$
```

注意，修改这些未版本化的属性的能力一定要明确的添加给版本库管理员（见“修正提交消息”一节）。因为属性没有版本化，如果编辑的时

候不小心，就会冒丢失信息的风险，版本库管理员可以设置方法来防范这种意外，缺省情况下，修改未版本化的属性是禁止的。

用户必须在可能的情况下使用 **svn propedit**，而不是 **svn propset**。然而这两个命令的结果是相同的，前一个会允许他们查看修改以前的内容，可以帮助用户验证，实际上，作出他们所期望的修改，当修改未版本化修订版本属性时，这一点特别需要。另外，这个命令也可以通过文本编辑器或命令行轻松的修改多行属性。

属性和 Subversion 工作流程

现在你已经熟悉了所有与属性相关的 **svn** 子命令，让我们看看属性修改如何影响 Subversion 的工作流。我们前面提到过，文件和目录的属性是版本化的，这一点类似于版本化的文件内容。后果之一，就是 Subversion 具有了同样的机制来合并—用干净或者冲突的方式—其他人的修改应用到你的修改。

就像文件内容，你的属性修改是本地修改，只有使用 **svn commit** 命令提交后才会保存到版本库中，属性修改也可以很容易的取消—**svn revert** 命令会恢复你的文件和目录为编辑前状态，包括内容、属性和其它的信息。另外，你可以使用 **svn status** 和 **svn diff** 接受感兴趣的文件和目录属性的状态信息。

```
$ svn status calc/button.c
 M     calc/button.c
```

```
$ svn diff calc/button.c

Property changes on: calc/button.c

___________________________________________________________________
_______

Name: copyright

   + (c) 2006 Red-Bean Software



$
```

注意 **status** 子命令显示的 M 在第二列而不是在第一列，这是因为我们修改了 *calc/button.c* 的属性，而不是它的文本内容，如果我们都修改了，我们也会看到 M 出现在第一列（见“查看你的修改概况”一节）。

属性冲突

与文件内容一样，本地的属性修改也会同别人的提交冲突，如果你更新你的工作拷贝目录并且接收到有资源属性修改与你的修改冲突，Subversion 会报告资源处于冲突状态。

```
% svn update calc

M  calc/Makefile.in

 C calc/button.c

Updated to revision 143.

$
```

Subversion 也会在冲突资源的同一个目录创建一个*.prej* 扩展名的文件，保存冲突的细节。你一定要检查这个文件的内容来决定如何解决冲突，

在你解决冲突之前，你会在使用 **svn status** 时看到这个资源的输出的第二列是一个 C，提交本地修改的尝试会失败。

```
$ svn status calc

 C     calc/button.c

?      calc/button.c.prej

$ cat calc/button.c.prej

prop 'linecount': user set to '1256', but update set to
'1301'.

$
```

为了解决属性冲突，只需要确定冲突的属性保存了它们应该的值，然后使用 **svn resolved** 命令告诉 Subversion 你已经手工解决了问题。

你也许已经注意到了 Subversion 在显示属性时的非标准方式。你还可以运行 **svn diff** 并且重定向输出来产生一个有用的补丁文件，**patch** 程序会忽略属性补丁—作为规则，它会忽略任何不理解的噪音。很遗憾，这意味着完全应用 **svn diff** 产生的补丁时，任何属性修改必须手工实施。

自动设置属性

属性是 Subversion 一个强大的特性，成为本章和其它章讨论的许多 Subversion 特性的关键组成部分—文本区别和合并支持、关键字替换、新行的自动转换等等。但是为了从属性得到完全的利益，他们必须设置到正确的文件和目录。不幸的是，在日常工作中很容易忘记这一步工作，

特别是当没有设置属性不会引起明显的错误时（至少相对与未能添加一个文件到版本控制这种操作），为了帮助你在需要添加属性的文件上添加属性，Subversion 提供了一些简单但是有用的特性。

当你使用 **svn add** 或是 **svn import** 准备加入一个版本控制的文件时，Subversion 会自动运行一个基本探测来检查文件是包含了可读还是不可读的内容，首先，在支持执行允许位的操作系统，Subversion 会自动会为设置执行位的文件设置 svn:executable 属性（更多信息见“文件的可执行性”一节）。第二，它会运行非常基础的启发式检查来检测文件是否可读，如果不是，Subversion 会自动设置文件的 svn:mime-type 属性为 application/octet-stream（原始的“一组字节”的 MIME 类型）。如果 Subversion 猜测错误，或者是你希望使用 svn:mime-type 属性更精确的设置—或许是 image/png 或者 application/x-shockwave-flash—你可以一直删除或编辑那个属性（关于 Subversion 使用 MIME 类型的更多信息，见“文件内容类型”一节。）

Subversion 也通过运行配置系统（见“运行配置区”一节）提供了自动属性特性，允许你创建文件名到属性名称与值影射，这个影射在你的运行配置区域设置，它们会影响添加和导入操作，而且不仅仅会覆盖 Subversion 所有缺省的 MIME 类型判断操作，也会设置额外的 Subversion 或者自定义的属性。举个例子，你会创建一个影射文件说在任何时候你添加了一个 JPEG 文件—一些符合*.jpg 的文件—Subversion 一定会自动设置它们的 svn:mime-type 属性为 image/jpeg。或

者是任何匹配*.cpp 的文件，必须把 svn:eol-style 设置为 native，并且 svn:keywords 设置为 Id。自动属性支持是 Subversion 工具箱中属性相关最垂手可得的工具，见“配置”一节来查看更多的配置支持。

文件移植性

幸运的是，对于许多在不同操作系统下工作的用户，Subversion 命令行程序的行为方式几乎完全一致，如果你知道在一个平台上如何运行 **svn**，你也就学会了在其他平台上运行。

然而，这一点在本软件的其他几类地方或 Subversion 保持的实际文件并不一定都是正确的。例如，在一个 Windows 系统，“文本文件”的定义与 Linux 环境下的类似，但是也有区别—行结束的字符串并不相同。当然也有其他的区别，Unix 平台支持（Subversion 也支持）符号链；Windows 不知吃，Unix 使用文件系统执行位来检测可执行性；而 Windows 使用文件扩展名。

因为 Subversion 不是要将世界上的所有此类事情统一起来，所以我们最好是尽可能让我们在多个计算机和操作系统上使用版本化文件和目录时能够更简单，本节描述了 Subversion 是如何做的。

文件内容类型

Subversion 同很多应用一样利用多用途网际邮件扩展（MIME）内容类型，svn:mime-type 属性为 Subversion 的许多目的服务，除了保存一个

文件的 MIME 分类以外，这个 svn:mime-type 属性值也描述了一些 Subversion 自己使用的行为特性。

识别文件类型

大多数现代操作系统通过文件名的扩展名推断文件的类型和格式，例如，以*.txt* 为后缀的文件通常被认为是可读的，不需要通过复杂的解码处理就可以被阅读。对于后缀名为*.png* 文件，则被认为是 Portable Network Graphics 类型—不是可读的格式，只能通过识别 PNG 格式并且可以将信息转化为光栅的软件使用。

不幸的是，一些扩展名已经随时间改变了意义。当个人电脑第一次出现时，一个叫做 *README.DOC* 的文件一定是一个纯文本文件了，就像现在的*.txt* 文件。但是在九十年代中期，你可以打赌这个文件已经不是文本文件，而变成了微软的私有的、不可读的 Word 文档格式。但是这个变化不是一夜完成的—在一段时间里一定会有用户在看到一个*.DOC* 时感到困惑。[11]

计算机网络的普及导致对于文件名和其关联内容的更多混淆，当信息存放在网络上并由服务器脚本动态生成时，会没有文件名。Web 服务器，需要使用其他方式告诉浏览器它们所请求内容的信息，这样浏览器才能智能的根据信息作一些事情，或者是使用注册的程序显示数据，或者是提示用户将文件下载到某个地方。

最终，一个描述数据流内容的标准出现了。1996 年，5 个描述 MIME 的 RFC 中的第一个 RFC2045 发布了，这个 RFC 描述了媒体类型和子类型概念，并且提出了表示这些类型的推荐语法。现在，MIME 媒体类型—MIME 类型已经广泛应用在电邮应用，Web 服务器和其他软件，成为了防止文件内容混淆的标准机制。

举个例子，一个好处就是 Subversion 在更新时通常可以提供基于上下文的行为基础的合并，如果一个文件 `svn:mime-type` 属性设置为非文本的 MIME 类型（通常是那些不是 `text/`开头的类型，但也有例外），Subversion 会假定这个文件保存了二进制内容—也就是不可读的数据。一个好处就是 Subversion 通常在更新工作拷贝时提供了一个前后相关的以行为基础的修改合并，但是对于二进制数据文件，没有“行”的概念，所以对这些文件，Subversion 不会在更新时尝试执行合并操作，相反，任何时候你在本地已经修改的一个二进制文件有了更新，你的文件扩展名会修改为*.orig*，然后 Subversion 保存一个新的工作拷贝文件来保存更新时得到的修改，但原来的文件名已经不是你自己的本地修改。这个行为模式是用来保护用户在对不可文本合并的文件尝试执行文本的合并时失败的情形。

另外，如果设置了 `svn:mime-type` 属性，Subversion 的 Apache 模块会使用这个值来在 HTTP 头里输入 `Content-type:`，这给了 web 浏览器如何显示版本库的一个文件提供了至关重要的线索。

文件的可执行性

在多数操作系统，执行一个文件或命令的能力是由执行位管理的，这些位缺省是关闭的，必须由用户根据需要显式的指定，但是记住应该为哪些检出的文件设置可执行位会是一件很麻烦的事情，所以 Subversion 提供了 svn:executable 这个属性来保持打开执行位，在工作拷贝得到这些文件时设置执行位。

这个属性对于没有可执行权限位的文件系统无效，如 FAT32 和 NTFS。[12]也就是说，尽管它没有定义的值，在设置这个属性时，Subversion 会强制它的值为*，最后，这个属性只对文件有效，目录无效。

行结束字符串

除非使用版本化文件的 svn:mime-type 属性注明，Subversion 会假定这个文件保存了可读的数据，一般来讲，Subversion 只使用这些信息来判断一个文件是否可以用上下文区别的报告，否则，对 Subversion 来说只是字节。

这意味着缺省情况下，Subversion 不会关注任何*行结束标记（end-of-line，EOL）*，不幸的是不同的操作系统在文本文件使用不同的行结束标志，举个例子，Windows 平台下的 A 编辑工具使用一对 ASCII 控制字符—回车（CR）和一个换行（LF），而 Unix 软件，只使用一个 LF 来表示一个行的结束。

并不是所有操作系统的工具准备好了理解与*本地行结束样式*不一样的行结束格式，一个常见的结果是 Unix 程序会把 Windows 文件中的 CR 当作一个不同的字符（通常表现为^M），而 Windows 程序会把 Unix 文件合并为一个非常大的行，因为没有发现标志行结束的回车加换行（或者是 CRLF）字符。

对外来 EOL 标志的敏感会让在多种操作系统分享文件的人们感到沮丧，例如，考虑有一个源代码文件，开发者会在 Windows 和 Unix 系统上编辑这个文件，如果所有的用户使用的工具可以展示文件的行结束，那就没有问题了。

但实践中，许多常用的工具不会正确的读取外来的 EOL 标志，或者只是在保存文件时将文件的行结束符转化为本地的样式，如果是前者，他需要一个外部的转化工具（如 **dos2unix**，或是他的伴侣 **unix2dos**）来准备需要编辑的文件。后一种情况不需要额外的准备工作，两种方法都会造成文件会与原来的文件在每一行上都不一样！在提交之前，用户有两个选择，或者选择用一个转化工具恢复文件的行结束样式，或者是简单的提交文件—包含新的 EOL 标志。

这个情景的结局看起来像是要浪费时间对提交的文件作不必要的修改，浪费时间是痛苦的，但是如果提交修改了文件的每一行，判断文件修改了哪一行会是一件复杂的工作，bug 在哪一行修正的？哪一行导致了语法错误？

这个问题的解决方案是 svn:eol-style 属性，当这个属性设置为一个正确的值时，Subversion 使用它来判断针对行结束样式执行何种特殊的操作，而不会随着多种操作系统的每次提交而发生剧烈变化，正确的值有：

native

这会导致保存 EOL 标志的文件使用 Subversion 运行的操作系统的本地编码，换句话说，如果一个 Windows 用户取出一个工作拷贝包含的文件设置 native 的属性为 svn:eol-style，这个文件会使用 CRLF 的 EOL 标志，一个 Unix 用户取出相同的文件会看到他的文件使用 LF 的 EOL 标志。

注意 Subversion 实际上使用 LF 的 EOL 标志，而不会考略操作系统，尽管这对用户来说是透明的。

CRLF

这会导致这个文件使用 CRLF 序列作为 EOL 标志，不管使用何种操作系统。

LF

这会导致文件使用 LF 字符作为 EOL 标志，不管使用何种操作系统。

CR

这会导致文件使用 CR 字符作为 EOL 标志，不管使用何种操作系统。这种行结束样式不是很常见，它用在一些老的苹果机（Subversion 不会运行的机器上）上。

忽略未版本控制的条目

在任何工作拷贝，将版本化文件和目录与没有也不准备版本化的文件分开会是非常常见的情况。文本编辑器的备份文件会将目录搞乱，代码编译过程中生成的中间文件，甚至最终文件也不是你希望版本化的，用户在见到这些文件和目录（经常是版本控制工作拷贝中）的任何时候都会将他们删除。

期望让 Subversion 的工作拷贝摆脱混乱保持干净是可笑的，实际上 Subversion 将工作拷贝是普通目录作为它的一项*特性*。但是这些没有版本化的文件和目录会给 Subversion 用户带来一些烦恼，例如，因为 **`svn add`** 和 **`svn import`** 命令都是会递归执行的，并不知道哪些文件你不希望版本化，很容易意外的添加一些文件。因为 **`svn status`** 会报告工作拷贝中包括未版本化文件和目录的信息，如果这种文件很多，它的输出会变得非常嘈杂。

所以 Subversion 提供了两种方法让你指明哪些文件可以被漠视，一种方法需要你修改 Subversion 的运行配置系统（见“运行配置区”一节），这样会使所有的 Subversion 操作都利用这个配置，通常来说，这是在某一个计算机上的操作，或者是某个计算机某个用户的操作。另一种方

法利用了 Subversion 目录属性支持，与版本化的目录树紧密结合，因而会影响所有拥有这个目录树工作拷贝的人。两种机制都使用文件模式。

Subversion 运行配置系统提供一个 global-ignores 选项，其中的值是空格分开的文件名模式（或 glob）。这些模式会应用到可以添加到版本控制的候选者，也就是 **svn status** 显示出来的未版本化文件。如果文件名与其中的某个模式匹配，Subversion 会当这个文件不存在。这个文件模式最好是全局不期望版本化的模式，例如编辑器 Emacs 的备份文件*~和.*~。

如果是在版本化目录上发现 svn:ignore 属性，其内容是一列以行分割的文件模式，Subversion 用来判断在这个目录下对象是否被忽略。这些模式不会覆盖在运行配置设置的全局忽略，而是向其添加忽略模式。不像全局忽略选项，在 svn:ignore 属性中设置的值只会应用到其设置的目录，而不会应用到其子目录。svn:ignore 属性是告诉 Subversion 在每个用户的工作拷贝对应目录忽略相同的文件的好方法，例如编译输出或一使用一个本书相关的例子一本书从 DocBook XML 文件生成的 HTML、PDF 或 PostScript。

Subversion 对于忽略文件模式的支持仅限于将未版本化文件和目录添加到版本控制时，如果一个文件已经在 Subversion 控制下，忽略模式机制不会再有效果，不要期望 Subversion 会阻止你提交一个符合忽略条件的修改一Subversion 一直认为它是版本化的对象。

CVS 用户的忽略模式

Subversion 的 svn:ignore 属性与 CVS 的*.cvsignore* 文件的语法和功能非常类似，实际上，如果你移植一个 CVS 的工作拷贝到 Subversion，你可以直接使用*.cvsignore* 作为 **svn propset** 输入文件参数：

```
$ svn propset svn:ignore -F .cvsignore .
property 'svn:ignore' set on '.'
$
```

但是 CVS 和 Subversion 处理忽略模式的方式有一些不同，这两个系统在不同的时候使用忽略模式，忽略模式应用的对象也由微小的不同，另外 Subversion 不可以使用!模式来去取消忽略模式。

全局忽略模式只是一种个人喜好，可能更接近于用户的特定工具链，而不是特定工作拷贝的需要，所以余下的小节将关注 svn:ignore 属性和它的使用。

假定你的 **svn status** 有如下输出：

```
$ svn status calc
 M      calc/button.c
?       calc/calculator
?       calc/data.c
?       calc/debug_log
?       calc/debug_log.1
?       calc/debug_log.2.gz
```

```
?      calc/debug_log.3.gz
```

在这个例子里，你对 *button.c* 文件作了一些属性修改，但是你的工作拷贝也有一些未版本化的文件：你从源代码编译的最新*计算器*程序，一系列调试输出日志文件，现在你知道你的编译系统一直会编译生成*计算器*程序。[13]而且你知道你的测试组件总是会留下这些调试日志，这对所有的工作拷贝都是一样的，不仅仅是你的。你也知道你不会有兴趣在 **svn status** 命令中显示这些信息，所以使用 **svn propedit svn:ignore calc** 来为 *calc* 目录增加一些忽略模式，举个例子，你或许会添加如下的值作为 svn:ignore 的属性：

```
calculator

debug_log*
```

当你添加完这些属性，你会在 *calc* 目录有一个本地修改，但是注意你的 **svn status** 输出有什么其他的不同：

```
$ svn status

 M     calc

 M     calc/button.c

?      calc/data.c
```

现在，所有多余的输出不见了！当然，你的*计算器*程序和所有的日志文件还在工作拷贝中，Subversion 仅仅是不再提醒你它们的存在和未版本

化。现在所有讨厌的噪音都已经不再显示，只留下了你感兴趣的条目—如你忘记添加到版本控制的源代码文件 *data.c*。

当然，不仅仅只有这种简略的工作拷贝状态输出，如果想查看被忽略的文件，可以使用 Subversion 的--no-ignore 选项：

```
$ svn status --no-ignore
 M     calc
 M     calc/button.c
I      calc/calculator
?      calc/data.c
I      calc/debug log
I      calc/debug log.1
I      calc/debug log.2.gz
I      calc/debug log.3.gz
```

我们在前面提到过，**svn add** 和 **svn import** 也会使用这个忽略模式列表，这两个操作都包括了询问 Subversion 来开始管理一组文件和目录。比强制用户挑拣目录树中那个文件要纳入版本控制的方式更好，Subversion 使用忽略模式来检测那个文件不应该在大的迭代添加和导入操作中进入版本控制系统。再次说明，操作 Subversion 文件和目录时你可以使用--no-ignore 选项忽略这个忽略列表。

关键字替换

Subversion 具备添加*关键字*的能力—一些有用的，关于版本化的文件动态信息的片断—不必直接添加到文件本身。关键字通常会用来描述文件最后一次修改的一些信息，因为这些信息每次都有改变，更重要的一点，这是在文件修改*之后*，除了版本控制系统，对于任何企图保持数据最新的过程都是一场混乱，作为人类作者，信息变得陈旧是不可避免的。

举个例子，你有一个文档希望显示最后修改的日期，你需要麻烦每个作者提交之前做这件事情，也要修改文档的一部分来描述何时作的修改，但是迟早会有人忘记做这件事，不选择简单的告诉 Subversion 来执行替换 LastChangedDate 关键字的操作，你通过在目标位置放置一个 *keyword anchor* 来控制关键字插入的位置，这个 anchor 只是一个格式为$*KeywordName*$字符串。

所有作为 anchor 出现在文件里的关键字是大小写敏感的：为了关键字的扩展，你必须使用正确的大写，你必须考虑 svn:keywords 的属性值也是大小写敏感—特定的关键字名会忽略大小写，但是这个特性已经被废弃了。

Subversion 定义了用来替换的关键字列表，这个列表保存了如下五个关键字，有一些也包括了可用的别名：

Date

这个关键字保存了文件最后一次在版本库修改的日期，看起来类似于$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $，它也可以用 LastChangedDate 来指定。

Revision

这个关键字描述了这个文件最后一次修改的修订版本，看起来像$Revision: 144 $，也可以通过 LastChangedRevision 或者 Rev 引用。

Author

这个关键字描述了最后一个修改这个文件的用户，看起来类似$Author: harry $，也可以用 LastChangedBy 来指定。

HeadURL

这个关键字描述了这个文件在版本库最新版本的完全 URL，看起来类似$HeadURL: http://svn.collab.net/repos/trunk/README $，可以缩写为 URL。

Id

这个关键字是其他关键字一个压缩组合，它看起来就像$Id: calc.c 148 2006-07-28 21:30:43Z sally $，可以解释为文件 *calc.c* 上一次修改的修订版本号是 148，时间是 2006 年 7 月 28 日，作者是 sally。

前面的一些描述使用了类似“最后已知的”短语，请记住关键字扩展是客户端操作，你的客户端只“知道”在你更新工作拷贝时版本库发生的

修改，如果你从不更新工作拷贝，即使文件在版本库里有规律的修改，这些关键字也不会扩展为不同的值。

只在你的文件增加关键字 anchor 不会做什么特别的事情，Subversion 不会尝试对你的文件内容执行文本替换，除非明确的被告知这样做，毕竟，你可以撰写一个关于如何使用关键字的文档[14]，你不希望 Subversion 会替换你漂亮的关于不需要替换的关键字 anchor 实例！

为了告诉 Subversion 是否替代某个文件的关键字，我们要再次求助于属性相关的子命令，当 svn:keywords 属性设置到一个版本化的文件，这些属性控制了哪些关键字将会替换到这个文件，这个属性的值是空格分隔的前面列表的名称或是别名列表。

举个例子，假定你有一个版本化的文件 *weather.txt*，内容如下：

```
Here is the latest report from the front lines.

$LastChangedDate$

$Rev$

Cumulus clouds are appearing more frequently as summer
approaches.
```

当没有 svn:keywords 属性设置到这个文件，Subversion 不会有任何特别操作，现在让我们允许 LastChangedDate 关键字的替换。

```
$ svn propset svn:keywords "Date Author" weather.txt

property 'svn:keywords' set on 'weather.txt'
```

```
$
```

现在你已经对 *weather.txt* 的属性作了修改，你会看到文件的内容没有改变（除非你之前做了一些属性设置），注意这个文件包含了 Rev 的关键字 anchor，但我们没有在属性值中包括这个关键字，Subversion 会高兴的忽略替换这个文件中的关键字，也不会替换 svn:keywords 属性中没有出现的关键字。

在你提交了属性修改后，Subversion 会立刻更新你的工作文件为新的替代文本，你将无法找到$LastChangedDate$的关键字 anchor，你会看到替换的结果，这个结果也保存了关键字的名字，与美元符号（$）绑定在一起，而且我们预测的，Rev 关键字不会被替换，因为我们没有要求这样做。

注意我们设置 svn:keywords 属性为“Date Author”，关键字 anchor 使用别名$LastChangedDate$并且正确的扩展。

```
Here is the latest report from the front lines.

$LastChangedDate: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006)
$

$Rev$

Cumulus clouds are appearing more frequently as summer
approaches.
```

如果有其他人提交了 *weather.txt* 的修改，你的此文件的拷贝还会显示同样的替换关键字值—直到你更新你的工作拷贝，此时你的 *weather.txt* 重的关键字将会被替换来反映最新的提交信息。

$GlobalRev$在哪里？

新用户经常为如何使用Rev关键字迷惑，自从版本库有了单独的全局增长的修订版本号码，许多人以为Rev关键字是反映修订版本号码的，但实际上Rev是文件最后修改的修订版本，而不是最后更新的。理解这一点，会减少一些混淆，但是还有一些挫折—如果没有 Subversion 关键字的支持，你怎么才能在你的文件自动得到全局修订版本号到。

为此你需要外置处理，Subversion 中有一个工具 **svnversion** 就是为此设计。**svnversion** 遍历你的工作拷贝，然后输出它发现的修订版本，你可以使用这个程序，外加一些工具，将修订版本信息嵌入到你的文件。关于 **svnversion** 的更多信息，见“ **svnversion** ”一节。

Subversion 1.2 引入了另一种关键字的语法，提供了额外和有用的，尽管是非典型的功能。你现在可以告诉 Subversion 为替代的关键字维护一个固定长度（从消耗字节的观点），通过在关键字名后使用双冒号（::），然后紧跟一组空格，你就定义了固定宽度。当 Subversion 使用替代值代替你的关键字，只会替换这些空白字符，保持关键字字段长度保持不变，如果替代值比定义的字段短，会有替代字段后保留空格；如果替代值太长，就会在最后的美元符号终止符前用井号（#）截断。

例如，你有一篇文档，其中一段是一些反映 Subversion 关键字的表格数据，使用原始的 Subversion 关键字替换语法，你的文件或许像这样：

```
$Rev$:     Revision of last commit

$Author$:  Author of last commit

$Date$:    Date of last commit
```

现在，表格看起来很漂亮，但是当你提交文件（当然，关键字替换功能已打开），你会看到：

```
$Rev: 12 $:     Revision of last commit

$Author: harry $:  Author of last commit

$Date: 2006-03-15 02:33:03 -0500 (Wed, 15 Mar 2006) $:    Date
of last commit
```

结果并不漂亮，你可能会尝试重新调整文件使之更像一个列表。只有关键字的长度是相同的时候才能保证保持样式，如果进入另一个修订版本（如从 99 到 100），或者是另一个有较长用户名的人提交了文件，表格又会变形。然而，如果你使用 Subversion 1.2，你可以使用新的固定长度的关键字语法，定义合适的字段宽度，然后你的文件可能如此：

```
$Rev::              $:  Revision of last commit

$Author::           $:  Author of last commit

$Date::             $:  Date of last commit
```

你提交这个文件的修改，这一次 Subversion 注意到了新的固定长度的关键字语法，根据你在双冒号之间指定的空格长度调整格式，并且紧跟

一个美元符号。经过替换，字段的长度没有发生变化—Rev 和 Author 多了一些空格，而较长的 Date 字段被一个分号截断：

```
$Rev:: 13           $:  Revision of last commit

$Author:: harry     $:  Author of last commit

$Date:: 2006-03-15 0#$:  Date of last commit
```

固定长度关键字在执行复杂文件格式的替换中非常易用，也可以处理那些很难通过其他程序（例如 Microsoft Office 文档）进行修改的文件。

需要意识到，因为关键字字段的长度是以字节为单位，可能会破坏多字节值，例如一个用户名包含多字节的 UTF-8 字符，可能会遭遇从某个字符中间截断的情况，从字节角度看仅仅是一种截断，但是从 UTF-8 字符串角度看可能是错误和曲解的，当载入文件时，破坏的 UTF-8 文本可能导致整个文件的破坏，整个文件无法操作。所以，当限制关键字为固定大小时，需要选择一个可以扩展的大小。

锁定

Subversion 的拷贝-修改-合并版本控制模型的关键是其合并算法，也就是如何处理多个用户修改同时修改一个文件产生冲突时的算法。

Subversion 本身只提供了一个这样的算法，其三方区别算法可以足够聪明的的行粒度的数据处理，Subversion 也支持使用外置比较工具（“外置 diff3”一节中有描述），有一些可以做得非常好，或许可以提供以单词或字母粒度的算法。但是，这些工具的共同点是基于文本的，当你

讨论非文本文件格式时，这看起来有一点残酷。如果你无法找到一个工具支持这种类型的合并，你的拷贝-修改-合并模型就会遇到麻烦。

让我们看一个使用这个模型的真实例子，Harry 和 Sally 是同一个项目的图形设计师，汽车技工的间接营销。海报的设计一个小车，需要一些主要部分的工作，使用 PNG 文件格式。海报的布局几乎完成，Harry 和 Sally 都看上了一个从损坏小车得到的特别照片—一个 1967 的淡蓝色的 Ford Mustang，挡泥板有一些溅迹。

现在，作为图像设计的惯例，计划的改变导致车的颜色很重要，所以 Sally 将工作拷贝更新到 HEAD，启动图形编辑软件，修改图像将车的颜色修改为樱桃红，同时 Harry 那一天特别有灵感，所以决定如果这个车受到更大的撞击可能会有更好的效果。他也更新到 HEAD，然后在车挡风玻璃上制作了一些裂痕，他设法在 Sally 完成前结束修改，因为受到自己不可阻挡天赋的鼓舞，提交了图像。没过多久，Sally 结束了她的工作，尝试提交。但是如我们所料，Subversion 提交失败，告诉 Sally 她的图像已经过期了。

这里就是麻烦的地方，如果 Harry 和 Sally 修改的是文本文件，她只需要简单得更新工作拷贝，接收 Harry 的修改。在最坏的情况下，他们会修改文件的同一部分，Sally 需要人工解决冲突。但是现在不是文本文件—而是二进制图像，没法估计合并的结果会是什么样子的，已存的软件不可能从基线图像分离出 Harry 和 Sally 的工作，并组合出一个挡风玻璃坏掉的红色 Mustang。

很显然，如果能够将 Harry 和 Sally 的工作串行话事情会变得平滑，也就是说 Harry 可以等到 Sally 的红车然后再画上破坏的挡风玻璃，或者 Sally 在破坏之后改变颜色。就像在“拷贝-修改-合并 方案”一节讨论的，如果 Harry 和 Sally 之间有完美的交流，就不会有这种问题发生。[15]但是作为一种版本控制系统，实际上是一种交流的形式，使得软件遵循非并行编辑的串行化也不是一件坏事，这里 Subversion 实现了锁定-修改-解锁模型，这里我们要讨论 Subversion 的*锁定*特性，与其他版本控制系统的“保留检出”机制类似。

Subversion 的锁定特性为两个主要目的服务：

- *顺序访问资源*。允许用户得到一个排他的修改文件权，这个用户可以确定不可合并的修改不会被浪费—他对这个修改的提交会成功。
- *辅助交流*。通过要求用户对某个版本化对象串行工作，用户可以知道对象正在被别人修改，这样可以防止浪费精力和时间去修改一个不可合并和提交的对象。

当我们引用 Subversion 锁定特性时，这是在讨论一个处理版本化文件的行为特性[16]（声明对一个文件排他性修改特权），包括对文件的锁定和解锁（释放排他性修改权限），察看包括文件被谁锁定的报告，以及提醒企图修改锁定文件的用户。在本小节，我们会覆盖锁定特性的大部分内容。

“锁定”的三种含义

在本小节，和几乎本书的每一个地方“lock”和“locking”描述了一种避免用户之间冲突提交的排他机制，但是很不幸，Subversion 中还有另外两种锁，因此需要在本书格外关心。

第一种是*工作拷贝锁*，Subversion 内部用来防止不同客户端同时操作同一份工作拷贝的锁，这种锁使用 **`svn status`** 输出中第三列出现的 L 表示，可以使用 **`svn cleanup`** 删除，“有时你只需要清理”一节有介绍。

第二种，*数据库锁*，在 Berkeley DB 后端内部使用，防止多个程序访问数据库发生冲突，一个导致版本库“楔住”的错误发生后产生，“Berkeley DB 恢复”一节有描述。

在发生问题之前你完全可以忘记上面两种锁，在本书，“锁定”意味着第一种锁，除非是在从上下文中十分明确或明确指出的。

创建锁定

在 Subversion 的版本库，一个*锁*是一份元数据，可以排它赋予某个用户修改权，这个用户被称作*锁的拥有者*。每个锁都有一个唯一标识，通常是一长串字符，叫做*锁令牌*。版本库管理锁，控制着锁的创建，权限控制和删除。如果提交包含了修改或者删除锁

为了描述锁的产生，我们回到前面那个关于多个图形设计师共同工作的例子，Harry 决定修改一个 JPEG 图像，为了防止其他用户此时提交这个文件的修改（也是警告别人他正在修改它），他使用 **svn lock** 命令锁定了版本库中的这个文件：

```
$ svn lock banana.jpg -m "Editing file for tomorrow's release."
'banana.jpg' locked by user 'harry'.
$
```

前一个例子描述了许多新事物，第一，注意 Harry 在 **svn lock** 中使用了--message (-m)选项，类似于 **svn commit**，**svn lock** 命令可以有描述锁定原因的注释（通过--message (-m)或--file (-F)）。然而不像 **svn commit**，**svn lock** 不会自动强制启动你喜欢的编辑器，锁定注释是可选的，但是为了方便交流我们还是推荐使用。

第二，锁定成功了，这意味着文件没有被别人锁定，Harry 的文件是最新的版本。如果 Harry 的工作拷贝文件不是最新的，版本库会拒绝请求，强制 Harry 执行 **svn update** 并重新运行锁定命令，同样，如果此文件已经被别的用户锁定了，锁定命令也会失败。

就像你看到的，**svn lock** 打印了锁定成功的确认信息。此时，通过 **svn status** 和 **svn info** 的输出我们可以看到文件已经锁定。

```
$ svn status
     K banana.jpg
```

```
$ svn info banana.jpg

Path: banana.jpg

Name: banana.jpg

URL: http://svn.example.com/repos/project/banana.jpg

Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec

Revision: 2198

Node Kind: file

Schedule: normal

Last Changed Author: frank

Last Changed Rev: 1950

Last Changed Date: 2006-03-15 12:43:04 -0600 (Wed, 15 Mar 2006)

Text Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)

Properties Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun
2006)

Checksum: 3b110d3b10638f5d1f4fe0f436a5a2a5

Lock Token:
opaquelocktoken:0c0f600b-88f9-0310-9e48-355b44d4a58e

Lock Owner: harry

Lock Created: 2006-06-14 17:20:31 -0500 (Wed, 14 Jun 2006)

Lock Comment (1 line):

Editing file for tomorrow's release.


$
```

svn info 命令不会联系版本库，当对工作拷贝路径应用 **svn info** 命令时，可以揭示令牌的一个重要事实—它们缓存在工作拷贝。有锁定令牌是非常重要的，这给了工作拷贝权利利用这个锁的能力。**svn status** 会在文件后面显示一个 K（locKed 的缩写），表明了拥有锁定令牌。

关于锁定令牌

一个锁不是一个认证令牌，而是一个*授权*令牌，这个令牌不是一个受保护的秘密，事实上，任何人都可以通过 **svn info URL** 发现这个唯一令牌。一个锁定令牌只有在工作拷贝中才有特别的意义，它是锁定建立在这个工作拷贝的证据，而不是其它用户在其他地方，仅仅检验锁定拥有者还不能防止出现意外。

例如，你在办公室电脑上锁定了一个文件，或许修改正在进行中。很有可能在你的家用计算机上的一个工作拷贝（或别的 Subversion 客户端）里你又不小心修改了同一个文件，仅仅因为检验了你就是锁定的拥有者。换句话说，锁定令牌防止你通过一个 Subversion 相关软件的工作破坏另一个的工作。（在我们的例子里，如果你真的需要在另一个工作拷贝修改这个文件，你必须*打破*锁定再重新锁定文件。）

现在 Harry 已经锁定了 *banana.jpg*，Sally 不能修改或删除这个文件：

```
$ svn delete banana.jpg
D         banana.jpg
```

```
$ svn commit -m "Delete useless file."
Deleting       banana.jpg
svn: Commit failed (details follow):
svn: DELETE of
'/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-df4224ddc35
d/banana.jpg':
423 Locked (http://svn.example.com)
$
```

但是，当完成了香蕉的黄色渐变，就可以提交文件的修改，因为认证为锁定的拥有者，也因为他的工作拷贝有正确的锁定令牌：

```
$ svn status
M    K banana.jpg
$ svn commit -m "Make banana more yellow"
Sending       banana.jpg
Transmitting file data .
Committed revision 2201.
$ svn status
$
```

需要注意到提交之后，**svn status** 显示工作拷贝已经没有锁定令牌了，这是 **svn commit** 的标准行为方式—它会遍历工作拷贝（或者从目标列表，如果有列表的话），并且作为提交的一部分发送所有遇到的锁定令牌到服务器。当提交完全成功，前面用到的所有版本库锁定都会被释放—*即使是没有提交的文件*。这样的原因是不鼓励用户滥用锁定，或者

是长时间的保持锁定。例如，假定 Harry 不小心锁定了 *images* 目录的 30 个文件，因为他不确定要修改什么文件，他最后只修改了四个文件，当他运行 **svn commit images**，会释放所有的 30 个锁定。

自动释放锁定的特性可以通过 **svn commit** 的--no-unlock 选项关闭，当你要提交文件，同时期望继续修改而必须保留锁定时非常有用。这个特性也可以半永久性的设定，方法是设置运行中 *config* 文件（见“运行配置区”一节）的 no-unlock = yes。

当然，锁定一个文件不会强制一个人要提交修改，任何时候都可以通过运行 **svn unlock** 命令释放锁定：

```
$ svn unlock banana.c

'banana.c' unlocked.
```

发现锁定

最明显的方式就是因为锁定而不能提交一个文件，最简单的方式是 **svn status --show-updates**：

```
$ svn status -u

M             23   bar.c

M    O        32   raisin.jpg

      *       72   foo.h

Status against revision:    105

$
```

在这个例子里，Sally 可以见到不仅她的 *foo.h* 是过期的，而且发现两个计划要提交的文件被锁定了。O 符号表示其他人所订了文件。如果她尝试提交，*raisin.jpg* 的锁定会阻止她，Sally 会纳闷谁锁定了文件，什么时候，为什么。再一次，**svn info** 拥有答案：

```
$ svn info http://svn.example.com/repos/project/raisin.jpg

Path: raisin.jpg

Name: raisin.jpg

URL: http://svn.example.com/repos/project/raisin.jpg

Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec

Revision: 105

Node Kind: file

Last Changed Author: sally

Last Changed Rev: 32

Last Changed Date: 2006-01-25 12:43:04 -0600 (Sun, 25 Jan 2006)

Lock Token:
opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b

Lock Owner: harry

Lock Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)

Lock Comment (1 line):

Need to make a quick tweak to this image.

$
```

就像 **svn info** 可以检验工作拷贝的对象，它也可以检验版本库的对象，如果 **svn info** 的主要参数是工作拷贝路径，所有工作拷贝的缓

存信息都会显示，发现了锁定就意味着工作拷贝拥有锁定令牌（如果一个文件被另一个用户在另一个工作拷贝锁定，工作拷贝路径上运行 **svn info** 不会显示锁定信息）。如果 **svn info** 的主参数是 URL，就会反映版本库中最新版本的对象信息，任何对锁定的提及描述了当前对象的锁定。

所以在这个特定的例子里，Sally 可以看到 Harry 在二月十六日为了“做修改”而锁定了这个文件，现在已经六月了，她怀疑他可能是忘记了这个锁定，她会打电话给 Harry 去询问他应该释放这个锁定，如果他不再，她就要自己强制解除这个锁定或者是找管理员去做。

解除和偷窃锁定

版本库锁定并不是神圣不可侵犯的，在 Subversion 的缺省配置状态，不只是创建者可以释放锁定，任何人都可以。当有其他人期望消灭锁定时，我们称之为*打破*锁定。

从管理员的位子上很容易打破锁定，**svnlook** 和 **svnadmin** 程序都有能力从版本库直接显示和删除锁定。（关于这些工具的信息可以看“管理员的工具箱”一节。）

```
$ svnadmin lslocks /usr/local/svn/repos

Path: /project2/images/banana.jpg

UUID Token:
opaquelocktoken:c32b4d88-e8fb-2310-abb3-153ff1236923

Owner: frank
```

```
Created: 2006-06-15 13:29:18 -0500 (Thu, 15 Jun 2006)

Expires:

Comment (1 line):

Still improving the yellow color.



Path: /project/raisin.jpg

UUID Token:
opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b

Owner: harry

Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)

Expires:

Comment (1 line):

Need to make a quick tweak to this image.



$ svnadmin rmlocks /usr/local/svn/repos /project/raisin.jpg

Removed lock on '/project/raisin.jpg'.

$
```

更有趣的选项是允许用户互相打破锁定，为此，Sally 只需要使用 unlock 命令的--force 选项：

```
$ svn status -u

M             23   bar.c

M    O        32   raisin.jpg

      *       72   foo.h

Status against revision:     105
```

```
$ svn unlock raisin.jpg

svn: 'raisin.jpg' is not locked in this working copy

$ svn info raisin.jpg | grep URL

URL: http://svn.example.com/repos/project/raisin.jpg

$ svn unlock http://svn.example.com/repos/project/raisin.jpg

svn: Unlock request failed: 403 Forbidden
(http://svn.example.com)

$ svn unlock --force
http://svn.example.com/repos/project/raisin.jpg

'raisin.jpg' unlocked.

$
```

Sally 初始的 unlock 命令失败了，因为她直接在自己的工作拷贝上运行了 **svn unlock**，而这里没有锁定令牌。为了直接从版本库删除锁定，她需要给 **svn unlock** 传递 URL 参数，她的这一次尝试又失败了，因为她不是锁定的拥有者（也没有锁定令牌）。当她使用了--force 选项后，认证和授权的要求被忽略了，远程的锁定被打破了。

当然，简单的打破锁定也许还不够，在这个例子里，Sally 不仅想要打破 Harry 遗忘的锁定，她也希望自己重新锁定。她可以通过运行 **svn unlock --force** 紧接着 **svn lock**，但是有可能有人在这两次命令之间锁定了文件，最简单的方式是*窃取*这个锁定，将打破和重新锁定变成一种原子操作，为此需要运行 **svn lock** 加--force 选项：

```
$ svn lock raisin.jpg

svn: Lock request failed: 423 Locked (http://svn.example.com)
```

```
$ svn lock --force raisin.jpg

'raisin.jpg' locked by user 'sally'.

$
```

在任何情况下，无论锁定被打破还是窃取，Harry 都会感到惊讶。Harry 的工作拷贝还保留有原来的锁定令牌，但是锁定已经不存在了，锁定令牌可以说已经*死掉了*。锁定令牌指代的锁定被打破（版本库中不再存在）或者是窃取了（被另一个锁定代替了），任何一种情况下，Harry 都可以使用 **svn status** 询问版本库：

```
$ svn status

     K raisin.jpg

$ svn status -u

     B        32   raisin.jpg

$ svn update

  B  raisin.jpg

$ svn status

$
```

如果版本库锁定被打破了，**svn status --show-updates** 会在文件旁边显示一个 B (Broken)。如果有一个新的锁，就会显示一个 T (sTolen) 符号。最终，**svn update** 会注意到所有死掉的锁定并且把它们从工作拷贝中删除掉。

锁定策略

不同的系统有不同的锁定限制程度的观念。有些人认为锁定必须不顾任何代价的严格执行，只有原始的创建者和管理员可以释放。他们认为如果有人打破了锁定，混乱就会放任，锁定就完全失去了意义。另外一些人认为锁定是第一个和最首要的交流工具，如果用户经常的打破别人的锁定，代表了团队的文化失败和软件之外的问题。

Subversion 缺省是比较“宽松的”方式，但也允许管理员创建钩子脚本来建立严格的控制策略。具体来说，*pre-lock* 和 *pre-unlock* 钩子允许管理员决定什么时候创建和释放锁定。根据锁定是否已经存在，这两个钩子脚本可以决定是否允许特定用户打破或窃取锁定。也有 *post-lock* 和 *post-unlock* 钩子，可以用来发送锁定动作的通知邮件。关于版本库钩子的更多信息可以看“实现版本库钩子”一节。

锁定交流

我们已经见到了如何利用 **`svn lock`** 和 **`svn unlock`** 来创建、释放、打破和窃取锁定，这就满足了顺序访问文件的要求，但是浪费时间这个大问题该如何呢？

例如，假定 Harry 锁定了一个图片，并开始编辑。同时，几英里之外的 Sally 希望做同样的工作，她没想到运行 **`svn status --show-updates`**，她不知道 Harry 已经锁定了文件。她花费了数小时来修改文件，当她真被提交时发现文件已经被锁定或者是她的文件已

经过期了。她的修改不能和 Harry 的合并，他们中的一人需要抛弃自己的工作，许多时间被浪费了。

Subversion 针对此问题的解决方案是提供一种机制，提醒用户在开始编辑*以前*必须锁定这个文件，这个机制就是提供一种特别的属性 --svn:needs-lock。当有这个值时，*除非*用户锁定这个文件，否则文件一直是只读的。当得到一个锁定令牌（运行 **svn lock** 的结果），文件变成可读写，当释放这个锁后，文件又变成只读。

根据这个原理，如果一个图像文件有这个属性，Sally 打开编辑文件就会立刻注意到有些特别，大多数程序会在打开只读文件时立刻警告，至少所有的程序会防止她保存修改，这提醒了她编辑之前需要锁定文件，这样她就发现了原来存在的锁定：

```
$ /usr/local/bin/gimp raisin.jpg

gimp: error: file is read-only!

$ ls -l raisin.jpg

-r--r--r--   1 sally   sally   215589 Jun  8 19:23 raisin.jpg

$ svn lock raisin.jpg

svn: Lock request failed: 423 Locked (http://svn.example.com)

$ svn info http://svn.example.com/repos/project/raisin.jpg |
grep Lock

Lock Token:
opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b

Lock Owner: harry

Lock Created: 2006-06-08 07:29:18 -0500 (Thu, 08 June 2006)
```

```
Lock Comment (1 line):

Making some tweaks.  Locking for the next two hours.

$
```

我们鼓励用户和管理员都应该给不能根据上下文的文件添加 svn:needs-lock 属性，这是鼓励好的锁定习惯和防止浪费的主要技术手段。

需要注意到这个属性是依赖于锁定系统的交流工具，不管是否有这个属性，文件都可以锁定。相反的，无论有没有这个属性，并不会要求提交需要首先锁定文件。

这个系统并不是毫无瑕疵，即使有这个属性，只读提醒也有可能失效。有些程序“偷偷的篡改了”文件的只读属性，悄无声息的允许用户编辑和保存文件，不幸的是，Subversion 对此无能为力—即使到了现今，还是没有任何工具能够代替人与人的良好交流。[17]

外部定义

有时候创建一个由多个不同检出得到的工作拷贝是非常有用的，举个例子，你或许希望不同的子目录来自不同的版本库位置，或者是不同的版本库。你可以手工设置这样一个工作拷贝—使用 **svn checkout** 来创建这种你需要的嵌套的工作拷贝结构。但是如果这个结构对所有的用户是很重要的，每个用户需要执行同样的检出操作。

很幸运，Subversion 提供了*外部定义*的支持，一个外部定义是一个本地路经到 URL 的影射—也有可能一个特定的修订版本—一些版本化的资源。在 Subversion 你可以使用 svn:externals 属性来定义外部定义，你可以用 **svn propset** 或 **svn propedit**（见“操作属性”一节）创建和修改这个属性。它可以设置到任何版本化的路经，它的值是一个多行的子目录，可选的修订版本标记和完全有效的 Subversion 版本库 URL 的列表（相对于设置属性的版本化目录）。

```
$ svn propget svn:externals calc

third-party/sounds
http://sounds.red-bean.com/repos

third-party/skins
http://skins.red-bean.com/repositories/skinproj

third-party/skins/toolkit -r21
http://svn.red-bean.com/repos/skin-maker
```

svn:externals 的方便之处是这个属性设置到版本化的路径后，任何人可以从那个目录取出一个工作拷贝，同样得到外部定义的好处。换句话说，一旦一个人努力来定义这些嵌套的工作拷贝检出，其他任何人不需要再麻烦了—Subversion 会在原先的工作拷贝检出之后，也会检出外部工作拷贝。

外部定义的相对目标子目录不需要存在于你的或其它用户的系统中—Subversion 会在检出工作拷贝时创建这些文件。实际上，你一定不要使用外部定义来产生已经在版本控制的路径。

注意前一个外部定义实例，当有人取出了一个 *calc* 目录的工作拷贝，Subversion 会继续来取出外部定义的项目。

```
$ svn checkout http://svn.example.com/repos/calc
A  calc
A  calc/Makefile
A  calc/integer.c
A  calc/button.c
Checked out revision 148.

Fetching external item into calc/third-party/sounds
A  calc/third-party/sounds/ding.ogg
A  calc/third-party/sounds/dong.ogg
A  calc/third-party/sounds/clang.ogg
…
A  calc/third-party/sounds/bang.ogg
A  calc/third-party/sounds/twang.ogg
Checked out revision 14.

Fetching external item into calc/third-party/skins
…
```

如果你希望修改外部定义，你可以使用普通的属性修改子命令，当你提交一个 svn:externals 属性修改后，当你运行 **svn update** 时，

Subversion 会根据修改的外部定义同步检出的项目，同样的事情也会发生在别人更新他们的工作拷贝接受你的外部定义修改时。

因为 svn:externals 的值是多行的，所以我们强烈建议使用 **svn propedit**，而不是使用 **svn propset**。

你一定要要考虑在所有的外部定义中使用明确的修订版本，这样做意味着你已经决定了何时拖出外部信息不同的快照，和精确的拖出哪个快照。除了不会受到第三方版本库的意外修改的影响以外，当你的工作拷贝回溯到以前的版本库时，使用明确的修订版本号会让外部定义回到以前的那个修订版本，也意味着外部定义的工作拷贝更新会匹配以前修订版本的样子。对于软件项目，这可能是编译复杂代码基的老快照成功和失败的区别。

svn status 命令也认识外部定义，会为外部定义的子目录显示 X 状态码，然后迭代这些子目录来显示外部项目的子目录状态信息。

Subversion 目前对外部定义的支持可能会引起误导，首先，一个外部定义只可以指向目录，而不是文件。第二，外部定义不可以指向相对路径（如*../../skins/myskin*）。第三，通过外部定义创建的工作拷贝与主工作拷贝没有连接起来（与设置 svn:externals 属性的工作拷贝的版本库），所以 Subversion 会以不关联的工作拷贝操作。所以举个例子，如果你希望提交一个或多个外部定义的拷贝，你必须在这些工作拷贝显示的运行 **svn commit**—对主工作拷贝的提交不会迭代到外部定义的部分。

另外，因为定义本身使用绝对路径，移动和拷贝路径他们附着的路径不会影响他们作为外部的检出（尽管相对的本地目标子目录会这样，当然，根据重命名的目录改变）。在特定情形下这看起来有些迷惑—甚至让人沮丧。举个例子，你的顶级目录叫作 *my-project*，你在它的子目录（*my-project/some-dir*）创建了一个外部定义，而这个外部定义指向的是另一个子目录（*my-project/external-dir*）的最新版本。

```
$ svn checkout http://svn.example.com/projects .
A    my-project
A    my-project/some-dir
A    my-project/external-dir
…
Fetching external item into 'my-project/some-dir/subdir'
Checked out external at revision 11.

Checked out revision 11.
$ svn propget svn:externals my-project/some-dir
subdir
http://svn.example.com/projects/my-project/external-dir

$
```

现在你使用 **svn move** 将目录 *my-project* 改名，此刻，你的外部定义还是指向 *my-project* 目录，即使这个目录已经不存在了。

```
$ svn move -q my-project renamed-project
```

```
$ svn commit -m "Rename my-project to renamed-project."
Deleting       my-project
Adding         my-renamed-project

Committed revision 12.
$ svn update

Fetching external item into 'renamed-project/some-dir/subdir'
svn: Target path does not exist
$
```

当然，如果版本库存在多种 URL 模式时，使用绝对 URL 来引用外部定义会导致问题。例如，如果你的 Subversion 服务器已经配置为任何用户可以使用 http://或 https://检出，但是只能通过 https://提交，你现在有了一个很有趣的问题。如果你的外部定义使用 http://形式，则你不能从这个工作拷贝提交任何内容。另一方面，如果他们使用 https://方式的 URL，任何因为不支持 https://的客户使用 http://检出的工作拷贝不能得到外部项目。也需要意识到，如果你需要重定位你的工作拷贝（使用 **svn switch --relocate**），外部定义不会重新定位。

最后，你或许经常希望 **svn** 子命令不会识别或其它作为外部定义处理的结果的外部工作拷贝上的操作，在这种情况下，你可以对子命令使用 --ignore-externals 选项。

Peg 和实施修订版本

文件和目录的拷贝、改名和移动能力使你可以创建一个项目，然后删除它，然后在同一个位置添加一个新的—这是在我们的计算机中经常发生的操作，而你的版本控制系统不应该成为你这样操作的障碍。

Subversion 的文件管理操作是这样的开放，提供了几乎和普通文件一样的操作版本化文件的灵活性，但是灵活意味着在整个版本库的生命周期中，一个给定的版本化的资源可能会出现在许多不同的路径，一个给定的路径会展示给我们许多完全不同的版本化资源。当然这些功能也增加了你与这些路径和资源交互的难度。

Subversion 可以非常聪明的注意到一个对象的包括一个“地址改变”历史变化，举个例子，如果你询问一个曾经上周改过名的文件的所有的日志信息，Subversion 会很高兴提供所有的日志—重命名发生的修订版本，外加相关版本之前和之后的修订版本日志，所以大多数时间里，你不需要考虑这些事情，但是偶尔，Subversion 会需要你的帮助来清除混淆。

这个最简单的例子发生在当一个目录或者文件从版本控制中删除时，然后一个新的同样名字目录或者文件添加到版本控制，显然你删除的和你后来添加的不是同样的东西，它们仅仅是有同样的路径，例如 */trunk/object*。什么，这意味着询问 Subversion 来查看*/trunk/object* 的历史？你是询问当前这个位置的东西还是你在这个位置删除的那个对象？你是希望询问对这个对象的所有操作还是这个路径的所有对象？很明显，Subversion 需要线索知道你真实的想法。

由于移动，版本化对象的历史会变得非常扭曲。举个例子，你会有一个目录叫做 *concept*，保存了一些你用来试验的初生的软件项目，最终，这个项目变得足够成熟，说明这个注意确实需要一些翅膀了，所以你决定给这个项目一个名字。[18]假定你叫你的软件为 Frabnaggilywort，此刻，把你的目录命名为反映项目名称的名字是有意义的，所以 *concept* 改名为 *frabnaggilywort*。生活还在继续，Frabnaggilywort 发布了 1.0 版本，并且被许多希望改进他们生活的分散用户天天使用。

这是一个美好的故事，但是没有在这里结束，作为主办人，你一定想到了另一件事，所以你创建了一个目录叫做 *concept*，周期重新开始。实际上，这个循环在几年里开始了多次，每一个想法从使用旧的 *concept* 目录开始，然后有时在想法成熟之后重新命名，有时你放弃了这个注意而删除了这个目录。或者更加变态一点，或许你把 *concept* 改成其他名字之后又因为一些原因重新改回 *concept*。

当这样的情景发生时，指导 Subversion 工作在重新使用的路径上的尝试就像指导一个芝加哥西郊的乘客驾车到东面的罗斯福路并且左转到主大道。仅仅 20 分钟，你可以穿过惠顿、格伦埃林何朗伯德的“主大道”，但是它们不是一样的街道，我们的乘客—和我们的 Subversion—需要更多的细节来做正确的事情。

在 1.1 版本，Subversion 提供了一种方法来说明你所指是哪一个街道，叫做 *peg 修订版本*，通过这个修订版本我们可以唯一确定一条历史线路，因为一个版本化的文件会在任何时间占用某个路径—路径和 peg 修订

版本的合并是可以指定一个历史的特定线路。Peg 修订版本可以在 Subversion 命令行客户端中用 *at 语法*指定，之所以使用这个名称是因为会在关联的修订版本的路径后面追加一个“at 符号”（@）。

但是我们在本书多次提到的--revision (-r)到底是什么？修订版本（或者是修订版本集）叫做*实施的修订版本*（或者叫做*实施的修订版本范围*），一旦一个特定历史线路通过一个路径和 peg 修订版本指定，Subversion 会使用实施的修订版本执行要求的操作。类似的，为了指出这个到我们芝加哥的道路，如果我们被告知到惠顿主大道 606 号，[19] 我们可以把“主大道”看作路径，把“惠顿”当作我们的 peg 修订版本。这两段信息确认了我们可以旅行（主大道的北方或南方）的唯一路径，也会保持我们不会在前前后后寻找目标时走到错误的主大道。现在我们把“606 N.”作为我们实施的修订版本，我们*精确的*知道到哪里。

Peg 版本算法

Subversion 命令行在解决路径和修订版本混淆时需要 peg 修订版本算法，这里是一个用以说明的例子：

```
$ svn command -r OPERATIVE-REV item@PEG-REV
```

如果 *OPERATIVE-REV* 比 *PEG-REV* 更老，则算法如下：

- 来到修订版本 *PEG-REV*，找到 *item*，在版本库定位到一个唯一的对象。

- 追踪对象的历史背景（通过任何可能的改名）来到修订版本 *OPERATIVE-REV* 的祖先。
- 对那个祖先执行请求的动作，无论它的位置，无论它是什么名字，无论当时是否存在。

但是如果 *OPERATIVE-REV* 比 *PEG-REV* 更年轻时会怎么样？这为定位 *OPERATIVE-REV* 中的路径的理论问题增加了一些复杂性，因为在 *PEG-REV* 和 *OPERATIVE-REV* 之间，路径在历史中可以出现多次（由于拷贝操作），而且那还不是全部—Subversion 不会保存向前跟踪历史的足够信息，所以算法会有一点不同：

- 来到修订版本 *PEG-REV*，找到 *item*，在版本库定位到一个唯一的对象。
- 追踪对象的历史背景（通过任何可能的改名）来到修订版本 *PEG-REV* 的祖先。
- 在 *PEG-REV* 中检验对象的位置（顺序）与在 *OPERATIVE-REV* 中相同，如果那是问题，则至少两个位置是直接关联的，所以在 *OPERATIVE-REV* 的位置执行请求动作。否则，关联没有建立，所以会报告没有可用位置的错误。（有一天，我们希望 Subversion 可以更灵活和优雅的处理这种场景。）

注意，即使你没有明确提供 peg 修订版本或操作修订版本，他们依然是存在的。为了使用的简便，对于工作拷贝项目的缺省 peg 修订版本是

BASE，而版本库 URL 的缺省值是 HEAD。当没有提供操作修订版本时，缺省是与 peg 修订版本一样。

也就是说很久以前我们创建了我们的版本库，在修订版本 1 添加我们第一个 *concept* 目录，并且在这个目录增加一个 *IDEA* 文件与 concept 相关，在几个修订版本之后，真实的代码被添加和修改，我们在修订版本 20，修改这个目录为 *frabnaggilywort*。通过修订版本 27，我们有了一个新的概念，所以一个新的 *concept* 目录用来保存这些东西，一个新的 *IDEA* 文件来描述这个概念，然后经过 5 年 20000 个修订版本，就像他们都有一个非常浪漫的历史。

现在，一年之后，我们想知道 *IDEA* 在修订版本 1 时是什么样子，但是 Subversion 需要知道我们是想询问*当前*文件在修订版本 1 时的样子，还是希望知道 *concepts/IDEA* 在修订版本 1 时的那个文件？确定这些问题有不同的答案，并且因为 peg 修订版本，你可以用两种方式询问。为了知道当前的 *IDEA* 文件在旧版本的样子，我们可以运行：

```
$ svn cat -r 1 concept/IDEA

svn: Unable to find repository location for 'concept/IDEA' in
revision 1
```

当然，在这个例子里，当前的 *IDEA* 文件在修订版本 1 中并不存在，所以 Subversion 给出一个错误，这个上面的命令是长的 peg 修订版本命令一个缩写，扩展的写法是：

```
$ svn cat -r 1 concept/IDEA@BASE
```

```
svn: Unable to find repository location for 'concept/IDEA' in
revision 1
```

当执行时，它包含期望的结果。

如果工作拷贝路径或 URL 中确实有一个 at 记号，peg 修订版本语法是否会导致问题？深刻理解的读者可能会产生这样的疑问。毕竟，**svn** 是如何知道 news@11 是我的目录树中的一个目录，还是修订版本 11 的 *news* 文件？幸好，**svn** 会一直假定后者。你只需要在路径最后添加一个 at 符号，例如 news@11@，**svn** 只关心最后一个 at 标记，如果遗漏了最后的修订版本号，不会认为不合法。这个法则甚至可以应用到以 at 结尾的路径—你可以使用 filename@@来引用 *filename@*。

然后让我们询问另一个问题—在修订版本 1 ， 占据 *concepts/IDEA* 路径的文件的内容到底是什么？我们会使用一个明确的 peg 修订版本来帮助我们完成。

```
$ svn cat concept/IDEA@1

The idea behind this project is to come up with a piece of software

that can frab a naggily wort.  Frabbing naggily worts is tricky

business, and doing it incorrectly can have serious
ramifications, so

we need to employ over-the-top input validation and data
verification

mechanisms.
```

注意我们这一次没有提供操作修订版本，那是因为如果没有指定操作修订版本，Subversion 假定缺省的操作修订版本是 peg 修订版本。

正像你看到的，这看起来是正确的输出，这些文本甚至提到“frabbing naggily worts”，所以这就是现在叫做 Frabnaggilywort 项目的那个文件，实际上，我们可以使用显示的 peg 修订版本和实施修订版本的组合核实这一点。我们知道在 HEAD，Frabnaggilywort 项目坐落在 *frabnaggilywort* 目录，所以我们指定我们希望看到 HEAD 的 *frabnaggilywort/IDEA* 路经在历史上的修订版本 1 的内容。

```
$ svn cat -r 1 frabnaggilywort/IDEA@HEAD

The idea behind this project is to come up with a piece of software

that can frab a naggily wort.  Frabbing naggily worts is tricky

business, and doing it incorrectly can have serious
ramifications, so

we need to employ over-the-top input validation and data
verification

mechanisms.
```

而且 peg 修订版本和实施修订版本也不需要这样琐碎，举个例子，我们的 *frabnaggilywort* 已经在 HEAD 删除，但我们知道在修订版本 20 它是存在的，我们希望知道 *IDEA* 从修订版本 4 到 10 的区别，我们可以使用 peg 修订版本 20 和 *IDEA* 文件的修订版本 20 的 URL 的组合，然后使用 4 到 10 作为我们的实施修订版本范围。

```
$ svn diff -r 4:10
http://svn.red-bean.com/projects/frabnaggilywort/IDEA@20
```

```
Index: frabnaggilywort/IDEA

===================================================================
=======

--- frabnaggilywort/IDEA  (revision 4)

+++ frabnaggilywort/IDEA  (revision 10)

@@ -1,5 +1,5 @@

-The idea behind this project is to come up with a piece of
software

-that can frab a naggily wort.  Frabbing naggily worts is tricky

-business, and doing it incorrectly can have serious
ramifications, so

-we need to employ over-the-top input validation and data
verification

-mechanisms.

+The idea behind this project is to come up with a piece of

+client-server software that can remotely frab a naggily wort.

+Frabbing naggily worts is tricky business, and doing it
incorrectly

+can have serious ramifications, so we need to employ
over-the-top

+input validation and data verification mechanisms.
```

幸运的是，几乎所有的人不会面临如此复杂的情形，但是如果是，记住 peg 修订版本是帮助 Subversion 清除混淆的额外提示。

网络模型

在某些情况下，你需要理解 Subversion 客户端如何与服务器通讯。Subversion 网络层是抽象的，意味着 Subversion 客户端不管其操作的对象都会使用相同的行为方式，不管是使用 HTTP 协议（http://）与 Apache HTTP 服务器通讯或是使用自定义 Subversion 协议（svn://）与 **svnserve** 通讯，基本的网络模型是相同的。在本小节，我们要解释网络模型基础，包括 Subversion 如何管理认证和授权信息。

请求和响应

Subversion 客户端花费大量的时间来管理工作拷贝，当它需要远程版本库的信息，它会做一个网络请求，然后服务器给一个恰当的回答，具体的网络协议细节对用户不可见，客户端尝试去访问一个 URL，根据 URL 模式的不同，会使用特定的协议与服务器联系（见版本库的 URL）。

用户可以运行 **svn --version** 来查看客户端可以使用的 URL 模式和协议。

当服务器处理一个客户端请求，它通常会要求客户端确定它自己的身份，它会发出一个认证请求给客户端，而客户端通过提供*凭证*给服务器作为响应，一旦认证结束，服务器会响应客户端最初请求的信息。注意这个系统与 CVS 之类的系统不一样，它们会在请求之前，预先提供凭证（“logs in”）给服务器，在 Subversion 里，服务器通过请求客户端适时地“拖入”凭证，而不是客户端“推”出，这使得这种操作更加的

优雅。例如，如果一个服务器配置为世界上的任何人都可以读取版本库，在客户使用 **svn checkout** 时，服务器永远不会发起一个认证请求。

如果客户端的请求会在版本库创建新的修订版本（例如 **svn commit**），Subversion 就会使用认证过的用户名作为此次提交的作者。也就是说经过认证的用户名作为 svn:author 属性的值保存到新的修订本里（见“Subversion 属性”一节）。如果客户端没有经过认证（换句话说，服务器没有发起过认证请求），这时修订本的 svn:author 的值是空的。

客户端凭证缓存

许多服务器配置为每次请求要求认证，对被强制每次输入用户名密码，许多用户会感到很讨厌。幸运的是，Subversion 客户端对此有一个修补—存在一个在磁盘上保存认证凭证缓存的系统，缺省情况下，当一个命令行客户端成功的响应了服务器的认证请求，它会保存一个认证文件到用户的私有运行配置区（类 Unix 系统下会在*~/.subversion/auth/*，Windows 下在*%APPDATA%/Subversion/auth/*，运行配置系统在“运行配置区”一节会有更多细节描述）。成功的凭证会缓存在磁盘，以主机名、端口和认证域的组合作为唯一性区别。

当客户端接收到一个认证请求，它会首先查找用户磁盘中的认证凭证缓存，如果没有发现，或者是缓存的凭证认证失败，客户端会提示用户提供需要的信息。

十分关心安全的人们一定会想“把密码缓存在磁盘？太可怕了，永远不要这样做！”

Subversion 开发者认识到这种关注的正确性，所以 Subversion 使用操作系统和环境提供的机制来减少泄露这些信息的风险，下面是在大多数平台上这种含义的列表：

- 在 Windows 2000 或更新的系统上，Subversion 客户端使用标准 Windows 加密服务来加密磁盘上的密码。因为加密密钥是 Windows 管理的，与用户的登陆凭证相关，只有用户可以解密密码。（注意：如果用户的 Windows 账户密码被管理员重置，所有的缓存密码就不可以解密了，此时 Subversion 客户端就会当它们根本不存在，在需要时继续询问密码。）
- 类似的，在 Mac OS X，Subversion 客户端在登陆 keyring（使用 Keychain 管理）保存了所有的版本库密码，使用户用帐号密码保护。用户选择的设置可以强加额外的政策，例如在需要用户密码时要求输入用户帐号密码。
- 对于其他类 Unix 系统，没有标准的加密服务。然而 *auth/*缓存区只有用户（拥有者）可以访问，而不是全世界都可以，操作系统的访问许可可以保护密码文件。

当然，对于真正的妄想狂，没有任何机制是完美的。这类人希望用无限的安全来牺牲便利性，Subversion 提供了各种方法来完全关闭凭证缓存。

你可以关闭凭证缓存，只需要一个简单的命令，使用参数--no-auth-cache：

```
$ svn commit -F log msg.txt --no-auth-cache

Authentication realm: <svn://host.example.com:3690> example
realm

Username:  joe

Password for 'joe':



Adding         newfile

Transmitting file data .

Committed revision 2324.



# password was not cached, so a second commit still prompts us



$ svn delete newfile

$ svn commit -F new msg.txt

Authentication realm: <svn://host.example.com:3690> example
realm

Username:  joe

…
```

或许，你希望永远关闭凭证缓存，你可以编辑你的运行运行配置区的 *config* 文件，只需要把 store-auth-creds 设置为 no，这样在影响的主机上的 Subversion 操作就不会有凭证缓存在磁盘。通过修改系统级的运行配置区，这个功能也会影响到本机的所有用户（详细内容见“配置区布局”一节）。

```
[auth]

store-auth-creds = no
```

有时候，用户希望从磁盘缓存删除特定的凭证，为此你可以浏览到 *auth/* 区域，删除特定的缓存文件，凭证都是作为一个单独的文件缓存，如果你打开每一个文件，你会看到键和值，svn:realmstring 描述了这个文件关联的特定服务器的域：

```
$ ls ~/.subversion/auth/svn.simple/

5671adf2865e267db74f09ba6f872c28

3893ed123b39500bca8a0b382839198e

5c3c22968347b390f349ff340196ed39



$ cat
~/.subversion/auth/svn.simple/5671adf2865e267db74f09ba6f872c
28



K 8

username

V 3

joe

K 8

password

V 4

blah

K 15
```

```
svn:realmstring

V 45

<https://svn.domain.com:443> Joe's repository

END
```

一旦你定位了正确的缓存文件，只需要删除它。

svn--username--password--username--passwordsvn

这里是Subversion客户端在收到认证请求的时候的行为方式最终总结：

1. 首先，检查用户是否通过命令选项（--username 和/或--password）指定了任何凭证信息，如果没有，或者这些选项没有认证成功，然后
2. 查找运行中的 *auth/*区域保存的服务器名，端口和认证域信息，来确定用户是否已经有了恰当的认证缓存，如果没有，或者缓存凭证认证失败，然后
3. 最终，客户端返回要求用户（除非使用--non-interactive 选项或客户端对等的方式）。

如果客户端通过以上的任何一种方式成功认证，它会尝试在磁盘缓存凭证（除非用户已经关闭了这种行为方式，在前面提到过。）

---

[9] 如果你熟悉 XML，其实这就是 XML 的"Name"语法的 ASCII 子集。

[10] 修正提交日志信息的拼写错误，文法错误和“简单的错误”是 --revprop 选项最常见用例。

[11] 你认为那样过于粗狂？在同一个时代里，WordPerfect 也使用*.DOC* 作为它们私有文件格式的扩展名！

[12] Windows 文件系统使用文件扩展名（如.EXE、.BAT 和.COM）来标示可执行文件。

[13] 这不是编译系统的基本功能吗？

[14] … 或者可能是一本书的一个小节 …

[15] Communication wouldn't have been such bad medicine for Harry and Sally's Hollywood namesakes, either, for that matter.

[16] Subversion 目前不允许锁定目录。

[17] 除非是，或许一个经典的火神精神融合。

[18] “你不是被期望去命名它，一旦你取了名字，你开始与之联系在一起。” — Mike Wazowski

[19] 伊利诺伊州惠顿主大道 606 号市惠顿离市中心，让它作为—“历史中心”？看起来是恰当的…。

分支与合并

目录

常规的供方分支管理过程

svn load dirs.pl

总结

“君子务本”

--孔子

分支、标签和合并是所有版本控制系统的共同概念，如果你并不熟悉这些概念，我们会在这一章里很好的介绍，如果你很熟悉，非常希望你有兴趣知道 Subversion 是怎样实现这些概念的。

分支是版本控制的基础组成部分，如果你允许 Subversion 来管理你的数据，这个特性将是你所必须依赖的，这一章假定你已经熟悉了 Subversion 的基本概念（第 1 章 *基本概念*）。

什么是分支？

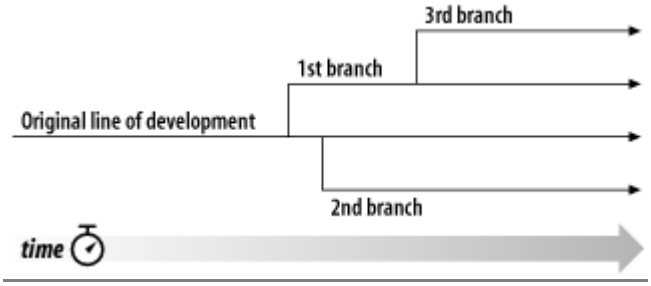
假设你的工作是维护本公司一个部门的手册文档，一天，另一个部门问你要相同的手册，但一些地方会有“区别”，因为他们有不同的需要。

这种情况下你会怎样做？显而易见的方法是：作一个版本的拷贝，然后分别维护两个版本，只要任何一个部门告诉要做一些小修改，你必须选择在对应的版本进行更改。

你也许希望在两个版本同时作修改，举个例子，你在第一个版本发现了一个拼写错误，很显然这个错误也会出现在第二个版本里。两份文档几乎相同，毕竟，只有许多特定的微小区别。

这是*分支*的基本概念—正如它的名字，开发的一条线独立于另一条线，如果回顾历史，可以发现两条线分享共同的历史，一个分支总是从一个备份开始的，从那里开始，发展自己独有的历史（见 图 4.1 “分支与开发”）。

图 4.1. 分支与开发



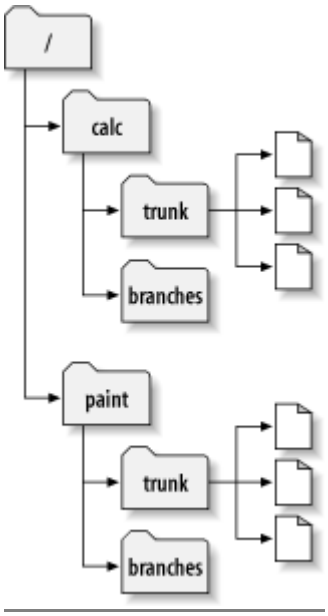
Subversion 允许你并行的维护文件和目录的分支，它允许你通过拷贝数据建立分支，记住，分支互相联系，它也帮助你从一个分支复制修改到另一个分支。最终，它可以让你的工作拷贝反映到不同的分支上，所以你在日常工作可以“混合和比较”不同的开发线。

使用分支

在这一点上，你必须理解每一次提交是怎样建立整个新的文件系统树（叫做“修订版本”）的，如果没有，可以回头去读“修订版本”一节。

对于本章节，我们会回到第 1 章 *基本概念*的同一个例子，还记得你和你的合作者 Sally 分享一个包含两个项目的版本库，*paint* 和 *calc*。注意 图 4.2 “开始规划版本库”，然而，现在每个项目的都有一个 *trunk* 和 *branches* 子目录，它们存在的理由很快就会清晰起来。

图 4.2. 开始规划版本库



像以前一样，假定 Sally 和你都有“calc”项目的一份拷贝，更准确地说，你有一份*/calc/trunk* 的工作拷贝，这个项目的所有的文件在这个子目录里，而不是在*/calc* 下，因为你的小组决定使用*/calc/trunk* 作为开发使用的“主线”。

假定你有一个任务，将要对项目做基本的重新组织，这需要花费大量时间来完成，会影响项目的所有文件，问题是你不会希望打扰 Sally，她正在处理这样或那样的程序小 Bug，一直使用整个项目（*/calc/trunk*）的最新版本，如果你一点一点的提交你的修改，你一定会干扰 Sally 的工作。

一种策略是自己闭门造车：你和 Sally 可以停止一个到两个星期的共享，也就是说，开始作出本质上的修改和重新组织工作拷贝的文件，但是在完成这个任务之前不做提交和更新。这样会有很多问题，首先，这样并

不安全，许多人习惯频繁的保存修改到版本库，工作拷贝一定有许多意外的修改。第二，这样并不灵活，如果你的工作在不同的计算机（或许你在不同的机器有两份*/calc/trunk* 的工作拷贝），你需要手工的来回拷贝修改，或者只在一个计算机上工作，这时很难做到共享你即时的修改，一项软件开发的“最佳实践”就是允许审核你做过的工作，如果没有人看到你的提交，你失去了潜在的反馈。最后，当你完成了公司主干代码的修改工作，你会发现合并你的工作拷贝和公司的主干代码会是一件非常困难的事情，Sally（或者其他人）也许已经对版本库做了许多修改，已经很难和你的工作拷贝结合—当你单独工作几周后运行 **svn update** 时就会发现这一点。

最佳方案是创建你自己的分支，或者是版本库的开发线。这允许你保存破坏了一半的工作而不打扰别人，尽管你仍可以选择性的同你的合作者分享信息，你将会看到这是怎样工作的。

创建分支

建立分支非常的简单—使用 **svn copy** 命令给你的工程做个拷贝，Subversion 不仅可以拷贝单个文件，也可以拷贝整个目录，在目前情况下，你希望作*/calc/trunk* 的拷贝，新的拷贝应该在哪里？在你希望的任何地方—它只是在于项目的政策，我们假设你们项目的政策是在*/calc/branches* 建立分支，并且你希望把你的分支叫做 my-calc-branch，你希望建立一个新的目录*/calc/branches/my-calc-branch*，作为/calc/trunk 的拷贝开始它的生命周期。

有两个方法作拷贝，我们先介绍一个混乱的方法，只是让概念更清楚，首先取出一个项目的根目录，*/calc*：

```
$ svn checkout http://svn.example.com/repos/calc bigwc
A  bigwc/trunk/
A  bigwc/trunk/Makefile
A  bigwc/trunk/integer.c
A  bigwc/trunk/button.c
A  bigwc/branches/
Checked out revision 340.
```

建立一个备份只是传递两个目录参数到 **svn copy** 命令：

```
$ cd bigwc
$ svn copy trunk branches/my-calc-branch
$ svn status
A  +   branches/my-calc-branch
```

在这个情况下，**svn copy** 命令迭代的将 *trunk* 工作目录拷贝到一个新的目录 branhes/my-calc-branch，像你从 **svn status** 看到的，新的目录是准备添加到版本库的，但是也要注意 A 后面的“+”号，这表明这个准备添加的东西是一份*备份*，而不是新的东西。当你提交修改，Subversion 会通过拷贝*/calc/trunk* 建立*/calc/branches/my-calc-branch* 目录，而不是通过网络传递所有数据：

```
$ svn commit -m "Creating a private branch of /calc/trunk."
```

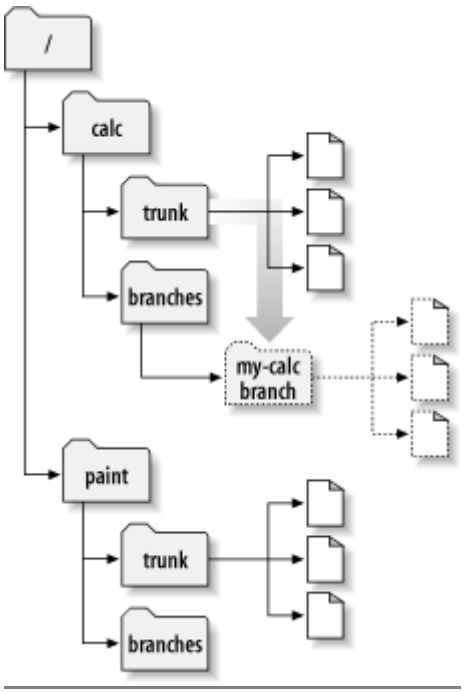
```
Adding         branches/my-calc-branch

Committed revision 341.
```

现在，我们必须告诉你建立分支最简单的方法：**svn copy** 可以直接对两个 URL 操作。

```
$ svn copy http://svn.example.com/repos/calc/trunk \

          
http://svn.example.com/repos/calc/branches/my-calc-branch \

      -m "Creating a private branch of /calc/trunk."



Committed revision 341.
```

从版本库的视点来看，其实这两种方法没有什么区别，两个过程都在版本 341 建立了一个新目录作为*/calc/trunk* 的一个备份，这些可以在图 4.3 “版本库与复制”看到，注意第二种方法，只是执行了一个 *立即*提交。[20] 这是一个简单的过程，因为你不需要取出版本库一个庞大的镜像，事实上，这个技术不需要你有工作拷贝，这是大多数用户创建分支的方式。

图 4.3. 版本库与复制



廉价复制

Subversion 的版本库有特殊的设计，当你复制一个目录，你不需要担心版本库会变得十分巨大—Subversion 并不是拷贝所有的数据，相反，它建立了一个*已存在*目录树的入口，如果你是 Unix 用户，可以把它理解成硬链接，在对拷贝的文件和目录操作之前，Subversion 还仅仅把它当作硬链接，只有为了区分不同版本的对象时才会复制数据。

这就是为什么经常听到 Subversion 用户谈论“廉价的拷贝”，与目录的大小无关—这个操作会使用很少的时间，事实上，这个特性是 Subversion 提交工作的基础：每一次版本都是前一个版本的一个“廉价的拷贝”，只有少数项目修改了。（要阅读更多关于这部分的内容，访问 Subversion 网站并且阅读设计文档中的“bubble up”方法）。

当然，拷贝与分享的内部机制对用户来讲是不可见的，用户只是看到拷贝树，这里的要点是拷贝的时间与空间代价很小。如果你完全在版本库里创建分支（通过运行 **svn copy URL1 URL2**），这是一个快速的，时间基本固定的操作，只要你希望，可以随意创建分支。

在分支上工作

现在你已经在项目里建立分支了，你可以取出一个新的工作拷贝来开始使用：

```
$ svn checkout
http://svn.example.com/repos/calc/branches/my-calc-branch

A  my-calc-branch/Makefile

A  my-calc-branch/integer.c

A  my-calc-branch/button.c

Checked out revision 341.
```

这一份工作拷贝没有什么特别的，它只是版本库另一个目录的一个镜像罢了，当你提交修改时，Sally 在更新时不会看到改变，她是*/calc/trunk* 的工作拷贝。（确定要读本章后面的“使用分支”一节：**svn switch** 命令是建立分支工作拷贝的另一个选择。）

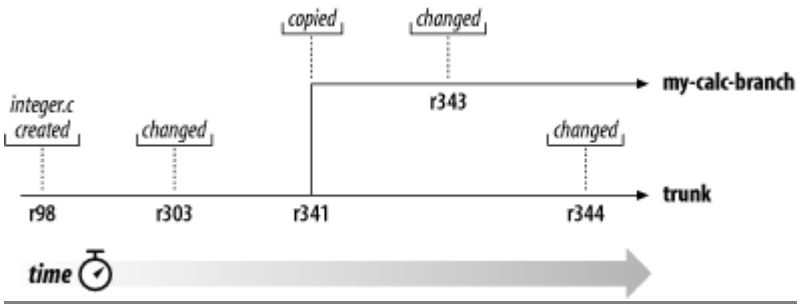
我们假定本周就要过去了，如下的提交发生：

- 你修改了*/calc/branches/my-calc-branch/button.c*，生成修订版本 342。
- 你修改了*/calc/branches/my-calc-branch/integer.c*，生成修订版本 343。

- Sally 修改了*/calc/trunk/integer.c*，生成了修订版本 344。

现在有两个独立开发线，图 4.4 “一个文件的分支历史”显示了 *integer.c* 的历史。

图 4.4. 一个文件的分支历史



当你看到 *integer.c* 的改变时，你会发现很有趣：

```
$ pwd
/home/user/my-calc-branch

$ svn log -v integer.c
------------------------------------------------------------------------
r343 | user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/branches/my-calc-branch/integer.c

* integer.c:  frozzled the wazjub.
```

```
------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

Creating a private branch of /calc/trunk.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  adding this file to the project.
```

```
------------------------------------------------------------------------
------------
```

注意，Subversion 追踪分支上的 *integer.c* 的历史，包括所有的操作，甚至追踪到拷贝之前。这表示了建立分支也是历史中的一次事件，因为在拷贝整个*/calc/trunk/*时已经拷贝了一份 *integer.c*。现在看 Sally 在她的工作拷贝运行同样的命令：

```
$ pwd

/home/sally/calc



$ svn log -v integer.c

------------------------------------------------------------------------
------------

r344 | sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) |
2 lines

Changed paths:

   M /calc/trunk/integer.c



* integer.c:  fix a bunch of spelling errors.



------------------------------------------------------------------------
------------

r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) |
2 lines

Changed paths:

   M /calc/trunk/integer.c
```

```
* integer.c:  changed a docstring.



------------------------------------------------------------------------

r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines

Changed paths:

   M /calc/trunk/integer.c



* integer.c:  adding this file to the project.



------------------------------------------------------------------------
```

sally 看到她自己的 344 修订，你做的 343 修改她看不到，从 Subversion 看来，两次提交只是影响版本库中不同位置上的两个文件。然而，Subversion *显示*了两个文件有共同的历史，在分支拷贝之前，他们使用同一个文件，所以你和 Sally 都看到版本号 303 到 98 的修改。

分支背后的关键概念

在本节，你需要记住两件重要的课程。首先，Subversion 并没有内在的分支概念—只有拷贝，当你拷贝一个目录，这个结果目录就是一个“分支”，只是因为你给了它这样一个含义而已。你可以换一种角度考虑，或者特别对待，但是对于 Subversion 它只是一个普通的拷贝，只不过碰巧包含了一些额外的历史信息。第二，因为拷贝机制，Subversion

的分支是以*普通文件系统目录*存在的，这与其他版本控制系统不同，它们都为分支定义了另一维度的“标签”。

在分支间复制修改

现在你与 Sally 在同一个项目的并行分支上工作：你在私有分支上，而 Sally 在主干（*trunk*）或者叫做开发主线上。

由于有众多的人参与项目，大多数人拥有主干拷贝是很正常的，任何人如果进行一个长周期的修改会使得主干陷入混乱，所以通常的做法是建立一个私有分支，提交修改到自己的分支，直到这阶段工作结束。

所以，好消息就是你和 Sally 不会互相打扰，坏消息是有时候分离会太远。记住“闭门造车”策略的问题，当你完成你的分支后，可能因为太多冲突，已经无法轻易合并你的分支和主干的修改。

相反，在你工作的时候你和 Sally 仍然可以继续分享修改，这依赖于你决定什么值得分享，Subversion 给你在分支间选择性“拷贝”修改的能力，当你完成了分支上的所有工作，所有的分支修改可以被拷贝回到主干。

复制特定的修改

在上一章节，我们提到你和 Sally 对 *integer.c* 在不同的分支上做过修改，如果你看了 Sally 的 344 版本的日志信息，你会知道她修正了一些拼写错误，毋庸置疑，你的拷贝的文件也一定存在这些拼写错误，所以你以

后的对这个文件修改也会保留这些拼写错误，所以你会在将来合并时得到许多冲突。最好是现在接收 Sally 的修改，而不是作了许多工作之后才来做。

是时间使用 **svn merge** 命令，这个命令的结果非常类似 **svn diff** 命令（在第 2 章 *基本使用*的内容），两个命令都可以比较版本库中的任何两个对象并且描述其区别，举个例子，你可以使用 **svn diff** 来查看 Sally 在版本 344 作的修改：

```
$ svn diff -c 344 http://svn.example.com/repos/calc/trunk


Index: integer.c
===================================================================
--- integer.c       (revision 343)
+++ integer.c       (revision 344)
@@ -147,7 +147,7 @@
     case 6:  sprintf(info->operating system, "HPFS (OS/2 or
NT)"); break;
     case 7:  sprintf(info->operating system, "Macintosh");
break;
     case 8:  sprintf(info->operating system, "Z-System");
break;
-    case 9:  sprintf(info->operating system, "CPM"); break;
+    case 9:  sprintf(info->operating system, "CP/M"); break;
     case 10:  sprintf(info->operating system, "TOPS-20");
break;
```

```
    case 11:  sprintf(info->operating system, "NTFS (Windows
NT)"); break;

    case 12:  sprintf(info->operating system, "QDOS"); break;

@@ -164,7 +164,7 @@

    low = (unsigned short) read byte(gzfile);  /* read LSB */

    high = (unsigned short) read byte(gzfile); /* read MSB */

    high = high << 8;  /* interpret MSB correctly */

-    total = low + high; /* add them togethe for correct total
*/

+    total = low + high; /* add them together for correct total
*/



    info->extra header = (unsigned char *) my malloc(total);

    fread(info->extra header, total, 1, gzfile);

@@ -241,7 +241,7 @@

     Store the offset with ftell() ! */



  if ((info->data offset = ftell(gzfile))== -1) {

-    printf("error: ftell() retturned -1.\n");

+    printf("error: ftell() returned -1.\n");

    exit(1);

  }



@@ -249,7 +249,7 @@

  printf("I believe start of compressed data is %u\n",
info->data offset);

  #endif
```

```
-  /* Set postion eight bytes from the end of the file. */
+  /* Set position eight bytes from the end of the file. */

   if (fseek(gzfile, -8, SEEK_END)) {
     printf("error: fseek() returned non-zero\n");
```

svn merge 命令几乎完全相同，但不是打印区别到你的终端，它会直接作为*本地修改*作用到你的本地拷贝：

```
$ svn merge -c 344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
M  integer.c
```

svn merge 的输出告诉你的 *integer.c* 文件已经作了补丁（patched），现在已经保留了 Sally 修改—修改从主干“拷贝”到你的私有分支的工作拷贝，现在作为一个本地修改，在这种情况下，要靠你审查本地的修改来确定它们工作正常。

在另一种情境下，事情并不会运行得这样正常，也许 *integer.c* 也许会进入冲突状态，你必须使用标准过程（见第 2 章 *基本使用*）来解决这种状态，或者你认为合并是一个错误的决定，你只需要运行 **svn revert** 放弃本地修改。

但是当你审查过你的合并结果后，你可以使用 **svn commit** 提交修改，在那一刻，修改已经合并到你的分支上了，在版本控制术语中，这种在分支之间拷贝修改的行为叫做*搬运*修改。

当你提交你的修改时，确定你的日志信息中说明你是从某一版本搬运了修改，举个例子：

```
$ svn commit -m "integer.c: ported r344 (spelling fixes) from
trunk."

Sending        integer.c

Transmitting file data .

Committed revision 360.
```

你将会在下一节看到，这是一条非常重要的“最佳实践”。

为什么不使用补丁？

也许你的脑中会出现一个问题，特别如果你是 Unix 用户，为什么非要使用 **svn merge**？为什么不简单的使用操作系统的 **patch** 命令来进行相同的工作？例如：

```
$ svn diff -c 344
http://svn.example.com/repos/calc/trunk > patchfile

$ patch -p0  < patchfile

Patching file integer.c using Plan A...

Hunk #1 succeeded at 147.

Hunk #2 succeeded at 164.

Hunk #3 succeeded at 241.
```

```
Hunk #4 succeeded at 249.

done
```

在这种情况下，确实没有区别，但是 **svn merge** 有超越 **patch** 的特别能力，使用 **patch** 对文件格式有一定的限制，它只能针对文件内容，没有方法表现*目录树*的修改，例如添加、删除或是改名。如果 Sally 的修改包括增加一个新的目录，**svn diff** 不会注意到这些，**svn diff** 只会输出有限的补丁格式，所以有些问题无法表达。 但是 **svn merge** 命令会通过直接作用你的工作拷贝来表示目录树的结构和属性变化。

一个警告：为什么 **svn diff** 和 **svn merge** 在概念上是很接近，但语法上有许多不同，一定阅读第 9 章 *Subversion 完全参考*来查看其细节或者使用 **svn help** 查看帮助。举个例子，**svn merge** 需要一个工作拷贝作为目标，就是一个地方来施展目录树修改，如果一个目标都没有指定，它会假定你要做以下某个普通的操作：

1. 你希望合并目录修改到工作拷贝的当前目录。
2. 你希望合并修改到你的当前工作目录的相同文件名的文件。

如果你合并一个目录而没有指定特定的目标，**svn merge** 假定第一种情况，在你的当前目录应用修改。如果你合并一个文件，而这个文件（或是一个有相同的名字文件）在你的当前工作目录存在，**svn merge** 假定第二种情况，你想对这个同名文件使用合并。

如果你希望修改应用到别的目录，你需要说出来。举个例子，你在工作拷贝的父目录，你需要指定目标目录：

```
$ svn merge -c 344 http://svn.example.com/repos/calc/trunk
my-calc-branch

U   my-calc-branch/integer.c
```

合并背后的关键概念

你已经看到了 **svn merge** 命令的例子，你将会看到更多，如果你对合并是如何工作的感到迷惑，这并不奇怪，很多人和你一样。许多新用户（特别是对版本控制很陌生的用户）会对这个命令的正确语法感到不知所措，不知道怎样和什么时候使用这个特性，不要害怕，这个命令实际上比你想象的简单！有一个简单的技巧来帮助你理解 **svn merge** 的行为。

迷惑的主要原因是这个命令的*名称*，术语“合并”不知什么原因被用来表明分支的组合，或者是其他什么神奇的数据混合，这不是事实，一个更好的名称应该是 **svn diff-and-apply**，这是发生的所有事件：首先两个版本库树比较，然后将区别应用到本地拷贝。

这个命令包括三个参数：

1. 初始的版本树（通常叫做比较的*左边*），
2. 最终的版本树（通常叫做比较的*右边*），
3. 一个接收区别的工作拷贝（通常叫做合并的*目标*）。

一旦这三个参数指定以后，两个目录树将要做比较，比较结果将会作为本地修改应用到目标工作拷贝，当命令结束后，结果同你手工修改或者是使用 **svn add** 或 **svn delete** 没有什么区别，如果你喜欢这结果，你可以提交，如果不喜欢，你可以使用 **svn revert** 恢复修改。

svn merge 的语法允许非常灵活的指定三个必要的参数，如下是一些例子：

```
$ svn merge http://svn.example.com/repos/branch1@150 \
           http://svn.example.com/repos/branch2@212 \
           my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
```

第一种语法使用 *URL@REV* 的形式直接列出了所有参数，第二种语法可以用来作为比较同一个 URL 的不同版本的简略写法，最后一种语法表示工作拷贝是可选的，如果省略，默认是当前目录。

合并的最佳实践

手工跟踪合并

合并修改听起来很简单，但是实践起来会是很头痛的事，如果你重复合并两个分支，你也许会合并*两次*同样的修改。当这种事情发生时，有时

候事情会依然正常，当对文件打补丁时，Subversion 如果注意到这个文件已经有了相应的修改，而不会作任何操作，但是如果已经应用的修改又被修改了，你会得到冲突。

理想情况下，你的版本控制系统应该会阻止对一个分支做两次改变操作，必须自动的记住那一个分支的修改已经接收了，并且可以显示出来，用来尽可能帮助自动化的合并。

不幸的是，Subversion 不是这样一个系统，类似于 CVS，Subversion 并不记录任何合并操作，[21]当你提交本地修改，版本库并不能判断出你是通过 **`svn merge`** 还是手工修改得到这些文件。

这对你这样的用户意味着什么？这意味着除非 Subversion 以后发展这个特性，你必须手工的记录这些信息。最佳的方式是使用提交日志信息，像前面的例子提到的，推荐你在日志信息中说明合并的特定版本号（或是版本号的范围），之后，你可以运行 **`svn log`** 来查看你的分支包含哪些修改。这可以帮助你小心的依序运行 **`svn merge`** 命令而不会进行多余的合并。

在下一小节，我们要展示一些这种技巧的例子。

预览合并

首先，一定要记住合并的工作拷贝*没有*本地更改，并且最近已更新过。如果你的工作拷贝用这样的方法“清理”，你会发现一些头痛的事情。

因为合并只是导致本地修改，它不是一个高风险的操作，如果你在第一次操作错误，你可以运行 **svn revert** 来再试一次。

有时候你的工作拷贝很可能已经改变了，合并会针对存在的那一个文件，这时运行 **svn revert** 不会恢复你在本地作的修改，两部分的修改无法识别出来。

在这个情况下，人们很乐意能够在合并之前预测一下，一个简单的方法是使用运行 **svn merge** 同样的参数运行 **svn diff**，另一种方式是传递--dry-run 选项给 merge 命令来预览：

```
$ svn merge --dry-run -c 344
http://svn.example.com/repos/calc/trunk

U  integer.c



$ svn status

#  nothing printed, working copy is still unchanged.
```

--dry-run 选项实际上并不修改本地拷贝，它只是显示实际合并时的状态信息，对于得到潜在合并的“整体”预览，这个命令很有用，因为 **svn diff** 包括太多细节。

合并冲突

就像 **svn update** 命令，**svn merge** 会把修改应用到工作拷贝，因此它也会造成冲突，因为 **svn merge** 造成的冲突有时候会有些不同，本小节会解释这些区别。

作为开始，我们假定本地没有修改，当你 **svn update** 到一个特定修订版本时，修改会“干净的”应用到工作拷贝，服务器产生比较两树的增量数据：一个工作拷贝和你关注的版本树的虚拟快照，因为比较的左边同你拥有的完全相同，增量数据确保你把工作拷贝转化到右边的树。

但是 **svn merge** 没有这样的保证，会导致很多的混乱：用户可以询问服务器比较*任何*两个树，即使一个与工作拷贝毫不相关的！这意味着有潜在的人为错误，用户有时候会比较两个错误的树，创建的增量数据不会干净的应用，**svn merge** 会尽力应用更多的增量数据，但是有一些部分也许会难以完成，就像 Unix 下 **patch** 命令有时候会报告“failed hunks”错误，**svn merge** 会报告“skipped targets”：

```
$ svn merge -r 1288:1351 http://svn.example.com/repos/branch

U  foo.c

U  bar.c

Skipped missing target: 'baz.c'

U  glub.c

C  glorb.h



$
```

在前一个例子中，*baz.c* 也许会存在于比较的两个分支快照里，但工作拷贝里不存在，比较的增量数据要应用到这个文件，这种情况下会发生什么？“skipped”信息意味着用户可能是在比较错误的两棵树，这是经典的用户错误，当发生这种情况，可以使用迭代恢复（**svn revert**

--recursive）合并所作的修改，删除恢复后留下的所有未版本化的文件和目录，并且使用另外的参数运行 **svn merge**。

也应当注意前一个例子显示 *glorb.h* 发生了冲突，我们已经规定本地拷贝没有修改：冲突怎么会发生呢？因为用户可以使用 **svn merge** 将过去的任何变化应用到当前工作拷贝，变化包含的文本修改也许并不能干净的应用到工作拷贝文件，即使这些文件没有本地修改。

另一个 **svn update** 和 **svn merge** 的小区别是冲突产生的文件的名字不同，在“解决冲突（合并别人的修改）”一节，我们看到过更新产生的文件名字为 *filename.mine*、*filename.rOLDREV* 和 *filename.rNEWREV*，当 **svn merge** 产生冲突时，它产生的三个文件分别为 *filename.working*、*filename.left* 和 *filename.right*。在这种情况下，术语“left”和“right”表示了两棵树比较时的两边，在两种情况下，不同的名字会帮助你区分冲突是因为更新造成的还是合并造成的。

关注还是忽视祖先

当与 Subversion 开发者交谈时你一定会听到提及术语 *祖先*，这个词是用来描述两个对象的关系：如果他们互相关联，一个对象就是另一个的祖先，或者相反。

举个例子，假设你提交版本 100，包括对 *foo.c* 的修改，则 foo.c@99 是 foo.c@100 的一个“祖先”，另一方面，假设你在版本 101 删除这个文件，而在 102 版本提交一个同名的文件，在这个情况下，*foo.c@99*

与 *foo.c@102* 看起来是关联的（有同样的路径），但是事实上他们是完全不同的对象，它们并不共享同一个历史或者说“祖先”。

指出 **svn diff** 和 **svn merge** 区别的重要性在于，前一个命令忽略祖先，如果你询问 **svn diff** 来比较文件 *foo.c* 的版本 99 和 102，你会看到行为基础的区别，diff 命令只是盲目的比较两条路径，但是如果你使用 **svn merge** 是比较同样的两个对象，它会注意到他们是不关联的，而且首先尝试删除旧文件，然后添加新文件，输出会是一个删除紧接着一个增加：

```
D  foo.c

A  foo.c
```

大多数合并包括比较包括祖先关联的两条树，因此 **svn merge** 这样运作，然而，你也许会希望 merge 命令能够比较两个不相关的目录树，举个例子，你有两个目录树分别代表了供应方软件项目的不同版本（见“供方分支”一节），如果你使用 **svn merge** 进行比较，你会看到第一个目录树被删除，而第二个树添加上！在这个情况下，你仅仅是希望 **svn merge** 以路径为基础比较两棵树，而忽略文件和目录的不相关性，当为合并命令添加--ignore-ancestry 选项时，就会像 **svn diff** 一样工作。（相反，--notice-ancestry 会导致 **svn diff** 像 merge 命令一样工作。）

合并和移动

一个普遍的愿望是重构源程序，特别是 Java 软件项目。在改名中文件和目录变乱，通常导致每个项目成员的极大破坏。听起来好像应该使用分支，不是吗？只是创建分支，变乱事情，然后合并回主干，不对吗？

唉，这个场景下这样并不正确，可以看作 Subversion 当前的弱点，这个问题是因为 Subversion 的 **update** 还不是足够的强壮，特别是针对拷贝和移动操作。

当你使用 **svn copy** 复制文件时，版本库会记住新文件的出处，但是它不能将这个信息传递给使用 **svn update** 或 **svn merge** 的客户端，不是告诉客户端“ 将文件拷贝到新的位置”，而是传递一整个新文件。这样会导致问题，特别是因为这件事也发生在改名的文件。 一个鲜为人知的事实是 Subversion 缺乏真正的重命名—**svn move** 命令只是一个 **svn copy** 和 **svn delete** 的组合。

例如，假定我们在一个私有分支工作，你将 *integer.c* 改名为 *whole.c*，你这是在分支上创建了原来文件的一个拷贝，并且删除了原来的文件。同时， 回到 *trunk*， Sally 提交了一些 *integer.c* 的修改，所以你需要将分支合并到主干：

```
$ cd calc/trunk



$ svn merge -r 341:405
http://svn.example.com/repos/calc/branches/my-calc-branch

D   integer.c
```

```
A   whole.c
```

第一眼看起来不是很差，但是很可能这不是你和 Sally 希望的，合并操作已经删除了最新版本的 *integer.c*（包含了 Sally 最新的修改），而且盲目的添加了你的 *whole.c* 文件—是旧版本的 *integer.c* 复制品。最终的结果是将你的“rename”合并到分支，并且从最新修订版本删除了 Sally 最近的修改。

这不是真的数据丢失；Sally 的修改还在版本库的历史中，但是。在 Subversion 改进之前，最好小心对分支进行合并和改名。

常见用例

分支和 **svn merge** 有很多不同的用法，这个小节描述了最常见的用法。

合并分支到另一分支

为了完成这个例子，我们将时间往前推进，假定已经过了几天，在主干和你的分支上都有许多更改，假定你完成了分支上的工作，已经完成了特性或 bug 修正，你想合并所有分支的修改到主干上，让别人也可以使用。

这种场景下如何使用 **svn merge**？记住这个命令比较两个目录树，然后应用比较结果到工作拷贝，所以要接受这种变化，你需要主干的工作拷贝，我们假设你有一个最初的主干工作拷贝（完全更新），或者是你最近取出了*/calc/trunk* 的一个干净的工作拷贝。

但是要哪两个树进行比较呢？乍一看，回答很明确，只要比较最新的主干与分支。但是你要意识到—这个想法是*错误的*，伤害了许多新用户！因为 **svn merge** 的操作很像 **svn diff**，比较最新的主干和分支树不仅仅会描述你在分支上所作的修改，这样的比较会展示太多的不同，不仅包括分支上的增加，也包括了主干上的删除操作，而这些删除根本就没有在分支上发生过。

为了表示你的分支上的修改，你只需要比较分支的初始状态与最终状态，在你的分支上使用 **svn log** 命令，你可以看到你的分支在 341 版本建立，你的分支最终的状态用 HEAD 版本表示，这意味着你希望能够比较版本 341 和 HEAD 的分支目录，然后应用这些分支的修改到主干目录的工作拷贝。

查找分支产生的版本（分支的“基准”）的最好方法是在 **svn log** 中使用--stop-on-copy 选项，log 子命令通常会显示所有关于分支的变化，包括创建分支的过程，就好像你在主干上一样，--stop-on-copy 会在 **svn log** 检测到目标拷贝或者改名时中止日志输出。

所以，在我们的例子里，

```
$ svn log -v --stop-on-copy \

_____
http://svn.example.com/repos/calc/branches/my-ca
lc-branch

…
```

```
------------------------------------------------------------------------

r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines

Changed paths:

   A /calc/branches/my-calc-branch (from /calc/trunk:340)



$
```

正如所料，最后打印出的版本正是由 *my-calc-branch* 拷贝生成的版本。

如下是最终的合并过程，然后：

```
$ cd calc/trunk

$ svn update

At revision 405.



$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch

U   integer.c

U   button.c

U   Makefile



$ svn status

M   integer.c

M   button.c

M   Makefile
```

```
# ...examine the diffs, compile, test, etc...

$ svn commit -m "Merged my-calc-branch changes r341:405 into the
trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 406.
```

再次说明，日志信息中详细描述了合并到主干的的修改范围，记住一定要这么做，这是你以后需要的重要信息。

举个例子，你希望在分支上继续工作一周，来进一步加强你的修正，这时版本库的 HEAD 版本是 480，你准备好了另一次合并，但是我们在“合并的最佳实践”一节提到过，你不想合并已经合并的内容，你只想合并新的东西，技巧就是指出什么是“新”的。

第一步是在主干上运行 **svn log** 察看最后一次与分支合并的日志信息：

```
$ cd calc/trunk
$ svn log
…
------------------------------------------------------------------------
------------
```

```
r406 | user | 2004-02-08 11:17:26 -0600 (Sun, 08 Feb 2004) | 1
line



Merged my-calc-branch changes r341:405 into the trunk.

------------------------------------------------------------
------------

…
```

阿哈!因为分支上 341 到 405 之间的所有修改已经在版本 406 合并了，现在你只需要合并分支在此之后的修改—通过比较 406 和 HEAD。

```
$ cd calc/trunk

$ svn update

At revision 480.



# We notice that HEAD is currently 480, so we use it to do the
merge:



$ svn merge -r 406:480
http://svn.example.com/repos/calc/branches/my-calc-branch

U   integer.c

U   button.c

U   Makefile



$ svn commit -m "Merged my-calc-branch changes r406:480 into the
trunk."

Sending        integer.c

Sending        button.c
```

```
Sending        Makefile

Transmitting file data ...

Committed revision 481.
```

现在主干有了分支上第二波修改的完全结果，此刻，你可以删除你的分支（我们会在以后讨论），或是继续在你分支上工作，重复这个步骤。

取消修改

svn merge 另一个常用的做法是取消已经做得提交，假设你愉快的在 */calc/trunk* 工作，你发现 303 版本对 *integer.c* 的修改完全错了，它不应该被提交，你可以使用 **svn merge** 来“取消”这个工作拷贝上所作的操作，然后提交本地修改到版本库，你要做得只是指定一个相反的区别。（你可以通过指定--revision 303:302--change -303

```
$ svn merge -c -303 http://svn.example.com/repos/calc/trunk

U  integer.c



$ svn status

M  integer.c



$ svn diff

…

# verify that the change is removed

…
```

```
$ svn commit -m "Undoing change committed in r303."

Sending        integer.c

Transmitting file data .

Committed revision 350.
```

我们可以把版本库修订版本想象成一组修改（一些版本控制系统叫做*修改集*），通过-r 选项，你可以告诉 **svn merge** 来应用修改集或是一个修改集范围到你的工作拷贝，在我们的情况例子里，我们使用 **svn merge** 合并修改集#303 到工作拷贝。

Subversion 与修改集

每一个人对于“修改集”的概念都有些不一样，至少对于版本控制系统的“修改集特性”这一概念有着不同的期望，根据我们的用途，可以说修改集只是一个有唯一名字的一系列修改集合，修改也许包括文件内容的修改，目录树结构的修改，或是元数据的调整，更通常的说法，一个修改集就是我们可以引用的有名字的补丁。

在 Subversion 里，一个全局的修订版本号 N 标示一个版本库中的树：它代表版本库在 N 次提交后的样子，它也是一个修改集的隐含名称：如果你比较树 N 与树 N-1，你可以得到你提交的补丁。出于这个原因，想象“版本 N”并不只是一棵树，也是一个修改集。如果你使用一个问题追踪工具来管理 bug，你可以使用版本号来表示特定的补丁修正了 bug —举个例子，“这个问题是在版本 9238 修正的”，然后其他人可以运行 **svn log -r9238** 来查看修正这个 bug 的修改集，或者使用 **svn**

diff -r9237:9238 来看补丁本身。Subversion 的合并命令也使用版本号作为参数，可以将特定修改集从一个分支合到另一个分支：**svn merge -r9237:9238** 将会合并修改集#9238 到本地拷贝。

记住回滚修改和任何一个 **svn merge** 命令都一样，所以你应该使用 **svn status** 或是 **svn diff** 来确定你的工作处于期望的状态中，然后使用 **svn commit** 来提交，提交之后，这个特定修改集不会反映到 HEAD 版本了。

继续，你也许会想：好吧，这不是真的取消提交吧！是吧？版本 303 还依然存在着修改，如果任何人取出 *calc* 的 303-349 版本，他还会得到错误的修改，对吧？

是的，这是对的。当我们说“删除”一个修改时，我们只是说从 HEAD 删除，原始的修改还保存在版本库历史中，在多数情况下，这是足够好的。大多数人只是对追踪 HEAD 版本感兴趣，在一些特定情况下，你也许希望毁掉所有提交的证据（或许某个人提交了一个秘密文件），这不是很容易的，因为 Subversion 设计用来不丢失任何信息，每个修订版本都是依赖其它修订版本的不可变目录树 ，从历史删除一个版本会导致多米诺效应，会在后面的版本导致混乱甚至会影响所有的工作拷贝。[22]

找回删除的项目

版本控制系统非常重要的一个特性就是它的信息从不丢失，即使当你删除了文件或目录，它也许从 HEAD 版本消失了 ，但这个对象依然存在于

历史的早期版本 ，一个新手经常问到的问题是“怎样找回我的文件和目录？”。

第一步首先要知道需要拯救的项目是**什么**，这里有个很有用的比喻：你可以认为任何存在于版本库的对象生活在一个二维的坐标系统里，第一维是一个特定的版本树，第二维是在树中的路径，所以你的文件或目录的任何版本可以通过这样一对坐标定义。（记住常见的“peg 修订版本”语法— foo.c@224 — 在前面的“Peg 和实施修订版本”一节提到过。）

首先，你需要 **svn log** 来察看你需要找回的坐标对，一个好的策略是使用 **svn log --verbose** 来察看包含删除项目的目录，--verbose 选项显示所有改变的项目的每一个版本 ，你只需要找出你删除文件或目录的那一个版本。你可以通过目测找出这个版本，也可以使用另一种工具来检查日志的输出 （通过 **grep** 或是在编辑器里增量查找）。

```
$ cd parent-dir

$ svn log -v

…

------------------------------------------------------------------------

r808 | joe | 2003-12-26 14:29:40 -0600 (Fri, 26 Dec 2003) | 3 lines

Changed paths:

   D /calc/trunk/real.c

   M /calc/trunk/integer.c
```

```
Added fast fourier transform functions to integer.c.

Removed real.c because code now in double.c.

…
```

在这个例子里，你可以假定你正在找已经删除了的文件 *real.c*，通过查找父目录的历史 ，你知道这个文件在 808 版本被删除，所以存在这个对象的版本在此之前 。结论：你想从版本 807 找回*/calc/trunk/real.c*。

以上是最重要的部分—重新找到你需要恢复的对象。现在你已经知道该恢复的文件，而你有两种选择。

一种是对版本反向使用 **`svn merge`** 到 808（我们已经学会了如何取消修改，见“取消修改”一节），这样会重新添加 *real.c*，这个文件会列入增加的计划，经过一次提交，这个文件重新回到 HEAD。

在这个例子里，这不是一个好的策略，这样做不仅把 *real.c* 加入添加到计划，也取消了对 *integer.c* 的修改，而这不是你期望的。确实，你可以恢复到版本 808，然后对 *integer.c* 执行取消 **`svn revert`** 操作，但这样的操作无法扩大使用，因为如果从版本 808 修改了 90 个文件怎么办？

所以第二个方法不是使用 **`svn merge`**，而是使用 **`svn copy`** 命令，精确的拷贝版本和路径“坐标对”到你的工作拷贝：

```
$ svn copy -r 807 \
```

```
_________
http://svn.example.com/repos/calc/trunk/real.c ./real.c



$ svn status

A  +   real.c



$ svn commit -m "Resurrected real.c from revision 807,
/calc/trunk/real.c."

Adding        real.c

Transmitting file data .

Committed revision 1390.
```

加号标志表明这个项目不仅仅是计划增加中，而且还包含了历史，Subversion 记住了它是从哪个拷贝过来的。在将来，对这个文件运行 **`svn log`** 会看到这个文件在版本 807 之前的历史，换句话说，*real.c* 不是新的，而是原先删除的那一个的后代。

尽管我们的例子告诉我们如何找回文件，对于恢复删除的目录也是一样的。

常用分支模式

版本控制在软件开发中广泛使用，这里是团队里程序员最常用的两种分支/合并模式的介绍，如果你不是使用 Subversion 软件开发，可随意跳过本小节，如果你是第一次使用版本控制的软件开发者，请更加注意，以下模式被许多老兵当作最佳实践，这个过程并不只是针对 Subversion，

在任何版本控制系统中都一样，但是在这里使用 Subversion 术语会感觉更方便一点。

发布分支

大多数软件存在这样一个生命周期：编码、测试、发布，然后重复。这样有两个问题，第一，开发者需要在质量保证小组测试假定稳定版本时继续开发新特性，新工作在软件测试时不可以中断，第二，小组必须一直支持老的发布版本和软件；如果一个 bug 在最新的代码中发现，它一定也存在已发布的版本中，客户希望立刻得到错误修正而不必等到新版本发布。

这是版本控制可以做的帮助，典型的过程如下：

- *开发者提交所有的新特性到主干。*每日的修改提交到*/trunk*：新特性，bug 修正和其他。
- *这个主干被拷贝到“发布”分支。*当小组认为软件已经做好发布的准备（如，版本 1.0）然后*/trunk* 会被拷贝到*/branches/1.0*。
- *项目组继续并行工作，*一个小组开始对分支进行严酷的测试，同时另一个小组在*/trunk* 继续新的工作（如，准备 2.0），如果一个 bug 在任何一个位置被发现，错误修正需要来回运送。然而这个过程有时候也会结束，例如分支已经为发布前的最终测试“停滞”了。
- *分支已经作了标签并且发布，*当测试结束，*/branches/1.0* 作为引用快照已经拷贝到*/tags/1.0.0*，这个标签被打包发布给客户。

- *分支多次维护*。当继续在*/trunk* 上为版本 2.0 工作，bug 修正继续从*/trunk* 运送到*/branches/1.0*，如果积累了足够的 bug 修正，管理部门决定发布 1.0.1 版本：拷贝*/branches/1.0* 到*/tags/1.0.1*，标签被打包发布。

整个过程随着软件的成熟不断重复：当 2.0 完成，一个新的 2.0 分支被创建，测试、打标签和最终发布，经过许多年，版本库结束了许多版本发布，进入了“维护”模式，许多标签代表了最终的发布版本。

特性分支

一个*特性分支*是本章中那个重要例子中的分支，你正在那个分支上工作，而 Sally 还在*/trunk* 继续工作，这是一个临时分支，用来作复杂的修改而不会干扰*/trunk* 的稳定性，不象发布分支（也许要永远支持），特性分支出生，使用了一段时间，合并到主干，然后最终被删除掉，它们在有限的时间里有用。

还有，关于是否创建特性分支的项目政策也变化广泛，一些项目永远不使用特性分支：大家都可以提交到*/trunk*，好处是系统的简单—没有人需要知道分支和合并，坏处是主干会经常不稳定或者不可用，另外一些项目使用分支达到极限：没有修改*曾经*直接提交到主干，即使最细小的修改都要创建短暂的分支，然后小心的审核合并到主干，然后删除分支，这样系统保持主干一直稳定和可用，但是造成了巨大的负担。

许多项目采用折中的方式，坚持每次编译*/trunk* 并进行回归测试，只有需要多次不稳定提交时才需要一个特性分支，这个规则可以用这样一个问题检验：如果开发者在好几天里独立工作，一次提交大量修改（这样*/trunk* 就不会不稳定。），是否会有太多的修改要来回顾？如果答案是“是”，这些修改应该在特性分支上进行，因为开发者增量的提交修改，你可以容易的回头检查。

最终，有一个问题就是怎样保持一个特性分支“同步”于工作中的主干，在前面提到过，在一个分支上工作数周或几个月是很有风险的，主干的修改也许会持续涌入，因为这一点，两条线的开发会区别巨大，合并分支回到主干会成为一个噩梦。

这种情况最好通过有规律的将主干合并到分支来避免，制定这样一个政策：每周将上周的修改合并到分支，注意这样做时需要小心，需要手工记录合并的过程，以避免重复的合并（在“手工跟踪合并”一节描述过），你需要小心的撰写合并的日志信息，精确的描述合并包括的范围（在“合并分支到另一分支”一节中描述过），这看起来像是胁迫，可是实际上是容易做到的。

在一些时候，你已经准备好了将“同步的”特性分支合并回到主干，为此，开始做一次将主干最新修改和分支的最终合并，这样以后，除了你的分支修改的部分，最新的分支和主干将会绝对一致，所以在这个特别的例子里，你会通过直接比较分支和主干来进行合并：

```
$ cd trunk-working-copy

$ svn update
At revision 1910.

$ svn merge http://svn.example.com/repos/calc/trunk@1910 \
____________
http://svn.example.com/repos/calc/branches/mybranch@1910
U  real.c
U  integer.c
A  newdirectory
A  newdirectory/newfile
…
```

通过比较 HEAD 修订版本的主干和 HEAD 修订版本的分支，你确定了只在分支上的增量信息，两条开发线都有了分枝的修改。

可以用另一种考虑这种模式，你每周按时同步分支到主干，类似于在工作拷贝执行 **svn update** 的命令，最终的合并操作类似于在工作拷贝运行 **svn commit**，毕竟，工作拷贝不就是一个非常浅的分支吗？只是它一次只可以保存一个修改。

使用分支

svn switch 命令改变存在的工作拷贝到另一个分支，然而这个命令在分支上工作时不是严格必要的，它只是提供了一个快捷方式。在前面的

例子里，完成了私有分支的建立，你取出了新目录的工作拷贝，相反，你可以简单的告诉 Subversion 改变你的*/calc/trunk* 的工作拷贝到分支的路径：

```
$ cd calc


$ svn info | grep URL
URL: http://svn.example.com/repos/calc/trunk


$ svn switch
http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile
Updated to revision 341.


$ svn info | grep URL
URL:
http://svn.example.com/repos/calc/branches/my-calc-branch
```

完成了到分支的“跳转”，你的目录与直接取出一个干净的版本没有什么不同。这样会更有效率，因为分支只有很小的区别，服务器只是发送修改的部分来使你的工作拷贝反映分支。

svn switch 命令也可以带--revision（-r）参数，所以你不需要一直移动你的工作拷贝到分支的 HEAD。

当然，许多项目比我们的 *calc* 要复杂的多，有更多的子目录，Subversion 用户通常用如下的法则使用分支：

1. 拷贝整个项目的“trunk”目录到一个新的分支目录。
2. 只是转换工作拷贝的*部分*目录到分支。

换句话说，如果一个用户知道分支工作只发生在部分子目录，我们使用 **svn switch** 来跳转部分目录（有时候只是单个文件），这样的话，他们依然可以继续得到普通的“trunk”主干的更新，但是已经跳转的部分则被免去了更新（除非分支上有更新）。这个特性给“混合工作拷贝”概念添加了新的维度—不仅工作拷贝的版本可以混合，在版本库中的位置也可以混合。

如果你的工作拷贝包含许多来自不同版本库目录跳转的子树，它会工作如常。当你更新时，你会得到每一个目录适当的补丁，当你提交时，你的本地修改会一直作为一个单独的原子修改提交到版本库。

注意，因为你的工作拷贝可以在混合位置的情况下工作正常，但是所有的位置必须在同一个版本库，Subversion 的版本库不能互相通信，这个特性还不在 Subversion 未来的计划里。

切换和更新

你注意到 **svn switch** 和 **svn update** 的输出很像？switch 命令只是 update 命令的一个超集。

当你运行 **svn update** 时，你会告诉版本库比较两个目录树，版本库这样做，并且返回给客户区别的描述，**svn switch** 和 **svn update** 两个命令唯一区别就是 **update** 会一直比较同一路径。

也就是了，如果你的工作拷贝是*/calc/trunk* 的一个镜像，当运行 **svn update** 时会自动地比较你的工作拷贝的*/calc/trunk* 与 HEAD 版本的*/calc/trunk*。如果你使用 **svn switch** 跳转工作拷贝到分支，则会比较你的工作拷贝的*/calc/trunk* 与相应分支目录的 HEAD 版本。

换句话说，一个更新通过时间移动你的工作拷贝，一个转换通过时间和空间移动工作拷贝。

因为 **svn switch** 是 **svn update** 的一个变种，具有相同的行为，当新的数据到达时，任何工作拷贝的已经完成的本地修改会被保存，这里允许你作各种聪明的把戏。

举个例子，你的工作拷贝目录是*/calc/trunk*，你已经做了很多修改，然后你突然发现应该在分支上修改更好，没问题！你可以使用 **svn switch**，而你本地修改还会保留，你可以测试并提交它们到分支。

标签

另一个常见的版本控制系统概念是标¾（*tag*），一个标签只是一个项目某一时间的“快照”，在 Subversion 里这个概念无处不在—每一次提交的修订版本都是一个精确的快照。

然而人们希望更人性化的标签名称，像 release-1.0。他们也希望可以对一个子目录快照，毕竟，记住 release-1.0 是修订版本 4822 的某一小部分不是件很容易的事。

建立简单标签

svn copy 再次登场，你希望建立一个*/calc/trunk* 的一个快照，就像 HEAD 修订版本，建立这样一个拷贝：

```
$ svn copy http://svn.example.com/repos/calc/trunk \

          http://svn.example.com/repos/calc/tags/release-1.0
\

     -m "Tagging the 1.0 release of the 'calc' project."



Committed revision 351.
```

这个例子假定*/calc/tags* 目录已经存在（如果不是，可以使用 **svn mkdir** 创建。），拷贝完成之后，一个表示当时 HEAD 版本的/calc/trunk 目录的镜像已经永久的拷贝到 *release-1.0* 目录。当然，你会希望更精确一点，以防其他人在你不注意的时候提交修改，所以，如果你知道*/calc/trunk* 的版本 350 是你想要的快照，你可以使用 **svn copy** 加参数 -r 350。

但是等一下：标签的产生过程与建立分支是一样的？是的，实际上在 Subversion 中标签与分支没有区别，都是普通的目录，通过 copy 命令得到，与分支一样，一个目录之所以是标签只是*人们*决定这样使用它，

只要没有人提交这个目录，它永远是一个快照，但如果人们开始提交，它就变成了分支。

如果你管理一个版本库，你有两种方式管理标签，第一种方法是禁止命令：作为项目的政策，我们要决定标签所在的位置，确定所有用户知道如何处理拷贝的目录（也就是确保他们不会提交他们），第二种方法看来很过分：使用访问控制脚本来阻止任何想对标签目录做的非拷贝的操作（见第 6 章 *服务配置*）这种方法通常是不必要的，如果一个人不小心提交了到标签目录一个修改，你可以简单的取消，毕竟这是版本控制啊。

建立复杂标签

有时候你希望你的“快照”能够很复杂，而不只是一个单独修订版本的一个单独目录。

举个例子，假定你的项目比我们的的例子 *calc* 大的多：假设它保存了一组子目录和许多文件，在你工作时，你或许决定创建一个包括特定特性和 Bug 修正的工作拷贝，你可以通过选择性的回溯文件和目录到特定修订版本（使用 **`svn update -r`**）来实现，或者转换文件和目录到特定分支（使用 **`svn switch`**），这样做之后，你的工作拷贝成为版本库不同版本和分支的司令部，但是经过测试，你会知道这是你需要的一种精确数据组合。

是时候进行快照了，拷贝 URL 在这里不能工作，在这个例子里，你希望把本地拷贝的布局做镜像并且保存到版本库中，幸运的是，**svn copy** 包括四种不同的使用方式（在第 9 章 *Subversion 完全参考*可以详细阅读），包括拷贝工作拷贝到版本库：

```
$ ls

my-working-copy/



$ svn copy my-working-copy
http://svn.example.com/repos/calc/tags/mytag



Committed revision 352.
```

现在在版本库有一个新的目录*/calc/tags/mytag*，这是你的本地拷贝的一个快照—混合了修订版本，URL 等等。

一些人也发现这一特性一些有趣的使用方式，有些时候本地拷贝有一组本地修改，你希望你的协作者看到这些，不使用 **svn diff** 并发送一个补定文件（不会捕捉到目录、符号链和属性的修改），而是使用 **svn copy** 来“上传”你的工作拷贝到一个版本库的私有区域，你的协作者可以选择完整的取出你的工作拷贝，或使用 **svn merge** 来接受你的精确修改。

虽然这是上传快速工作拷贝快照的一个好方法，但这*不是*初始创建分支的好方法。分支创建必须是它本身的事件，而这个方法创建的分支包含

了额外修改，都包含在一个单独修订版本里。这让我们很难识别分支点的单个修订版本号码。

你是否发现你做出了复杂的修改（在*/trunk* 的工作拷贝），并突然发现，“这些修改必须在它们自己的分支？”处理这个问题的技术可以总结为两步：

```
$ svn copy http://svn.example.com/repos/calc/trunk
\

          
http://svn.example.com/repos/calc/branches/newbr
anch

Committed revision 353.



$ svn switch
http://svn.example.com/repos/calc/branches/newbr
anch

At revision 353.
```

就像 **svn update** 命令，**svn switch** 会保留工作拷贝的本地修改，此刻，你的工作拷贝反映到新建的分支上，而你的下一次 **svn commit** 会发送修改到服务器。

分支维护

你一定注意到了 Subversion 极度的灵活性，因为它用相同的底层机制（目录拷贝）实现了分支和标签，因为分支和标签是作为普通的文件系

统出现，会让人们感到害怕，因为它太灵活了，在这个小节里，我们会提供安排和管理数据的一些建议。

版本库布局

有一些标准的，推荐的组织版本库的方式，许多人创建一个 *trunk* 目录来保存开发的“主线”，一个 *branches* 目录存放分支拷贝，一个 *tags* 目录保存标签拷贝，如果一个版本库只是存放一个项目，人们会在顶级目录创建这些目录：

```
/trunk
/branches
/tags
```

如果一个版本库保存了多个项目，管理员会通过项目来布局（见“规划你的版本库结构”一节关于“项目根目录”）：

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

当然，你可以自由的忽略这些通常的布局方式，你可以创建任意的变化，只要是对你和你的项目有益，记住无论你选择什么，这不会是一种永久

的承诺，你可以随时重新组织你的版本库。因为分支和标签都是普通的目录，**svn move** 命令可以任意的改名和移动它们，从一种布局到另一种大概只是一系列服务器端的移动，如果你不喜欢版本库的组织方式，你可以任意修改目录结构。

记住，尽管移动目录非常容易，你必须体谅你的用户，你的修改会让你的用户感到迷惑，如果一个用户的拥有一个版本库目录的工作拷贝，你的 **svn move** 命令也许会删除最新的版本的这个路径，当用户运行 **svn update**，会被告知这个工作拷贝引用的路径已经不再存在，用户需要强制使用 **svn switch** 转到新的位置。

数据的生命周期

另一个 Subversion 模型的可爱特性是分支和标签可以有有限的生命周期，就像其它的版本化的项目，举个例子，假定你最终完成了 *calc* 项目你的个人分支上的所有工作，在合并了你的所有修改到*/calc/trunk* 后，没有必要继续保留你的私有分支目录：

```
$ svn delete
http://svn.example.com/repos/calc/branches/my-calc-branch \

           -m "Removing obsolete branch of calc project."



Committed revision 375.
```

你的分支已经消失了，当然不是真的消失了：这个目录只是在 HEAD 修订版本里消失了，如果你使用 **svn checkout**、**svn switch** 或者 **svn list** 来检查一个旧的版本，你仍会见到这个旧的分支。

如果浏览你删除的目录还不足够，你可以把它找回来，恢复数据对 Subversion 来说很简单，如果你希望恢复一个已经删除的目录（或文件）到 HEAD，仅需要使用 **svn copy -r** 来从旧的版本拷贝出来：

```
$ svn copy -r 374
http://svn.example.com/repos/calc/branches/my-calc-branch \

_____________
http://svn.example.com/repos/calc/branches/my-calc-branch



Committed revision 376.
```

在我们的例子里，你的个人分支只有一个相对短的生命周期：你会为修复一个 Bug 或实现一个小的特性来创建它，当任务完成，分支也该结束了。在软件开发过程中，有两个“主要的”分支一直存在很长的时间也是很常见的情况，举个例子，假定我们是发布一个稳定的 *calc* 项目的时候了，但我们仍会需要几个月的时间来修复 Bug，你不希望添加新的特性，但你不希望告诉开发者停止开发，所以作为替代，你为软件创建了一个“分支”，这个分支更改不会很多：

```
$ svn copy http://svn.example.com/repos/calc/trunk \

_______
http://svn.example.com/repos/calc/branches/stable-1.0 \

       -m "Creating stable branch of calc project."
```

```
Committed revision 377.
```

而且开发者可以自由的继续添加新的（试验的）特性到*/calc/trunk*，你可以宣布这样一种政策，只有 bug 修正提交到*/calc/branches/stable-1.0*，这样的话，人们继续在主干上工作，某个人会选择在稳定分支上做出一些 Bug 修正，甚至在稳定版本发布之后。你或许会在这个维护分支上工作很长时间—也就是说，你会一直继续为客户提供这个版本的支持。

供方分支

当开发软件时有这样一个情况，你版本控制的数据可能关联于或者是依赖于其他人的数据，通常来讲，你的项目的需要会要求你自己的项目对外部实体提供的数据保持尽可能最新的版本，同时不会牺牲稳定性，这种情况总是会出现—只要某个小组的信息对另一个小组的信息有直接的影响。

举个例子，软件开发者会工作在一个使用第三方库的应用，Subversion 恰好是和 Apache 的 Portable Runtime library（见“Apache 可移植运行库”一节）有这样一个关系。Subversion 源代码依赖于 APR 库来实现可移植需求。在 Subversion 的早期开发阶段，项目紧密地追踪 APR 的 API 修改，经常在库代码的“流血的边缘”粘住，现在 APR 和 Subversion 都已经成熟了，Subversion 只尝试同步 APR 的经过良好测试的，稳定的 API 库。

现在，如果你的项目依赖于其他人的信息，有许多方法可以用来尝试同步你的信息，最痛苦的，你可以为项目所有的贡献者发布口头或书写的指导，告诉他们确信他们拥有你们的项目需要的特定版本的第三方信息。如果第三方信息是用 Subversion 版本库维护，你可以使用 Subversion 的外部定义来有效的“强制”特定的版本的信息在你的工作拷贝的的位置（见“外部定义”一节）。

但是有时候，你希望在你自己的版本控制系统维护一个针对第三方数据的自定义修改，回到软件开发的例子，程序员为了他们自己的目的会需要修改第三方库，这些修改会包括新的功能和 bug 修正，在成为第三方工具官方发布之前，只是内部维护。或者这些修改永远不会传给库的维护者，只是作为满足软件开发需要的单独的自定义修改存在。

现在你会面对一个有趣的情形，你的项目可以用某种脱节的样式保持它关于第三方数据自己的修改，如使用补丁文件或者是完全的可选版本的文件和目录。但是这很快会成为维护的头痛的事情，需要一种机制来应用你对第三方数据的自定义修改，并且迫使在第三方数据的后续版本重建这些修改。

这个问题的解决方案是使用*供方分支*，一个供方分支是一个目录树保存了第三方实体或供应方的信息，每一个供应方数据的版本吸收到你的项目叫做*供方 drop*。

供方分支提供了两个关键的益处，第一，通过在我们的版本控制系统保存现在支持的供方 drop，你项目的成员不需要指导他们是否有了正确版本的供方数据，他们只需要作为不同工作拷贝更新的一部份，简单的接受正确的版本就可以了。第二，因为数据存在于你自己的 Subversion 版本库，你可以在恰当的位置保存你的自定义修改—你不需要一个自动的（或者是更坏，手工的）方法来交换你的自定义行为。

常规的供方分支管理过程

管理供方分支通常会像这个样子，你创建一个顶级的目录（如*/vendor*）来保存供方分支，然后你导入第三方的代码到你的子目录。然后你将拷贝这个子目录到主要的开发分支（例如*/trunk*）的适当位置。你一直在你的主要开发分支上做本地修改，当你的追踪的代码有了新版本，你会把带到供方分支并且把它合并到你的*/trunk*，解决任何你的本地修改和他们的修改的冲突。

也许一个例子有助于我们阐述这个算法，我们会使用这样一个场景，我们的开发团队正在开发一个计算器程序，与一个第三方的复杂数字运算库 libcomplex 关联。我们从供方分支的初始创建开始，并且导入供方 drop，我们会把每株分支目录叫做 *libcomplex*，我们的代码 drop 会进入到供方分支的子目录 *current*，并且因为 **`svn import`** 创建所有的需要的中间父目录，我们可以使用一个命令完成这一步。

```
$ svn import /path/to/libcomplex-1.0 \
```

```
_____
http://svn.example.com/repos/vendor/libcomplex/current \

          -m 'importing initial 1.0 vendor drop'

…
```

我们现在在*/vendor/libcomplex/current* 有了 libcomplex 当前版本的代码，现在我们为那个版本作标签（见“标签”一节），然后拷贝它到主要开发分支，我们的拷贝会在 *calc* 项目目录创建一个新的目录 *libcomplex*，它是这个我们将要进行自定义的供方数据的拷贝版本。

```
$ svn copy
http://svn.example.com/repos/vendor/libcomplex/current  \

         http://svn.example.com/repos/vendor/libcomplex/1.0
\

         -m 'tagging libcomplex-1.0'

…

$ svn copy http://svn.example.com/repos/vendor/libcomplex/1.0
\

         http://svn.example.com/repos/calc/libcomplex
\

         -m 'bringing libcomplex-1.0 into the main branch'

…
```

我们取出我们项目的主分支—现在包括了第一个供方释放的拷贝—我们开始自定义 libcomplex 的代码，在我们知道之前，我们的 libcomplex 修改版本是已经与我们的计算器程序完全集成了。[23]

几周之后，libcomplex 得开发者发布了一个新的版本—版本 1.1—包括了我们很需要的一些特性和功能。我们很希望升级到这个版本，但不希望失去在当前版本所作的修改。我们本质上会希望把我们当前基线版本是的 libcomplex1.0 的拷贝替换为 libcomplex 1.1，然后把前面自定义的修改应用到新的版本。但是实际上我们通过一个相反的方向解决这个问题，应用 libcomplex 从版本 1.0 到 1.1 的修改到我们修改的拷贝。

为了执行这个升级，我们取出一个我们供方分支的拷贝，替换 *current* 目录为新的 libcomplex 1.1 的代码，我们只是拷贝新文件到存在的文件上，或者是解压缩 libcomplex 1.1 的打包文件到我们存在的文件和目录。此时的目标是让我们的 *current* 目录只保留 libcomplex 1.1 的代码，并且保证所有的代码在版本控制之下，哦，我们希望在最小的版本控制历史扰动下完成这件事。

完成了这个从 1.0 到 1.1 的代码替换，**`svn status`** 会显示文件的本地修改，或许也包括了一些未版本化或者丢失的文件，如果我们做了我们应该做的事情，未版本化的文件应该都是 libcomplex 在 1.1 新引入的文件—我们运行 **`svn add`** 来将它们加入到版本控制。丢失的文件是存在于 1.1 但是不是在 1.1，在这些路径我们运行 **`svn delete`**。最终一旦我们的 *current* 工作拷贝只是包括了 libcomplex1.1 的代码，我们可以提交这些改变目录和文件的修改。

我们的 *current* 分支现在保存了新的供方 drop，我们为这个新的版本创建一个新的标签（就像我们为 1.0 版本 drop 所作的），然后合并这从个标签前一个版本的区别到主要开发分支。

```
$ cd working-copies/calc

$ svn merge
http://svn.example.com/repos/vendor/libcomplex/1.0      \

__________
http://svn.example.com/repos/vendor/libcomplex/current  \

          libcomplex

… # resolve all the conflicts between their changes and our
changes

$ svn commit -m 'merging libcomplex-1.1 into the main branch'

…
```

在这个琐碎的用例里，第三方工具的新版本会从一个文件和目录的角度来看，就像前一个版本。没有任何 libcomplex 源文件会被删除、被改名或是移动到别的位置—新的版本只会保存针对上一个版本的文本修改。在完美世界，我们对呢修改会干净得应用到库的新版本，不会产生任何并发和冲突。

但是事情总不是这样简单，实际上源文件在不同的版本间的移动是很常见的，这种过程复杂性可以确保我们的修改会一直对新的版本代码有效，可以很快使形势退化到我们需要在新版本手工的重新创建我们的自定义修改。一旦 Subversion 知道了给定文件的历史—包括了所有以前的

位置—合并到新版本的进程就会很简单，但是我们需要负责告诉 Subversion 供方 drop 之间源文件布局的改变。

svn_load_dirs.pl

不仅仅包含一些删除、添加和移动的供方 drops 使得升级第三方数据后续版本的过程变得复杂，所以 Subversion 提供了一个 **svn_load_dirs.pl** 脚本来辅助这个过程，这个脚本自动进行我们前面提到的常规供方分支管理过程的导入步骤，从而使得错误最小化。你仍要负责使用合并命令合并第三方的新 版本数据合并到主要开发分支，但是 **svn_load_dirs.pl** 帮助你快速到达这一步骤。

一句话，**svn_load_dirs.pl** 是一个增强的 **svn import**，具备了许多重要的特性：

- 它可以在任何有一个存在的版本库目录与一个外部的目录匹配时执行，会执行所有必要的添加和删除并且可以选则执行移动。
- 它可以用来操作一系列复杂的操作，如那些需要一个中间媒介的提交—如在操作之前重命名一个文件或者目录两次。
- 它可以随意的为新导入目录打上标签。
- 它可以随意为符合正则表达式的文件和目录添加任意的属性。

svn_load_dirs.pl 利用三个强制的参数，第一个参数是 Subversion 工作的基本目录 URL，第二个参数在 URL 之后—相对于第一个参数—指向当前的供方分支将会导入的目录，最后，第三个参数是一个需要导

入的本地目录，使用前面的例子，一个典型的 **svn_load_dirs.pl** 调用看起来如下：

```
$ svn_load_dirs.pl
http://svn.example.com/repos/vendor/libcomplex \

                current                                          \

                /path/to/libcomplex-1.1

…
```

你可以说明你会希望 **svn_load_dirs.pl** 同时打上标签，这使用-t 命令行选项，需要指定一个标签名，这个标签是第一个参数的一个相对 URL。

```
$ svn_load_dirs.pl -t libcomplex-1.1                              \

                
http://svn.example.com/repos/vendor/libcomplex \

                current                                          \

                /path/to/libcomplex-1.1

…
```

当你运行 **svn_load_dirs.pl**，它会检验你的存在的“current”供方 drop，并且与提议的新供方 drop 比较，在这个琐碎的例子里，没有文件只出现在一个版本里，脚本执行新的导入而不会发生意外。然而如果版本之间有了文件布局的区别，**svn_load_dirs.pl** 会询问你如何解决这个区别，例如你会有机会告诉脚本 libcomplex 版本 1.0 的 *math.c*

文件在 1.1 已经重命名为 *arithmetic.c*，任何没有解释为移动的差异都会被看作是常规的添加和删除。

这个脚本也接受单独配置文件用来为*添加到*版本库的文件和目录设置匹配正则表达式的属性。配置文件通过 **svn_load_dirs.pl** 的-p 命令行选项指定，这个配置文件的每一行都是一个空白分割的两列或者四列值：一个 Perl 样式的正则表达式来匹配添加的路径、一个控制关键字（break 或者是 cont）和可选的属性名和值。

```
\.png$             break   svn:mime-type   image/png
\.jpe?g$           break   svn:mime-type   image/jpeg
\.m3u$             cont    svn:mime-type   audio/x-mpegurl
\.m3u$             break   svn:eol-style   LF
.*                 break   svn:eol-style   native
```

对每一个添加的路径，会按照顺序为匹配正则表达式的文件配置属性，除非控制标志是 break（意味着不需要更多的路径匹配应用到这个路径）。如果控制说明是 cont—continue 的缩写—然后匹配工作会继续到配置文件的下一行。

任何正则表达式，属性名或者属性值的空格必须使用单引号或者双引号环绕，你可以使用反斜杠（\）换码符来回避引号，反斜杠只会在解析配置文件时回避引号，所以不能保护必要正则表达式字符之外的的其它字符。

总结

我们已经在本章覆盖了许多基础知识，我们讨论了标签和分支的概念，然后描述了 Subversion 怎样用 **svn copy** 命令拷贝目录实现了这些概念，我们也已经展示了怎样使用 **svn merge** 命令来在分支之间拷贝修改，或是撤销错误的修改。我们仔细研究了使用 **svn switch** 来创建混合位置的工作拷贝，然后我们也讨论了怎样管理和组织版本库中分支的生命周期。

记住 Subversion 的颂歌：分支和标签是廉价的，所以可以自由的使用！在同一时间，不要忘记使用好的合并习惯，廉价的拷贝只在你小心的跟踪你的合并操作时有用。

---

[20] Subversion 不支持跨版本库的拷贝，当使用 **svn copy** 或者 **svn move** 直接操作 URL 时你只能在同一个版本库内操作。

[21] 然而，写这些的时候，这些特性正在实现中！

[22] Subversion 项目有计划，不管用什么方式，会有一天要实现 **svnadmin obliterate** 命令来进行永久删除操作，而此时可以看“svndumpfilter”一节找到可行的方案。

[23] 而且完全没有 bug，当然！

版本库管理

目录

Subversion 版本库是保存任意数量项目版本化数据的中央仓库，因此，版本库成为管理员关注的对象。版本库的维护一般并不需要太多的关注，但为了避免一些潜在的问题和解决一些实际问题，理解怎样适当的配置和维护还是非常重要的。

在这一章里，我们将讨论如何建立和配置一个 Subversion 版本库，还会讨论版本库的维护，包括 **`svnlook`** 和 **`svnadmin`** 工具的使用实例。我们将说明一些常见的问题和错误，并提供一些安排版本库数据的建议。

如果您只是以普通用户的身份访问版本库对数据进行版本控制（就是说通过 Subversion 客户端），您完全可以跳过本章。但是如果您已经是或打算成为 Subversion 版本库的管理员，[24]您一定要关注一下本章的内容。

Subversion 版本库的定义

在进入版本库管理这块宽泛的主题之前，让我们进一步确定一下版本库的定义，它是怎样工作的？让人有什么感觉？它希望茶是热的还是冰的，加糖或柠檬吗？作为一名管理员，你应该既能够从物理具体细节的视角－版本库如何响应一个非 Subversion 的工具，也能够从逻辑视角－数据在版本库中如何展示。

通过典型的文件浏览器应用程序或命令行为基础的文件系统浏览工具查看，Subversion 版本库只是另一个目录。也有一些子目录下包含可读的数据文件，也有一些子目录包含不可读的数据文件。Subversion 设计

的其他地方，模块化被认真考虑，等级化的组织可以减少混乱，所以脱离细节粗略看一下典型的版本库可以有效地揭示版本库的基本组件。

```
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

下面是一个你看到列出目录的快速总揽。（不要因为术语陷入困境—这些组件的细节介绍可以从本章或其他章节找到。）

conf

一个存储版本库配置文件的目录。

dav

提供给 Apache 和 mod_dav_svn 的目录，让它们存储自己的数据。

db

你的版本化数据的数据存储方式。

format

包含了一个用来表示版本库布局版本号整数的文件。

hooks

一个存储钩子脚本模版的目录（还有钩子脚本本身， 如果你安装了的话）。

locks

一个存储 Subversion 版本库锁定文件的目录，被用来追踪对版本库的访问。

README.txt

这个文件只是用来告诉它的阅读者，他现在看的是 Subversion 的版本库。

当然，当通过 Subversion 库访问时，这些平常的文件和目录立刻变成了虚拟文件系统的实现，由自定义的事件触发完成。这个文件系统的目录和文件都有自己的概念，与真实的文件系统（例如 NTFS、FAT32、ext3 等等）很类似，但是也有特别的地方—它在修订版本间锁定目录和文件，保持你的所有修改可以永远访问的，这是你的所有版本化数据存放的地方。

版本库开发策略

因为 Subversion 版本库本身和所依赖技术设计的简单性，创建和配置版本库是一件相对直接的任务。需要做一些的预备决定，但是设置 Subversion 版本库的实际工作非常直接，在做过几次之后就会发现不必费太多心思去做这件事。

下面是一些你需要预先考虑的事情：

- 你的版本库将要存放什么数据（或多个版本库），这些数据如何组织？

- 版本库存放在哪里，如何被访问？
- 你需要什么类型的访问控制和版本库事件报告？
- 你希望使用哪种数据存储方式？

在本节，我们要尝试帮你回答这些问题。

规划你的版本库结构

在 Subversion 版本库中，移动版本化的文件和目录不会损失任何信息，甚至也可以将版本库的的一组数据无损历史的移植到另一个版本库，但是这样一来那些经常访问版本库并且以为文件总是在同一个路径的用户可能会受到干扰。为将来着想，最好预先对你的版本库布局进行规划。以一种高效的“布局”开始项目，可以减少将来很多不必要的麻烦。

假如你是一个版本库管理员，需要向多个项目提供版本控制支持。那么，你首先要决定的是，用一个版本库支持多个项目，还是为每个项目建立一个版本库，还是两种方法的混合方式。

使用一个版本库支持多个项目有很多好处，最明显的无过于不需要维护好几个版本库。单一版本库就意味着只有一个钩子程序，只需要备份一个数据库，当 Subversion 进行不兼容升级时，只需要一次转储和装载操作，等等。还有，你可以轻易的在项目之间移动数据，还不会损失任何历史版本信息。

单一版本库的缺点是，不同的项目通常都有不同的版本库触发事件需求，例如需要发送提交通知邮件到不同的邮件列表，需要不同的鉴定提交是否合法的定义。这些都不是不可逾越的问题，当然一之需要你的钩子程序能够察看版本库的布局，而不是假定整个版本库与同一组人关联。还有，别忘了 Subversion 的修订版本号是针对整个版本库的，这些号码没有任何魔力。即使最近没有对某个项目作出修改，版本库的修订版本号还是会因为其它项目的修改而不停的提升，许多人并不喜欢这样的事实。[25]

可以采用折中的办法。比如，可以把许多项目按照彼此之间的关联程度划分为几个组合，然后为每一个项目组合建立一个版本库。这样，在相关项目之间共享数据依旧很简单，而如果修订版本号有了变化，至少开发人员知道，改变的东西多少和他们有些关系。

在决定了如何用版本库组织项目以后，就该决定如何设置版本库的目录层次了。由于 Subversion 按普通的目录复制方式完成分支和标签操作（参见第 4 章*分支与合并*），Subversion 社区建议为每一个项目建立一个*项目根目录*—项目的“顶级”目录—然后在根目录下建立三个子目录：*trunk*，保存项目的开发主线；*branches*，保存项目的各种开发分支；*tags*，保存项目的标签，也就是创建后永远不会修改的分支（可能会删除）。[26]

举个例子，一个版本库可能会有如下的布局：

```
/
  calc/
    trunk/
    tags/
    branches/
  calendar/
    trunk/
    tags/
    branches/
  spreadsheet/
    trunk/
    tags/
    branches/
  …
```

项目在版本库中的根目录地址并不重要。如果每个版本库中只有一个项目，那么就可以认为项目的根目录就是版本库的根目录。如果版本库中包含多个项目，那么可以将这些项目划分成不同的组合（按照项目的目标或者是否需要共享代码甚至是字母顺序）保存在不同子目录中，下面的例子给出了一个类似的布局：

```
/
  utils/
    calc/
      trunk/
```

```
        tags/
        branches/
    calendar/
        trunk/
        tags/
        branches/
    …
  office/
    spreadsheet/
        trunk/
        tags/
        branches/
    …
```

按照你认为合适的方式安排版本库的布局，Subversion 自身并不强制或者偏好某一种布局形式，对于 Subversion 来说，目录就是目录。最后，在设计版本库布局的时候，不要忘了考虑一下项目参与者们的意见。

为了完整性，我们需要提一下另一种常见的布局，在这种布局中 *trunk*、*tags* 和 *branches* 都在根目录下，而你的项目在各个子目录下，例如：

```
/
  trunk/
    calc/
    calendar/
    spreadsheet/
```

```
    …
  tags/
    calc/
    calendar/
    spreadsheet/
    …
  branches/
    calc/
    calendar/
    spreadsheet/
    …
```

这种布局没有什么不对的，但是它只是或不是你的用户的直觉。特别是在大的，有许多用户的多项目情况下，用户可能只熟悉版本库中一两个项目。但是项目作为分支的方式可以鼓励项目的个性和将注意力集中在一个单独的实体。尽管这也是一个社会问题，因为实践的原因，我们很愿意对安排提出一些建议—当一个项目的历史都在一个目录里时，很容易查询（或是修改、移植）单个项目的历史—过去、现在、标签和分支—单独为那个项目。

决定在哪里与如何部署你的版本库

在创建 Subversion 版本库之前，一个明显的问题是所有的东西要存放在什么地方，这与很多问题关联，包括版本库如何访问（通过 Subversion 服务器或直接访问）、被谁访问（防火墙后的用户或全部是在 Internet

上）、你将围绕 Subversion 提供哪些服务（版本库浏览接口，e-mail 为基础的提交通知等）、你的数据备份策略，等等。

我们在第 6 章 *服务配置*覆盖了服务器的选择和配置，我们也提供一些可能会使你必须决定使用某种服务器的问题的答案。例如，特定的部署策略可能会需要从多个计算机通过远程文件系统访问版本库，这个情况下（下一小节会读到）要求你不能选择一种版本库后端数据存储方式，因为只有一种后端在这种场景下可以工作。

列出所有的 Subversion 可能的部署方法是不可能的，超出了本书的范围，我们只是简单的鼓励你使用这部分内容和参考材料验证你的想法，并往前计划。

选择数据存储格式

在 Subversion1.1 中，版本库中有两种数据存储方式—通常叫做“后端”或其他容易混淆的名字，如“（版本化的） 文件系统”，每一个版本库都会使用一种。一种是在 Berkeley DB 数据库中存储数据，我们称之为“BDB 后端”；另一种是使用普通的文件，自定义格式，Subversion 开发者根据习惯称之为 *FSFS*[27] —一种使用本地操作系统文件存储数据的版本化文件系统直接实现—而不是通过某个数据库层或其他抽象层来保存数据。

表 5.1 “”从总体上比较了 Berkeley DB 和 FSFS 版本库。

表 5.1.

分类	特性	Berkeley DB	FSFS
可靠性	数据完整性	当正确部署，非常可靠；Berkeley DB 4.4 支持自动恢复	较老的版本较少被描述，但是有数据毁坏 bug
	对操作中断的敏感	很敏感；系统崩溃或者权限问题会导致数据库“塞住”，需要定期进行恢复。	十分敏感
可用性	可只读加载	不能	可以
	存储平台无关	不能	可以
	可从网络文件系统访问	通常，不	可以
	组访问权处理	对于用户的 umask 设置十分敏感，最好只由一个用户访问。	对 umask 设置不敏感
伸缩性	版本库磁盘使用情况	较大（特别是没有清除日志时）	较小
	修订版本树的数量	数据库，没有限制	许多古老的本地文件系统在处理单一目录包含上千个条目时出现问题。
	有很多文件的目录	较慢	较快
性能	检出最新的代码	没有有意义的区别	没有有意义的区别
	大的提交	整体较慢，但是在整个提交周期中消耗被分摊	较快，但是最后较长的延时可能会导致客户端操作超时

两种后端都有优点和缺点，没有一种更加“正式”，尽管新的 FSFS 在 Subversion1.2 成为缺省数据存储，两者用来存储版本化数据都是可靠的。但是就象你在表 5.1 “”看到的，FSFS 后端在部署场景中提供了更多的灵活性，更灵活意味着你很难错误的配置。那些原因—加上不使

用 Berkeley DB 意味着在这个系统有更少的组件—这就是为什么今天几乎所有的人都使用 FSFS 来创建新的版本库。

幸运的是，大多数访问 Subversion 的程序不会在意其所用的后端数据存储。而且你不必一定要使用你最初的数据存储方法—如果后来你改变了主意，Subversion 提供了移植版本库数据到另一个版本库的方法，我们会在后面详细讨论。

下面的小节提供了数据存储类型更加详细的介绍。

Berkeley DB

在 Subversion 的初始设计阶段，开发者因为多种原因而决定采用 Berkeley DB，比如它的开源协议、事务支持、可靠性、性能、简单的 API、线程安全、支持游标等。

Berkeley DB 提供了真正的事务支持－这或许是它最强大的特性，访问你的 Subversion 版本库的多个进程不必担心偶尔会破坏其他进程的数据。事务系统提供的隔离对于任何给定的操作，Subversion 版本库代码看到的只是数据库的静态视图－而不是一个在其他进程影响不断变化的数据库－并能够根据该视图作出决定。如果该决定正好同其他进程所做操作冲突，整个操作会回滚，就像什么都没有发生一样，并且 Subversion 会优雅的再次对更新的静态视图进行操作。

Berkeley DB 另一个强大的特性是热备份－不必“脱机”就可以备份数据库环境的能力。我们将会在“版本库备份”一节讨论如何备份你的版本库，能够不停止系统对版本库做全面备份的好处是显而易见的。

Berkeley DB 同时是一个可信赖的数据库系统。Subversion 利用了 Berkeley DB 可以记日志的便利，这意味着数据库先在磁盘上写一个日志文件，描述它将要做的修改，然后再做这些修改。这是为了确保如果如果任何地方出了差错，数据库系统能恢复到先前的*检查点*—一个日志文件认为没有错误的位置，重新开始事务直到数据恢复为一个可用的状态。关于 Berkeley DB 日志文件的更多信息请查看“管理磁盘空间”一节。

但是每朵玫瑰都有刺，我们也必须记录一些 Berkeley DB 已知的缺陷。首先，Berkeley DB 环境不是跨平台的。你不能简单的拷贝一个在 Unix 上创建的 Subversion 版本库到一个 Windows 系统并期望它能够正常工作。尽管 Berkeley DB 数据库的大部分格式是不受架构约束的，但环境还是有一些方面没有独立出来。其次，使用 Berkeley DB 的 Subversion 不能在 95/98 系统上运行—如果你需要将版本库建在一个 Windows 机器上，请装到 Windows2000 或 WindowsXP 上。

然而 Berkeley DB 对于在网络共享上工作提出了一组规范，[28]大多数网络文件系统和应用没有实现这个要求，所以不能允许在网络共享上的 BDB 后端版本库被多个客户端同时访问（首先要知道版本库存放在网络共享上是非常普遍的）。

如果你尝试在不顺从的远程文件系统上使用 Berkeley DB，结果是不可预知的—你会立刻看到神秘的错误，或者是在发生隐含错误之后几个月之后才发现。你必须认真考虑在网络共享情况下使用 FSFS 数据存储。

最后，因为 Berkeley DB 的库直接链接到了 Subversion 中，它对于中断比典型的关系型数据库系统更为敏感。大多数 SQL 系统，举例来说，有一个主服务进程来协调对数据库表的访问。如果一个访问数据库的程序因为某种原因出现问题，数据库守护进程察觉到连接中断会做一些清理。因为数据库守护进程是唯一访问数据库表的进程，应用程序不需要担心访问许可的冲突。但是，这些情况与 Berkeley DB 不同。Subversion（和使用 Subversion 库的程序）直接访问数据库的表，这意味着如果有一个程序崩溃，就会使数据库处于一个暂时的不一致、不可访问的状态。当这种情况发生时，管理员需要让 Berkeley DB 恢复到一个检查点，这的确有点讨厌。除了崩溃的进程，还有一些情况能让版本库出现异常，比如程序在数据库文件的所有权或访问权限上发生冲突。

Berkeley DB 4.4（对应 Subversion 1.4 和更高）提供了在需要恢复时自动恢复 Berkeley DB 环境的能力，当 Subversion 进程发现任何以前进程未清理的连接，就会执行所有可能的恢复，然后就当什么都没有发生一样继续执行。这样不会完全消除版本库楔住的可能，但是大大减少了人工干预恢复的数量。

因为 Berkeley DB 是这样快速和可伸缩，最好是使用某种单用户单服务进程方式处理—例如 Apache 的 **`httpd`** 或 **`svnserve`**（见第 6 章 *服务*

配置）—而最好不要使用许多不同的用户通过 file://或 svn+ssh://的 URL 访问的方法。如果使用多个用户直接访问 Berkeley DB 版本库的，请确定要读“支持多种版本库访问方法”一节。

FSFS

在 2004 年中期，另一种版本库存储系统慢慢形成了：一种不需要数据库的存储系统。FSFS 版本库在单一文件中存储修订版本树，所以版本库中所有的修订版本都在一个子文件夹中有限的几个文件里。事务在单独的子目录中被创建，创建完成后，一个单独的事务文件被创建并移动到修订版本目录，这保证提交是原子性的。因为一个修订版本文件是持久不可改变的，版本库也可以做到“热”备份，就象 Berkeley DB 版本库一样。

修订版本文件格式代表了一个修订版本的目录结构，文件内容，和其它修订版本树中相关信息。不像 Berkeley DB 数据库，这种存储格式可跨平台并且与 CPU 架构无关。因为没有日志或用到共享内存的文件，数据库能被网络文件系统安全的访问和在只读环境下检查。缺少数据库花消同时也意味着版本库的总体体积可以稍小一点。

FSFS 也有一种不同的性能特性。当提交大量文件时，FSFS 可以更快的追加条目。另一方面，FSFS 通过写入与上一个版本比较的变化来记录新版本，这也意味着获取最新修订版本时会比 Berkeley DB 慢一点，提交时 FSFS 也会有一个更长的延迟，在某些极端情况下会导致客护端在等待回应时超时。

最重要的区别是当出现错误时 FSFS 不会楔住的能力。如果使用 Berkeley DB 的进程发生许可错误或突然崩溃，数据库会一直无法使用，直到管理员恢复。假如在应用 FSFS 版本库时发生同样的情况，版本库不会受到任何干扰，最坏情况下也就是会留下一些事务数据。

FSFS 的唯一真实的争议是其相对于 Berkeley DB 的不成熟，不像 Berkeley DB 有着多年历史的，而且有专门的开发团队，强大的 Oracle 会提供支持。[29]FSFS 在工程上更新一点，在 Subversion1.4 之前，我们还未一些确实很严重的数据一致性问题颤抖，尽管只在非常罕见的情况下发生，然而还是发生了。但是，FSFS 还是很快被一些最大的开放和私有 Subversion 版本库所采用，并且承诺了在跨平台时的有较少的麻烦。

创建和配置你的版本库

在“版本库开发策略”一节，我们我们看了一些在创建和配置 Subversion 版本库之前需要做的重要决定，现在我们最终要干活了！在本小节，我们要看看如何真实的创建一个 Subversion 版本库，并配置它在特定版本库事件执行自定义动作。

创建版本库

创建一个 Subversion 版本库出乎寻常的简单。 Subversion 提供的 **svnadmin** 工具，有一个执行这个功能的子命令（create）。

```
$ svnadmin create /path/to/repos
```

这样在目录*/path/to/repos* 使用默认数据存储方式创建了一个新的版本库。在 Subversion 1.2 之前，缺省值是 Berkeley DB；而现在是 FSFS。你可以通过`--fs-type` 参数明确地指定文件系统类型，可选的值包括 `fsfs` 和 `bdb`。

```
$ # Create an FSFS-backed repository
$ svnadmin create --fs-type fsfs /path/to/repos
$
# Create a Berkeley-DB-backed repository
$ svnadmin create --fs-type bdb /path/to/repos
$
```

运行这个命令之后，你有了一个 Subversion 版本库。

你可能已经注意到了，**svnadmin** 命令的路径参数只是一个普通的文件系统路径，而不是一个 **svn** 客户端程序访问版本库时使用的 URL。**svnadmin** 和 **svnlook** 都被认为是服务器端工具—它们在版本库所在的机器上使用，用来检查或修改版本库，不能通过网络来执行任务。一个 Subversion 的新手通常会犯的错误，就是试图将 URL（甚至“本地”`file:`路径）传给这两个程序。

这个命令在目录*/path/to/repos* 创建了一个新的版本库。这个新的版本库会以修订版本版本 0 开始其生命周期，里面除了最上层的根目录(*/*)，什

么都没有。刚开始，修订版本 0 有一个修订版本属性 svn:date，设置为版本库创建的时间。

现在你有了一个版本库，可以用户化了。

一般来说，版本库除了一小部分—例如配置文件和钩子脚本，你不要（也不需要）手动干预版本库。**svnadmin** 工具应该足以用来处理对版本库的任何修改，或者你也可以使用第三方工具（比如 Berkeley DB 的工具包）来调整部分版本库。*不*要尝试通过处理版本库数据存储文件手工修改版本控制历史，

实现版本库钩子

*钩子*是通过版本库事件触发，例如新版本的创建或一个未版本化属性的修改。一些钩子（叫做“pre hooks”）在事件发生前运行，可以用来报告发生了什么以及防止它发生。还有一些钩子（“post hooks”）在版本库事件之后发生，只是用来报告。每个钩子能够获得事件的足够信息，例如提出的（或完成的）版本库修改细节，还有触发事件的用户名。

默认情况下，*hooks* 子目录中包含各种版本库钩子模板。

```
$ ls repos/hooks/

post-commit.tmpl     post-unlock.tmpl
pre-revprop-change.tmpl

post-lock.tmpl               pre-commit.tmpl
pre-unlock.tmpl

post-revprop-change.tmpl  pre-lock.tmpl     start-commit.tmpl
```

对每种 Subversion 版本库支持的钩子的都有一个模板，通过查看这些脚本的内容，你能看到是什么事件触发了脚本及如何给传脚本传递数据。同时，这些模版也是如何使用这些脚本，结合 Subversion 支持的工具来完成有用任务的例子。要实际安装一个可用的钩子，你需要在 *repos/hooks* 目录下安装一些与钩子同名（如 **`start-commit`** 或者 **`post-commit`**）的可执行程序或脚本。

在 Unix 平台上，这意味着要提供一个与钩子同名的脚本或程序（可能是 shell 脚本，Python 程序，编译过的 c 语言二进制文件或其他东西）。当然，脚本模板文件不仅仅是展示了一些信息—在 Unix 下安装钩子最简单的办法就是拷贝这些模板，并且去掉.tmpl 扩展名，然后自定义钩子的内容，确定脚本是可运行的。Windows 用文件的扩展名来决定一个程序是否可运行，所以你要使程序的基本名与钩子同名，同时，它的扩展名是 Windows 系统所能辨认的，例如 *exe*、*com* 和批处理的 *bat*。

由于安全原因，Subversion 版本库在一个空环境中执行钩子脚本—就是没有设置任何环境变量，甚至没有$PATH 或%PATH%。由于这个原因，许多管理员会感到很困惑，它们的钩子脚本手工运行时正常，可在 Subversion 中却不能运行。要注意，必须在你的钩子中设置好环境变量或为你的程序指定好绝对路径。

Subversion 会试图以当前访问版本库的用户身份执行钩子。通常，对版本库的访问总是通过 Apache HTTP 服务器和 mod_dav_svn 进行，因此，执行钩子的用户就是运行 Apache 的用户。钩子本身需要具有操作

系统级的访问许可，用户可以运行它。另外，其它被钩子直接或间接使用的文件或程序（包括 Subversion 版本库本身）也要被同一个用户访问。换句话说，要注意潜在的访问控制问题，它可能会让你的钩子无法按照你的目的顺利执行。

Subversion 版本库有 9 种钩子实现，你可以在“版本库钩子”一节获得每个的信息。作为一个版本库管理员，你需要决定你要实现的钩子（通过提供家当名称和执行许可的程序）类型和方法，这种决策需要对版本库的部署非常熟悉。例如，如果你使用服务器配置方式，通过版本库检测用户名称和权限，你不需要通过钩子系统实现访问控制。

在 Subversion 社区和其他地方都不缺 Subversion 钩子，这些脚本覆盖了广泛的工具—基本的访问控制，政策相关检查，问题追踪集成，email 或提交通知等等。关于最常用的钩子程序的讨论，可以看附录 D, *第三方工具*，如果你希望写你自己的，可以看第 8 章 *嵌入 Subversion*。

尽管经过调整钩子脚本可以作任何事情，但钩子脚本的作者仍会受到一些限制：不要修改使用钩子脚本修改提交事务，因为使用钩子脚本自动修改错误或提交文件的政策违例的尝试会导致问题。Subversion 会在客户端缓存对应的版本库数据，如果你这样修改了提交事务，这些缓存就进入了未知的状态，这种不一致会导致令人吃惊和预想不到的行为。作为对事物修改的替换，你可以简单的在 *pre-commit* 确认事物信息并且拒绝提交，如果这样满足不了需求，作为额外的奖赏，你的用户会学会小心顺从的工作习惯。

Berkeley DB 配置

Berkeley DB 环境是对一个或多个数据库、日志文件、区域文件和配置文件的封装。Berkeley DB 环境对许多参数有自己的缺省值，例如任何时间里可用的数据库锁定数目、日志文件的最大值等。Subversion 文件系统会使用 Berkeley DB 的默认值。 不过，有时候你的特定版本库与它独特的数据集合和访问类型，可能需要不同的配置选项。

你的版本库的 Berkeley 配置文件位于 *db* 目录的 *db/DB_CONFIG*， Subversion 在创建版本库时自己创建了这个文件。这个文件初始时包含了一些默认选项，也包含了 Berkeley DB 在线文档，使你能够了解这些选项是做什么的。当然，你也可以为你的 *DB_CONFIG* 文件添加任何 Berkeley DB 支持的选项。需要注意到，虽然 Subversion 不会尝试读取并解析这个文件，或使用其中的设置，你一定要避免会导致 Berkeley DB 按照 Subversion 代码不习惯的方式工作的修改。另外，*DB_CONFIG* 的修改在复原数据库环境（用 **`svnadmin recover`**）之前不会产生任何效果。

版本库维护

维护一个 Subversion 版本库是一项令人沮丧的工作，主要因为有数据库后端与生俱来的复杂性。做好这项工作需要知道一些工具——它们是什么，什么时候用以及如何使用。这一节将会向你介绍 Subversion 自

带的版本库管理工具，以及如何使用它们来完成诸如版本库移植、升级、备份和整理之类的任务。

管理员的工具箱

Subversion 提供了一些用来创建、查看、修改和修复版本库的工具。让我们首先详细了解一下每个工具，然后，我们再看一下仅在 Berkeley DB 后端分发版本中提供的版本数据库工具。

svnadmin

svnadmin 程序是版本库管理员最好的朋友。除了提供创建 Subversion 版本库的功能，这个程序使你可以维护这些版本库。**svnadmin** 的语法同其他 Subversion 命令类似：

```
$ svnadmin help

general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]

Type 'svnadmin help <subcommand>' for help on a specific subcommand.

Type 'svnadmin --version' to see the program version and FS modules.



Available subcommands:

   crashtest

   create

   deltify

…
```

我们已经讨论了 **svnadmin** 的 create 子命令（参照“创建和配置你的版本库”一节），本章后面我们会详细讲解大多数其他的子命令，关于所有的子命令你可以参考“ **svnadmin** ”一节。

svnlook

svnlook 是 Subversion 提供的用来查看版本库中不同的修订版本和事务（正在产生的修订版本）。这个程序不会修改版本库内容－这是个“只读”的工具。**svnlook** 通常用在版本库钩子程序中，用来记录版本库即将提交（**用在 pre-commit 钩子时**）或者已经提交的（用在 **post-commit** 钩子时）修改。版本库管理员可以将这个工具用于诊断。

svnlook 的语法很直接：

```
$ svnlook help

general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]

Note: any subcommand which takes the '--revision' and '--transaction'

      options will, if invoked without one of those options, act on

      the repository's youngest revision.

Type 'svnlook help <subcommand>' for help on a specific subcommand.

Type 'svnlook --version' to see the program version and FS modules.

…
```

几乎 **svnlook** 的每一个子命令都能操作修订版本或事务树，显示树本身的信息，或是它与版本库中上一个修订版本的不同。你可以用 --revision (-r) 和 --transaction (-t)选项指定要查看的修订版本或事务。如果没有指定--revision (-r)和--transaction (-t)选项，**svnlook** 会检查版本库最新的（或者说“HEAD”）修订版本。所以当 19 是位于*/path/to/repos* 的版本库的最新版本时，如下的两个名字起到相同的效果：

```
$ svnlook info /path/to/repos
$ svnlook info /path/to/repos -r 19
```

这些子命令的唯一例外是 **svnlook youngest**，它不需要任何选项，只会打印出版本库的最新修订版本号。

```
$ svnlook youngest /path/to/repos
19
```

请记住只能浏览未提交的事物，大多数版本库没有这样的事物，因为事物要么是已经提交的(也就是你可以--revision (-r)访问的修订版本)，要么是退出的和删除的。

svnlook 的输出被设计为人和机器都易理解，拿 info 子命令举例来说：

```
$ svnlook info /path/to/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
```

```
27

Added the usual

Greek tree.
```

info 子命令的输出定义如下：

1. 作者，后接换行。
2. 日期，后接换行。
3. 日志消息的字数，后接换行。
4. 日志信息本身， 后接换行。

这种输出是人可阅读的，像是时间戳这种有意义的条目，使用文本表示，而不是其他比较晦涩的方式（例如许多无聊的人推荐的十亿分之一秒的数量）。这种输出也是机器可读的—因为日志信息可以有多行，没有长度的限制，**svnlook** 在日志消息之前提供了消息的长度，这使得脚本或者其他这个命令的封装器能够针对日志信息做出许多职能的决定，或仅仅是在这个输出成为最后一个字节之前应该略过多少字节。

svnlook 还可以做很多别的查询：显示我们先前提到的信息的一些子集，递归显示版本目录树，报告指定的修订版本或事务中哪些路径曾经被修改过，显示对文件和目录做过的文本和属性的修改，等等。

“ **svnlook** ”一节是 **svnlook** 命令能接受子命令的完全特性参考。

svndumpfilter

虽然在管理员的日常工作中并不会经常使用，不过 **svndumpfilter** 提供了一项特别有用的功能—可以简单快速的作为 Subversion 版本库历史的以路径为基础的过滤器。

svndumpfilter 的语法如下：

```
$ svndumpfilter help

general usage: svndumpfilter SUBCOMMAND [ARGS & OPTIONS ...]

Type "svndumpfilter help <subcommand>" for help on a specific
subcommand.

Type 'svndumpfilter --version' to see the program version.

—

Available subcommands:

   exclude

   include

   help (?, h)
```

有意义的子命令只有两个。你可以使用这两个子命令说明你希望保留和不希望保留的路径：

exclude

将指定路径的数据从转储数据流中排除。

include

将指定路径的数据添加到转储数据流中。

关于这些子命令和 **svndumpfilter** 的唯一目的的，可以见“过滤版本库历史”一节。

svnsync

svnsync 程序是 Subversion 1.4 版的新特性，提供了维护一个只读版本库镜像的全部功能。这个程序只有一个工作—将一个版本库的历史转移到另一个，尽管有几种方法，但这种方法的主要特点是可以远程操作—“源”，“目标”[30]版本库以及 **svnsync** 程序可能在不同的计算机上。

就像你期望的，**svnsync** 的语法与本节提到的其他命令非常类似。

```
$ svnsync help

general usage: svnsync SUBCOMMAND DEST_URL [ARGS & OPTIONS ...]

Type 'svnsync help <subcommand>' for help on a specific
subcommand.

Type 'svnsync --version' to see the program version and RA
modules.



Available subcommands:

   initialize (init)

   synchronize (sync)

   copy-revprops

   help (?, h)

$
```

我们会在“版本库复制”一节详细讨论使用 **svnsync** 实现版本库复制。

Berkeley DB 工具

如果你使用 Berkeley DB 版本库，那么所有纳入版本控制的文件系统结构和数据都储存在一系列数据库的表中，而这个目录就是版本库的 *db/*。这个子目录是一个标准的 Berkeley DB 环境目录，可以应用任何 Berkeley 数据库工具进行操作，通常这些工具随 Berkeley DB 发布。

对于 Subversion 的日常使用来说，这些工具并没有什么用处。大多数 Subversion 版本库必须的数据库操作都集成到 **svnadmin** 工具中。比如，**svnadmin list-unused-dblogs** 和 **svnadmin list-dblogs** 实现了 Berkeley **db_archive** 命令功能的一个子集，而 **svnadmin recover** 则起到了 **db_recover** 工具的作用。

当然，还有一些 Berkeley DB 工具有时是有用的。**db_load** 和 **db_dump** 分别将 Berkeley DB 数据库中的键值对以特定的格式读写文件。Berkeley 数据库本身不支持跨平台转移，这两个工具在这样的情况下就可以实现在平台间转移数据库的功能，而无需关心操作系统或机器架构。就像我们以前描述的，你可以使用 **svnadmin dump** 和 **svnadmin load** 实现类似的目的，但是 **db_dump** 和 **db_load** 可以更快一点，它们也可以协助 Berkeley DB 的 hacker 来篡改 BDB 后端的数据，这是 Subversion 工具不允许的。此外，**db_stat** 工具能够提供关于 Berkeley DB 环境的许多有用信息，包括详细的锁定和存储子系统的统计信息。

关于 Berkeley DB 工具的更多信息，可以访问 Oracle 网站的 Berkeley DB 文档部分，在 http://www.oracle.com/technology/documentation/berkeley-db/db/。

修正提交消息

有时用户输入的日志信息有错误（比如拼写错误或者内容错误）。如果配置版本库时设置了（使用 pre-revprop-change 和 post-revprop-change 钩子；参见“实现版本库钩子”一节）允许用户在提交后修改日志信息的选项，那么用户可以使用 **svn** 程序的 propset 命令（参见第 9 章 *Subversion 完全参考*）“修正”日志信息中的错误。不过为了避免永远丢失信息，Subversion 版本库通常设置为仅能由管理员修改非版本化属性（这也是默认的选项）。

如果管理员想要修改日志信息，那么可以使用 **svnadmin setlog** 命令。这个命令从指定的文件中读取信息，取代版本库中某个修订版本的日志信息（svn:log 属性）。

```
$ echo "Here is the new,  correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

即使是 **svnadmin setlog** 命令也受到限制。pre- 和 post-revprop-change 钩子同样会被触发，因此必须进行相应的设置才能允许修改非版本化属性。不过管理员可以使用 **svnadmin setlog** 命令的--bypass-hooks 选项跳过钩子。

不过需要注意的是，一旦跳过钩子也就跳过了钩子所提供的所有功能，比如邮件通知（通知属性有改动）、系统备份（可以用来跟踪非版本化的属性变更）等等。换句话说，要留心你所作出的修改，以及你作出修改的方式。

管理磁盘空间

虽然存储器的价格在过去的几年里以让人难以致信的速度滑落，但是对于那些需要对大量数据进行版本管理的管理员们来说，磁盘空间的消耗依然是一个重要的因素。版本库每增加一个字节都意味着需要多一个字节的磁盘空间进行备份，对于多重备份来说，就需要消耗更多的磁盘空间。Berkeley DB 版本库的主要存储机制是基于一个复杂的数据库系统建立的，因此了解一些数据性质是有意义的，比如哪些数据必须保持在线，哪些数据需要备份、哪些数据可以安全的删除等等。

Subversion 如何节约磁盘空间

为了尽可能减小版本库的体积，Subversion 在版本库中采用了*增量化技术*（或称为“增量存储技术”）。增量化技术可以将一组数据表示为相对于另一组数据的不同。如果这两组数据十分相似，增量化技术就可以仅保存其中一组数据以及两组数据的差别，而不需要同时保存两组数据，从而节省了磁盘空间。每次一个文件的新版本提交到版本库，版本库就会将之前的版本（之前的多个版本）相对于新版本做增量化处理。采用了这项技术，版本库的数据量大小基本上是可以估算出来的—主要是版本化的文件的大小—并且远小于“全文”保存所需的数据量。而

Subversion 1.4 以后，空间存储变得更为节省—现在文件内容的全文本身都是压缩的了。

由于 Subversion 版本库的增量化数据保存在单一 Berkeley DB 数据库文件中，减少数据的体积并不一定能够减小数据库文件的大小。但是，Berkeley DB 会在内部记录未使用的数据库文件区域，并且在增加数据库文件大小之前会首先使用这些未使用的区域。因此，即使增量化技术不能立杆见影的节省磁盘空间，也可以极大的减慢数据库的膨胀速度。

删除终止的事务

尽管不太常见，Subversion 的提交进程也有失败，同时留下将要生成的修订版本—未提交的事物和所有随之的文件和目录修改。出现这种情况可能有以下原因：客户端的用户粗暴的结束了操作，操作过程中出现网络故障，等等。不管是什么原因，死亡的事务总是有可能会出现。这类事务不会产生什么负面影响，仅仅是消耗了一点点磁盘空间。不过，严厉的管理员总是希望能够将它们清除出去。

可以使用 **svnadmin** 的 `lstxns` 命令列出当前的事务名。

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

将输出的结果条目作为 **svnlook**（设置--transaction (-t)选项）的参数，就可以获得事务的详细信息，如事务的创建者、创建时间，事务已作出的更改类型，由这些信息可以判断出是否可以将这个事务安全的删除。如果可以安全删除，那么只需将事务名作为参数输入到 **svnadmin rmtxns**，就可以将事务清除掉了。其实 rmtxns 子命令可以直接以 lstxns 的输出作为输入进行清理。

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`
$
```

在按照上面例子中的方法清理版本库之前，你或许应该暂时关闭版本库和客户端的连接。这样在你开始清理之前，不会有正常的事务进入版本库。例 5.1 “txn-info.sh（报告异常事务）”中的 shell 脚本可以用来迅速获得版本库中异常事务的信息。

例 5.1. txn-info.sh（报告异常事务）

```
#!/bin/sh


### Generate informational output for all outstanding
transactions in
### a Subversion repository.


REPOS="${1}"
if [ "x$REPOS" = x ] ; then
  echo "usage: $0 REPOS_PATH"
```

```
  exit

fi



for TXN in `svnadmin lstxns ${REPOS}`; do

  echo "---[ Transaction
${TXN} ]-------------------------------------------"

  svnlook info "${REPOS}" -t "${TXN}"

done
```

该命令的输出主要由多个 **svnlook info**（参见“svnlook”一节）的输出组成，类似于下面的例子：

```
$ txn-info.sh myrepos

---[ Transaction
19 ]-------------------------------------------

sally

2001-09-04 11:57:19 -0500 (Tue,  04 Sep 2001)

0

---[ Transaction
3a1 ]------------------------------------------

harry

2001-09-10 16:50:30 -0500 (Mon,  10 Sep 2001)

39

Trying to commit over a faulty network.

---[ Transaction
a45 ]------------------------------------------
```

```
sally

2001-09-12 11:09:28 -0500 (Wed,  12 Sep 2001)

0

$
```

一个废弃了很长时间的事务通常是提交错误或异常中断的结果。事务的时间戳可以提供给我们一些有趣的信息，比如一个进行了 9 个月的操作居然还是活动的等等。

简言之，作出事务清理的决定前应该仔细考虑一下。许多信息源—比如 Apache 的错误和访问日志，已成功完成的 Subversion 提交日志等等—都可以作为决策的参考。当然，管理员还可以直接和那些似乎已经死亡事务的提交者直接交流（比如通过邮件），来确认该事务确实已经死亡了。

删除不使用的 Berkeley DB 日志文件

目前为止，Subversion 版本库中耗费磁盘空间的最大凶手是日志文件，每次 Berkeley DB 在修改真正的数据文件之前都会进行预写入（pre-writes）操作。这些文件记录了数据库从一个状态变化到另一个状态的所有动作——数据库文件反映了特定时刻数据库的状态，而日志文件则记录了所有状态变化的信息。因此，日志文件会以很快的速度膨胀起来。

幸运的是，从版本 4.2 开始，Berkeley DB 的数据库环境无需额外的操作即可删除无用的日志文件。如果编译 **`svnadmin`** 时使用了高于 4.2

版本的 Berkeley DB，那么由此 **svnadmin** 程序创建的版本库就具备了自动清除日志文件的功能。如果想屏蔽这个功能，只需设置 **svnadmin create** 命令的--bdb-log-keep 选项即可。如果创建版本库以后想要修改关于此功能的设置，只需编辑版本库中 *db* 目录下的 *DB_CONFIG* 文件，注释掉包含 set_flags DB_LOG_AUTOREMOVE 内容的这一行，然后运行 **svnadmin recover** 强制设置生效就行了。查阅“Berkeley DB 配置”一节获得更多关于数据库配置的帮助信息。

如果不自动删除日志文件，那么日志文件会随着版本库的使用逐渐增加。这多少应该算是数据库系统的特性，通过这些日志文件可以在数据库严重损坏时恢复整个数据库的内容。但是一般情况下，最好是能够将无用的日志文件收集起来并删除，这样就可以节省磁盘空间。使用 **svnadmin list-unused-dblogs** 命令可以列出无用的日志文件：

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033
…
$ rm `svnadmin list-unused-dblogs /path/to/repos`
## disk space reclaimed!
```

BDB 后端的版本库的日志文件如果是用来作为备份或容灾恢复计划时，不要使用日志文件的自动删除特性。从日志文件重新构建版本库数据只

有在所有的日志文件都存在时才能完成，如果有一些文件在别的程序将其拷贝之前就已经被删除了，不完整的备份日志文件就没有用了。

Berkeley DB 恢复

就像在“Berkeley DB”一节提到的，如果没有正确的关闭，Berkeley DB 版本库有时候会进入冻结的状态。当发生这种情况时，管理员需要恢复版本库进入一致的状态。当然这种情况只发生在 BDB 版本库，FSFS 版本库不会有这种情况。对于使用 Subversion 1.4 和 Berkeley DB 4.4 或更新版本的用户，你一定发现 Subversion 对于这种情况已经更富弹性，但是 Berkeley DB 楔住的情况还是会发生，管理员需要知道如何安全的处理种情况。

Berkeley DB 使用一种锁机制保护版本库中的数据。锁机制确保数据库不会同时被多个访问进程修改，也就保证了从数据库中读取到的数据始终是稳定而且正确的。当一个进程需要修改数据库中的数据时，首先必须检查目标数据是否已经上锁。如果目标数据没有上锁，进程就将它锁上，然后作出修改，最后再将锁解除。而其它进程则必须等待锁解除后才能继续访问数据库中的相关内容。（你对这种锁无能为力，作为一个用户，可以应用版本库的版本化文件；我们会在“锁定”的三种含义讨论因为术语冲突导致的概念混淆。）

在操作 Subversion 版本库的过程中，致命错误（如内存或硬盘空间不足）或异常中断可能会导致某个进程没能及时将锁解除。结果就是后端

的数据库系统被“塞住”了。一旦发生这种情况，任何访问版本库的进程都会挂起(每个访问进程都在等待锁被解除，但是锁已经无法解除了)。

如果你的版本库出现这种情况，没什么好惊慌的。Berkeley DB 的文件系统采用了数据库事务、检查点以及预写入日志等技术来确保只有灾难性的事件[31]才能永久性的破坏数据库环境。所以虽然一个过于稳重的版本库管理员通常都会按照某种方案进行大量的版本库离线备份，不过不要急着通知你的管理员进行恢复。

然后，使用下面的方法试着“恢复”你的版本库：

1. 确保没有其它进程访问（或者试图访问）版本库。对于网络版本库，这意味着关闭 Apache HTTP Server 或 svnserve。
2. 成为版本库的拥有者和管理员。这一点很重要，如果以其它用户的身份恢复版本库，可能会改变版本库文件的访问权限，导致在版本库“恢复”后依旧无法访问。
3. 运行命令 **svnadmin recover /path/to/repos**。输出如下：

```
4. Repository lock acquired。
5. Please wait; recovering the repository may take some
   time...
6.
7. Recovery completed.
8. The latest repos revision is 19.
```

此命令可能需要数分钟才能完成。

9. 重新启动服务进程。

这个方法能修复几乎所有版本库锁住的问题。记住，要以数据库的拥有者和管理员的身份运行这个命令，而不一定是 root 用户。恢复过程中可能会使用其它数据存储区（例如共享内存区）重建一些数据库文件。如果以 root 用户身份恢复版本库，这些重建的文件拥有者将变成 root 用户，也就是说，即使恢复了到版本库的连接，一般的用户也无权访问这些文件。

如果因为某些原因，上面的方法没能成功的恢复版本库，那么你可以做两件事。首先，将破损的版本库保存到其它地方，然后从最新的备份中恢复版本库。然后，发送一封邮件到 Subversion 用户列表（地址是：<users@subversion.tigris.org>），写清你所遇到的问题。对于 Subversion 的开发者来说，数据安全是最重要的问题。

版本库数据的移植

Subversion 文件系统将数据保存在许多数据库表中，而这些表的结构只有 Subversion 开发者们才了解（也只有他们才感兴趣），不过，有些时候我们会想到把所有或一部分数据转移到另一个版本库。

Subversion 提供了转储版本库的功能，一个版本库转储流（当存放在磁盘上叫做“dumpfile”）是一种可移植的，普通文件格式，可以用来描述版本库的不同版本—什么发生了修改，谁做的，何时等等。这种转储流是解析版本化历史的主要机制—全部或部分，包含或部包含修改—在

版本库之间。Subversion 也提供了创建和加载这些转储流的工具—对应的 **svnadmin dump** 和 **svnadmin load** 子命令。

虽然 Subversion 版本库转储格式包含了人可读的部分和熟悉的结构（类似 RFC-822 格式，大多数邮件使用的），它不是纯文本的格式，这种格式必须作为二进制文件格式处理，对修改高度敏感。例如，许多文本编辑器会破坏这种文件的内容，通常是因为自动换行符替换。

有很多导出和加载 Subversion 版本库数据的方法，在 Subversion 的早期阶段，最主要的原因是 Subversion 本身的进化。随着 Subversion 的成熟，对于数据后端模式的改变会导致更多的兼容性问题，所以用户需要使用旧版本的 Subversion 将版本库数据导出，然后用新版的版本库加载内容到新建的版本库。目前，这种类型的模式修改从 Subversion 1.0 版本还没有发生，而且 Subversion 开发者也许诺不会强制用户在小版本（如 1.3 到 1.4）升级之间导入和导出版本库。但是也有一些其它原因导出和导入，包括重新部署 Berkeley DB 到版本库到新的 OS 或 CPU 架构，在 Berkeley DB 和 FSFS 后端之间切换，或者（我们会在“过滤版本库历史”一节覆盖）从版本库历史中清理文件。

无论你是什么原因需要移植版本库历史，都可以直接使用 **svnadmin dump** 和 **svnadmin load**。**svnadmin dump** 命令会将版本库中的修订版本数据按照特定的格式输出到转储流中，转储数据会输出到标准输出，而提示信息会输出到标准错误。这就是说，可以将转储数据存储到文件中，而同时在终端窗口中监视运行状态，例如：

```
$ svnlook youngest myrepos

26

$ svnadmin dump myrepos > dumpfile

* Dumped revision 0.

* Dumped revision 1.

* Dumped revision 2.

…

* Dumped revision 25.

* Dumped revision 26.
```

最后，版本库中的指定的修订版本数据被转储到一个独立的文件中（在上面的例子中是 *dumpfile*）。注意，**svnadmin dump** 从版本库中读取修订版本树与其它“读者”（比如 **svn checkout**）的过程相同，所以可以在任何时候安全的运行这个命令。

另一个命令，**svnadmin load**，从标准输入流中读取 Subversion 转储数据，并且高效的将数据转载到目标版本库中。这个命令的提示信息输出到标准输出流中：

```
$ svnadmin load newrepos < dumpfile

<<< Started new txn, based on original revision 1

     * adding path : A ... done.

     * adding path : A/B ... done.

     …

------- Committed new rev 1 (loaded from original rev 1) >>>
```

```
<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.

------- Committed new rev 2 (loaded from original rev 2) >>>

…

<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

load 命令的结果就是添加一些新的修订版本—与使用普通 Subversion 客户端直接提交到版本库相同。正像一次简单的提交，你也可以使用钩子脚本在每次 load 的开始和结束执行一些操作。通过传递 --use-pre-commit-hook 和--use-post-commit-hook 选项给 **svnadmin load**，你可以告诉 Subversion 的对每一个加载修订版本执行

pre-commit 和 post-commit 钩子脚本，可以利用这个选项确保这种提交也能通过一般提交的检验。当然，你要小心使用这个选项，你一定不想接受一大堆提交邮件。你可以查看“实现版本库钩子”一节来得到更多相关信息。

既然 **svnadmin** 使用标准输入流和标准输出流作为转储和装载的输入和输出，那么更漂亮的用法是（管道两端可以是不同版本的 **svnadmin**：

```
$ svnadmin create newrepos

$ svnadmin dump oldrepos | svnadmin load newrepos
```

默认情况下，转储文件的体积可能会相当庞大——比版本库自身大很多。这是因为在转储文件中，每个文件的每个版本都以完整的文本形式保存下来。这种方法速度很快，而且很简单，尤其是直接将转储数据通过管道输入到其它进程中时（比如一个压缩程序，过滤程序，或者一个装载进程）。不过如果要长期保存转储文件，那么可以使用--deltas 选项来节省磁盘空间。设置这个选项，同一个文件的数个连续修订版本会以增量式的方式保存—就像储存在版本库中一样。这个方法较慢，但是转储文件的体积则基本上与版本库的体积相当。

之前我们提到 **svnadmin dump** 输出指定范围内的修订版本，使用 --revision (-r)选项可以指定一个单独的修订版本，或者一个修订版本的范围。如果忽略这个选项，所有版本库中的修订版本都会被转储。

```
$ svnadmin dump myrepos -r 23 > rev-23.dumpfile
```

```
$ svnadmin dump myrepos -r 100:200 > revs-100-200.dumpfile
```

Subversion 在转储修订版本时，仅会输出与前一个修订版本之间的差异，通过这些差异足以从前一个修订版本中重建当前的修订版本。换句话说，在转储文件中的每一个修订版本仅包含这个修订版本作出的修改。这个规则的唯一一个例外是当前 **svnadmin dump** 转储的第一个修订版本。

默认情况下，Subversion 不会把转储的第一个修订版本看作对前一个修订版本的更改。 首先，转储文件中没有比第一个修订版本更靠前的修订版本了！其次，Subversion 不知道装载转储数据时（如果真的需要装载的话）的版本库是什么样的情况。为了保证每次运行 **svnadmin dump** 都能得到一个独立的结果，第一个转储的修订版本默认情况下会完整的保存目录、文件以及属性等数据。

不过，这些都是可以改变的。如果转储时设置了--incremental 选项，**svnadmin** 会比较第一个转储的修订版本和版本库中前一个修订版本，就像对待其它转储的修订版本一样。转储时也是一样，转储文件中将仅包含第一个转储的修订版本的增量信息。这样的好处是，可以创建几个连续的小体积的转储文件代替一个大文件，比如：

```
$ svnadmin dump myrepos -r 0:1000 > dumpfile1

$ svnadmin dump myrepos -r 1001:2000 --incremental > dumpfile2

$ svnadmin dump myrepos -r 2001:3000 --incremental > dumpfile3
```

这些转储文件可以使用下列命令装载到一个新的版本库中：

```
$ svnadmin load newrepos < dumpfile1

$ svnadmin load newrepos < dumpfile2

$ svnadmin load newrepos < dumpfile3
```

另一个有关的技巧是，可以使用--incremental 选项在一个转储文件中增加新的转储修订版本。举个例子，可以使用 post-commit 钩子在每次新的修订版本提交后将其转储到文件中。或者，可以编写一个脚本，在每天夜里将所有新增的修订版本转储到文件中。这样，**svnadmin dump** 命令就变成了很好的版本库备份工具，以防万一出现系统崩溃或其它灾难性事件。

转储还可以用来将几个独立的版本库合并为一个版本库。使用 **svnadmin load** 的--parent-dir 选项,可以在装载的时候指定根目录。也就是说，如果有三个不同版本库的转储文件，比如 *calc-dumpfile*，*cal-dumpfile*，和 *ss-dumpfile*，可以在一个新的版本库中保存所有三个转储文件中的数据：

```
$ svnadmin create /path/to/projects

$
```

然后在版本库中创建三个目录分别保存来自三个不同版本库的数据：

```
$ svn mkdir -m "Initial project roots" \

     file:///path/to/projects/calc \

     file:///path/to/projects/calendar \

     file:///path/to/projects/spreadsheet
```

```
Committed revision 1.

$
```

最后，将转储文件分别装载到各自的目录中：

```
$ svnadmin load /path/to/projects --parent-dir calc <
calc-dumpfile

…

$ svnadmin load /path/to/projects --parent-dir calendar <
cal-dumpfile

…

$ svnadmin load /path/to/projects --parent-dir spreadsheet <
ss-dumpfile

…

$
```

我们再介绍一下 Subversion 版本库转储数据的最后一种用途——在不同的存储机制或版本控制系统之间转换。因为转储数据的格式的大部分是可以阅读的，所以使用这种格式描述变更集（每个变更集对应一个新的修订版本）会相对容易一些。事实上，**cvs2svn** 工具（参见 “迁移 CVS 版本库到 Subversion”一节）正是将 CVS 版本库的内容转换为转储数据格式，如此才能将 CVS 版本库的数据导入 Subversion 版本库之中。

过滤版本库历史

因为 Subversion 使用底层的二进制区别和压缩算法（也可以选择完全非透明数据库系统）储存各类数据，手工调整是不明智的，即使这样做并不困难，我们也不鼓励这样做。然而，一旦你的数据存进了版本库，Subversion 没有提供删除数据的简单办法。[32]但是不可避免的，总会有些时候你需要处理版本库的历史数据。你也许想把一个不应该出现的文件从版本库中彻底清除（无论任何原因不应该在那个位置出现）。或者，你曾经用一个版本库管理多个工程，现在又想把它们分开。要完成这样的工作，管理员们需要更易于管理和扩展的方法表示版本库中的数据，Subversion 版本库转储文件格式就是一个很好的选择。

就像我们在“版本库数据的移植”一节中说的，Subversion 版本库转储文件记录了所有版本数据的变更信息，而且以易于阅读的格式保存。可以使用 **svnadmin dump** 命令生成转储文件，然后用 **svnadmin load** 命令生成一个新的版本库。（参见 “版本库数据的移植”一节）。转储文件易于阅读意味着你可以查看和修改它。当然，问题是如果你有一个运行了三年的版本库，那么生成的转储文件会很庞大，阅读和手工修改起来都会花费很多时间。

这正是 **svndumpfilter** 发挥作用的地方，这个程序可以对版本库转储流进行特定路径的过滤。这是一个独特而很有意义的用法，可以帮助你快速方便的修改转储的数据。使用时，只需提供一个你想要保留的（或者不想保留的）路径列表，然后把你的版本库转储文件送进这个过滤器。

最后你就可以得到一个仅包含你想保留路径（明确的或含蓄的）的转储数据流。

现在我来演示如何使用这个命令。我们会在其它章节（参见 “规划你的版本库结构”一节）讨论关于如何选择设定版本库布局的问题，比如应该使用一个版本库管理多个项目还是使用一个版本库管理一个项目，或者如何在版本库中安排数据等等。不过，有些时候，即使在项目已经展开以后，你还是希望对版本库的布局做一些调整。最常见的情况是，把原来存放在同一个版本库中的几个项目分开，各自成家。

假设有一个包含三个项目的版本库： calc，calendar，和 spreadsheet。它们在版本库中的布局如下：

```
/
  calc/
    trunk/
    branches/
    tags/
  calendar/
    trunk/
    branches/
    tags/
  spreadsheet/
    trunk/
    branches/
```

```
    tags/
```

现在要把这三个项目转移到三个独立的版本库中。首先，转储整个版本库：

```
$ svnadmin dump /path/to/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

然后，将转储文件三次送入过滤器，每次仅保留一个顶级目录，就可以得到三个转储文件：

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile >
ss-dumpfile
…
$
```

现在你必须要作出一个决定了。这三个转储文件中，每个都可以用来创建一个可用的版本库，不过它们保留了原版本库的精确路径结构。也就

是说，虽然项目 calc 现在独占了一个版本库，但版本库中还保留着名为 *calc* 的顶级目录。如果希望 *trunk*、*tags* 和 *branches* 这三个目录直接位于版本库的根路径下，你可能需要编辑转储文件，调整 Node-path 和 Copyfrom-path 头参数，将路径 *calc/*删除。同时，你还要删除转储数据中创建 *calc* 目录的部分。一般来说，就是如下的一些内容：

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0

…
```

如果你打算通过手工编辑转储文件来移除一个顶级目录，注意不要让你的编辑器将换行符转换为本地格式（比如将\r\n 转换为\n）。否则文件的内容就与所需的格式不相符，这个转储文件也就失效了。

剩下的工作就是创建三个新的版本库，然后将三个转储文件分别导入：

```
$ svnadmin create calc; svnadmin load calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar; svnadmin load calendar <
cal-dumpfile
<<< Started new transaction, based on original revision 1
```

```
     * adding path : Makefile ... done.

     * adding path : cal.c ... done.

…
$ svnadmin create spreadsheet; svnadmin load spreadsheet < ss-dumpfile

<<< Started new transaction, based on original revision 1

     * adding path : Makefile ... done.

     * adding path : ss.c ... done.

…
$
```

svndumpfilter 的两个子命令都可以通过选项设定如何处理“空”修订版本。如果某个指定的修订版本仅包含路径的更改，过滤器就会将它删除，因为当前为空的修订版本通常是无用的甚至是让人讨厌的。为了让用户有选择的处理这些修订版本，**svndumpfilter** 提供了以下命令行选项：

--drop-empty-revs

不生成任何空修订版本，忽略它们。

--renumber-revs

如果空修订版本被剔除（通过使用--drop-empty-revs 选项），依次修改其它修订版本的编号，确保编号序列是连续的。

--preserve-revprops

如果空修订版本被保留，保持这些空修订版本的属性（日志信息，作者，日期，自定义属性，等等）。如果不设定这个选项，空修订版本将仅保留初始时间戳，以及一个自动生成的日志信息，表明此修订版本由 **svndumpfilter** 处理过。

尽管 **svndumpfilter** 十分有用，能节省大量的时间，但它却是把不折不扣的双刃剑。首先，这个工具对路径语义极为敏感。仔细检查转储文件中的路径是不是以斜线开头。也许 Node-path 和 Copyfrom-path 这两个头参数对你有些帮助。

```
…
Node-path: spreadsheet/Makefile
…
```

如果这些路径以斜线开头，那么你传递给 **svndumpfilter include** 和 **svndumpfilter exclude** 的路径也必须以斜线开头（反之亦然）。如果因为某些原因转储文件中的路径没有统一使用或不使用斜线开头，[33]也许需要修正这些路径，统一使用斜线开头或不使用斜线开头。

此外，复制操作生成的路径也会带来麻烦。Subversion 支持在版本库中进行复制操作，也就是复制一个存在的路径，生成一个新的路径。问题是，**svndumpfilter** 保留的某个文件或目录可能是由某个 **svndumpfilter** 排除的文件或目录复制而来的。也就是说，为了确保转储数据的完整性，**svndumpfilter** 需要切断这些复制自被排除路径的文件与源文件的关系，还要将这些文件的内容以新建的方式添加到转

储数据中。但是由于 Subversion 版本库转储文件格式中仅包含了修订版本的更改信息，因此源文件的内容基本上无法获得。如果你不能确定版本库中是否存在类似的情况，最好重新考虑一下到底保留/排除哪些路径。

最后，**svndumpfilter** 就是字面上的意思，如果你尝试将目录 *trunk/my-project* 中的内容迁移到其自己版本库，你可以使用 **svndumpfilter include** 命令保持 *trunk/my-project* 目录下的所有修改。但是结果转储文件对于将要被加载入的版本库没有任何假定，特别的，目录 *trunk/my-project* 可能从创建这个目录的修订版本开始，而它不会包含以自己创建 *trunk* 目录的指示（因为 *trunk* 没有匹配 include 过滤）。在尝试将转储流存放到版本库之前，你需要确定任何转储流将要存在的目录必须存在于目标版本库。

版本库复制

有许多场景下会存在一个 Subversion 版本库的版本历史与另一个完全相同。或许最明显的就是在主版本库因为硬件故障或网络已出或其他原因而不可用时，维护一个简单的备份版本库。其他的场景包括，部署一个镜像版本库来分流压力，作为软升级机制等等。

Subversion 1.4 提供了管理这种场景的工具—**svnsync**。**svnsync** 实质上就是通知版本库“重放”修订版本，一次一个，然后将修订版本信息模拟提交到另一个版本库。**svnsync** 运行不需要能够本地访问版本

库—它的参数是版本库 URL，所有的工作是通过 Subversion 版本库访问层（RA）接口实现的，所有要做的就是读源版本库，然后读写访问目标版本库。

当对远程源版本库使用 **svnsync** 时，Subversion 版本库的服务器必须是 Subversion1.4 或更高的版本。

假定你已经有了一个希望镜像的源版本库，下一步就是你要有一个作为镜像的目标版本库。目标版本库可以使用任意文件系统数据存储后端（见“选择数据存储格式”一节），但是其中一定不能有历史版本。**svnsync** 的通讯议对于源和目标版本库版本历史的不一致非常敏感，因此，虽然 **svnsync** 无法要求目标版本库是只读的，[34]最好的办法就是只允许镜像进程修改目标版本库内容。

*不要*做出会对镜像版本库产生版本库历史偏移的修改，所有提交和版本库的属性修改必须是由 **svnsync** 执行的。

对于目标版本库的另一种需求是 **svnsync** 可以修改特定版本化属性。**svnsync** 在目标版本库的修订版本 0 的特别属性上记录了簿记信息，因为 **svnsync** 在版本库的钩子系统的框架下工作的，版本库缺省的状态（关闭了版本库属性修改；见 pre-revprop-change）是不够的。你会需要明确的实现 pre-revprop-change 钩子，而且你的脚本必须允许 **svnsync** 设置它的特别属性，有了这些准备工作，你就可以开始镜像版本库修订版本了。

实现授权措施允许复制进程的操作，同时防止其他用户修改镜像版本库内容是一个好主意。

让我们在一个典型的镜像场景中浏览一下 **svnsync** 的使用，我们急着讨论实践推荐，但是如果你们不需要或者感到不适合你们的环境，你可以不必去关注。

作为开发者喜欢的版本控制系统的一个服务，我们会 Subversion 的源代码版本库镜像到 Internet，存放在不同的主机上，而不仅仅只有最初的 Subversion 版本库。远程主机的全局设置允许匿名用户读取版本库的信息，但是需要认证的用户才能修改版本库。（请原谅我们在此刻这里曲解 Subversion 服务器配置的细节—这些内容在第 6 章 *服务配置*。）因为没有更多的理由来建立更有趣的例子，我们会在第三个机器上创建复制进程，我们正在使用的一个例子。

首先，我们会创建一个作为镜像的版本库，下面两步需要我们能够通过 shell 访问镜像版本库的机器。一旦版本库配置完成，我们不必再直接碰它了。

```
$ ssh admin@svn.example.com \
    "svnadmin create /path/to/repositories/svn-mirror"
admin@svn.example.com's password: ********
$
```

此刻，我们有了我们的版本库，因为我们服务器的配置，这个版本库现在“存在于”Internet。现在，因为除了复制进程我们不希望任何其他修改，我们需要将这个进程同其他可能的提交者区分开来。为此，我们的进程使用专用的用户，只有特定用户 syncuser 的提交和属性修改可以被执行。

我们会使用版本库的钩子系统来允许复制进程完成我们的任务，我们通过实现两个版本库事件钩子 pre-revprop-change 和 start-commit 来强制这个过程。我们的 *pre-revprop-change* 钩子脚本可以在例 5.2 “镜像版本库的 pre-revprop-change 钩子”找到，只是验证尝试修改属性的用户是 syncuser，如果是，则允许修改；否则，拒绝修改。

例 5.2. 镜像版本库的 pre-revprop-change 钩子

```
#!/bin/sh

USER="$3"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may change revision
properties" >&2
exit 1
```

这里覆盖了修订版本属性修改，我们现在需要来确认只有用户 syncuser 允许提交新版本到版本库，我们使用了一个像例 5.3 “镜像版本库的 start-commit 钩子”的 *start-commit* 钩子。

例 5.3. 镜像版本库的 start-commit 钩子

```
#!/bin/sh

USER="$2"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may commit new revisions" >&2
exit 1
```

在安装了我们的钩子脚本和确定它们可以被 Subversion 服务器执行后，我们完成了镜像版本库的配置，现在我们开始实际的镜像。

对于 **svnsync**，我们首先需要在目标版本库上注册源版本库，我们通过 **svnsync initialize** 实现这一步。注意，**svnsync** 子命令提供了许多类似 **svn** 认证相关的选项，包括：--username、--password、--non-interactive、--config-dir 和--no-auth-cache。

```
$ svnsync help init

initialize (init): usage: svnsync initialize DEST URL
SOURCE URL
```

```
Initialize a destination repository for synchronization from
another repository.

The destination URL must point to the root of a repository with
no committed revisions.  The destination repository must allow
revision property changes.

You should not commit to, or make revision property changes in,
the destination repository by any method other than 'svnsync'.
In other words, the destination repository should be a read-only
mirror of the source repository.

Valid options:
  --non-interactive       : do no interactive prompting
  --no-auth-cache         : do not cache authentication tokens
  --username arg          : specify a username ARG
  --password arg          : specify a password ARG
  --config-dir arg        : read user configuration files from
directory ARG

$ svnsync initialize http://svn.example.com/svn-mirror \
                     http://svn.collab.net/repos/svn \
                     --username syncuser --password syncpass
Copied properties for revision 0.
```

```
$
```

我们的目标版本库现在记住了它是 Subversion 公共源代码版本库的镜像，注意我们在 **svnsync** 提供了一个用户名和密码—这是我们的镜像版本库 pre-revprop-change 钩子的要求。

提供给 **svnsync** 的 URL 必须是指向目标和源版本库的根目录，这个工具不支持对版本库子树的镜像处理。

svnsync 的最初版本（在 Subversion 1.4）有一些缺陷—用来认证的 --username 和--password 命令行参数同时作用于源和目标版本库。显然，我们无法保证同步的用户认证信息是相同的，如果不一样，用户使用非交互模式（--non-interactive 选项）来运行 **svnsync** 时会遇到这个问题。

现在有趣的部分开始了，通过一个单独的子命令，我们可以告诉 **svnsync** 将所有未镜像的修订版本从源版本库拷贝到目标版本库。[35]**svnsync synchronize** 子命令会查看目标版本库特定修订版本的属性，并且检测同步的版本库是哪一个，以及最新镜像的修订版本是 0。然后它会查询源版本库，检测其最新的修订版本。最后，它会询问源版本库服务器来开始重演从修订版本 0 到最新修订版本。**svnsync** 从源版本库服务器得到返回的结果，然后将其作为新的提交转发到目标版本库服务器。

```
$ svnsync help synchronize
```

```
synchronize (sync): usage: svnsync synchronize DEST URL


Transfer all pending revisions from source to destination.

…
$ svnsync synchronize http://svn.example.com/svn-mirror \
                    --username syncuser --password syncpass
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 23406.
Copied properties for revision 23406.
Committed revision 23407.
Copied properties for revision 23407.
Committed revision 23408.
Copied properties for revision 23408.
```

镜像修订版本有一点特别有趣，首先是到目标版本库的修订版本提交，然后跟着属性修改。这是因为最初的提交是通过用户 syncuser 执行的，而时间戳是提交的时间，而且 Subversion 底层的版本库访问接口不允许在提交时任意修改修订版本属性，所以 **svnsync** 会立即使用属性修

改，将源版本库发现的所有修订版本属性拷贝到目标版本库，这其中就包括了修改作者和时间戳使之与源版本库一致的效果。

值得注意的是 **svnsync** 会小心簿记所有的操作，可以安全的中断并重新开始，而不必破坏镜像数据的完整性。如果在 **svnsync synchronize** 时出现网络故障，只需要重新运行 **svnsync synchronize**，她会从中断处开始。实际上，随着新的修订版本在源版本库出现，这样就可以保证你的镜像不会过时。

然而，这个进程还有一点不雅的地方，因为 Subversion 属性修改可以发生在整个生命周期的任何时候，不会留下任何审计痕迹来说明所作的修改，扶植进程需要对此额外关注。如果你已经镜像了某个版本库的 15 个修订版本，而某个人修改了修订版本 12 的属性，你需要告诉它手工使用（或一些额外的工具）**svnsync copy-revprops** 子命令，只是简单的重新复制某个特定修订版本的属性。

```
$ svnsync help copy-revprops

copy-revprops: usage: svnsync copy-revprops DEST URL REV



Copy all revision properties for revision REV from source to

destination.

…

$ svnsync copy-revprops http://svn.example.com/svn-mirror 12 \

                    --username syncuser --password syncpass

Copied properties for revision 12.
```

```
$
```

版本库复制只是一个壳，你一定会希望利用这个进程的自动化。例如，如果我们的例子是一个“拖和推”设置，你或许希望在 post-commit 和 post-revprop-change 钩子实现中从你的主版本库将修改推倒一个或多个镜像，这样就可以近乎实时的保持镜像的时效性。

而且，这样做并不平凡，在人证用户只有部分读权限时 **svnsync** 也会优雅的镜像，它只会拷贝允许查看的版本库内容，显然这种镜像不适合备份方案。

只要用户与版本库和镜像的交互继续，是可以有一个工作拷贝直接与这两个版本库交互。但是你需要跳出几个圈子才能做到这样。第一，你需要保证主和镜像版本库有相同的 UUID（通常缺省不是相同），你可以加载一个包含住版本库的 UUID 转储文件来设置镜像版本库的 UUID。

```
$ cat - <<EOF | svnadmin load --force-uuid dest
SVN-fs-dump-format-version: 2


UUID: 65390229-12b7-0310-b90b-f21a5aa7ec8e
EOF
$
```

现在两个版本库有了相同的 UUID，你可以使用 **svn switch --relocate** 指向任何你希望操作的版本库，详细方法见 svn switch。这里也可能有危险，尽管如果主和镜像版本库没有同步的关闭，一个工

作拷贝对于主版本库没有过时，而重定位的镜像却是过时的，显然期望存在的修订版本缺失会造成困惑。如果发生这个情况，你可以将工作拷贝重新定位到主版本库，然后等待镜像版本库变成最新，或者将工作拷贝恢复到你知道的版本库修订版本，再尝试重新定位。

最后我们需要意识到，**svnsync** 只支持修订版本为基础的复制，它没有包括诸如钩子实现，版本库或服务器配置数据，未提交事务或关于用户锁定版本库路径的信息，只有 Subversion 版本库转储文件格式在复制时包含这些信息。

版本库备份

尽管现代计算机的诞生带来了许多便利，但有一件事听起来是完全正确的—有时候，事情变的糟糕，很糟糕，动力损耗、网络中断、坏掉的内存和损坏的硬盘都是对魔鬼的一种体验，即使对于最尽职的管理员，命运也早已注定。所以我们来到了这个最重要的主题—怎样备份你的版本库数据。

Subversion 版本库管理有两种备份方法—完全和增量。一个完全的版本库备份包含了在重大灾难后重建版本库所需的所有信息，通常，这意味着对版本库目录（包括 Berkeley DB 或 FSFS 环境）的完全复制，增量备份的内容要少一些，只包含在上次备份后改变的部分。

随着完全备份的使用，这种幼稚的方法或许看起来有点不够健全，但是除非你临时关闭所有访问版本库的进程，否则这种递归的拷贝目录会有

产生错误拷贝的风险。Berkeley DB 的情况下，其文档中记述了按照什么顺序拷贝可以保证正确的备份拷贝，FSFS 也有类似的顺序。但是你不必自己实现这种算法，因为 Subversion 的开发团队已经这样做了。**svnadmin hotcopy** 关注了在热拷贝版本库时的所有细节，它的调用就像 Unix 的 **cp** 或 Windows 的 **copy** 一样琐碎：

```
$ svnadmin hotcopy /path/to/repos /path/to/repos-backup
```

作为结果的备份是一个完全功能的版本库，当发生严重错误时可以作为你的活动版本库的替换。

当进行 Berkeley DB 版本库的备份时，你可以指导 **svnadmin hotcopy** 清理源版本库中无用的 Berkeley DB 日志文件（见“删除不使用的 Berkeley DB 日志文件”一节），只需要简单的在命令行里提供 `--clean-logs`。

```
$ svnadmin hotcopy --clean-logs /path/to/bdb-repos
/path/to/bdb-repos-backup
```

还有一些附加的加工命令，Subversion 源程序中的 *tools/backup/* 目录包含了 **hot-backup.py** 脚本，这个脚本在 **hot-backup.py** 之上增加了备份管理功能，你可以保存每个版本库最近的配置号码。为了防止与以前的备份冲突，它会自动管理备份版本库目录名字，“循环”利用备份名，删除掉旧的，保存新的。即使你也有一个增量的备份，你还是会希望有规律的运行这个程序。例如，你会在一个调度程序（例如 Unix

系统的 **cron**）中调用 **hot-backup.py** 会导致它在半夜执行（或者是任何你认为安全的时间间隔）。

一些管理员使用不同的备份机制，通过生成和保存版本库转储数据。我们在“版本库数据的移植”一节中描述如何使用 **svnadmin dump --incremental** 来对一个修订版本或一个修订版本范围执行增量备份。当然，通过取消--incremental 选项可以得到完整的备份。在备份信息中方法的值非常灵活—不会与特定平台，版本化的文件系统类型或 Subversion 和 Berkeley DB 的版本绑定。但是灵活带来了代价，数据恢复会占用更长的时间—比每个新版本提交更长。此外，在非完全的量转储生成时，对已经备份修订版本的修订版本属性的修改不会被采纳，因为这些原因，我们不建议你单独依赖转储为基础的备份方法。

如你所见，几种备份方式都有各自的优点，最简单的方式是完全热备份，将会每次建立版本库的完美复制品，这意味着如果当你的活动版本库发生了什么事情，你可以用备份恢复。但不幸的是，如果你维护多个备份，每个完全的备份会吞噬掉和你的活动版本库同样的空间。与之相对照的是增量备份，能够快速生成小的备份，但是恢复过程将会很痛苦，通常要包括多个增量拷贝的应用。其他方法都有自己的特点，管理员需要在创建拷贝和恢复的代价之间寻求平衡。

svnsync（见“版本库复制”一节）实际上提供了一种更易实施的妥协方法，如果你有规律的同步镜像版本库，则在必要时，镜像版本库就成了主版本库发生问题时的一个合适替代者。这个方法最大的缺点是只有

版本化的数据得到了同步—版本库的配置信息，用户指定的路径锁定和其它以物理形式存在于版本库路径而不存在于版本库虚拟文件系统的项目不会被 svnsync 处理。

在每一种备份情境下，版本库管理员需要意识到对未版本化的修订版本属性的修改对备份的影响，因为这些修改本身不会产生新的修订版本，所以不会触发 post-commit 的钩子程序，也不会触发 pre-revprop-change 和 post-revprop-change 的钩子。[36]而且因为你可以改变修订版本的属性，而不需要遵照时间顺序—你可在任何时刻修改任何修订版本的属性—因此最新版本的增量备份不会捕捉到以前特定修订版本的属性修改。

通常说来，在每次提交时，只有妄想狂才会备份整个版本库，然而，假设一个给定的版本库拥有一些恰当粒度的冗余机制（如每次提交的邮件）。版本库管理员也许会希望将版本库的热备份引入到系统级的每夜备份，对大多数版本库，归档的提交邮件为保存资源提供了足够的冗余措施，至少对于最近的提交。但是它是你的数据—你喜欢怎样保护都可以。

通常情况下，最好的版本库备份方式是混合的，你可以平衡完全和增量备份，另外配合提交邮件的归档。Subversion 开发者，举个例子，使用 **`hot-backup.py`** 对 Subversion 版本库进行完全备份并使用 **`rsync`** 同步这些备份；同时保存所有的提交日至和修改通知邮件；并且使用许多志愿者维护的 **`svnsync`** 镜像版本库。你们的解决方案可能非常类似，

但是要实现满足需要和便利性的平衡。无论你做了什么，你需要一次次的验证你的备份—就像要检查备用轮胎是否有个窟窿？当然，所有做的事情都无法回避我们的硬件来自钢铁的命运，[37]它将帮助你从艰难的时光恢复过来。

总结

现在，你应该已经对如何创建、配置以及维护 Subversion 版本库有了个基本的认识。我们向您介绍了几个可以帮助您工作的工具。通过这一章，我们说明了一些常见的管理误区，并提出了避免陷入误区的建议。

剩下的，就是由你决定在你的版本库中存放一些什么有趣的资料，并最终通过网络获得这些资料。下一章是关于网络的内容。

---

[24] 这可能听起来很崇高， 但我们所指的只是那些对管理别人工作拷贝数据之外的神秘领域感兴趣的人。

[25] 无论是在忽略情况下建立或很少考虑过如何产生正确的软件开发矩阵，都不应该愚蠢的担心全局的修订版本号码，这不应该成为安排项目和版本库的理由。

[26] *trunk*、*tags* 和 *branches* 可以使用“TTB 目录”来表示。

[27] 通常读作“fuzz-fuzz”，如果 Jack Repenning 说起这个问题。（本书，假定读者认为是“eff-ess-eff-ess”。）

[28] Berkeley DB 需要底层的文件系统实现严格的 POSIX 锁定语法，更重要的是，将文件直接映射到内存的能力。

[29] Oracle 在 2006 情人节购买了 Sleepycat 和它的旗舰软件 Berkeley DB。

[30] 或者是，“sync” ？

[31] 比如：硬盘 + 大号电磁铁 = 毁灭。

[32] 那就是你是用版本控制的原因，对吗？

[33] 尽管 **svnadmin dump** 对是否以斜线作为路径的开头有统一的规定——这个规定就是不以斜线作为路径的开头——其它生成转储文件的程序不一定会遵守这个规定。

[34] 实际上，它不是真的完全只读，或者 **svnsync** 本身有时间将版本库历史拷入。

[35] 要预先警告一下，尽管对于普通读者只需要几秒钟就可以理解下面的输出，而对于整个镜像过程花费的时间可能会非常长。

[36] **svnadmin setlog** 可以被绕过钩子程序被调用。

[37] 你知道的—只是对各种变化莫测的问题的统称。

服务配置

目录

一个 Subversion 的版本库可以和客户端同时运行在同一个机器上，使用 file:///访问，但是一个典型的 Subversion 设置应该包括一个单独的服务器，可以被办公室的所有客户端访问—或者有可能是整个世界。

本小节描述了怎样将一个 Subversion 的版本库暴露给远程客户端，我们会覆盖 Subversion 已存在的服务器机制，讨论各种方式的配置和使用。经过阅读本小节，你可以决定你需要哪种网络设置，并且明白怎样在你的主机上进行配置。

概述

Subversion 的设计包括一个抽象的网络层，这意味着版本库可以通过各种服务器进程访问，而且客户端“版本库访问”的 API 允许程序员写出相关协议的插件，理论上讲，Subversion 可以使用无限数量的网络协议实现，目前实践中只有两种服务器。

Apache 是最流行的 web 服务器，通过使用 **mod_dav_svn** 模块，Apache 可以访问版本库，并且可以使客户端使用 HTTP 的扩展协议 WebDAV/DeltaV 进行访问，因为 Apache 是一个非常易于扩展的 web 服务器，它提供了许多“易于获取的”特性，例如加密的 SSL 通讯，日志和与第三方工具的集成，以及内置的版本库 web 浏览功能。

在另一个角落是 **svnserve**：一个更小，轻型的服务器程序，同客户端使用自定义的协议。因为协议是为 Subversion 专门设计的，并且是有状态的（不像 HTTP），它提供了更快的网络操作—但也有一些代价。

它只理解 CRAM-MD5 的认证，然而它非常易于配置，是开始使用 Subversion 的小团队的最佳选择。

第三个选择是使用 SSH 连接包裹的 **svnserve**，尽管这个场景依然使用 **svnserve**，它与传统的 **svnserve** 部署非常不同，SSH 在多所有的通讯中使用加密方式，SSH 也使用排他的认证，所以在服务器主机（**svnserve** 与之不同，它包含了自己的私有用户帐号）上必须要有真实的系统帐户。最后，因为这些配置需要每个用户发起一个私有的临时 **svnserve** 进程，这与允许一组本地用户通过 file://协议访问等同（从访问许可的视点）。因此路径为基础的访问控制变得没有意义，因为每个用户都可以直接访问版本库。

下面是三种典型服务器部署的总结。

表 6.1.

特性	Apache + mod_dav_svn	svnserve	svnserve over SSH
认证选项	HTTP(S) basic auth、X.509 certificates、LDAP、NTLM 或任何 Apache httpd 已经具备的方式	CRAM-MD5	SSH
用户帐号选项	私有的'users'文件	私有的'users'文件	系统帐号
授权选项	可以授予整个版本库的读/写权限，也可以指定目录的。	可以授予整个版本库的读/写权限，也可以指定目录的。	只能对版本库整体赋予读/写权限
加密	通过可选的 SSL	无	SSH 通道的
Logging	对每个 HTTP 请求记录完全的 Apache 日志，通过选项“高级”记录普通的客户端操作。	no logging	no logging
交互性	可以部分的被其他 WebDAV 客户端	只同 svn 客户端通	只同 svn 客户

特性	Apache + mod_dav_svn	svnserve	svnserve over SSH
	使用	讯	端通讯
Web 浏览能力	有限的内置支持,或者通过第三方工具，如 ViewVC	只有通过第三方工具，如 ViewVC	只有通过第三方工具，如 ViewVC
速度	有些慢	快一点	快一点
初始设置	有些复杂	极为简单	相当简单

选择一个服务器配置

那你应该用什么服务器？什么最好？

显然，对这个问题没有正确的答案，每个团队都有不同的需要，不同的服务器都有各自的代价。Subversion 项目没有更加认可哪种服务，或认为哪个服务更加“正式”一点。

下面是你选择或者不选择某一个部署方式的原因。

svnserve 服务器

为什么你会希望使用它：

- 设置快速简单。
- 网络协议是有状态的，比 WebDAV 快很多。
- 不需要在服务器创建系统帐号。
- 不会在网络传输密码。

为什么你会希望避免它：

- 网络协议没有加密。
- 只有一个认证方法选择。
- 在这个服务器上明文保存密码。
- 没有任何类型的日志，甚至是错误。

`svnserve` 使用 SSH 通道

为什么你会希望使用它：

- 网络协议是有状态的，比 WebDAV 快很多。
- 你可以利用现有的 ssh 帐号和用户基础。
- 所有网络传输是加密的。

为什么你会希望避免它：

- 只有一个认证方法选择。
- 没有任何类型的日志，甚至是错误。
- 需要用户在同一个系统组，使用共享 ssh 密钥。
- 如果使用不正确，会导致文件许可问题。

Apache 的 HTTP 服务器

为什么你会希望使用它：

- 允许 Subversion 使用大量已经集成到 Apache 的用户认证系统。

- 不需要在服务器创建系统帐号。
- 完全的 Apache 日志。
- 网络传输可以通过 SSL 加密。
- HTTP(S) 通常可以穿越公司防火墙。
- 通过 web 浏览器访问内置的版本库浏览。
- 版本库可以作为网络驱动器加载，实现透明的版本控制，见“自动版本化”一节。

为什么你会希望避免它：

- 比 svnserve 慢很多，因为 HTTP 是无状态的协议，需要更多的传递。
- 初始设置可能复杂

推荐

通常，本书的作者推荐希望尝试开始使用 Subversion 的小团队使用 **`svnserve`**；这是设置最简单，维护最少的方法，而当你的需求改变时，你可以转换到复杂的部署方式。

下面是一些常见的建议和小技巧，基于多年对用户的支持：

- 如果你尝试为你的团队建立最简单的服务器，安装 **`svnserve`** 是最简单的，最快速的方法。注意，无论如何，如果你的整个部署都是在局域网或者 VPN 中，版本库数据可以在网络上没有限制的

传递。如果版本库部署在 internet，你会希望确定版本库的内容不是敏感的（例如只包含开源代码。）

- 如果你希望与现有的认证系统（LDAP、Active Directory、NTLM、X.509 等）集成，你你只能选择 Apache 服务器，同样的，如果你绝对需要服务器端的日志（服务日志或客户端活动），也需要 Apache 服务器。
- 如果你已经决定使用 Apache 或 **`svnserve`**，应该单独创建一个运行服务器进程的 svn 用户，也需要确定版本库目录属于 svn 用户。从安全的角度，这样很好的利用了操作系统的文件系统许可保护了版本库数据，只有 Subversion 服务进程可以修改其内容。
- 如果你有一个严重依赖于 SSH 帐号的基础，而且你的用户已经在服务器上有了帐号，那建立一个通过 ssh 的 svnserve 方案就非常有意义，否则，我们不会建议这种方案。通常还是通过 **`svnserve`** 或 Apache 管理的用户访问版本库比较安全，而不是使用完全的系统帐户。如果你很希望加密的通讯，那可能还是需要选择这个方案，但我们更加推荐 SSL 的 Apache 方案。
- 不要被简单的让所有用户使用 `file://` 的 URL 访问版本库的方案诱惑，即使，版本库版本库已经对网络共享的所有用户可见，这也不是一个好方案。这样删除了用户和版本库之间的所有保护层：用户可能会偶然（或有意的）毁坏版本库数据库，这样也很难在检查或升级时将版本库脱机，而且这样会造成曾混乱的文件许可问题（见“支持多种版本库访问方法”一节。），注意那就是我

们警告使用 svn+ssh://的原因—从安全角度讲，这样与作为本地用户访问 file://是一样的，如果管理员不小心，会造成同样的问题。

svnserve，一个自定义的服务器

svnserve 是一个轻型的服务器，可以同客户端通过在 TCP/IP 基础上的自定义有状态协议通讯，客户端通过使用开头为 svn://或者 svn+ssh://**svnserve** 的 URL 来访问一个 **svnserve** 服务器。这一小节将会解释运行 **svnserve** 的不同方式，客户端怎样实现服务器的认证，怎样配置版本库恰当的访问控制。

调用服务器

有许多不同方法运行 **svnserve**：

- 作为一个独立守护进程启动 **svnserve**，监听请求。
- 当特定端口收到一个请求，就会使 UNIX 的 **inetd** 守护进程临时调用 **svnserve** 处理。
- 使用 SSH 在加密通道发起临时 **svnserve** 服务。
- 以 Windows service 服务方式运行 **svnserve**。

svnserve 作为守护进程

使用 **svnserve** 最简单的方式是作为独立“守护”进程运行，使用-d 选项：

```
$ svnserve -d

$             # svnserve is now running, listening on port 3690
```

当以守护模式运行 **svnserve** 时，你可以使用--listen-port=和--listen-host=选项来自定义“绑定”的端口和主机名。

一旦 **svnserve** 已经运行，它会将你系统中所有版本库发布到网络，一个客户端需要指定版本库在 URL 中的*绝对*路径，举个例子，如果一个版本库是位于*/usr/local/repositories/project1*，则一个客户端可以使用 svn://host.example.com/usr/local/repositories/project1 来进行访问，为了提高安全性，你可以使用 **svnserve** 的-r 选项，这样会限制只输出指定路径下的版本库，例如：

```
$ svnserve -d -r /usr/local/repositories

…
```

使用-r 可以有效地改变文件系统的根位置，客户端可以使用去掉前半部分的路径，留下的要短一些的（更加有提示性）URL：

```
$ svn checkout svn://host.example.com/project1

…
```

使用 svnserve 通过 inetd

如果你希望 **inetd** 启动进程，你需要使用-i（--inetd）选项，在这个例子里，我们显示了在命令行中运行 svnserve -i 的输出，但是请注意

这不是如何实际启动 daemon; 请继续阅读例子后的文章，学习如何配置 **inetd** 启动 **svnserve**。

```
$ svnserve -i

( success ( 1 2 ( ANONYMOUS ) ( edit-pipeline ) ) )
```

当用参数--inetd 调用时，**svnserve** 会尝试使用自定义协议通过 *stdin* 和 *stdout* 来与 Subversion 客户端通话，这是使用 **inetd** 工作的标准方式，IANA 为 Subversion 协议保留 3690 端口，所以在类 Unix 系统你可以在*/etc/services* 添加如下的几行（如果不存在的话）：

```
svn             3690/tcp   # Subversion

svn             3690/udp   # Subversion
```

如果系统是使用经典的类 Unix 的 **inetd** 守护进程，你可以在 */etc/inetd.conf* 添加这几行：

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

确定“svnowner”用户拥有访问版本库的适当权限，现在如果一个客户连接来到你的服务器的端口 3690，**inetd** 会产生一个 **svnserve** 进程来做服务。当然，你也可以添加-r 到命令行，限制暴露出的版本库。

通过通道使用 svnserve

第三种方式使用-t 选项的“管道模式”，这个模式假定一个分布式服务程序如 **RSH** 或 **SSH** 已经验证了一个用户，并且*以这个用户*调用了一个

私有 **svnserve** 进程，**svnserve** 运作如常（通过 *stdin* 和 *stdout* 通讯），并且可以设想通讯是自动转向到一种通道并传递回客户端，当 **svnserve** 被这样的通道代理调用，确定认证用户对版本数据库有完全的读写权限，这与本地用户通过 file:///URI 访问版本库同样重要。

这个选项将在“SSH 隧道”一节详细讨论。

svnserve 作为 Windows 服务

如果你的 Windows 系统是 Windows NT (2000, 2003, XP, Vista)的后代，你可以将 **svnserve** 作为 Windows 服务运行，这是比使用--daemon (-d) 选项直接运行守护进程感觉更好。使用守护进程模式，需要打开命令行窗口，输入命令，然后保持命令行窗口不关闭，而作为 Windows 服务时，在后台运行，可以在启动时自动执行，并且可以使用同其他 Windows 服务一致的管理界面启动和停止服务。

你需要使用命令行工具 **SC.EXE** 定义新的服务，就像 **inetd** 的配置行，你必须在 Windows 启动时指明 **svnserve** 的调用：

```
C:\> sc create svn
       binpath= "C:\svn\bin\svnserve.exe --service -r
C:\repos"
       displayname= "Subversion Server"
       depend= Tcpip
       start= auto
```

这样定义了一个新的 Windows 服务，叫做“svn”，会在启动时（在这个例子里，根目录是 *C:\repos*。）执行特定的 **svnserve.exe**，可是前面这个例子产生了一些错误。

首先，要注意 **svnserve.exe** 必须使用--service 选项启动。**svnserve** 的其它选项必须在同一行上指定，但你不能使用冲突的选项，例如 --daemon (-d)、--tunnel 或--inetd (-i)，而选项-r 或--listen-port 都没有问题。第二，调用 **SC.EXE** 时必须注意空格：key= value 的模式中 key= 之间必须没有空格，而且在与 value 之间只能有一个空格。最后，必须注意执行的命令行中的空格，如果目录名中包含了空格（或其它需要回避的字符），为了回避这些字符，请将整个 binpath 值放在双引号中：

```
C:\> sc create svn
       binpath= "\"C:\program files\svn\bin\svnserve.exe\"
--service -r C:\repos"
       displayname= "Subversion Server"
       depend= Tcpip
       start= auto
```

也需要注意单词 binpath 会造成误解—它的值是一个*命令行*，而不是可执行的路径，所以我们为了防止有嵌入的空格而使用了引号围绕。

一旦定义了服务，就可以使用标准 GUI 工具（服务管理控制面板）进行停止、启动和查询，或者是通过命令行：

```
C:\> net stop svn
```

```
C:\> net start svn
```

也可以通过删除其定义删除服务：sc delete svn，只需要确定首先停止服务，**SC.EXE** 有许多子命令和选项，更多信息可以运行 sc /?查看。

内置的认证和授权

如果一个客户端连接到 **svnserve** 进程，如下事情会发生：

- 客户端选择特定的版本库。
- 服务器处理版本库的 *conf/svnserve.conf* 文件，并且执行里面定义的所有认证和授权政策。
- 依赖于位置和授权政策，
    - 如果没有收到认证请求，客户端可能被允许匿名访问，或者
    - 客户端收到认证请求，或者
    - 如果操作在“通道模式”，客户端会宣布自己已经在外部得到认证。

在撰写本文时，服务器还只知道怎样发送 CRAM-MD5[38]认证请求，本质上讲，就是服务器发送一些数据到客户端，客户端使用 MD5 哈希算法创建这些数据组合密码的指纹，然后返回指纹，服务器执行同样的计算并且来计算结果的一致性，*真正的密码并没有在互联网上传递。*

当然也有可能，如果客户端在外部通过通道代理认证，如 **SSH**，在那种情况下，服务器简单的检验作为那个用户的运行，然后使用它作为认证用户名，更多信息请看“SSH 隧道”一节。

像你已经猜测到的，版本库的 *svnserve.conf* 文件是控制认证和授权政策的中央机构，这文件与其它配置文件格式相同（见“运行配置区”一节）：小节名称使用方括号标记（[和]），注释以井号（#）开始，每一小节都有一些参数可以设置（variable = value），让我们浏览这个文件并且学习怎样使用它们。

创建一个用户文件和认证域

此时，*svnserve.conf* 文件的[general]部分包括所有你需要的变量，开始先定义一个保存用户名和密码的文件和一个认证域：

```
[general]
password-db = userfile
realm = example realm
```

realm 是你定义的名称，这告诉客户端连接的“认证命名空间”，Subversion 会在认证提示里显示，并且作为凭证缓存（见“客户端凭证缓存”一节。）的关键字（还有服务器的主机名和端口），password-db 参数指出了保存用户和密码列表文件，这个文件使用同样熟悉的格式，举个例子：

```
[users]
```

```
harry = foopassword

sally = barpassword
```

password-db 的值可以是用户文件的绝对或相对路径，对许多管理员来说，把文件保存在版本库 *conf/*下的 *svnserve.conf* 旁边是一个简单的方法。另一方面，可能你的多个版本库使用同一个用户文件，此时，这个文件应该在更公开的地方，版本库分享用户文件时必须配置为相同的域，因为用户列表本质上定义了一个认证域，无论这个文件在哪里，必须设置好文件的读写权限，如果你知道运行 **svnserve** 的用户，限定这个用户对这个文件有读权限是必须的。

设置访问控制

svnserve.conf 有两个或多个参数需要设置：它们确定未认证（匿名）和认证用户可以做的事情，参数 anon-access 和 auth-access 可以设置为 none、read 或者 write，设置为 none 会限制所有方式的访问，read 允许只读访问，而 write 允许对版本库完全的读/写权限，例如：

```
[general]

password-db = userfile

realm = example realm


# anonymous users can only read the repository

anon-access = read


# authenticated users can both read and write
```

```
auth-access = write
```

实例中的设置实际上是参数的缺省值，你一定不要忘了设置它们，如果你希望更保守一点，你可以完全封锁匿名访问：

```
[general]
password-db = userfile
realm = example realm

# anonymous users aren't allowed
anon-access = none

# authenticated users can both read and write
auth-access = write
```

服务进程不仅仅理解对版本库的整体访问控制，也可以细粒度的控制版本库某个文件或目录的访问，为了使用这个特性，你需要定一个包含详细规则的文件，并将变量 authz-db 指向到这个文件。

```
[general]
password-db = userfile
realm = example realm

# Specific access rules for specific locations
authz-db = authzfile
```

authzfile 得语法会在“基于路径的授权”一节讨论，注意变量 authz-db 并不比 anon-access 和 auth-access 更高级，如果定义了所有的变量，要想被允许访问必须满足所有的规则。

SSH 隧道

svnserve 的内置认证会非常容易得到，因为它避免了创建真实的系统帐号，另一方面，一些管理员已经创建好了 SSH 认证框架，在这种情况下，所有的项目用户已经拥有了系统帐号和有能力“SSH 到”服务器。

SSH 与 **svnserve** 结合很简单，客户端只需要使用 svn+ssh://的 URL 模式来连接：

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harry@host.example.com's password:  *****

foo
bar
baz
…
```

在这个例子里，Subversion 客户端会调用一个 **ssh** 进程，连接到 host.example.com，使用用户 harry 认证，然后会有一个 **svnserve** 私有进程以用户 harry 运行。**svnserve** 是以管道模式调用的（-t），它的网络协议是通过 **ssh** “封装的”，被管道代理的 **svnserve** 会知道程序是以用户 harry 运行的，如果客户执行一个提交，认证的用户名会作为版本的参数保存到新的修订本。

这里要理解的最重要的事情是 Subversion 客户端*不*是连接到运行中的 **svnserve** 守护进程，这种访问方法不需要一个运行的守护进程，也不需要在必要时唤醒一个，它依赖于 **ssh** 来发起一个 **svnserve** 进程，然后网络断开后终止进程。

当使用 svn+ssh://的 URL 访问版本库时，记住是 **ssh** 提示请求认证，而*不*是 **svn** 客户端程序。这意味着密码不会有自动缓存（见“客户端凭证缓存”一节），Subversion 客户端通常会建立多个版本库的连接，但用户通常会因为密码缓存特性而没有注意到这一点，当使用 svn+ssh://的 URL 时，用户会为 **ssh** 在每次建立连接时重复的询问密码感到讨厌，解决方案是用一个独立的 SSH 密码缓存工具，像类 Unix 系统的 **ssh-agent** 或者是 Windows 下的 **pageant**。

当在一个管道上运行时，认证通常是基于操作系统对版本库数据库文件的访问控制，这同 Harry 直接通过 file:///的 URL 直接访问版本库非常类似，如果有多个系统用户要直接访问版本库，你会希望将他们放到一个常见的组里，你应该小心的使用 umasks。（确定要阅读“支持多种

版本库访问方法”一节）但是即使是在管道模式时，文件 *svnserve.conf* 还是可以阻止用户访问，如设置 auth-access = read 或 auth-access = none。[39]

你会认为 SSH 管道的故事该结束了，但还不是，Subversion 允许你在运行配置文件 *config*（见“运行配置区”一节）创建一个自定义的管道行为方式，举个例子，假定你希望使用 RSH 而不是 SSH，在 *config* 文件的[tunnels]部分作如下定义：

```
[tunnels]

rsh = rsh
```

现在你可以通过指定与定义匹配的 URL 模式来使用新的管道定义：svn+rsh://host/path。当使用新的 URL 模式时，Subversion 客户端实际上会在后台运行 **rsh host svnserve -t** 这个命令，如果你在 URL 中包括一个用户名（例如，svn+rsh://username@host/path），客户端也会在自己的命令中包含这部分（**rsh username@host svnserve -t**），但是你可以定义比这个更加智能的新的管道模式：

```
[tunnels]

joessh = $JOESSH /opt/alternate/ssh -p 29934
```

这个例子里论证了一些事情，首先，它展现了如何让 Subversion 客户端启动一个特定的管道程序（这个在*/opt/alternate/ssh*），在这个例子里，使用 svn+joessh://的 URL 会以-p 29934 参数调用特定的 SSH 程序—对连接到非标准端口的程序非常有用。

第二点，它展示了怎样定义一个自定义的环境变量来覆盖管道程序中的名字，设置 SVN_SSH 环境变量是覆盖缺省的 SSH 管道的一种简便方法，但是如果你需要为多个服务器做出多个不同的覆盖，或许每一个都联系不同的端口或传递不同的 SSH 选项，你可以使用本例论述的机制。现在如果我们设置 JOESSH 环境变量，它的值会覆盖管道中的变量值—会执行**$JOESSH** 而不是**/opt/alternate/ssh -p 29934**。

SSH 配置技巧

不仅仅是可以控制客户端调用 **ssh** 方式，也可以控制服务器中的 **sshd** 的行为方式，在本小节，我们会展示怎样控制 **sshd** 执行 **svnserve**，包括如何让多个用户分享同一个系统帐户。

初始设置

作为开始，定位到你启动 **svnserve** 的帐号的主目录，确定这个账户已经安装了一套 SSH 公开/私有密钥对，用户可以通过公开密钥认证，因为所有如下的技巧围绕着使用 SSH*authorized_keys* 文件，密码认证在这里不会工作。

如果这个文件还不存在，创建一个 *authorized_keys* 文件（在 UNIX 下通常是*~/.ssh/authorized_keys*），这个文件的每一行描述了一个允许连接的公钥，这些行通常是下面的形式：

```
ssh-dsa AAAABtce9euch.... user@example.com
```

第一个字段描述了密钥的类型，第二个字段是未加密的密钥本身，第三个字段是注释。然而，这是一个很少人知道的事实，可以使用一个 command 来处理整行：

```
command="program" ssh-dsa AAAABtce9euch.... user@example.com
```

当 command 字段设置后，SSH 守护进程运行命名的程序而不是通常 Subversion 客户端询问的 **svnserve -t**。这为实施许多服务器端技巧开启了大门，在下面的例子里，我们简写了文件的这些行：

```
command="program" TYPE KEY COMMENT
```

控制调用的命令

因为我们可以指定服务器端执行的命令，我们很容易来选择运行一个特定的 **svnserve** 程序来并且传递给它额外的参数：

```
command="/path/to/svnserve -t -r /virtual/root" TYPE KEY
COMMENT
```

在这个例子里，*/path/to/svnserve* 也许会是一个 **svnserve** 程序的包裹脚本，会来设置 umask（见“支持多种版本库访问方法”一节）。它也展示了怎样在虚拟根目录定位一个 **svnserve**，就像我们经常在使用守护进程模式下运行 **svnserve** 一样。这样做不仅可以把访问限制在系统的一部分，也可以使用户不需要在 svn+ssh://URL 里输入绝对路径。

多个用户也可以共享同一个帐号，作为为每个用户创建系统帐户的替代，我们创建一个公开/私有密钥对，然后在 *authorized_users* 文件里放置各自的公钥，一个用户一行，使用--tunnel-user 选项：

```
  command="svnserve -t --tunnel-user=harry" TYPE1 KEY1
harry@example.com

  command="svnserve -t --tunnel-user=sally" TYPE2 KEY2
sally@example.com
```

这个例子允许 Harry 和 Sally 通过公钥认证连接同一个的账户，每个人自定义的命令将会执行。--tunnel-user 选项告诉 **svnserve -t** 命令采用命名的参数作为经过认证的用户，如果没有--tunnel-user，所有的提交会作为共享的系统帐户提交。

最后要小心：设定通过公钥共享账户进行用户访问时还会允许其它形式的 SSH 访问，即使你设置了 *authorized_keys* 的 command 值，举个例子，用户仍然可以通过 SSH 得到 shell 访问，或者是通过服务器执行 X11 或者是端口转发。为了给用户尽可能少的访问权限，你或许希望在 command 命令之后指定一些限制选项：

```
  command="svnserve -t
--tunnel-user=harry",no-port-forwarding,\

          no-agent-forwarding,no-X11-forwarding,no-pty \

          TYPE1 KEY1 harry@example.com
```

httpd，Apache 的 HTTP 服务器

Apache 的 HTTP 服务器是一个 Subversion 可以利用的“重型”网络服务器，通过一个自定义模块，**`httpd`** 可以让 Subversion 版本库通过 WebDAV/DeltaV 协议在客户端前可见，WebDAV/DeltaV 协议是 HTTP 1.1 的扩展（见 http://www.webdav.org/来查看详细信息）。这个协议利用了无处不在的 HTTP 协议是广域网的核心这一点，添加了写能力—更明确一点，版本化的写—能力。结果就是这样一个标准化的健壮的系统，作为 Apache 2.0 软件的一部分打包，被许多操作系统和第三方产品支持，网络管理员也不需要打开另一个自定义端口。[40]这样一个 Apache-Subversion 服务器具备了许多 **`svnserve`** 没有的特性，但是也有一点难于配置，灵活通常会带来复杂性。

下面的讨论包括了对 Apache 配置指示的引用，给了一些使用这些指示的例子，详细地描述不在本章的范围之内，Apache 小组维护了完美的文档，公开存放在他们的站点 http://httpd.apache.org。例如，一个一般的配置参考位

于 http://httpd.apache.org/docs-2.0/mod/directives.html。

同样，当你修改你的 Apache 设置，很有可能会出现一些错误，如果你还不熟悉 Apache 的日志子系统，你一定需要认识到这一点。在你的文件 *httpd.conf* 里会指定 Apache 生成的访问和错误日志（CustomLog 和 ErrorLog 指示）的磁盘位置。Subversion 的 mod_dav_svn 使用 Apache 的错误日志接口，你可以浏览这个文件的内容查看信息来查找难于发现的问题根源。

为什么是 Apache 2？

如果你系统管理员，很有可能是你已经运行了 Apache 服务器，并且有一些高级经验。写本文的时候，Apache 1.3 是 Apache 最流行的版本，这个世界因为许多原因而放缓升级到 2.X 系列：如人们害怕改变，特别是像 web 服务器这种重要的变化，有些人需要一些在 Apache 1.3 API 下工作的插件模块，在等待 2.X 的版本。无论什么原因，许多人会在首次发现 Subversion 的 Apache 模块只是为 Apache 2 API 写的后开始担心。

对此问题的适当反应是：不需要担心，同时运行 Apache 1.3 和 Apache 2 非常简单，只需要安装到不同的位置，用 Apache 2 作为 Subversion 的专用服务器，并且不使用 80 端口，客户端可以访问版本库时在 URL 里指定端口：

```
$ svn checkout
http://host.example.com:7382/repos/project

…
```

先决条件

为了让你的版本库使用 HTTP 网络，你基本上需要两个包里的四个部分。你需要 Apache **httpd**2.0 和包括的 **mod_dav** DAV 模块，Subversion 和与之一同分发的 **mod_dav_svn** 文件系统提供者模块，如果你有了这些组件，网络化你的版本库将非常简单，如：

- 配置好 httpd 2.0，并且使用 mod_dav 启动，

- 为 mod_dav 安装 mod_dav_svn 插件，它会使用 Subversion 的库访问版本库，并且
- 配置你的 *httpd.conf* 来输出（或者说暴露）版本库。

你可以通过从源代码编译 **httpd** 和 Subversion 来完成前两个项目，也可以通过你的系统上的已经编译好的二进制包来安装。最新的使用 Apache HTTP 的 Subversion 的编译方法和 Apache 的配置方式可以看 Subversion 源代码树根目录的 *INSTALL* 文件。

基本的 Apache 配置

一旦你安装了必须的组件，剩下的工作就是在 *httpd.conf* 里配置 Apache，使用 LoadModule 来加载 mod_dav_svn 模块，这个指示必须先与其它 Subversion 相关的其它配置出现，如果你的 Apache 使用缺省布局安装，你的 **mod_dav_svn** 模块一定在 Apache 安装目录（通常是在 */usr/local/apache2*）的 *modules* 子目录，LoadModule 指示的语法很简单，影射一个名字到它的共享库的物理位置：

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

注意，如果 **mod_dav** 是作为共享对象编译（而不是静态链接到 **httpd** 程序），你需要为它使用 LoadModule 语句，一定确定它在 **mod_dav_svn** 之前：

```
LoadModule dav_module         modules/mod_dav.so
```

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

在你的配置文件后面的位置，你需要告诉 Apache 你在什么地方保存 Subversion 版本库（也许是多个），位置指示有一个很像 XML 的符号，开始于一个开始标签，以一个结束标签结束，配合中间许多的其它配置。Location 指示的目的是告诉 Apache 在特定的 URL 以及子 URL 下需要特殊的处理，如果是为 Subversion 准备的，你希望可以通过告诉 Apache 特定 URL 是指向版本化的资源，从而把支持转交给 DAV 层，你可以告诉 Apache 将所有路径部分（URL 中服务器名称和端口之后的部分）以*/repos/*开头的 URL 交由 DAV 服务提供者处理。一个 DAV 服务提供者的版本库位于*/absolute/path/to/repository*，可以使用如下的 *httpd.conf* 语法：

```
<Location /repos>
  DAV svn
  SVNPath /absolute/path/to/repository
</Location>
```

如果你计划支持多个具备相同父目录的 Subversion 版本库，你有另外的选择，SVNParentPath 指示，来表示共同的父目录。举个例子，如果你知道会在*/usr/local/svn* 下创建多个 Subversion 版本库，并且通过类似 http://my.server.com/svn/repos1，http://my.server.com/svn/repos2 的 URL 访问，你可以用后面例子中的 *httpd.conf* 配置语法：

```
<Location /svn>
```

```
  DAV svn


  # any "/svn/foo" URL will map to a repository
/usr/local/svn/foo
  SVNParentPath /usr/local/svn
</Location>
```

使用上面的语法，Apache 会代理所有 URL 路径部分为*/svn/*的请求到 Subversion 的 DAV 提供者，Subversion 会认为 SVNParentPath 指定的目录下的所有项目是真实的 Subversion 版本库，这通常是一个便利的语法，不像是用 SVNPath 指示，我们在此不必为创建新的版本库而重启 Apache。

请确定当你定义新的 Location，不会与其它输出的位置重叠。例如你的主要 DocumentRoot 是*/www*，不要把 Subversion 版本库输出到<Location /www/repos>，如果一个请求的 URI 是*/www/repos/foo.c*，Apache 不知道是直接到 *repos/foo.c* 访问这个文件还是让 **mod_dav_svn** 代理从 Subversion 版本库返回 *foo.c*。服务器返回的结果通常是 301 Moved Permanently。

服务器名称和复制请求

Subversion 利用 COPY 请求类型来执行服务器端的文件和目录拷贝，作为一个健全的 Apache 模块的一部分，拷贝源和拷贝的目标通常坐落在同一个机器上，为了满足这个需求，你或许需要告诉 mod_dav 服务器

主机的名称，通常你可以使用 *httpd.conf* 的 ServerName 指示来完成此目的。

```
ServerName svn.example.com
```

如果你通过 NameVirtualHost 指示使用 Apache 的虚拟主机，你或许需要 ServerAlias 指示来指定额外的名称，再说一次，可以查看 Apache 文档的来得到更多细节。

在本阶段，你一定要考虑访问权限问题，如果你已经作为普通的 web 服务器运行过 Apache，你一定有了一些内容—网页、脚本和其他。这些项目已经配置了许多在 Apache 下可以工作的访问许可，或者更准确一点，允许 Apache 与这些文件一起工作。Apache 当作为 Subversion 服务器运行时，同样需要正确的访问许可来读写你的 Subversion 版本库。

你会需要检验权限系统的设置满足 Subversion 的需求，同时不会把以前的页面和脚本搞乱。这或许意味着修改 Subversion 的访问许可来配合 Apache 服务器已经使用的工具，或者可能意味着需要使用 *httpd.conf* 的 User 和 Group 指示来指定 Apache 作为运行的用户和 Subversion 版本库的组。并不是只有一条正确的方式来设置许可，每个管理员都有不同的原因来以特定的方式操作，只需要意识到许可关联的问题经常在为 Apache 配置 Subversion 版本库的过程中被疏忽。

认证选项

此时，如果你配置的 *httpd.conf* 保存如下的内容

```
<Location /svn>

  DAV svn

  SVNParentPath /usr/local/svn

</Location>
```

…这样你的版本库对全世界是可以“匿名”访问的，直到你配置了一些认证授权政策，你通过 Location 指示来使 Subversion 版本库可以被任何人访问，换句话说，

- 任何人可以使用 Subversion 客户端来从版本库 URL 取出一个工作拷贝（或者是它的子目录），
- 任何人可以在浏览器输入版本库 URL 交互浏览的方式来查看版本库的最新修订版本，并且
- 任何人可以提交到版本库。

当然，你也许已经设置了 *pre-commit* 钩子来防止提交（见“实现版本库钩子”一节），但是就像你读到的，也可以使用 Apache 内置的方法来限制访问。

基本 HTTP 认证

最简单的客户端认证方式是通过 HTTP 基本认证机制，简单的使用用户名和密码来验证一个用户所自称的身份，Apache 提供了一个 **htpasswd** 工具来管理可接受的用户名和密码，这些就是你希望赋予

Subversion 特别权限的用户，让我们给 Sally 和 Harry 赋予提交权限，首先，我们需要添加他们到密码文件。

```
$ ### First time: use -c to create the file

$ ### Use -m to use MD5 encryption of the password, which is more secure

$ htpasswd -cm /etc/svn-auth-file harry

New password: *****

Re-type new password: *****

Adding password for user harry

$ htpasswd -m /etc/svn-auth-file sally

New password: *******

Re-type new password: *******

Adding password for user sally

$
```

下一步，你需要在 *httpd.conf* 的 Location 区里添加一些指示来告诉 Apache 如何来使用这些密码文件，AuthType 指示指定系统使用的认证类型，这种情况下，我们需要指定 Basic 认证系统，AuthName 是你提供给认证域一个任意名称，大多数浏览器会在向用户询问名称和密码的弹出窗口里显示这个名称，最终，使用 AuthUserFile 指示来指定使用 **htpasswd** 创建的密码文件的位置。

添加完这三个指示，你的<Location>区块一定像这个样子：

```
<Location /svn>
```

```
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
</Location>
```

这个<Location>区块还没有结束，还不能做任何有用的事情，它只是告诉 Apache 当需要授权时，要去向 Subversion 客户端索要用户名和密码。我们这里遗漏的，是一些告诉 Apache *什么样*客户端需要授权的指示。哪里需要授权，Apache 就会在哪里要求认证，最简单的方式是保护所有的请求，添加 Require valid-user 来告诉 Apache 任何请求需要认证的用户：

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

一定要阅读后面的部分（“授权选项”一节）来得到 Require 的细节，和授权政策的其他设置方法。

需要警惕：HTTP 基本认证的密码是用明文传输，因此非常不可靠的，如果你担心密码偷窥，最好是使用某种 SSL 加密，所以客户端认证使用 https://而不是 http://，为了方便，你可以配置 Apache 为自签名认证。[41]参考 Apache 的文档（和 OpenSSL 文档）来查看怎样做。

SSL 证书管理

商业应用需要越过公司防火墙的版本库访问，防火墙需要小心的考虑非认证用户“吸取”他们的网络流量的情况，SSL 让那种形式的关注更不容易导致敏感数据泄露。

如果 Subversion 使用 OpenSSL 编译，它就会具备与 Subversion 服务器使用 https://的 URL 通讯的能力，Subversion 客户端使用的 Neon 库不仅仅可以用来验证服务器证书，也可以必要时提供客户端证书，如果客户端和服务器交换了 SSL 证书并且成功地互相认证，所有剩下的交流都会通过一个会话关键字加密。

怎样产生客户端和服务器端证书以及怎样使用它们已经超出了本书的范围，许多书籍，包括 Apache 自己的文档，描述这个任务，现在我们*可以*覆盖的是普通的客户端怎样来管理服务器与客户端证书。

当通过 https://与 Apache 通讯时，一个 Subversion 客户端可以接收两种类型的信息：

- 一个服务器证书
- 一个客户端证书的要求

如果客户端接收了一个服务器证书，它需要去验证它是可以相信的：这个服务器是它自称的那一个吗？OpenSSL 库会去检验服务器证书的签名人或者是*核证机构*（CA）。如果 OpenSSL 不可以自动信任这个 CA，或者是一些其他的问题（如证书过期或者是主机名不匹配），Subversion 命令行客户端会询问你是否愿意仍然信任这个证书：

```
$ svn list https://host.example.com/repos/project



Error validating server certificate for
'https://host.example.com:443':

 - The certificate is not issued by a trusted authority. Use the

   fingerprint to validate the certificate manually!

Certificate information:

 - Hostname: host.example.com

 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56
2006 GMT

 - Issuer: CA, example.com, Sometown, California, US

 - Fingerprint:
7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b



(R)eject, accept (t)emporarily or accept (p)ermanently?
```

这个对话看起来很熟悉，这是你会在 web 浏览器（另一种 HTTP 客户端，就像 Subversion）经常看到的问题，如果你选择(p)ermanent 选项，服务器证书会存放在你存放那个用户名和密码缓存（见“客户端凭证缓

存”一节。）的私有运行区 *auth/*中，缓存后，Subversion 会自动记住在以后的交流中信任这个证书。

你的运行中 *servers* 文件也会给你能力可以让 Subversion 客户端自动信任特定的 CA，包括全局的或是每主机为基础的，只需要设置 `ssl-authority-files` 为一组逗号隔开的 PEM 加密的 CA 证书列表：

```
[global]

ssl-authority-files =
/path/to/CAcert1.pem;/path/to/CAcert2.pem
```

许多 OpenSSL 安装包括一些预先定义好的可以普遍信任的“缺省的”CA，为了让 Subversion 客户端自动信任这些标准权威，设置 `ssl-trust-default-ca` 为 `true`。

当与 Apache 通话时，Subversion 客户端也会收到一个证书的要求，Apache 是询问客户端来证明自己的身份：这个客户端是否是他所说的那一个？如果一切正常，Subversion 客户端会发送回一个通过 Apache 信任的 CA 签名的私有证书，一个客户端证书通常会以加密方式存放在磁盘，使用本地密码保护，当 Subversion 收到这个要求，它会询问你证书的路径和保护用的密码：

```
$ svn list https://host.example.com/repos/project



Authentication realm: https://host.example.com:443

Client certificate filename: /path/to/my/cert.p12
```

```
Passphrase for '/path/to/my/cert.p12':  ********

…
```

注意这个客户端证书是一个“p12”文件，为了让 Subversion 使用客户端证书，它必须是运输标准的 PKCS#12 格式，大多数浏览器可以导入和导出这种格式的证书，另一个选择是用 OpenSSL 命令行工具来转化存在的证书为 PKCS#12 格式。

再次，运行中 *servers* 文件允许你为每个主机自动响应这种要求，单个或两条信息可以用运行参数来描述：

```
[groups]

examplehost = host.example.com



[examplehost]

ssl-client-cert-file = /path/to/my/cert.p12

ssl-client-cert-password = somepassword
```

一旦你设置了 `ssl-client-cert-file` 和 `ssl-client-cert-password` 参数，Subversion 客户端可以自动响应客户端证书请求而不会打扰你。[42]

授权选项

此刻，你已经配置了认证，但是没有配置授权，Apache 可以要求用户认证并且确定身份，但是并没有说明这个身份的怎样允许和限制，这个部分描述了两种控制访问版本库的策略。

整体访问控制

最简单的访问控制形式是授权特定用户为只读版本库访问或者是读/写访问版本库。

你可以通过在<Location>区块添加 Require valid-user 指示来限制所有的版本库操作，使用我们前面的例子，这意味着只有客户端只可以是 harry 或者 sally，而且他们必须提供正确的用户名及对应密码，这样允许对 Subversion 版本库做任何事：

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # only authenticated users may access the repository
  Require valid-user
</Location>
```

有时候，你不需要这样严密，举个例子，Subversion 自己在 http://svn.collab.net/repos/svn 的源代码允许全世界的人执行版本库的只读操作（例如检出我们的工作拷贝和使用浏览器浏览版本库），但是

限定只有认证用户可以执行写操作。为了执行特定的限制，你可以使用 Limit 和 LimitExcept 配置指示，就像 Location 指示，这个区块有开始和结束标签，你需要在<Location>中添加这个指示。

在 Limit 和 LimitExcept 中使用的参数是可以被这个区块影响的 HTTP 请求类型，举个例子，如果你希望禁止所有的版本库访问，只是保留当前支持的只读操作,你可以使用 LimitExcept 指示,并且使用 GET,PROPFIND，OPTIONS 和 REPORT 请求类型参数，然后前面提到过的 Require valid-user 指示将会在<LimitExcept>区块中而不是在<Location>区块。

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # For any operations other than these, require an authenticated
user.
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

这里只是一些简单的例子，想看关于 Apache 访问控制 Require 指示的更深入信息，可以查看 Apache 文档中的教程集 http://httpd.apache.org/docs-2.0/misc/tutorials.html 中的 Security 部分。

每目录访问控制

也可以使用 Apache 的 httpd 模块 **mod_authz_svn** 更加细致的设置访问权限，这个模块收集客户端传递过来的不同的晦涩的 URL 信息，询问 **mod_dav_svn** 来解码，然后根据在配置文件定义的访问政策来裁决请求。

如果你从源代码创建 Subversion，**mod_authz_svn** 会自动附加到 **mod_dav_svn**，许多二进制分发版本也会自动安装，为了验证它是安装正确，确定它是在 *httpd.conf* 的 LoadModule 指示中的 **mod_dav_svn** 后面：

```
LoadModule dav_module        modules/mod_dav.so
LoadModule dav_svn_module    modules/mod_dav_svn.so
LoadModule authz_svn_module  modules/mod_authz_svn.so
```

为了激活这个模块，你需要配置你的 Location 区块的 AuthzSVNAccessFile 指示，指定保存路径中的版本库访问政策的文件。（一会儿我们将会讨论这个文件的格式。）

Apache 非常的灵活，你可以从三种模式里选择一种来配置你的区块，作为开始，你选择一种基本的配置模式。（下面的例子非常简单；见 Apache 自己的文档中的认证和授权选项来查看更多的细节。）

最简单的区块是允许任何人可以访问，在这个场景里，Apache 决不会发送认证请求，所有的用户作为“匿名”对待。

例 6.1. 匿名访问的配置实例。

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file
</Location>
```

在另一个极端，你可以配置为拒绝所有人的认证，所有客户端必须提供证明自己身份的证书，你通过 Require valid-user 指示来阻止无条件的认证，并且定义一种认证的手段。

例 6.2. 一个认证访问的配置实例。

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn
```

```
  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # only authenticated users may access the repository
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

第三种流行的模式是允许认证和匿名用户的组合，举个例子，许多管理员希望允许匿名用户读取特定的版本库路径，但希望只有认证用户可以读（或者写）更多敏感的区域，在这个设置里，所有的用户开始时用匿名用户访问版本库，如果你的访问控制策略在任何时候要求一个真实的用户名，Apache 将会要求认证客户端，为此，你可以同时使用 Satisfy Any 和 Require valid-user 指示。

例 6.3. 一个混合认证/匿名访问的配置实例。

```
<Location /repos>
  DAV svn
```

```
  SVNParentPath /usr/local/svn

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # try anonymous access first, resort to real
  # authentication if necessary.
  Satisfy Any
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

一旦你已经设置了 *httpd.conf* 模版之一，你需要在对应的路径创建包含访问规则的文件，在“基于路径的授权”一节中有描述。

禁用基于路径的检查

mod_dav_svn 模块做了许多工作来确定你标记为“不可读”的数据不会因意外而泄露，这意味着需要紧密监控通过 **svn checkout** 或是 **svn update** 返回的路径和文件内容，如果这些命令遇到一些根据认证策略

不是可读的路径，这个路径通常会被一起忽略，在历史或者重命名操作时—例如运行一个类似 **svn cat -r OLD foo.c** 的命令来操作一个很久以前改过名字的文件 — 如果一个对象的以前的名字检测到是只读的，重命令追踪就会终止。

所有的路径检查在有时会非常昂贵，特别是 **svn log** 的情况。当检索一列修订版本时，服务器会查看所有修订版本修改的路径，并且检查可读性，如果发现了一个不可读路径，它会从修订版本的修改路径中忽略（通常可以使用--verbose 选项查看），并且整个的日志信息会被禁止，不必多说，这种影响大量文件修订版本的操作会非常耗时。这是安全的代价：即使你并没有配置 **mod_authz_svn** 模块，**mod_dav_svn** 还是会询问 **httpd** 来对所有路径运行认证检查，**mod_dav_svn** 模块没有办法知道那个认证模块被安装，所以只能要求 Apache 调用时提供的内容。

在另一方面，也有一个安全舱门允许你用安全特性来交换速度，如果你不是坚持要求有每目录授权（如不使用 **mod_authz_svn** 和类似的模块），你就可以关闭所有的路径检查，在你的 *httpd.conf* 文件，使用 SVNPathAuthz 指示：

例 6.4. 禁用所有的路径检查

```
<Location /repos>
  DAV svn
  SVNParentPath /usr/local/svn
```

```
  SVNPathAuthz off

</Location>
```

SVNPathAuthz 指示缺省是“on”，当设置为“off”时，所有的路径为基础的授权都会关闭；**mod_dav_svn** 停止对每个目录调用授权检查。

额外的糖果

我们已经覆盖了关于认证和授权的 Apache 和 mod_dav_svn 的大多数选项，但是 Apache 还提供了许多很好的特性。

版本库浏览

使用 Apache/WebDAV 配置 Subversion 版本库时一个非常有用的好处是可以用普通的浏览器察看最新的版本库文件，因为 Subversion 使用 URL 来鉴别版本库版本化的资源，版本库使用的 HTTP 为基础的 URL 也可以直接输入到 Web 浏览器中，你的浏览器会发送一个 GET 请求到 URL，根据访问的 URL 是指向一个版本化的目录还是文件，mod_dav_svn 会负责列出目录列表或者是文件内容。

因为 URL 不能确定你所希望看到的资源的版本，mod_dav_svn 会一直返回最新的版本，这样会有一些美妙的副作用，你可以直接把

Subversion 的 URL 传递给文档作为引用，这些 URL 会一直指向文档最新的材料，当然，你也可以在别的网站作为超链使用这些 URL。

我可以看到老的修订版本吗？

通过一个普通的浏览器？一句话：不可以，至少是当你只使用 **mod_dav_svn** 作为唯一的工具时。

你的 Web 浏览器只会说普通的 HTTP，也就是说它只会 GET 公共的 URL，这个 URL 代表了最新版本的文件和目录，根据 WebDAV/DeltaV 规范，每种服务器定义了一种私有的 URL 语法来代表老的资源的版本，这个语法对客户端是不透明的，为了得到老的版本，一个客户端必须通过一种规范过程来“发现”正确的 URL；这个过程包括执行一系列 WebDAV PROPFIND 请求和理解 DeltaV 概念，这些事情一般是你的 web 浏览器做不了的。

为了回答这些问题，一个明显的看老版本文件和目录的方式是带 --revision (-r)参数的 **svn list** 和 **svn cat** 命令，为了在浏览器里察看老版本，你可以使用第三方的软件，一个好的例子是 ViewVC（http://viewvc.tigris.org/），ViewVC 最初写出来是为了在 web 显示 CVS 版本库，[43]最新的版本也已经可以理解 Subversion 版本库了。

正确的文件类型

当浏览 Subversion 版本库时，web 浏览器通过从 Apache 的 HTTP GET 返回内容中查看 Content-Type:头可以知道如何渲染文件的线索，这个值

是一种 MIME 类型。默认情况下，Apache 告诉浏览器所有的版本库文件都是缺省的 MIME 类型，通常是 text/plain，这样有时候会让人沮丧，如果一个用户希望版本库文件能够更有意义的渲染—例如一个 *foo.html*，在浏览时最好能够按照 HTML 方式渲染。

为了生效，我们只需要确认你的文件有正确的 svn:mime-type 设置，这将在“文件内容类型”一节详细讨论，你可以设置的你的客户端在文件首次添加到版本库时自动附加 svn:mime-type 属性；见“自动设置属性”一节。

所以在我们的例子中，如果一个人对 *foo.html* 将 svn:mime-type 设置为 text/html，Apache 就会告知浏览器使用 HTML 方式渲染文件，也可以给图片文件设置合适的 image/*类型，这样最终可以使整个 web 站点直接从版本库浏览，这样做通常没有问题，只要你的站点不包含动态生成的内容。

定制外观

你通常会在版本化的文件的 URL 之外得到更多地用处—毕竟那里是有趣的内容存在的地方，但是你会偶尔浏览一个 Subversion 的目录列表，你会很快发现展示列表生成的 HTML 非常基本，并且一定没有在外观上（或者是有趣上）下功夫，为了自定义这些目录显示，Subversion 提供了一个 XML 目录特性，一个单独的 SVNIndexXSLT 指示在你的 *httpd.conf* 文件版本库的 Location 块里，它将会指导 mod_dav_svn 在显示目录列表的时候生成 XML 输出，并且引用你选择的 XSLT 样式表文件：

```
<Location /svn>

  DAV svn

  SVNParentPath /usr/local/svn

  SVNIndexXSLT "/svnindex.xsl"

  …

</Location>
```

使用 SVNIndexXSLT 指示和创建一个 XSLT 样式表，你可以让你的目录列表的颜色模式与你的网站的其它部分匹配，否则，如果你愿意，你可以使用 Subversion 源分发版本中的 *tools/xslt/*目录下的样例样式表。记住提供给 SVNIndexXSLT 指示的路径是一个 URL 路径—浏览器需要阅读你的样式表来利用它们！

版本库列表

如果你通过 SVNParentPath 指示从一个 URL 维护一组版本库，也可以让 Apache 在浏览器显示所有存在的版本库，只需要通过 SVNListParentPath 指示激活：

```
<Location /svn>

  DAV svn

  SVNParentPath /usr/local/svn

  SVNListParentPath on

  …

</Location>
```

如果一个用户将浏览器指向 http://host.example.com/svn/，她一定会看到*/usr/local/svn* 下所有的 Subversion 版本库，很明显这是一件安全问题，所以这个特性默认是关闭的。

Apache 日志

因为 Apache 的核心是一个 HTTP 服务器，它包含了梦幻般灵活的日志特性。各种配置日志的方式可以超出了本书的范围，但是我们必须指出，即使是最原始的文件 *httpd.conf* 也可以让 Apache 产生两个日志：*error_log* 和 *access_log*。这些日志会出现在不同的地方，但通常是创建在 Apache 安装的日志区。（在 Unix 下，这个目录是 */usr/local/apache2/logs/*。）

error_log 描述了所有 Apache 运行中的内部错误，*access_log* 记录了 Apache 接收到的所有 HTTP 请求，这个日志很容易查看，例如包括 Subversion 客户端的 IP 地址，哪些用户正确认证和请求成功还是失败。

不幸的是，因为 HTTP 是无状态协议，即使最简单的 Subversion 客户端操作会产生多个网络请求，很难通过查看 *access_log* 来确定用户的操作—大多数操作看起来像是一系列神秘的 PROPPATCH、GET、PUT 和 REPORT 请求。更糟糕的是，许多客户段操作会发送几乎完全相同的一系列请求，所以更加难以区分。

mod_dav_svn 会成为一个辅助，通过激活“operational logging”属性，你可以告诉 mod_dav_svn 创建另外的日志文件，来描述你的客户度 uan 做了哪些高级操作。

为此，你需要利用 Apache 的 CustomLog 指示（在 Apache 自己的文档里有详细解释）指示，请确定在 Subversion 的 Location 指示之外配置这个指示。

```
<Location /svn>

  DAV svn

  …

</Location>



CustomLog logs/svn_logfile "%t %u %{SVN-ACTION}e"
env=SVN-ACTION
```

在这个例子里，我们告诉 Apache 在标准的 Apache*logs* 目录创建一个 *svn_logfile* 日志文件，%t 和%u 变量会被请求的时间和用户名代替，关键的部分是 SVN-ACTION 的两个实例，当 Apache 看到变量，会将变量的值替代为环境变量 SVN-ACTION，这个环境变量的值是 mod_dav_svn 在检测到高级客户段操作时自动设置的。

所以我们不选择翻译下面的传统的 *access_log* 文件：

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND
/svn/calc/!svn/vcc/default HTTP/1.1" 207 398
```

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/bln/59
HTTP/1.1" 207 449

[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc HTTP/1.1" 207
647

[26/Jan/2007:22:25:29 -0600] "REPORT
/svn/calc/!svn/vcc/default HTTP/1.1" 200 607

[26/Jan/2007:22:25:31 -0600] "OPTIONS /svn/calc HTTP/1.1" 200
188

[26/Jan/2007:22:25:31 -0600] "MKACTIVITY
/svn/calc/!svn/act/e6035ef7-5df0-4ac0-b811-4be7c823f998
HTTP/1.1" 201 227

…
```

… 你可以细读一个更加智能的 *svn_logfile* 文件：

```
[26/Jan/2007:22:24:20 -0600] - list-dir '/'

[26/Jan/2007:22:24:27 -0600] - update '/'

[26/Jan/2007:22:25:29 -0600] - remote-status '/'

[26/Jan/2007:22:25:31 -0600] sally commit r60
```

其它特性

Apache 作为一个健壮的 Web 服务器的许多特性也可以用来增加 Subversion 的功能性和安全性，Subversion 使用 Neon 与 Apache 通讯，这是一种一般的 HTTP/WebDAV 库，可以支持 SSL（Secure Socket Layer，将在后面讨论）。如果你的 Subversion 是以支持 SSL（安全套接层，过一会儿讨论）编译，则你可以使用 https://访问 Apache 服务器。

同样有用的是 Apache 和 Subversion 关系的一些特性，像可以指定自定义的端口（而不是缺省的 HTTP 的 80）或者是一个 Subversion 可以被访问的虚拟主机名，或者是通过 HTTP 代理服务器访问的能力，这些特性都是 Neon 所支持的，所以 Subversion 轻易得到这些支持。

最后，因为 **mod_dav_svn** 是使用一个半完成的 WebDAV/DeltaV 方言，所以通过第三方的 DAV 客户端访问也是可能的，几乎所有的现代操作系统（Win32、OS X 和 Linux）都有把 DAV 服务器影射为普通的网络“共享”的内置能力，这是一个复杂的主题；察看附录 C, *WebDAV 和自动版本*来得到更多细节。

基于路径的授权

Apache 和 **svnserve** 都可以给用户赋予（或拒绝）访问许可，通常是对整个版本库：一个用户可以读版本库（或不），而且他可以写版本库（或不）。如果可能，也可以定义细粒度的访问规则。一组用户可以有版本库的一个目录的读写权限，但是没有其它的；另一个目录可以是只对一少部分用户可读。

两种服务器都使用同样的文件格式描述路径为基础的规则，如果是 Apache，需要加载 **mod_authz_svn** 模块，然后添加 AuthzSVNAccessFile 指示（在文件 *httpd.conf* 中）指明你的规则文件。（完全解释可以看“每目录访问控制”一节。）如果你在使用 **svnserve**，你需要让你的 authz-db 变量（在 *svnserve.conf* 中）指向规则文件。

你真的需要基于路径的访问控制吗？

许多第一次设置 Subversion 的管理员会在未经太多的思考的情况下轻易选择使用路径为基础的访问控制，管理员通常知道团队的成员工作在哪个项目，所以很容易确定赋予哪些团队访问哪些目录，不能访问哪些目录。这看起来是很自然的事情，它满足了管理员紧密控制版本库访问的愿望。

注意，这个特性通常有一些看不见（和可见的）代价。可见的，需要更多的工作来确信用户对某个路径有读写权限；在一些情况下，会是非常大的性能损失。不可见的，考虑你创建的文化，大多数情况下，因为特定用户不能够在特定目录提交修改，所以社会契约不必通过技术来加强。团队有时候可以自然的互相协作；一些人会通过为他人提交不能正常工作目录的内容的方法帮助别人，你设置了一种不期望交流的障碍。你也要建立一套项目开发、新人加入等活动的规则，还有很多额外的工作。

记住这是一个版本控制系统，即使一些人不小心提交了一些不该提交的东西，很容易回退修改。如果一个用户故意提交到了错误的位置，这是一个社会问题，需要在 Subversion 之外解决。

所以在开始限制用户的访问权限之前，你要问你自己是否有一个真正的、正直的需要，或仅仅是为了对一个管理员来说这样“听起来不错”。决定是否值得影响服务器的速度，必须记住只有很小的风险；依靠技术手段解决社会问题并不好。[44]。

作为一个思考的例子，考虑 Subversion 项目本身有允许某个用户可以在那个目录提交的设置，而只是通过社交方式规定。这是一个社区信任的好模型，特别是对开源项目。当然，有时候需要正统的路径为基础的访问控制；在公司中，例如，只有部分数据是敏感的，只允许以小组人可以访问。

当你的服务器知道去查找规则文件时，就是需要定义规则的时候了。

访问文件的语法与 **svnserve.conf** 和运行中配置文件非常相似，以（#）开头的行会被忽略，在它的简单形式里，每一小节命名一个版本库和一个里面的路径，认证用户名是在每个小节中的选项名，每个选项的值描述了用户访问版本库的级别：r（只读）或者 rw（读写），如果用户没有提到，访问是不允许的。

具体一点：这个小节的名称是[repos-name:path]或者[path]的形式，如果你使用 SVNParentPath 指示，指定版本库的名字是很重要的，如果你漏掉了他们，[/some/dir]部分就会与*/some/dir* 的所有版本库匹配，如果你使用 SVNPath 指示，因此在你的小节中只是定义路径也很好—毕竟只有一个版本库。

```
[calc:/branches/calc/bug-142]

harry = rw

sally = r
```

在第一个例子里，用户 harry 对 calc 版本库中*/branches/calc/bug-142* 具备完全的读写权利，但是用户 sally 只有读权利，任何其他用户禁止访问这个目录。

当然，访问控制是父目录传递给子目录的，这意味着我们可以为 Sally 指定一个子目录的不同访问策略：

```
[calc:/branches/calc/bug-142]

harry = rw

sally = r



# give sally write access only to the 'testing' subdir

[calc:/branches/calc/bug-142/testing]

sally = rw
```

现在 Sally 可以读取分支的 *testing* 子目录，但对其他部分还是只可以读，同时，Harry 对整个分支还继续有完全的读写权限。

也可以通过继承规则明确的的拒绝某人的访问，只需要设置用户名参数为空：

```
[calc:/branches/calc/bug-142]

harry = rw

sally = r



[calc:/branches/calc/bug-142/secret]
```

```
harry =
```

在这个例子里，Harry 对 *bug-142* 目录树有完全的读写权限，但是对其中的 *secret* 子目录没有任何访问权利。

需要记住的是最详细的的路径会被匹配，服务器首先找到匹配自己的目录，然后父目录，然后父目录的父目录，就这样继续下去，更具体的路径控制会覆盖所有继承下来的访问控制。

缺省情况下，没有人对版本库有任何访问，这意味着如果你已经从一个空文件开始，你会希望给所有用户对版本库根目录具备读权限，你可以使用星号（*）实现，用来代表“所有用户”：

```
[/]
* = r
```

这是一个普通的设置；注意在小节名中没有提到版本库名称，这让所有版本库对所有的用户可读。当所有用户对版本库有了读权利，你可以赋予特定用户对特定子目录的 rw 权限。

星号（*）参数需要在这里详细强调：这是匹配匿名用户的*唯*一模式，如果你已经配置了你的 Location 区块允许匿名和认证用户的混合访问，所有用户作为 Apache 匿名用户开始访问，**mod_authz_svn** 会在要访问路径的定义中查找*值；如果找不到，Apache 就会要求真实的客户端认证。

访问文件也允许你定义一组的用户，很像 Unix 的*/etc/group* 文件：

```
[groups]

calc-developers = harry, sally, joe

paint-developers = frank, sally, jane

everyone = harry, sally, joe, frank, sally, jane
```

组可以被赋予通用户一样的访问权限，使用“at”（@）前缀来加以区别：

```
[calc:/projects/calc]

@calc-developers = rw



[paint:/projects/paint]

@paint-developers = rw

jane = r
```

组中也可以定义为包含其它的组：

```
[groups]

calc-developers = harry, sally, joe

paint-developers = frank, sally, jane

everyone = @calc-developers, @paint-developers
```

部分可读性和检出

如果你使用 Apache 作为 Subversion 服务器，并让版本库的某些子目录对特定用户不可读，然后你需要知道 **`svn checkout`** 会有一个不理想的执行方式。

当客户端通过 HTTP 请求检出或更新时，它会做出一个单独的服务器请求，并接收一个单独的（通常很大）服务器响应，当服务器接收到请求，这是 Apache 服务器要求用户认证的*唯*一机会，这有一些副作用。例如，如果版本库的一个特定子目录只对用户 Sally 可读，用户 Harry 检出父目录，他的客户端会在收到认证要求时返回用户名 Harry，因为服务器生成的响应很大，无法在达到特别子目录时重新发送认证请求；因此子目录会被一起略过，而不会询问用户是否使用 Sally 重新认证。如果版本库是匿名可访问的，则整个检出将不需要认证—再一次，略过不可读的目录，而不会中途要求认证。

支持多种版本库访问方法

你已经看到了一个版本库可以用多种方式访问，但是可以—或者说安全的—用几种方式同时并行的访问你的版本库吗？回答是可以，倘若你有一些深谋远虑的使用。

在任何给定的时间，这些进程会要求读或者写访问你的版本库：

- 常规的系统用户使用 Subversion 客户端（客户端程序本身）通过 `file://`URL 直接访问版本库；

- 常规的系统用户连接使用 SSH 调用的访问版本库的 **svnserve** 进程（就像它们自己运行一样）；
- 一个 **svnserve** 进程—是一个守护进程或是通过 **inetd** 启动的—作为一个固定的用户运行；
- 一个 Apache **httpd** 进程，以一个固定用户运行。

最通常的一个问题是管理进入到版本库的所有权和访问许可，是前面例子的所有进程 （或者说是用户）都有读写 Berkeley DB 的权限？假定你有一个类 Unix 的操作系统，一个直接的办法是在新的 svn 组添加所有潜在的用户，然后让这个组完全拥有版本库，但这样还不足够，因为一个进程会使用不友好的 umask 来写数据库文件—用来防止别的用户的访问。

所以下一步我们不选择为每个版本库用户设置一个共同的组的方法，而是强制每个版本库访问进程使用一个健全的 umask。对直接访问版本库的用户，你可以使用 **svn** 的包裹脚本来首先设置 **umask 002**，然后运行真实的 **svn** 客户端程序，你可以为 **svnserve** 写相同的脚本，并且增加 **umask 002** 命令到 Apache 自己的启动脚本 *apachectl* 中。例如：

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
```

```
/usr/bin/svn-real "$@"
```

另一个在类 Unix 系统下常见的问题是，当版本库在使用时，BerkeleyDB 有时候创建一个新的日志文件来记录它的东西，即使这个版本库是完全由 **svn** 组拥有，这个新创建的文件不是必须被同一个组拥有，这给你的用户造成了更多地许可问题。一个好的工作区应该设置组的 SUID 字节到版本库的 *db* 目录，这会导致所有新创建的日志文件拥有同父目录相同的组拥有者。

一旦你跳过了这些障碍，你的版本库一定是可以通过各种可能的手段访问了，这看起来有点凌乱和复杂，但是这个让多个用户分享对一个文件的写权限的问题是一个经典问题，并且经常是没有优雅的解决。

幸运的是，大多数版本库管理员不*需要*这样复杂的配置，用户如果希望访问本机的版本库，并不是一定要通过 `file://`的 URL—他们可以用 `localhost` 机器名联系 Apache 的 HTTP 服务器或者是 **`svnserve`**，协议分别是 `http://`或 `svn://`。为你的 Subversion 版本库维护多个服务器进程，版本库会变得超出需要的头痛，我们建议你选择最符合你的需要的版本库，并且坚持使用！

svn+ssh 服务器检查列表

让一些用户通过存在的 SSH 帐户来共享版本库而没有访问许可问题是一件很有技巧的事情，如果你为自己需要在（作为一个管理员）类 Unix

系统上做的事情感到迷惑，这里是一些快速的检查列表，总结了本小节讨论的事情：

- 所有的 SSH 用户需要能够读写版本库，把所有的 SSH 用户放到同一个组里。
- 让那个组拥有整个版本库。
- 设置组的访问许可为读/写。
- 你的用户在访问版本库时需要使用一个健全的 umask，确定 **svnserve**（*/usr/bin/svnserve* 或者是任何一个`$PATH` 说明的位置）是一个设置了 **umask 002** 和执行真正的 **svnserve** 程序的包裹脚本，对 **svnlook** 和 **svnadmin** 使用相同的措施，或者是使用一个健全的 umask 运行或者是使用上面说明的包裹。
- **svnlook** 和 **svnadmin** 的使用类似，使用健全的 umask 或者使用前面提到的包裹程序。

---

[38] 见 RFC 2195。

[39] 请注意，使用 **svnserve** 的访问控制进行权限控制将会失去意义，因为用户已经直接访问到了版本库数据。

[40] 他们讨厌这样做。

[41] 当使用自签名的服务器时仍会遭受“中间人”攻击，但是与偷取未保护的密码相比，这样的攻击比一个偶然的获取要艰难许多。

[42] 更多有安全意识的人不会希望在运行中 *servers* 文件保存客户端证书密码。

[43] 之前叫做“ViewCVS”。

[44] 本书的共同主题！

定制你的 Subversion 体验

目录

版本控制可以成为复杂的主题，和科学一样充满艺术性，为解决事情能提供了无数的方法。贯穿这本书，你已经阅读许多 Subversion 命令行子命令，以及可以改变运行方式的选项，在本章我们要查看一些自定义 Subversion 工作的方法—设置 Subversion 运行配置，使用外置帮助程序，Subversion 与操作系统配置的地区交互等等。

运行配置区

Subversion 提供了许多用户可以控制的可选行为方式，许多是用户希望添加到所有的 Subversion 操作中的选项，为了避免强制用户记住命令行参数并且在每个命令中使用，Subversion 使用配置文件，并且将配置文件保存在独立的 Subversion 配置区。

Subversion *配置区*是一个双层结构，保存了可选项的名称和值。通常，Subversion 配置区是一个保存*配置文件*的特殊目录（第一层结构），目录中保存了一些标准 INI 格式的文本文件（文件中的“section”形成第二层结构）。这些文件可以简单用你喜欢的文本编辑器编辑（如 Emacs 或 vi），而且保存了客户端可以读取的指示，用来指导用户的一些行为选项。

配置区布局

svn 命令行客户端第一次执行时，会创建一个用户配置区，在类 Unix 系统中，配置区位于用户主目录中，名为*.subversion*。在 Win32 系统，Subversion 创建一个名为 *Subversion* 的目录，这个目录通常位于用户配

置目录（顺便说一句，通常是一个隐藏目录）的 *Application Data* 子目录下。然而，在 Win32 平台上，此目录的具体位置在不同的系统上是不一样的，由 Windows 注册表决定。[45]我们以 Unix 下的名字 *.subversion* 来表示用户配置区。

除了用户配置区，Subversion 也提供了系统配置区，通过系统配置区，系统管理员可以为某个机器的所有用户建立缺省配置值。注意系统配置区不会规定强制性的策略—每个用户配置区都可以覆盖系统配置区中的配置项，而 **`svn`** 的命令行参数决定了最后的行为。在类 Unix 的平台上，系统配置区位于 */etc/subversion* 目录下，在 Windows 平台上，系统配置区位于 *Application Data*（再说一次，是由 Windows 注册表决定的）的 *Subversion* 目录中。与每用户配置区不同，**`svn`** 不会试图创建系统配置区。

目前，Subversion 的配置区包含三个文件—两个配置文件（*config* 和 *servers*），和一个 INI 文件格式的 *README.txt* 描述文件。配置文件创建的时候，Subversion 的选项都设置为默认值。配置文件中的选项都按功能划分成组，大多数选项还有详细的文字描述注释，说明这些选项的值对 Subversion 的主要影响。要修改选项，只需用文本编辑器打开并编辑配置文件。如果想要恢复缺省的配置，可以直接删除（或者重命名）配置目录，并且运行一些如 **`svn --version`** 之类的无关紧要的 **`svn`** 命令，一个包含缺省值的新配置目录就会创建起来。

用户配置区也缓存了认证信息，*auth* 目录下的子目录中缓存了一些 Subversion 支持的各种认证方法的信息,这个目录需要相应的用户权限才可以访问。

配置和 Windows 注册表

除了基于 INI 文件的配置区，运行在 Windows 平台的 Subversion 客户端也可以使用 Windows 注册表来保存配置数据。注册表中保存的选项名称和值的含义与 INI 文件中相同，“file/section”在注册表中表现为注册表键树的层级，使得双层结构得以保留下来。

Subversion 的系统配置值保存在键 `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` 下。举个例子，`global-ignores` 选项位于 *config* 文件的 `miscellany` 小节，在 Windows 注册表中，则位于 `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`。用户配置值存放在 `HKEY_CURRENT_USER\Software\Tigris.org\Subversion` 下。

基于注册表的配置项在基于文件的配置项之*前*解析，所以其配置项的值会被配置文件中相同配置项的值覆盖，换句话说，在 Windows 系统下这样查找配置信息；低位的位置优先于高位的位置：

1. 命令行选项
2. 用户 INI 配置文件

3. 用户注册表值

4. 系统 INI 配置文件

5. 系统注册表值

此外，虽然 Windows 注册表不支持“注释掉”这种概念，但是 Subversion 会忽略所有以井号（#）开始的字符，这允许你快速的取消一个选项而不需要删除整个注册表键，明显简化了恢复选项的过程。

svn 命令行客户端不会尝试写 Windows 注册表，也不会在注册表中创建默认配置区。不过可以使用 **REGEDIT** 创建所需的键。此外，还可以创建一个*.reg* 文件，并在文件浏览器中双击这个文件，文件中的数据就会合并到注册表中。

例 7.1. 注册表条目(.reg)样本文件。

```
REGEDIT4



[HKEY LOCAL MACHINE\Software\Tigris.org\Subversion\Servers\groups]



[HKEY LOCAL MACHINE\Software\Tigris.org\Subversion\Servers\global]

"#http-proxy-host"=""

"#http-proxy-port"=""

"#http-proxy-username"=""

"#http-proxy-password"=""

"#http-proxy-exceptions"=""
```

```
"#http-timeout"="0"

"#http-compression"="yes"

"#neon-debug-mask"=""

"#ssl-authority-files"=""

"#ssl-trust-default-ca"=""

"#ssl-client-cert-file"=""

"#ssl-client-cert-password"=""



[HKEY CURRENT USER\Software\Tigris.org\Subversion\Conf
ig\auth]

"#store-passwords"="yes"

"#store-auth-creds"="yes"



[HKEY CURRENT USER\Software\Tigris.org\Subversion\Conf
ig\helpers]

"#editor-cmd"="notepad"

"#diff-cmd"=""

"#diff3-cmd"=""

"#diff3-has-program-arg"=""



[HKEY CURRENT USER\Software\Tigris.org\Subversion\Conf
ig\tunnels]



[HKEY CURRENT USER\Software\Tigris.org\Subversion\Conf
ig\miscellany]

"#global-ignores"="*.o *.lo *.la #*# .*.rej *.rej .*~
*~ .#* .DS Store"
```

```
"#log-encoding"=""

"#use-commit-times"=""

"#no-unlock"=""

"#enable-auto-props"=""



[HKEY CURRENT USER\Software\Tigris.org\Subversion\Config\auto-props]
```

上面例子里显示的*.reg* 文件中，包含了一些最常用的配置选项和它们的缺省值。注意，上面的例子中不仅包含了系统设置（关于网络代理相关的选项），也包含了用户设置（指定的编辑器程序，是否保存密码，以及其它选项）。同时要注意的是，所有选项都注释掉了，要启用其中的选项，只需删除该选项名称前面的井号（#），然后设置相应的值就可以了。

配置选项

本节我们会详细讨论 Subversion 目前支持的运行配置选项。

服务器

servers 文件保存了 Subversion 关于网络层的配置选项，这个文件有两个特别的小节：groups 和 global。groups 小节是一个交叉引用表，其中的关键字是 *servers* 文件中其它小节的名字，值则是 *globs* 格式的，也就是包含通配符的字符序列，对应于接收 Subversion 请求的主机名。

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
…

[collabnet]
…
```

当通过网络访问 Subversion 服务器时，客户端会设法匹配正在尝试连接的服务器名字和 groups 小节中的 glob 名称，如果发现匹配，Subversion 会在 *servers* 文件中查找对应于这个 glob 名称的小节，并从该小节中去读取真实的网络配置设置。

如果没有能够匹配到 groups 中的 glob 名称，global 小节中的选项就会发生作用。global 小节中的选项与其他小节一样（当然是除了 groups 小节），这些选项是：

http-proxy-exceptions

> 这里指定了一组逗号分割的列表，其内容是无须代理服务器可以直接访问的版本库主机名模式，模式语法与 Unix 的 shell 中的文件名相同，其中任何匹配的版本库主机不会通过代理访问。

http-proxy-host

代理服务器的详细主机名，是 HTTP 为基础的 Subversion 请求必须通过的，缺省值为空，意味着 Subversion 不会尝试通过代理服务器进行 HTTP 请求，而会尝试直接连接目标机器。

http-proxy-port

代理服务器的详细端口，缺省值为空。

http-proxy-username

代理服务器的用户名，缺省值为空。

http-proxy-password

代理服务器的密码，缺省为空。

http-timeout

等待服务器响应的时间，以秒为单位，如果你的网络速度较慢，导致 Subversion 的操作超时，你可以加大这个数值，缺省值是 0，意思是让 HTTP 库 Neon 使用自己的缺省值。

http-compression

这说明是否在与设置好 DAV 的服务器通讯时使用网络压缩请求，缺省值是 yes（尽管只有在这个功能编译到网络层时压缩才会有效），设置 no 来关闭压缩，如调试网络传输时。

neon-debug-mask

只是一个整形的掩码，底层的 HTTP 库 Neon 用来选择产生调试的输出，缺省值是 0，意思是关闭所有的调试输出，关于 Subversion 使用 Neon 的详细信息，见第 8 章 *嵌入 Subversion*。

`ssl-authority-files`

这是一个分号分割的路径和文件列表，这些文件包含了 Subversion 客户端在用 HTTPS 访问时可以接受的认证授权（或者 CA）证书。

`ssl-trust-default-ca`

如果你希望 Subversion 可以自动相信 OpenSSL 携带的缺省的 CA，可以设置为 `yes`。

`ssl-client-cert-file`

如果一个主机（或是一些主机）需要一个 SSL 客户端证书，你会收到一个提示说需要证书的路径。通过设置这个路径你的 Subversion 客户端可以自动找到你的证书而不会打扰你。没有标准的存放位置；Subversion 会从任何你指定的路径得到这个文件。

`ssl-client-cert-password`

如果你的 SSL 客户端证书文件是用密码加密的，Subversion 会在每次使用证书时请你输入密码，如果你发现这很讨厌（并且不介意把密码存放在 *servers* 文件中），你可以设置这个参数为证书的密码，这样就不会再收到密码输入提示了。

配置

其它的 Subversion 运行选项保存在 *config* 文件中，这些运行选项与网络连接无关，只是一些正在使用的选项，但是为了应对未来的扩展，也按小节划分成组。

auth 小节保存了 Subversion 相关的认证和授权的设置，它包括：

store-passwords

这告诉 Subversion 是否缓存服务器认证要求时用户提供的密码，缺省值是 yes。设置为 no 可以关闭在存盘的密码缓存，你可以通过 **svn** 的--no-auth-cache 命令行参数（那些支持这个参数的子命令）来覆盖这个设置，详细信息请见“客户端凭证缓存”一节。

store-auth-creds

这个设置与 store-passwords 相似，不过设置了这个选项将会保存*所有*认证信息，如用户名、密码、服务器证书，以及其他任何类型的可以缓存的凭证。

helpers 小节控制完成 Subversion 任务的外部程序，正确的选项包括：

editor-cmd

Subversion 在提交操作时用来询问用户日志信息的程序，例如使用 **svn commit** 而没有指定--message（-m）或者--file（-F）选项。这个程序也会与 **svn propedit** 一起使用——一个临时文件跳出来包含已经存在的用户希望编辑的属性，然后用户可以对这

个属性进行编辑（见“属性”一节），这个选项的缺省值为空，检测编辑器的顺序如下（小号码位置优先于大号码位置）：

1. 命令行选项--editor-cmd
2. 环境变量 SVN_EDITOR
3. 配置选项 editor-cmd
4. 环境变量 VISUAL
5. 环境变量 EDITOR
6. 也有可能 Subversion 会有一个内置的缺省值（官方编译版本不是如此）

所有这些选项和变量（不像 diff-cmd）的值的开头都是 shell 中要执行的命令行，Subversion 会追加一个空格和一个需要编辑的临时文件,编辑器必须修改临时文件,并且返回一个 0 来表明成功。

diff-cmd

这里是比较程序的绝对路径，当 Subversion 生成了“diff”输出时（例如当使用 **svn diff** 命令）就会使用，缺省 Subversion 会使用一个内置的比较库—设置这个参数会强制它使用外部程序执行这个任务，此类程序的更多信息见“使用外置比较工具”一节。

diff3-cmd

这指定了一个三向的比较程序，Subversion 使用这个程序来合并用户和从版本库接受的修改，缺省 Subversion 会使用一个内置的比较库—设置这个参数会导致它会使用外部程序执行这个任务，此类程序的更多信息见“使用外置比较工具”一节。

diff3-has-program-arg

如果 diff3-cmd 选项设置的程序接受一个--diff-program 命令行参数，这个标记必须设置为 true。

tunnels 小节允许你定义一个 **svnserve** 和 svn://客户端连接使用的管道模式，更多细节见“SSH 隧道”一节。

miscellany 小节是一些没法归到别处的选项。[46]在本小节，你会找到：

global-ignores

当运行 **svn status** 命令时，Subversion 会和版本化的文件一样列出未版本化的文件和目录，并使用?字符（见 see“查看你的修改概况”一节）标记，有时候察看无关的未版本化文件会很讨厌—比如程序编译产生的对象文件—的显示出来。global-ignores 选项是一个空格分隔的列表，用来描述 Subversion 在它们版本化之前不想显示的文件和目录，缺省值是*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store。

就像 **svn status**，**svn add** 和 **svn import** 命令也会忽略匹配这个列表的文件，你可以用单个的--no-ignore 命令行参数来覆盖这个选项。

For information on more fine-grained control of ignored items, see “忽略未版本控制的条目”一节.

enable-auto-props

这里指示 Subversion 自动对新加的或者导入的文件设置属性，缺省值是 no，可以设置为 yes 来开启自动添加属性，这个文件的 auto-props 小节会说明哪些属性会被设置到哪些文件。

log-encoding

这个变量设置提交日志缺省的字符集，是--encoding 选项（见“**svn** 选项”一节）的永久形式，Subversion 版本库保存了一些 UTF-8 的日志信息，并且假定你的日志信息是用操作系统的本地编码，如果你提交的信息使用别的编码方式，你一定要指定不同的编码。

use-commit-times

通常你的工作拷贝文件会有最后一次被进程访问的时间戳，不管是你自己的编辑器还是用 **svn** 子命令。这通常对人们开发软件提供了便利，因为编译系统通常会通过查看时间戳来决定那些文件需要重新编译。

在其他情形，有时候如果工作拷贝的文件时间戳反映了上一次在版本库中更改的时间会非常好，**svn export** 命令会一直放置这些“上次提交的时间戳”放到它创建的目录树。通过设置这个 config 参数为 yes，**svn checkout**、**svn update**、**svn switch** 和 **svn revert** 命令也会为它们操作的文件设置上次提交的时间戳。

auto-props 小节控制 Subversion 客户端自动设置提交和导入的文件的属性的能力，它可以包含任意数量的键-值对，格式是 PATTERN = PROPNAME=PROPVALUE，其中 PATTERN 是一个文件模式，匹配一系列文件名，此行其它两项为属性和值。如果一个文件匹配多次，会导致有多个属性集；然而，没有手段保障自动属性不会按照配置文件中的顺序应用，所以你可以一个规则“覆盖”另一个。你可以在 *config* 文件找到许多自动属性的用法实例。最后，如果你希望开启自动属性，不要忘了设置 miscellany 小节的 enable-auto-props 为 yes。

本地化

*本地化*是让程序按照地区特定方式运行的行为，如果一个程序的格式、数字或者是日期是你的本地方式，或者是打印的信息（或者是接受的输入）是你本地的语言，这个程序被叫做已经*本地化了*，这部分描述了针对本地化的 Subversion 的步骤。

理解地区

许多现代操作系统都有一个“当前地区”的概念—也就是本地化习惯服务的国家和地区。这些习惯—通常是被一些运行配置机制选择—影响程序展现数据的方式，也有接受用户输入的方式。

在类 Unix 的系统，你可以运行 **locale** 命令来检查本地关联的运行配置的选项值：

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
```

输出是一个本地相关的环境变量和它们的值，在这个例子里，所有的变量设置为缺省的 C 地区，但是用户可以设置这些变量为特定的国家/语言代码组合。举个例子，如果有人设置 LC_TIME 变量为 fr_CA，然后程序会知道使用讲法语的加拿大期望的格式来显示时间和日期信息。如果一个人会设置 LC_MESSAGES 变量为 zh_TW，程序会知道使用繁体中文显示可读信息。如果设置 LC_ALL 的效果同分别设置所有的位置变量为同一个值有相同的效果。LANG 用来作为没有设置地区变量的缺省值，为了查看 Unix 系统所有的地区列表，运行 **locale -a** 命令。

在 Windows，地区配置是通过“地区和语言选项”控制面板管理的，可以从已存在的地区查看选择，甚至可以自定义（会是个很讨厌的复杂事情）许多显示格式习惯。

Subversion 对区域设置的支持

Subversion 客户端，**svn** 通过两种方式支持当前的地区配置。首先，它会注意 LC_MESSAGES 的值，然后尝试使用特定的语言打印所有的信息，例如：

```
$ export LC MESSAGES=de DE

$ svn help cat

cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.

Aufruf: cat ZIEL[@REV]...

…
```

这个行为在 Unix 和 Windows 上同样工作，注意，尽管有时你的操作系统支持某个地区，Subversion 客户端可能不能讲特定的语言。为了制作本地化信息，志愿者可以提供各种语言的翻译。翻译使用 GNU gettext 包编写，相关的翻译模块使用.*mo* 作为后缀名。举个例子，德国翻译文件为 *de.mo*。翻译文件安装到你的系统的某个位置，在 Unix 它们会在 */usr/share/locale/*，而在 Windows 它们通常会在 Subversion 安装的 *share**locale*\目录。一旦安装，一个命名在程序后面的模块会为此提供翻译。举个例子，*de.mo* 会最终安装到

/usr/share/locale/de/LC_MESSAGES/subversion.mo，通过查看安装的*.mo*文件，我们可以看到 Subversion 支持的语言。

第二种支持地区设置的方式包括 **svn** 怎样解释你的输入，版本库使用 UTF-8 保存了所有的路径，文件名和日志信息。在这种情况下，版本库是*国际化的*—也就是版本库准备接受任何人类的语言。这意味着，无论如何 Subversion 客户端要负责发送 UTF-8 的文件名和日志信息到版本库，为此，必须将数据从本地位置转化为 UTF-8。

举个例子，你创建了一个文件叫做 *caffè.txt*，然后提交了这个文件，你写的日志信息是“Adesso il caffè è più forte”，文件名和日志信息都包含非 ASCII 字符，但是因为你的位置设置为 `it_IT`，Subversion 知道把它们作为意大利语解释，在发送到版本库之前，它用意大利字符集转化数据为 UTF-8。

注意当版本库要求 UTF-8 文件名和日志信息时，它*不会*注意到文件的内容，Subversion 会把文件内容看作字节串，没有任何客户端和服务器会尝试理解或是编码这些内容。

字符集转换错误

当使用 Subversion，你或许会碰到一个字符集转化关联的错误：

```
svn: Can't convert string from native encoding to 'UTF-8':
…
svn: Can't convert string from 'UTF-8' to native encoding:
```

…

这个错误信息通常会发生在 Subversion 客户端从版本库接收到一个 UTF-8 串，但字符不能转化为当前的地区文字时，举个例子，如果你的地区设置是 en_US，但是一个写作者使用日本文件名提交，你会在 **svn update** 接受文件时会看到这个错误。

解决方案或者是设置你的地区为*可以*表示即将到来的 UTF-8 数据，或者是修改版本库的文件名或信息。（不要忘记和你的合作者拍拍手—项目必须首先决定通用的语言，这样所有的参与者会使用相同的地区设置。）

使用外置比较工具

选项--diff-cmd 和--diff3-cmd 的形式相似，也有类似名称的运行配置参数（见“配置”一节），这会导致一个错误的观念，也就是在 Subversion 中使用外置的比较（或“diff”）和合并工具会非常的容易，虽然 Subversion 可以使用大多数类似的工具，但是设置这些工具绝非易事。

Subversion 和外置比较和合并工具的接口可以追溯到很久以前，当时 Subversion 的唯一文本比较能力是建立在 GNU 的工具链之上，特别是 **diff** 和 **diff3** 工具，为了得到 Subversion 需要的方式，它使用非常复杂的选项和参数调用这些工具，而这些选项和参数都是工具特定的，渐渐的，Subversion 发展了自己的比较区别库作为备份机制。

[47]--diff-cmd 和--diff3-cmd 选项是添加到 Subversion 的命令行客户端，

所以用户可以更加容易的指明他们最喜欢的使用的 GNU diff 和 diff3 工具，而不是新奇的内置比较库，如果使用了这些选项，Subversion 会忽略内置的比较库，转而使用外置程序，使用冗长的参数列表，目前还是这种情况。

人们很快意识到使用简单的配置机制必须使 Subversion 使用位于特定位置的 GNU diff 和 diff3 工具，毕竟，Subversion 并不验证其被告之要执行的程序是否是 GNU 的工具链的比较工具。唯一可以配置的方面是外置工具在系统的位置—而不是选项集，参数顺序等等。Subversion 一直将这些 GNU 工具选项发给你的外置比较工具，而不管程序是否可以理解那些选项，那不是所有用户直觉的方式。

使用外置比较和合并工具的关键是使用包裹脚本将 Subversion 的输出转化为你的脚本程序可以理解的形式，然后将这些比较工具的输出转化为你的 Subversion 期望的格式—GNU 工具可能使用的格式，下面的小节覆盖了那些期望格式的细节。

何时启动文本比较或合并的决定完全是 Subversion 的决定，而这个决定是根据文件的 svn:mime-type 属性作出的，这意味着，例如，即使你有一个可以识别 Microsoft Word 格式的比较或合并工具，当你对一个 Word 文件设置为非人工可读（例如 application/msword）时，依然不会调用这个识别 Word 的工具。关于 MIME type 的设定，可以见“文件内容类型”一节。

外置 diff

Subversion 可以调用适合 GNU 参数的 diff 工具，并期望外置程序能够返回成功的错误代码。对于大多数可用的 diff 程序，只有第 6、7 参数，diff 两边文件的路径。需要注意 Subversion 对于每个修改的文件都要以异步方式（或“后台”）运行 diff 程序，你会得到许多并行的实例。最后，Subversion 期望你的程序在发现区别时返回错误代码 1，没有区别则返回 0—任何其他的返回值都被认为是严重错误。 [48]

例 7.2 “diffwrap.sh”和例 7.3 “diffwrap.bat”分别是 Bourne shell 和 Windows 批处理外置 diff 工具的包裹器模版。

例 7.2. diffwrap.sh

```
#!/bin/sh

# Configure your favorite diff program here.
DIFF="/usr/local/bin/my-diff-tool"

# Subversion provides the paths we need as the sixth and seventh
# parameters.
LEFT=${6}
RIGHT=${7}

# Call the diff command (change the following line to make sense for
```

```
# your merge program).

$DIFF --left $LEFT --right $RIGHT



# Return an errorcode of 0 if no differences were detected,
1 if some were.

# Any other errorcode will be treated as fatal.
```

例 7.3. diffwrap.bat

```
@ECHO OFF



REM Configure your favorite diff program here.

SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"



REM Subversion provides the paths we need as the sixth and
seventh

REM parameters.

SET LEFT=%6

SET RIGHT=%7



REM Call the diff command (change the following line to
make sense for

REM your merge program).

%DIFF% --left %LEFT% --right %RIGHT%



REM Return an errorcode of 0 if no differences were
detected, 1 if some were.
```

```
REM Any other errorcode will be treated as fatal.
```

外置 diff3

Subversion 按照符合 GNU 的 diff3 的参数调用合并程序，期望外置程序会返回成功的错误代码，并且完整合并的文件结果打印到标准输出（这样 Subversion 可以重定向这些东西到适当的版本控制下的文件）。对于大多数可选的合并程序，只有第 9、10 和 11 参数，分别代表“mine”、“older”和“yours”的路径。需要注意，因为 Subversion 依赖于你的合并程序的输出，你的包裹脚本在输出发送到 Subversion 之前不要退出。当最终退出，如果合并成功返回 0，如果有为解决的冲突则返回 1—其它返回值都是严重错误。

例 7.4 “diff3wrap.sh”和例 7.5 “diff3wrap.bat”分别是 Bourne shell 和 Windows 批处理外置 diff 工具的包裹器模版。

例 7.4. diff3wrap.sh

```
#!/bin/sh


# Configure your favorite diff3/merge program here.
DIFF3="/usr/local/bin/my-merge-tool"


# Subversion provides the paths we need as the ninth, tenth,
and eleventh
# parameters.
```

```
MINE=${9}

OLDER=${10}

YOURS=${11}



# Call the merge command (change the following line to make
sense for

# your merge program).

$DIFF3 --older $OLDER --mine $MINE --yours $YOURS



# After performing the merge, this script needs to print
the contents

# of the merged file to stdout.  Do that in whatever way
you see fit.

# Return an errorcode of 0 on successful merge, 1 if
unresolved conflicts

# remain in the result.  Any other errorcode will be treated
as fatal.
```

例 7.5. diff3wrap.bat

```
@ECHO OFF



REM Configure your favorite diff3/merge program here.

SET DIFF3="C:\Program Files\Funky Stuff\My Merge
Tool.exe"



REM Subversion provides the paths we need as the ninth,
tenth, and eleventh
```

```
REM parameters.  But we only have access to nine parameters
at a time, so we

REM shift our nine-parameter window twice to let us get
to what we need.

SHIFT

SHIFT

SET MINE=%7

SET OLDER=%8

SET YOURS=%9



REM Call the merge command (change the following line to
make sense for

REM your merge program).

%DIFF3% --older %OLDER% --mine %MINE% --yours %YOURS%



REM After performing the merge, this script needs to print
the contents

REM of the merged file to stdout.  Do that in whatever way
you see fit.

REM Return an errorcode of 0 on successful merge, 1 if
unresolved conflicts

REM remain in the result.  Any other errorcode will be
treated as fatal.
```

---

[45] APPDATA 环境变量指向 *Application Data* 目录，所以你可以通过*%APPDATA%\Subversion* 引用用户配置区目录。

[46] 就是一个大杂烩？

[47] Subversion 的开发者很好，但最好的也会发生错误。

[48] GNU 的 diff 手册这样说的：“返回 0 意味着没有区别，1 是有有区别，其它值意味着出现问题。”

嵌入 Subversion

目录

Subversion 有一个模块化的设计，以库的形式由 C 编写和实现。每个库都有一个定义良好的目的和 API，而且这些接口不仅仅为了 Subversion 本身使用，也可以为任何希望嵌入编程方式控制 Subversion 的软件。此外，Subversion 的 API 不仅仅可以为 C 程序使用，也可以使用如 Ptyhon、Perl、Java 或 Ruby 等高级语言调用。

本章是为那些希望编写代码或其他语言绑定与 Subversion 交互的人准备的。如果你围绕 Subversion 功能编写健壮的脚本来简化你的生活，设法开发 Subversion 与其他软件的复杂集成，或者只是对 Subversion 不同库模块提供功能感兴趣，这一章是为你准备的。然而，如果你不能预见你会以此种程度参与 Subversion，你可以放心的跳过本章，略过本章不会影响你对 Subversion 使用的体验。

分层的库设计

每个 Subversion 核心模块都属于三层中的某一层—版本库层、版本库访问（RA）层或是客户端层（见图 1 “Subversion 的架构”）。我们很快就会考察这些层，但首先让我们看一下 Subversion 库的摘要目录，为了一致性，我们将通过它们的无扩展 Unix 库名（例如 libsvn_fs、libsvn_wc 和 mod_dav_svn）来引用它们。

libsvn_client

客户端程序的主要接口

libsvn_delta

目录树和文本区别程序

libsvn_diff

上下文区别和合并例程

libsvn_fs

Subversion 文件系统库和模块加载器

libsvn_fs_base

Berkeley DB 文件系统后端

libsvn_fs_fs

本地文件系统（FSFS）后端

libsvn_ra

版本库访问通用组件和模块装载器

libsvn_ra_dav

WebDAV 版本库访问模块

libsvn_ra_local

本地版本库访问模块

libsvn_ra_serf

另一个(实验性的) WebDAV 版本库访问模块

libsvn_ra_svn

一个自定义版本库访问模块

libsvn_repos

版本库接口

libsvn_subr

各色各样的有用的子程序

libsvn_wc

工作拷贝管理库

mod_authz_svn

使用 WebDAV 访问 Subversion 版本库的 Apache 授权模块

mod_dav_svn

影射 WebDAV 操作为 Subversion 操作的 Apache 模块

单词“各色各样的（miscellaneous）”只在列表中出现过一次是一个好的迹象。Subversion 开发团队非常注意将功能归入合适的层和库，或许模块化设计最大的好处就是从开发者的角度看减少了复杂性。作为一个开发者，你可以很快就描画出一副“大图像”，以便于你更精确地，也相对容易地找出某一功能所在的位置。

模块化的另一个好处是我们有能力去构造一个全新的，能够完全实现相同 API 功能的库，以替换整个给定的模块，而又不会影响基础代码。在某种意义上，Subversion 已经这样做了。libsvn_ra_dav、libsvn_ra_local、libsvn_ra_serf 和 libsvn_ra_svn all 都实现了相同的接口，均与版本库层进行通讯—libsvn_ra_loca 与版本库直接连接其他几个则通过网络。

libsvn_fs_base 和 libsvn_fs_fs 库是另外一对以不同方式实现相同功能的库—都是可以与 libsvn_fs 库连接。

客户端本身也得益于 Subversion 设计的模块化，Subversion 的 libsvn_client 库提供了设计一个 Subversion 工作客户端（见“客户端层”一节）的绝大多数功能。所以尽管 Subversion 的发布版只有 **svn** 命令行客户端程序，依然有许多第三方的程序提供了各种形式的图形化客户端 UI。这些 GUI 使用的 API 与命令行客户端完全相同。模块化类型的 API 的促使了大量 Subversion 客户端和 IDE 集成插件使用 Subversion 本身。

版本库层

当提到 Subversion 版本库层时，我们通常会讨论两个基本概念—版本化文件系统实现（通过 libsvn_fs 访问，libsvn_fs_base 和 libsvn_fs_fs 支持），和包装在外的（以 libsvn_repos 实现）版本库逻辑。这些库提供了版本控制数据的存储和报告机制，这些层通过版本库访问层连接客户端层，从 Subversion 用户的角度，这些事情在整个过程的另一端。

Subversion 文件系统通过 libsvn_fs API 来访问，它并不是一个安装在操作系统之上的内核级的文件系统（例如 Linux ext2 或 NTFS），而是一个虚拟文件系统。它并未将“文件”和“目录”保存为真实的文件和目录（也就是用你熟知的 shell 程序可以浏览的那种），而是采用了一种抽象的后端存储方式，这个后端存储方式有两种—一个是 Berkeley

DB 数据库环境，另一个是普通文件表示。（要了解更多关于版本库后端的信息，请看“选择数据存储格式”一节）。除此之外，开发社区也非常有兴趣考虑在Subversion的未来版本 中提供某种使用其它后端数据库系统的能力，也许是开放式数据库连接（ODBC）的机制。实际上，Google 在 2006 中期启动 Google Code 主机服务项目之前做了一些类似的事情，它的部分开源项目组成员编写了新的Subversion文件系统，使用了他们的扩展性极好的 Bigtable 数据存储。

libsvn_fs 支持的文件系统 API 包含了所有其他文件系统的功能：你可以创建和删除文件和目录、拷贝和移动、修改文件内容等等。它也包含了一些不太常用的特性，如对任意文件和目录添加、修改和删除元数据（“properties”）的能力。此外，Subversion 文件系统是一个版本化的文件系统，意味着你修改你的目录树时，Subversion 会记住修改以前的样子。也可以回到所有初始化版本库之后（且仅仅之后）的版本。

所有你对目录树的修改包含在 Subversion 事务的上下文中，下面描述了修改文件系统的例程：

1. 开始 Subversion 的提交事务。
2. 作出修改（添加、删除、属性修改等等。）。
3. 提交事务。

一旦你提交了你的事务，你的文件系统修改就会永久的作为历史保存起来，每个这样的周期会产生一个新的树，所有的修订版本都是永远可以访问的一个不变的快照。

事务的其它信息

Subversion 的事务概念，特别是在 libsvn_fs 中的数据库附近的代码，很容易与低层提供支持的数据库事务混淆。两种类型事务都提供了原子和隔离操作，换句话说，事务给你能力可以用“全部或者没有”样式执行一系列的动作—所有的动作都完全成功，或者是所有的*没有*发生—而且不会干扰别人操作数据。

数据库事务通常围绕着一些对数据库本身的数据修改相关的小操作（如修改表行的内容），Subversion 是更大范围的事务，围绕着一些高一级的操作，如下一个修订版本文件系统的一组文件和目录的修改。如果这还不是很混乱，考虑这个：Subversion 在创建 Subversion 事务（所以如果 Subversion 创建事务失败，数据库会看起来我们从来没有尝试创建）时会使用一个数据库事务！

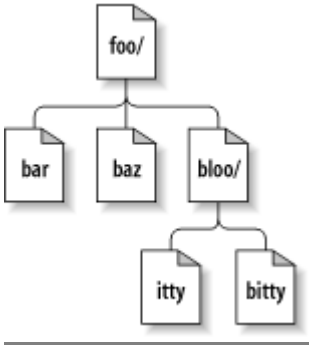
很幸运的是用户的文件系统 API，数据库提供的事务支持本身几乎完全从外表隐藏（也是一个完全模块化的模式所应该的）。只有当你开始研究文件系统本身的实现时，这些事情才可见（或者是开始感兴趣）。

大多数文件系统接口提供的功能作为一个动作发生在一个文件系统路径上，也就是，从文件系统的外部，描述和访问文件和目录独立版本的

主要机制是经过如*/foo/bar* 的路径，就像你在喜欢的 shell 程序中定位文件和目录。你通过传递它们的路径到相应的 API 功能来添加新的文件和目录，查询这些信息也是同样的机制。

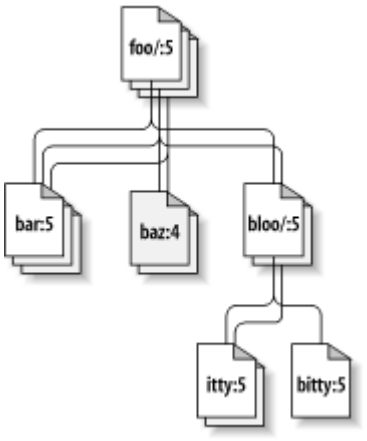
然而，不像大多数文件系统，一个单独的路径不足以在 Subversion 定位一个文件或目录，可以把目录树看作一个二维的系统，一个节点的兄弟代表了一种从左到右的动作，并且递减到子目录是一个向下的动作，图 8.1 “二维的文件和目录”展示了一个典型的树的形式。

图 8.1. 二维的文件和目录



当然，Subversion 文件系统有一个其它文件系统所没有的第三维—时间！[49]在一个文件系统接口，几乎所有的功能都有个*路径（path）*参数，也期望一个 *root* 参数。`svn_fs_root_t` 参数不仅描述了一个修订版本或一个 Subversion 事务（通常正好是一个修订版本），而且提供了用来区分修订版本 32 的*/foo/bar* 和修订版本 98 在同样路径的三维上下文环境。图 8.2 “版本时间—第三维！”展示了修订版本历史作为添加的纬度进入到 Subversion 文件系统领域。

图 8.2. 版本时间—第三维！



像之前我们提到的，libsvn_fs 的 API 感觉像是其它文件系统，只是有一个美妙的版本化能力。它设计为为所有对版本化的文件系统有兴趣的程序使用，不是巧合，Subversion 本身也对这个功能很有兴趣。但是虽然文件系统 API 一定必须对基本的文件和目录版本化提供足够的支持，Subversion 需要的更多—这是 libsvn_repos 到来的地方。

Subversion 版本库库（libsvn_repos）建立在（逻辑上讲）libsvn_fs 的 API 之上，不仅仅提供了版本化文件系统的功能，它没有包裹所有的文件系统功能—只有文件系统常规周期中的主要事件使用版本库接口包裹，如包括 Subversion 事务的创建和提交，修订版本属性的修改。这些特别的事件使用版本库库包裹是因为它们有一些关联的钩子。版本库钩子系统并没有与与版本化文件系统的紧密关联，所以它们存在于版本库的包裹库。

钩子机制需求是从文件系统代码的其它部分中抽象出单独的版本库库的一个原因，libsvn_repos 的 API 提供了许多其他有用的工具，它们可以做到：

- 在 Subversion 版本库和版本库包括的文件系统的上创建、打开、销毁和执行恢复步骤。
- 描述两个文件系统树的区别。
- 关于所有（或者部分）修订版本中的文件系统中的一组文件的提交日志信息的查询
- 产生可读的文件系统“导出”，一个文件系统修订版本的完整展现。
- 解析导出格式，加载导出的版本到一个不同的 Subversion 版本库。

伴随着 Subversion 的发展，版本库库会随着文件系统提供更多的功能和配置选项而不断成长。

版本库访问层

如果说 Subversion 版本库层是在“这条线的另一端”，那版本库访问层就是这条线。负责在客户端库和版本库之间编码数据，这一层包括 libsvn_ra 模块加载模块，RA 模块本身（现在包括了 libsvn_ra_dav、libsvn_ra_local、libsvn_ra_serf 和 libsvn_ra_svn），和所有一个或多个 RA 模块需要的附加库，例如与 Apache 模块 mod_dav_svn 通讯的 libsvn_ra_dav 或者是 libsvn_ra_svn 的服务器，**`svnserve`**。

因为 Subversion 使用 URL 来识别版本库资源，URL 模式的协议部分（通常是 `file:`、`http:`、`https:`或 `svn:`）用来监测那个 RA 模块用来处理通讯。每个模块注册一组它们知道如何“说话”的协议，所以 RA 加载器

可以在运行中监测在手边的任务中使用哪个模块。通过运行 **svn --version**，你可以监测 Subversion 命令行客户端所支持的 RA 模块和它们声明支持的协议：

```
$ svn --version
svn, version 1.4.3 (r23084)
   compiled Jan 18 2007, 07:47:40

Copyright (C) 2000-2006 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/).

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' scheme
  - handles 'https' scheme
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' scheme
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' scheme

$
```

RA 层导出的 API 包含了发送和接收版本化数据的必要功能，并且每一个存在的 RA 插件可以使用特定协议执行任务—libsvn_ra_dav 同配置了 mod_dav_svn 模块的 Apache HTTP 服务器使用 HTTP/WebDAV（可选 SSL 加密）通讯，libsvn_ra_svn 同 **`svnserve`** 使用自定义网络协议通讯。

对那些一直希望使用另一个协议来访问 Subversion 版本库的人，正好是为什么版本库访问层是模块化的！开发者可以简单的编写一个新的库来在一侧实现 RA 接口并且与另一侧的版本库通讯。你的新库可以使用存在的网络协议，或者发明你自己的。你可以使用进程间的通讯调用，或者—让我们发狂，我们会吗？—你甚至可以实现一个电子邮件为基础的协议，Subversion 提供了 API，你提供创造性。

客户端层

在客户端这一面，Subversion 工作拷贝是所有动作发生的地方。大多数客户端库实现的功能是为了管理工作拷贝的目的实现的—满是文件子目录的目录是一个或多个版本库位置的可编辑的本地“影射”—从版本库访问层来回传递修改。

Subversion 的工作拷贝库，libsvn_wc 直接负责管理工作拷贝的数据，为了完成这一点，库会在工作拷贝的每个目录的特殊子目录中保存关于工作拷贝的管理性信息。这个子目录叫做*.svn*，出现在所有工作拷贝目录里，保存了各种记录了状态和用来在私有工作区工作的文件和目录。

对那些熟悉 CVS 的用户，*.svn* 子目录与 *CVS* 工作拷贝管理目录的作用类似，关于*.svn* 管理区域的更多信息，见本章的“进入工作拷贝的管理区”一节。

Subversion 客户端库 libsvn_client 具备最广泛的职责；它的工作是结合工作拷贝库和版本库访问库的功能，然后为希望普通版本控制的应用提供最高级的 API。举个例子，`svn_client_checkout()`方法是用一个 URL 作为参数，传递这个 URL 到 RA 层然后在特定版本库打开一个会话。然后向版本库要求一个特定的目录树，然后把目录树发送给工作拷贝库，然后把完全的工作拷贝写到磁盘（*.svn* 目录和一切）。

客户端库是为任何程序使用设计的，尽管 Subversion 的源代码包括了一个标准的命令行客户端，用客户端库编写 GUI 客户端也是很简单，Subversion 新的 GUI（或者任何新的客户端，真的）不需要紧密围绕包含的命令行客户端—他们对具有相同功能、数据和回调机制的 libsvn_client 的 API 有完全的访问权利。事实上，Subversion 源代码中包含了一段 C 程序（可以在 *tools/examples/minimal_client.c*）例子，演示了如何利用 Subversion 客户端创建简单的客户端程序。

直接绑定—关于正确性

为什么 GUI 程序要直接访问 libsvn_client 而不以命令行客户端的包裹运行？除了效率以外，这也关系到潜在的正确性问题。一个命令行客户端程序（如 Subversion 提供的）如果绑定了客户端库，需要将反馈和请

求数据字节从 C 翻译为可读的输出，这种翻译是有损耗的，程序不能得到 API 所提供的所有信息，或者是得到紧凑的信息。

如果你已经包裹了这样一个命令行程序，第二个程序只能访问已经经过解释的（如我们提到的，不完全）信息，需要*再次*转化为*它本身的*展示格式。由于各层的包裹，原始数据的完整性越来越难以保证，结果很像对喜欢的录音带或录像带反复的拷贝（一个拷贝…）。

但是关于直接绑定 API 使用，而不是包裹程序，这是 Subversion 项目对其 API 兼容性的承诺。在小版本的变化（如从 1.3 到 1.4）中 API 的不会有函数原形的改变，简单来说就是你不需要将你程序源代码升级，因为你只是升级到了一个新版本的 Subversion。某些方法可能会被废弃，但依然工作，这给你了缓冲时间来最终适应新 API。Subversion 的命令行输出没有这种兼容性承诺，可能会在每个版本更改。

进入工作拷贝的管理区

像我们前面提到的，每个 Subversion 工作拷贝包含了一个特别的子目录叫做*.svn*，这个目录包含了关于工作拷贝目录的管理数据，Subversion 使用*.svn* 中的信息来追踪如下的数据：

- 工作拷贝中展示的目录和文件在版本库中的位置。
- 工作拷贝中当前展示的文件和目录的修订版本。
- 所有附加在文件和目录上的用户定义属性。
- 初始（未编辑）的工作拷贝文件的拷贝。

Subversion 工作拷贝管理区域的布局和内容主要是考虑的实现细节，不是被人来使用的。开发者被鼓励使用 Subversion 的 API 或工具来访问和处理工作拷贝数据，反对直接读写操作组成工作拷贝管理区域的文件。工作拷贝中管理数据采用的文件格式会不断改变—只是公共 API 成功的隐藏了这种改变。在本小节，我们将会探讨一些实现细节来安抚你们的焦虑。

条目文件

或许*.svn* 目录中最重要的单个文件就是 *entries* 了，这个条目文件是一个 XML 文档，包含了关于工作拷贝中的版本化的资源的大多数管理性信息，这个文件保留了版本库 URL、原始修订版本、文件校验数据、可知的最后提交信息（作者、修订版本和时间戳）和本地拷贝历史—实际上是 Subversion 客户端关于一个版本化（或者是将要版本化的）资源的所有感兴趣的信息！

熟悉 CVS 管理目录的人可能会发现，Subversion 的*.svn/entries* 实现了 CVS 的 *CVS/Entries*、*CVS/Root* 和 *CVS/Repository* 的功能。

.svn/entries 的格式曾经多次修改，最初是 XML 文件，现在使用自定义的—尽管依然是可读的文件格式。早期的 Subversion 需要频繁调试文件内容，所以选择了 XML 这种格式，随着 Subversion 的成熟，频繁调试的需求消失了，而产生了用户对性能的要求。当然，Subversion 的工作拷贝库可以从一种格式自动升级到另一种格式—按照老格式读取，然

后按照新格式写—避免了重新检出工作拷贝，但是也造成了不同版本 Subversion 程序访问同一份工作拷贝的复杂情形。

原始拷贝和属性文件

如我们前面提到的，*.svn* 也包含了一些原始的“text-base”文件版本，可以在*.svn/text-base* 看到。这些原始文件的好处是多方面的—察看本地修改和区别不需要经过网络访问，减少传递修改时的数据—但是随之而来的代价是每个版本化的文件都在磁盘至少保存两次，现在看来这是对大多数文件可以忽略不计的一个惩罚。但是，当你版本控制的文件增多之后形势会变得很严峻，我们已经注意到了应该可以选择使用“text-base”，但是具有讽刺意味的是，当版本化文件增大时，“text-base”文件的存在会更加重要—谁会希望在提交一个小修改时在网络上传递一个大文件？

同“text-base”文件的用途一样的还有属性文件和它们的“prop-base”拷贝，分别位于*.svn/props* 和*.svn/prop-base*。因为目录也有属性，所以也有*.svn/dir-props* 和*.svn/dir-prop-base* 文件。

使用 API

使用 Subversion 库 API 开发应用看起来相当的直接，所有的公共头文件放在源文件的 *subversion/include* 目录，从源代码编译和安装 Subversion 本身，需要这些头文件拷贝到系统位置。这些头文件包括了所有用户和 Subversion 库可以访问的功能和类型。Subversion 开发者

社区仔细的确保所有的公共 API 有完好的文档—直接引用头文件的文档。

你首先应该注意 Subversion 的数据类型和方法是命名空间保护的，每一个公共 Subversion 对象名以 svn_开头，然后紧跟一个这个对象定义（如 wc、client 和 fs 等等）所在的库的简短编码，然后是一个下划线（_）和后面的对象名称。半公开的方法（库使用，但是但库之外代码不可以使用并且只可以在库自己的目录看到）与这个命名模式不同，并不是库代码之后紧跟一个下划线，他们是用两个下划线（__）。给定源文件的私有方法没有特殊前缀，使用 static 声明。当然，一个编译器不会关心命名习惯，只是用来区分给定方法或数据类型的应用范围。

关于 Subversion 的 API 编程的另一个好的资源是 hacking 指南，可以在 http://subversion.tigris.org/hacking.html 找到，这个文档包含了有用的信息，同时满足 Subversion 本身的开发者和将 Subversion 作为第三方库的开发者。[50]

Apache 可移植运行库

伴随 Subversion 自己的数据类型，你会看到许多 apr 开头的数据类型引用—来自 Apache 可移植运行库（APR）的对象。APR 是 Apache 可移植运行库，源自为了服务器代码的多平台性，尝试将不同的操作系统特定字节与操作系统无关代码隔离。结果就提供了一个基础 API 的库，只有一些适度区别—或者是广泛的—来自各个操作系统。Apache HTTP

服务器很明显是 APR 库的第一个用户，Subversion 开发者立刻发现了使用 APR 库的价值。意味着 Subversion 没有操作系统特定的代码，也意味着 Subversion 客户端可以在 Server 存在的平台编译和运行。当前这个列表包括，各种类型的 Unix、Win32、OS/2 和 Mac OS X。

除了提供了跨平台一致的系统调用，[51]APR 给 Subversion 对多种数据类型有快速的访问，如动态数组和哈希表。Subversion 在代码中广泛使用这些类型，但是 Subversion 的 API 原型中最常见的 APR 类型是 `apr_pool_t`—APR 内存池，Subversion 使用内部缓冲池用来进行内存分配（除非外部库在 API 传递参数时需要一个不同的内存管理模式），[52]而且一个人如果针对 Subversion 的 API 编码不需要做同样的事情，他们可以在需要时给 API 提供缓冲池，这意味着 Subversion 的 API 使用者也必须链接到 APR，必须调用 `apr_initialize()`来初始化 APR 子系统，而且在使用 Subversion API 时必须创建和管理池，通常是使用 `svn_pool_create()`、`svn_pool_clear()`和 `svn_pool_destroy()`。

使用内存池编程

几乎每一个使用过 C 语言的开发者曾经感叹令人畏缩的内存管理，分配足够的内存，并且追踪内存的分配，在不需要时释放内存—这个任务会非常复杂。当然，如果没有正确地做到这一点会导致程序毁掉自己，或者更加严重一点，把电脑搞瘫。

另一方面高级语言使开发者完全摆脱了内存管理，[53]Java 和 Python 之类的语言使用*垃圾收集*原理，在需要的时候分配对象内存，在不使用时进行清理。

APR 提供了一种叫做池基础的中等的内存管理方法，允许开发者以一种低分辨率的方式控制内存—每块（或池“pool”）的内存，而不是每个对象。不是使用 malloc()和其他按照对象分配内存的方式，你要求 APR 从内存创建一段内存池，当你结束使用在池中创建的对象，你销毁池，可以有效地取消其中的对象消耗的内存。通过池，你不需要跟踪每个对象的内存释放，你的程序只需要跟踪这些对象，将对象分配到池中，而池的生命周期（池的创建和删除之间的时间）满足所有对象的需要。

URL 和路径需求

因为分布式版本控制操作是 Subversion 存在的重点，有意义来关注一下国际化（i18n）支持。毕竟，当“分布式”或许意味着“横跨办公室”，它也意味着“横跨全球”。为了更容易一点，Subversion 的所有公共接口只接受路径参数,这些参数是传统的，使用 UTF-8 编码。这意味着，举个例子，任何新的使用 libsvn_client 接口客户端库，在把这些参数传递给 Subversion 库前，需要首先将路径从本地代码转化为 UTF-8 代码，然后将 Subversion 传递回来的路径转换为本地代码，很幸运，Subversion 提供了一组任何程序可以使用的转化方法（见 *subversion/include/svn_utf.h*）。

同样，Subversion 的 API 需要所有的 URL 参数是正确的 URI 编码，所以，我们不会传递 file:///home/username/My File.txt 作为 My File.txt 的 URL，而要传递 file:///home/username/My%20File.txt。再次，Subversion 提供了一些你可以使用的助手方法—svn_path_uri_encode() 和 svn_path_uri_decode()，分别用来 URI 的编码和解码。

使用 C 和 C++ 以外的语言

除 C 语言以外，如果你对使用其他语言结合 Subversion 库感兴趣—如 Python 脚本或是 Java 应用—Subversion 通过简单包裹生成器（SWIG）提供了最初的支持。Subversion 的 SWIG 绑定位于 *subversion/bindings/swig*，并且慢慢的走向成熟进入可用状态。这个绑定允许你直接调用 Subversion 的 API 方法，使用包裹器会把脚本数据类型转化为 Subversion 需要的 C 语言库类型。

非常不幸，Subversion 的语言绑定缺乏对核心 Subversion 模块的关注，但是，花了很多力气处理创建针对 Python、Perl 和 Ruby 的功能绑定，在一定程度上，在这些接口上的工作量可以在其他语言的 SWIG（包括 C#、Guile、Java、MzScheme、OCaml、PHP、Tcl 等等）接口上得到重用。然而，为了完成复杂的 API，一些 SWIG 接口仍然需要额外的编程工作，关于 SWIG 本身的更多信息可以看项目的网站 http://www.swig.org/。

Subversion 也有 Java 的语言绑定，JavaJL 绑定（位于 Subversion 源目录树的 *subversion/bindings/java*）不是基于 SWIG 的，而是 javah 和手写 JNI 的混合，JavaHL 几乎覆盖 Subversion 客户端的 API，目标是作为 Java 基础的 Subversion 客户端和集成 IDE 的实现。

Subversion 的语言绑定缺乏 Subversion 核心模块的关注，但是通常可以作为一个产品信赖。大量脚本、应用、Subversion 的 GUI 客户端和其他第三方工具现在已经成功地运用了 Subversion 语言绑定来完成 Subversion 的集成。

这里使用其它语言的方法来与 Subversion 交互没有任何意义：Subversion 开发社区没有提供其他的绑定，你可以在 Subversion 项目链接页里（http://subversion.tigris.org/links.html）找到其他绑定的链接，但是有一些流行的绑定我觉得应该特别留意。首先是 Python 的流行绑定，Barry Scott 的 PySVN（http://pysvn.tigris.org/）。PySVN 鼓吹它们提供了更多 Python 样式的接口，而不像 Subversion 自己的 Python 绑定的 C 样式接口。对于希望寻求 Subversion 纯 Java 实现的人，可以看看 SVNKit（http://svnkit.com/），也就是从头使用 Java 编写的 Subversion。你必须要小心，SVNKit 没有采用 Subversion 的核心库，其行为方式没有确保与 Subversion 匹配。

代码样例

例 8.1 “使用版本库层”包含了一段 C 代码（C 编写）描述了我们讨论的概念，它使用了版本库和文件系统接口（可以通过方法名 svn_repos_ 和 svn_fs_ 分辨）创建了一个添加目录的修订版本。你可以看到 APR 库的使用，为了内存分配而传递，这些代码也揭开了一些关于 Subversion 错误处理的晦涩事实—所有的 Subversion 错误必须需要明确的处理以防止内存泄露（在某些情况下，应用失败）。

例 8.1. 使用版本库层

```
/* Convert a Subversion error into a simple boolean error
code.
 *
 * NOTE:  Subversion errors must be cleared (using
svn_error_clear())
 *       because they are allocated from the global pool,
else memory
 *       leaking occurs.
 */
#define INT_ERR(expr)                                   \
  do {                                                  \
    svn_error_t *__temperr = (expr);                    \
    if (__temperr)                                      \
      {                                                 \
        svn_error_clear(__temperr);                     \
        return 1;                                       \
      }                                                 \
    return 0;                                           \
```

```
  } while (0)


/* Create a new directory at the path NEW_DIRECTORY in the
Subversion
 * repository located at REPOS_PATH.  Perform all memory
allocation in
 * POOL.  This function will create a new revision for the
addition of
 * NEW_DIRECTORY.  Return zero if the operation completes
 * successfully, non-zero otherwise.
 */
static int
make_new_directory(const char *repos_path,
                   const char *new_directory,
                   apr_pool_t *pool)
{
  svn_error_t *err;
  svn_repos_t *repos;
  svn_fs_t *fs;
  svn_revnum_t youngest_rev;
  svn_fs_txn_t *txn;
  svn_fs_root_t *txn_root;
  const char *conflict_str;


  /* Open the repository located at REPOS_PATH.
   */
```

```
  INT ERR(svn repos open(&repos, repos path, pool));



  /* Get a pointer to the filesystem object that is stored
in REPOS.

   */

  fs = svn repos fs(repos);



  /* Ask the filesystem to tell us the youngest revision
that

   * currently exists.

   */

  INT ERR(svn fs youngest rev(&youngest rev, fs,
pool));



  /* Begin a new transaction that is based on YOUNGEST REV.
We are

   * less likely to have our later commit rejected as
conflicting if we

   * always try to make our changes against a copy of the
latest snapshot

   * of the filesystem tree.

   */

  INT ERR(svn fs begin txn(&txn, fs, youngest rev,
pool));



  /* Now that we have started a new Subversion transaction,
get a root

   * object that represents that transaction.

   */
```

```
  INT ERR(svn fs txn root(&txn root, txn, pool));

__

  /* Create our new directory under the transaction root,
at the path

   * NEW DIRECTORY.

   */

  INT ERR(svn fs make dir(txn root, new directory,
pool));



  /* Commit the transaction, creating a new revision of
the filesystem

   * which includes our added directory path.

   */

  err = svn repos fs commit txn(&conflict str, repos,

                                &youngest rev, txn, pool);

  if (! err)

    {

      /* No error? Excellent! Print a brief report of our
success.

       */

      printf("Directory '%s' was successfully added as new
revision "

             "'%ld'.\n", new directory, youngest rev);

    }

  else if (err->apr err == SVN ERR FS CONFLICT)

    {

      /* Uh-oh. Our commit failed as the result of a
conflict
```

```
      * (someone else seems to have made changes to the
same area
      * of the filesystem that we tried to modify).  Print
an error
      * message.
      */
     printf("A conflict occurred at path '%s' while
attempting "
            "to add directory '%s' to the repository at
'%s'.\n",
            conflict str, new directory, repos path);
    }
  else
    {
     /* Some other error has occurred.  Print an error
message.
      */
     printf("An error occurred while attempting to add
directory '%s' "
            "to the repository at '%s'.\n",
            new directory, repos path);
    }

  INT ERR(err);
}
```

请注意在例 8.1 “使用版本库层”中，代码可以非常容易使用 svn_fs_commit_txn()提交事务。但是文件系统的 API 对版本库库的钩子一无所知，如果你希望你的 Subversion 版本库在每次提交一个事务时自动执行一些非 Subversion 的任务（例如，给开发者邮件组发送一个描述事务修改的邮件），你需要使用 libsvn_repos 包裹的功能版本—这个功能会实际上首先运行一个如果存在的 pre-commit 钩子脚本，然后提交事务，最后会运行一个 post-commit 钩子脚本。钩子提供了一种特别的报告机制，不是真的属于核心文件系统库本身。（关于 Subversion 版本库钩子的更多信息，见“实现版本库钩子”一节。）

现在我们转换一下语言，例 8.2 “使用 Python 处理版本库层”使用 Subversion SWIG 的 Python 绑定实现了从版本库取得最新的版本，并且打印了取出时访问的目录。

例 8.2. 使用 Python 处理版本库层

```
#!/usr/bin/python


"""Crawl a repository, printing versioned object path
names."""


import sys

import os.path

import svn.fs, svn.core, svn.repos


def crawl_filesystem_dir(root, directory):
```

```
    """Recursively crawl DIRECTORY under ROOT in the
filesystem, and return

    a list of all the paths at or below DIRECTORY."""



    # Print the name of this path.

    print directory + "/"

    

    # Get the directory entries for DIRECTORY.

    entries = svn.fs.svn fs dir entries(root, directory)



    # Loop over the entries.

    names = entries.keys()

    for name in names:

        # Calculate the entry's full path.

        full path = directory + '/' + name



        # If the entry is a directory, recurse.  The
recursion will return

        # a list with the entry and all its children, which
we will add to

        # our running list of paths.

        if svn.fs.svn fs is dir(root, full path):

            crawl filesystem dir(root, full path)

        else:

            # Else it's a file, so print its path here.

            print full path
```

```
def crawl_youngest(repos_path):
    """Open the repository at REPOS_PATH, and recursively
crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference
to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj)

    # Open a root object representing the youngest (HEAD)
revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj,
youngest_rev)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "")

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
```

```
                        % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize the repository path.
    repos_path =
svn.core.svn_path_canonicalize(sys.argv[1])

    # Do the real work.
    crawl_youngest(repos_path)
```

同样的 C 程序需要处理 APR 内存池系统，但是 Python 自己处理内存，Subversion 的 Python 绑定也遵循这种习惯。在 C 语言中，为表示路径和条目的 hash 需要处理自定义的数据类型（例如 APR 提供的库），但是 Python 有 hash（叫做“dictionaries”），并且是内置数据类型，而且还提供了一系列操作这些类型的函数，所以 SWIG（通过 Subversion 的语言绑定层的自定义帮助）要小心的将这些自定义数据类型映射到目标语言的数据类型，这为目标语言的用户提供了一个更加直观的接口。

Subversion 的 Python 绑定也可以用来进行工作拷贝的操作，在本章前面的小节中，我们提到过 *libsvn_client* 接口，它存在的目的就是简化编写 Subversion 客户端的难度，例 8.3 “一个 Python 状态爬虫”是一个例子，讲的是如何使用 SWIG 绑定创建一个扩展版本的 **svn status** 命令。

例 8.3. 一个 Python 状态爬虫

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


"""Crawl a working copy directory, printing status
information."""


import sys

import os.path

import getopt

import svn.core, svn.client, svn.wc


def generate status code(status):

    """Translate a status value into a single-character
status code,

    using the same logic as the Subversion command-line
client."""

    code map = { svn.wc.svn wc status none       : ' ',

                 svn.wc.svn wc status normal     : ' ',

                 svn.wc.svn wc status added      : 'A',

                 svn.wc.svn wc status missing    : '!',

                 svn.wc.svn wc status incomplete  : '!',

                 svn.wc.svn wc status deleted    : 'D',

                 svn.wc.svn wc status replaced   : 'R',

                 svn.wc.svn wc status modified   : 'M',

                 svn.wc.svn wc status merged     : 'G',

                 svn.wc.svn wc status conflicted  : 'C',
```

```
                svn.wc.svn wc status obstructed  : '~',
                svn.wc.svn wc status ignored     : 'I',
                svn.wc.svn wc status external    : 'X',
                svn.wc.svn wc status unversioned : '?',
            }
    return code map.get(status, '?')


def do status(wc path, verbose):
    # Calculate the length of the input working copy path.
    wc path len = len(wc path)


    # Build a client context baton.
    ctx = svn.client.svn client ctx t()


    def  status callback(path, status,
root path len=wc path len):
        """A callback function for svn client status."""


        # Print the path, minus the bit that overlaps with
the root of
        # the status crawl
        text status =
generate status code(status.text status)
        prop status =
generate_status_code(status.prop_status)
        print '%s%s  %s' % (text status, prop status,
path[wc path len + 1:])
```

```
_______

    # Do the status crawl, using  status_callback() as our
callback function.

    svn.client.svn_client_status(wc_path, None,
 status_callback,

                                 1, verbose, 0, 0, ctx)



def usage_and_exit(errorcode):

    """Print usage message, and exit with ERRORCODE."""

    stream = errorcode and sys.stderr or sys.stdout

    stream.write("""Usage: %s OPTIONS WC-PATH

Options:

  --help, -h    : Show this usage message

  --verbose, -v : Show all statuses, even uninteresting
ones

""" % (os.path.basename(sys.argv[0])))

    sys.exit(errorcode)

____

if __name__ == '__main__':

    # Parse command-line options.

    try:

        opts, args = getopt.getopt(sys.argv[1:], "hv",
["help", "verbose"])

    except getopt.GetoptError:

        usage_and_exit(1)

    verbose = 0

    for opt, arg in opts:
```

```
        if opt in ("-h", "--help"):
            usage and exit(0)
        if opt in ("-v", "--verbose"):
            verbose = 1
    if len(args) != 1:
        usage and exit(2)

    # Canonicalize the repository path.
    wc_path = svn.core.svn_path_canonicalize(args[0])


    # Do the real work.
    try:
        do_status(wc_path, verbose)
    except svn.core.SubversionException, e:
        sys.stderr.write("Error (%d): %s\n" % (e[1],
e[0]))
        sys.exit(1)
```

就像例 8.2 “使用 Python 处理版本库层”中的例子，这个程序是池自由的，而且最重要的是使用 Python 的数据类型。svn_client_ctx_t() 是欺骗，因为 Subversion 的 API 没有这个方法—这仅仅是 SWIG 自动语言生成中的一点问题（这是对应复杂 C 结构的一种工厂方法）。也需要注意传递给程序的路径（象最后一个）是通过 svn_path_canonicalize()

执行的，因为要防止触发 Subversion 底层 C 库的断言，也就是防止导致程序立刻随意退出。

---

[49] 我们理解这一定会给科幻小说迷带来一个震撼，他们认为时间是*第四*维的，我们要为提出这样一个不同理论的断言而伤害了他们的作出道歉。

[50] 当然，Subversion 使用 Subversion 的 API。

[51] Subversion 使用尽可能多 ANSI 系统调用和数据类型。

[52] Neon 和 Berkeley DB 就是这种库的例子。

[53] 或仅仅是在紧密地程序优化中玩弄什么东西。

Subversion 完全参考

目录

本章是使用 Subversion 的一个完全手册，包括了命令行客户端（**svn**）和它的所有子命令，也有版本库管理程序（**svnadmin** 和 **svnlook**）和它们各自的子命令。

Subversion 命令行客户端：svn

为了使用命令行客户端，只需要输入 **svn** 和它的子命令[54]以及相关的选项或操作的对象—输入的子命令和选项没有特定的顺序，下面使用 **svn status** 的方式都是合法的：

```
$ svn -v status

$ svn status -v
```

```
$ svn status -v myfile
```

你可以在第 2 章 *基本使用*发现更多使用客户端命令的例子，以及“属性”一节中的管理属性的命令。

svn 选项

虽然 Subversion 的子命令有一些不同的选项，但有的选项是全局的—也就是说，每个选项保证是表示同样的事情，而不管是哪个子命令使用的。举个例子，--verbose（-v）一直意味着“冗长输出”，而不管使用它的命令是什么。

--auto-props

开启 auto-props，覆盖 *config* 文件中的 enable-auto-props 指示。

--change (-c) *ARG*

作为引用特定“修改”（也叫做修订版本）的方法，这个选项是“-r ARG-1:ARG”语法上的甜头。

--config-dir *DIR*

指导 Subversion 从指定目录而不是默认位置（用户主目录的.*subversion*）读取配置信息。

--diff-cmd *CMD*

指定用来表示文件区别的外部程序，当 **svn diff** 调用时，会使用 Subversion 的内置区别引擎，默认会提供统一区别输出，如果你希望使用一个外置区别程序，使用--diff-cmd。你可以通过--extensions（本小节后面有更多介绍）把选项传递到区别程序。

--diff3-cmd *CMD*

指定一个外置程序用来合并文件。

--dry-run

检验运行一个命令的效果，但没有实际的修改—可以用在磁盘和版本库。

--editor-cmd *CMD*

指定一个外部程序来编辑日志信息或是属性值。如何设定缺省编辑器见“配置”一节的 editor-cmd 小节。

--encoding *ENC*

告诉 Subversion 你的提交日志信息是通过提供的字符集编码的，缺省时是你的操作系统的本地编码，如果你的提交信息使用其它编码，你一定要指定这个值。

--extensions (-x) *ARGS*

指定一个或多个 Subversion 传递给提供文件区别的外部区别程序的参数，如果你要传递多个参数，你一定能够要用引号（例如，**`svn diff --diff-cmd /usr/bin/diff -x "-b -E"`**）括起所有的参数。这个选项*只有*在使用`--diff-cmd`选项时使用。

`--file` (`-F`) *FILENAME*

为特定子命令使用命名文件的的内容，尽管不同的子命令对这些内容做不同的事情。例如，**`svn commit`** 使用内容作为提交日志，而 **`svn propset`** 使用它作为属性值。

`--force`

强制一个特定的命令或操作运行。Subversion 有一些操作防止你做普通的使用，但是你可以传递 force 选项告诉 Subversion“我知道我做的事情，也知道这样的结果，所以让我做吧”。这个选项在程序上等同于在打开电源的情况下做你自己的电子工作—如果你不知道你在做什么，你很有可能会得到一个威胁的警告。

`--force-log`

将传递给`--message`（`-m`）或者`--file`（`-F`）的可疑参数指定为有效可接受。缺省情况下，如果选项的参数看起来会成为子命令的目标，Subversion 会提出一个错误，例如，你传递一个版本化的文件路径给`--file`（`-F`）选项，Subversion 会认为出了点错误，认为你将目标对象当成了参数，而你并没有提供其它的—未版本化

的文件作为日志信息的文件。为了确认你的意图并且不考虑这类错误，传递--force-log 选项给命令来接受它作为日志信息。

--help (-h 或-?)

如果同一个或多个子命令一起使用，会显示每个子命令内置的帮助文本，如果单独使用，它会显示常规的客户端帮助文本。

--ignore-ancestry

告诉 Subversion 在计算区别（只依赖于路径内容）时忽略祖先。

--ignore-externals

告诉 Subversion 忽略外部定义和外部定义管理的工作拷贝。

--incremental

打印适合串联的输出格式。

--limit *NUM*

只显示第一个 *NUM* 日志信息。

--message (-m) *MESSAGE*

表示你会在命令行中指定日志信息，紧跟这个开关，例如：

```
$ svn commit -m "They don't make Sunday."
```

--new *ARG*

使用 *ARG* 作为新的目标（结合 **svn diff** 使用）。

--no-auth-cache

阻止在 Subversion 管理区缓存认证信息（如用户名密码）。

--no-auto-props

关闭 auto-props，覆盖 *config* 文件中的 enable-auto-props 指示。

--no-diff-added

防止 Subversion 打印添加文件的区别。缺省的行为方式是，当添加一个文件时，**svn diff** 打印的信息和比较一个空白文件相同。

--no-diff-deleted

防止 Subversion 打印删除文件的区别信息，缺省的行为方式是当你删除了一个文件后运行 **svn diff** 打印的区别与删除文件所有的内容得到的结果一样。

--no-ignore

在状态列表中显示 global-ignores 配置选项或者是 svn:ignore 属性忽略的文件。见“配置”一节和“忽略未版本控制的条目”一节查看详情。

--no-unlock

不自动解锁文件(缺省的提交行为是解锁提交列出的所有文件)，更多信息见“锁定”一节。

--non-interactive

如果认证失败，或者是不充分的凭证时，防止出现要求凭证的提示（例如用户名和密码）。这在运行自动脚本时非常有用，只是让 Subversion 失败而不是提示更多的信息。

--non-recursive (-N)

防止子命令迭代到子目录，大多数子命令缺省是迭代的，但是一些子命令—通常是那些潜在的删除或者是取消本地修改的命令—不是。

--notice-ancestry

在计算区别时关注祖先。

--old *ARG*

使用 *ARG* 作为旧的目标（结合 **svn diff** 使用）。

--password *PASS*

指出在命令行中提供你的密码—另外，如果它是需要的，Subversion 会提示你输入。

--quiet (-q)

请求客户端在执行操作时只显示重要信息。

--recursive (-R)

让子命令迭代到子目录，大多数子命令缺省是迭代的。

--relocate *目的路径[PATH...]*

svn switch 子命令中使用，用来修改你的工作拷贝所引用的版本库位置。当版本库的位置修改了，而你有一个工作拷贝，希望继续使用时非常有用。见 **svn switch** 的例子。

--revision (-r) *REV*

指出你将为特定操作提供一个修订版本（或修订版本的范围），你可以提供修订版本号，修订版本关键字或日期（在华括号中）作为修订版本开关的参数。如果你希望提供一个修订版本范围，你可以提供用冒号隔开的两个修订版本，举个例子：

```
$ svn log -r 1729

$ svn log -r 1729:HEAD

$ svn log -r 1729:1744

$ svn log -r {2001-12-04}:{2002-02-17}

$ svn log -r 1729:{2002-02-17}
```

见“修订版本关键字”一节查看更多信息。

--revprop

操作针对修订版本属性，而不是 Subversion 文件或目录的属性。这个选项需要你传递--revision（-r）参数。

--show-updates (-u)

导致客户端显示本地拷贝哪些文件已经过期，这不会实际更新你的任何文件—只是显示了如果你运行 **svn update** 时更新的文件。

--stop-on-copy

导致 Subversion 子命令在传递历史时会在版本化资源拷贝时停止收集历史信息—也就是历史中资源从另一个位置拷贝过来时。

--strict

导致 Subversion 使用严格的语法，就是明确使用特定而不是含糊的子命令（也就是，**svn propget**）。

--targets *FILENAME*

告诉 Subversion 从你提供的文件中得到希望操作的文件列表，而不是在命令行列出所有的文件。

--username *NAME*

表示你要在命令行提供认证的用户名—否则如果需要，Subversion 会提示你这一点。

--verbose (-v)

请求客户端在运行子命令打印尽量多的信息，会导致 Subversion 打印额外的字段，每个文件的细节信息或者是关于动作的附加信息。

--version

打印客户端版本信息，这个信息不仅仅包括客户端的版本号，也有所有客户端可以用来访问 Subversion 版本库的版本库访问模块列表。

--xml

使用 XML 格式打印输出。

svn 子命令

下面是一些子命令：

名称

svn add — 添加文件、目录或符号链。

概要

```
svn add PATH...
```

描述

文件、目录或符号链到你的工作拷贝并且预定添加到版本库。它们会在下次提交上传并添加到版本库，如果你在提交之前改变了主意，你可以使用 **svn revert** 取消预定。

别名

无

改变

工作拷贝

是否访问版本库

否

选项

```
--targets FILENAME

--non-recursive (-N)

--quiet (-q)

--config-dir DIR

--no-ignore

--auto-props

--no-auto-props

--force
```

例子

添加一个文件到工作拷贝：

```
$ svn add foo.c
A         foo.c
```

当添加一个目录，**svn add** 缺省的行为方式是递归的：

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

你可以只添加一个目录而不包括其内容：

```
$ svn add --non-recursive otherdir
A         otherdir
```

通常情况下，命令 **svn add *** 会忽略所有已经在版本控制之下的目录，有时候，你会希望添加所有工作拷贝的未版本化文件，包括那些隐藏在深处的文件，可以使用 **svn add** 的--force 递归到版本化的目录下：

```
$ svn add * --force
A         foo.c
A         somedir/bar.c
A         otherdir/docs/baz.doc
```

```
...
```

名称

svn blame — 显示特定文件和 URL 内嵌的作者和修订版本信息。

概要

```
svn blame TARGET[@REV]...
```

描述

显示特定文件和 URL 内嵌的作者和修订版本信息。每一行文本在开头都放了最后修改的作者（用户名）和修订版本号。

别名

praise、annotate、ann

改变

无 2

是否访问版本库

是

选项

```
--revision (-r) ARG

--verbose (-v)

--incremental

--xml

--extensions (-x) ARG

--force

--username ARG

--password ARG

--no-auth-cache

--non-interactive

--config-dir ARG
```

例子

如果你希望在测试版本库看到 blame 标记的 *readme.txt* 源代码：

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt

     3      sally This is a README file.

     5      harry You should read this.
```

即使 **svn blame** 说明 Harry 最后在修订版本 5 被修改，你也需要验证 Harry 在修订版本修改行的*上下文*—也许他只是调整了空格。

名称

svn cat — 输出特定文件或 URL 的内容。

概要

```
svn cat TARGET[@REV]...
```

描述

输出特定文件或 URL 的内容。列出目录的内容可以使用 **svn list**。

别名

无

改变

无 2

是否访问版本库

是

选项

```
--revision (-r) REV

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

如果你希望不检出而察看版本库的 readme.txt 的内容：

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt

This is a README file.

You should read this.
```

如果你的工作拷贝已经过期（或者你有本地修改），并且希望察看工作拷贝的 HEAD 修订版本的一个文件，如果你给定一个路径，**svn cat** 会自动取得 HEAD 的修订版本：

```
$ cat foo.c

This file is in my local working copy

and has changes that I've made.



$ svn cat foo.c

Latest revision fresh from the repository!
```

名称

svn checkout — 从版本库取出一个工作拷贝。

概要

```
svn checkout URL[@REV]... [PATH]
```

描述

从版本库取出一个工作拷贝，如果省略 *PATH*，URL 的基名称会作为目标，如果给定多个 URL，每一个都会检出到 *PATH* 的子目录，使用 URL 基名称的子目录名称。

别名

co

改变

创建一个工作拷贝。

是否访问版本库

是

选项

```
--revision (-r) REV

--quiet (-q)

--non-recursive (-N)

--username USER

--password PASS

--no-auth-cache

--non-interactive

--ignore-externals
```

```
--config-dir DIR
```

例子

取出一个工作拷贝到 *mine* 目录：

```
$ svn checkout file:///tmp/repos/test mine

A  mine/a

A  mine/b

Checked out revision 2.

$ ls

mine
```

检出两个目录到两个单独的工作拷贝：

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz

A  test/a

A  test/b

Checked out revision 2.

A  quiz/l

A  quiz/m

Checked out revision 2.

$ ls

quiz  test
```

检出两个目录到两个单独的工作拷贝，但是将两个目录都放到 *working-copies*：

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz
working-copies

A  working-copies/test/a

A  working-copies/test/b

Checked out revision 2.

A  working-copies/quiz/l

A  working-copies/quiz/m

Checked out revision 2.

$ ls

working-copies
```

如果你打断一个检出（或其它打断检出的事情，如连接失败。），你可以使用同样的命令重新开始或者是更新不完整的工作拷贝：

```
$ svn checkout file:///tmp/repos/test test

A  test/a

A  test/b

^C

svn: The operation was interrupted

svn: caught SIGINT



$ svn checkout file:///tmp/repos/test test

A  test/c

A  test/d

^C

svn: The operation was interrupted
```

```
svn: caught SIGINT



$ cd test

$ svn update

A  test/e

A  test/f

Updated to revision 3.
```

名称

svn cleanup — 递归清理工作拷贝。

概要

```
svn cleanup [PATH...]
```

描述

递归清理工作拷贝，删除未完成的工作拷贝锁定，并恢复未完成的操作。如果你得到一个“工作拷贝已锁定”的错误，运行这个命令可以删除无效的锁定，让你的工作拷贝再次回到可用的状态。

如果，因为一些原因，运行外置的区别程序（例如，用户输入或是网络错误）有时候会导致一个 **svn update** 失败，使用--diff3-cmd 选项可以完全清除你的外置区别程序所作的合并，你也可以使用--config-dir 指定任何配置目录，但是你应该不会经常使用这些选项。

别名

无

改变

工作拷贝 2

是否访问版本库

否

选项

```
--diff3-cmd CMD

--config-dir DIR
```

例子

svn cleanup 没有输出，没有太多的例子，如果你没有传递 *PATH*，会使用“.”。

```
$ svn cleanup



$ svn cleanup /path/to/working-copy
```

名称

svn commit — 将修改从工作拷贝发送到版本库。

概要

```
svn commit [PATH...]
```

描述

将修改从工作拷贝发送到版本库。如果你没有使用--file 或--message 提供一个提交日志信息，**svn** 会启动你的编辑器来编写一个提交信息，见“配置”一节的 editor-cmd 小节。

svn commit 会返回所有找到的锁定令牌并释放所有提交 *PATHS* 的锁定，除非传递--no-unlock 参数。

如果你开始一个提交并且 Subversion 启动了你的编辑器来编辑提交信息，你仍可以退出而不会提交你的修改，如果你希望取消你的提交，只需要退出编辑器而不保存你的提交信息，Subversion 会提示你是选择取消提交、空信息继续还是重新编辑信息。

别名

ci（“check in”的缩写；不是“checkout”的缩写“co”。）

改变

工作拷贝，版本库

是否访问版本库

是

选项

```
--message (-m) TEXT

--file (-F) FILE

--quiet (-q)

--no-unlock

--non-recursive (-N)

--targets FILENAME

--force-log

--username USER

--password PASS

--no-auth-cache

--non-interactive

--encoding ENC

--config-dir DIR
```

例子

使用命令行提交一个包含日志信息的文件修改，当前目录（“.”）是没有说明的目标路径：

```
$ svn commit -m "added howto section."

Sending        a
```

```
Transmitting file data .

Committed revision 3.
```

提交一个修改到 *foo.c*（在命令行明确指明），并且 msg 文件中保存了提交信息：

```
$ svn commit -F msg foo.c

Sending        foo.c

Transmitting file data .

Committed revision 5.
```

如果你希望使用在--file 选项中使用在版本控制之下的文件作为参数，你需要使用--force-log 选项：

```
$ svn commit --file file under vc.txt foo.c

svn: The log message file is under version control

svn: Log message file is a versioned file; use '--force-log' to
override



$ svn commit --force-log --file file under vc.txt foo.c

Sending        foo.c

Transmitting file data .

Committed revision 6.
```

提交一个已经预定要删除的文件：

```
$ svn commit -m "removed file 'c'."

Deleting       c
```

```
Committed revision 7.
```

名称

svn copy — 拷贝工作拷贝的一个文件或目录到版本库。

概要

```
svn copy SRC DST
```

描述

拷贝工作拷贝的一个文件或目录到版本库。*SRC*和 *DST*既可以是工作拷贝（WC）路径也可以是 URL：

WC -> WC

拷贝并且预定一个添加的项目（包含历史）。

WC -> URL

将 WC 或 URL 的拷贝立即提交。

URL -> WC

检出 URL 到 WC，并且加入到添加计划。

URL -> URL

完全的服务器端拷贝，通常用在分支和标签。

你只可以在单个版本库中拷贝文件，Subversion 还不支持跨版本库的拷贝。

别名

cp

改变

如果目标是 URL 则包括版本库。

如果目标是 WC 路径，则是工作拷贝。

是否访问版本库

如果目标是版本库，或者需要查看修订版本号，则会访问版本库。

选项

```
--message (-m) TEXT

--file (-F) FILE

--revision (-r) REV

--quiet (-q)

--username USER

--password PASS

--no-auth-cache

--non-interactive

--force-log

--editor-cmd EDITOR
```

```
--encoding ENC

--config-dir DIR
```

例子

拷贝工作拷贝的一个项目（只是预定要拷贝—在提交之前不会影响版本库）：

```
$ svn copy foo.txt bar.txt

A         bar.txt

$ svn status

A  +   bar.txt
```

拷贝你的工作拷贝的一个项目到版本库的 URL（直接的提交，所以需要提供一个提交信息）：

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m
"Remote copy."



Committed revision 8.
```

拷贝版本库的一个项目到你的工作拷贝（只是预定要拷贝—在提交之前不会影响版本库）：

这是恢复死掉文件的推荐方式！

```
$ svn copy file:///tmp/repos/test/far-away near-here

A         near-here
```

最后，是在 URL 之间拷贝：

```
$ svn copy file:///tmp/repos/test/far-away
file:///tmp/repos/test/over-there -m "remote copy."



Committed revision 9.
```

这是在版本库里作“标签”最简单的方法—**svn copy** 那个修订版本（通常是 HEAD）到你的 tags 目录。

```
$ svn copy file:///tmp/repos/test/trunk
file:///tmp/repos/test/tags/0.6.32-prerelease -m "tag tree"



Committed revision 12.
```

不要担心忘记作标签—你可以在以后任何时候给一个旧版本作标签：

```
$ svn copy -r 11 file:///tmp/repos/test/trunk
file:///tmp/repos/test/tags/0.6.32-prerelease -m "Forgot to
tag at rev 11"



Committed revision 13.
```

名称

svn delete — 从工作拷贝或版本库删除一个项目。

概要

```
svn delete PATH...

svn delete URL...
```

描述

*PATH*指定的项目会在下次提交删除，文件（和没有提交的目录）会立即从版本库删除，这个命令不会删除任何未版本化或已经修改的项目；使用--force 选项可以覆盖这种行为方式。

URL 指定的项目会在直接提交中从版本库删除，多个 URL 的提交是原子操作。

别名

del, remove, rm

改变

如果操作对象是文件则是工作拷贝变化，对象是URL则会影响版本库。

是否访问版本库

对 URL 操作时访问

选项

```
--force

--force-log

--message (-m) TEXT
```

```
--file (-F) FILE

--quiet (-q)

--targets FILENAME

--username USER

--password PASS

--no-auth-cache

--non-interactive

--editor-cmd EDITOR

--encoding ENC

--config-dir DIR
```

例子

使用 **svn** 从工作拷贝删除文件只是预定要删除，当你提交，文件才会从版本库删除。

```
$ svn delete myfile

D        myfile



$ svn commit -m "Deleted file 'myfile'."

Deleting      myfile

Transmitting file data .

Committed revision 14.
```

然而直接删除一个 URL，你需要提供一个日志信息：

```
$ svn delete -m "Deleting file 'yourfile'"
file:///tmp/repos/test/yourfile



Committed revision 15.
```

如下是强制删除本地已修改文件的例子：

```
$ svn delete over-there

svn: Attempting restricted operation for modified resource

svn: Use --force to override this restriction

svn: 'over-there' has local modifications



$ svn delete --force over-there

D        over-there
```

名称

svn diff — 比较两条路径的区别。

概要

```
diff [-c M | -r N[:M]] [TARGET[@REV]...]

diff [-r N[:M]] --old=OLD-TGT[@OLDREV]
[--new=NEW-TGT[@NEWREV]] [PATH...]

diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

描述

显示两条路径的区别，**svn diff** 有三种使用方式：

运行 **svn diff** 以标准差别格式查看本地工作拷贝修改的内容。

显示 *TARGET* 在 *REV* 的样子时两个修订版本之间所作的修改，*TARGET* 可以是任何工作拷贝路径或任何 *URL*，如果 *TARGET* 是工作拷贝路径，则 *N* 缺省是 BASE，而 *M* 是工作拷贝；如果是 *URL*，则必须指定 *N*，而 *M* 缺省是 HEAD。“-c M”选项与“-r N:M”等价，其中 N = M-1。使用“-c -M”则相反：“-r M:N”的意思是 N = M-1。

显示在 *OLDREV* 的 *OLD-TGT* 和 *NEWREV* 的 *NEW-TGT* 之间的区别。如果提供 *PATH*，则与 *OLD-TGT* 和 *NEW-TGT* 关联，将输出限制在那些路径。*OLD-TGT* 和 *NEW-TGT* 可能是工作拷贝路经或 *URL[@REV]*。如果没有指定，*NEW-TGT* 缺省是 *OLD-TGT*。“-r N”设置 OLDREV 缺省为 N，而-r N:M 设置 *OLDREV* 缺省为 *N*，而 *NEWREV* 缺省为 *M*。

svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV] 的简写方式。

svn diff -r N:M URL 是 **svn diff -r N:M --old=URL --new=URL** 的简写。

svn diff [-r N[:M]] URL1[@N] URL2[@M] 是 **svn diff [-r N[:M]] --old=URL1 --new=URL2** 的简写。

TARGET 是一个 URL，然后可以使用前面提到的--revision 或“@”符号来指定 N 和 M。

如果 *TARGET* 是工作拷贝路径，则--revision 选项的含义是：

--revision N:M

服务器比较 *TARGET*@*N* 和 *TARGET*@*M*。

--revision N

客户端比较 *TARGET*@*N* 和工作拷贝。

（无--revision）

客户端比较 base 和 *TARGET* 的 *TARGET*。

如果使用其他语法，服务器会比较 *URL1* 和 *URL2* 各自的 *N* 和 *M*。如果省掉 *N* 或 *M*，会假定为 HEAD。

缺省情况下，**svn diff** 忽略文件的祖先，只会比较两个文件的内容。如果你使用--notice-ancestry，比较修订版本（也就是，当你运行 **svn diff** 比较两个内容相同，但祖先历史不同的对象会看到所有的内容被删除又再次添加）时就会考虑路径的祖先。

别名

di

改变

无 2

是否访问版本库

获得工作拷贝非 BASE 修订版本的区别时会

选项

```
--revision (-r) ARG

--change (-c) ARG

--old ARG

--new ARG

--non-recursive (-N)

--diff-cmd CMD

--extensions (-x) "ARGS"

--no-diff-deleted

--notice-ancestry

--summarize

--force

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

比较 BASE 和你的工作拷贝（**svn diff** 最经常的用法）：

```
$ svn diff COMMITTERS

Index: COMMITTERS

===================================================================

--- COMMITTERS      (revision 4404)

+++ COMMITTERS      (working copy)
```

查看文件 COMMITTERS 在修订版本 9115 修改的内容：

```
$ svn diff -c 9115 COMMITTERS

Index: COMMITTERS

===================================================================

--- COMMITTERS      (revision 3900)

+++ COMMITTERS      (working copy)
```

察看你的工作拷贝对旧的修订版本的修改：

```
$ svn diff -r 3900 COMMITTERS

Index: COMMITTERS

===================================================================

--- COMMITTERS      (revision 3900)

+++ COMMITTERS      (working copy)
```

使用“@”语法与修订版本 3000 和 35000 比较：

```
$ svn diff
http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000
http://svn.collab.net/repos/svn/trunk/COMMITTERS@3500

Index: COMMITTERS

===================================================================
=======

--- COMMITTERS      (revision 3000)

+++ COMMITTERS      (revision 3500)

…
```

使用范围符号来比较修订版本 3000 和 3500（在这种情况下只能传递一个 URL）：

```
$ svn diff -r 3000:3500
http://svn.collab.net/repos/svn/trunk/COMMITTERS

Index: COMMITTERS

===================================================================
=======

--- COMMITTERS      (revision 3000)

+++ COMMITTERS      (revision 3500)
```

使用范围符号比较修订版本 3000 和 3500*trunk* 中的所有文件：

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk

_________
```

使用范围符号比较修订版本 3000 和 3500*trunk* 中的三个文件：

```
$ svn diff -r 3000:3500 --old
http://svn.collab.net/repos/svn/trunk COMMITTERS README
HACKING

_________
```

如果你有工作拷贝，你不必输入这么长的 URL：

```
$ svn diff -r 3000:3500 COMMITTERS

Index: COMMITTERS

===================================================================
=======

--- COMMITTERS      (revision 3000)

+++ COMMITTERS      (revision 3500)
```

使用--diff-cmd*CMD*-x 来指定外部区别程序

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS

Index: COMMITTERS

===================================================================
=======

0a1,2

> This is a test

>
```

名称

svn export — 导出一个干净的目录树。

概要

```
svn export [-r REV] URL[@PEGREV] [PATH]

svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

描述

第一种从版本库导出干净工作目录树的形式是指定 URL，如果指定了修订版本 *REV*，会导出相应的版本，如果没有指定修订版本，则会导出 HEAD，导出到 *PATH*。如果省略 *PATH*，*URL* 的最后一部分会作为本地目录的名字。

从工作拷贝导出干净目录树的第二种形式是指定 *PATH1* 到 *PATH2*，所有的本地修改将会保留，但是不再版本控制下的文件不会拷贝。

别名

无

改变

本地磁盘

是否访问版本库

只有当从 URL 导出时会访问

选项

```
--revision (-r) REV
```

```
--quiet (-q)

--force

--username USER

--password PASS

--no-auth-cache

--non-interactive

--non-recursive (-N)

--config-dir DIR

--native-eol EOL

--ignore-externals
```

例子

从你的工作拷贝导出（不会打印每一个文件和目录）：

```
$ svn export a-wc my-export

Export complete.
```

从版本库导出目录（打印所有的文件和目录）：

```
$ svn export file:///tmp/repos my-export

A  my-export/test

A  my-export/quiz

…

Exported revision 15.
```

当使用操作系统特定的分发版本，使用特定的 EOL 字符作为行结束符号导出一棵树会非常有用。--native-eol 选项会这样做，但是如果影响的文件拥有 svn:eol-style = native 属性，举个例子，导出一棵使用 CRLF 作为行结束的树（可能是为了做一个 Windows 的.zip 文件分发版本）：

```
$ svn export file:///tmp/repos my-export --native-eol CRLF
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

你可以为--native-eol 选项指定 LR、CR 或 CRLF 作为行结束符。

名称

svn help — 求助！

概要

```
svn help [SUBCOMMAND...]
```

描述

当手边没有这本书时，这是你使用 Subversion 最好的朋友！

别名

?, h

使用-?、-h 和--help 选项与使用 **help** 子命令效果相同。

改变

无 2

是否访问版本库

否

选项

```
--config-dir DIR
```

名称

svn import — 递归提交一个路径的拷贝到版本库。

概要

```
svn import [PATH] URL
```

描述

递归提交一个路径的拷贝到 URL。如果省略 *PATH*，默认是“.”。版本库中对应的父目录必须已经创建。

别名

无

改变

版本库

是否访问版本库

是

选项

```
--message (-m) TEXT

--file (-F) FILE

--quiet (-q)

--non-recursive (-N)

--username USER

--password PASS

--no-auth-cache

--non-interactive

--force-log

--editor-cmd EDITOR

--encoding ENC

--config-dir DIR

--auto-props

--no-auto-props
```

```
--ignore-externals
```

例子

这将本地目录 *myproj* 导入到版本库的 *trunk/misc*，*trunk/misc* 在导入之前不需要存在—**svn import** 会递归的为你创建目录。

```
$ svn import -m "New import" myproj
http://svn.red-bean.com/repos/trunk/misc

Adding         myproj/sample.txt

…

Transmitting file data .........

Committed revision 16.
```

需要知道这样*不*会在版本库创建目录 *myproj*，如果你希望这样，请在 URL 后添加 *myproj*：

```
$ svn import -m "New import" myproj
http://svn.red-bean.com/repos/trunk/misc/myproj

Adding         myproj/sample.txt

…

Transmitting file data .........

Committed revision 16.
```

在导入数据之后，你会发现原先的目录树*并没有*纳入版本控制，为了开始工作，你还是要运行 **svn checkout** 得到一个干净的目录树工作拷贝。

名称

svn info — 显示本地或远程条目的信息。

概要

```
svn info [TARGET[@REV]...]
```

描述

打印你的工作拷贝路径和 URL 的信息，包括：

- 路经
- 名称
- URL
- 版本库的根
- 版本库的 UUID
- Revision
- 节点类型
- 最后修改的作者
- 最后修改的修订版本
- 最后修改的日期
- 锁定令牌
- 锁定拥有者
- 锁定创建时间
- Lock Expires (date)

Additional kinds of information available only for working copy paths are:

- Schedule
- Copied From URL
- Copied From Rev
- Text Last Updated
- Properties Last Updated
- Checksum
- Conflict Previous Base File
- Conflict Previous Working File
- Conflict Current Base File
- Conflict Properties File

别名

无

改变

无 2

是否访问版本库

对 URL 操作时访问

选项

```
--revision (-r) REV

--recursive (-R)

--targets FILENAME

--incremental

--xml

--username USER
```

```
--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

svn info 会展示工作拷贝所有项目的所有有用信息，它会显示文件的信息：

```
$ svn info foo.c

Path: foo.c

Name: foo.c

URL: http://svn.red-bean.com/repos/test/foo.c

Repository Root: http://svn.red-bean.com/repos/test

Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25

Revision: 4417

Node Kind: file

Schedule: normal

Last Changed Author: sally

Last Changed Rev: 20

Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)

Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)

Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan
2003)

Checksum: /3L38YwzhT93BWvgpdF6Zw==
```

它也会展示目录的信息：

```
$ svn info vendors

Path: vendors

URL: http://svn.red-bean.com/repos/test/vendors

Repository Root: http://svn.red-bean.com/repos/test

Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25

Revision: 19

Node Kind: directory

Schedule: normal

Last Changed Author: harry

Last Changed Rev: 19

Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
```

svn info 也可以针对 URL 操作（另外，可以注意一下例子中的 readme.doc 文件已经被锁定，所以也会显示锁定信息）：

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc

Path: readme.doc

Name: readme.doc

URL: http://svn.red-bean.com/repos/test/readme.doc

Repository Root: http://svn.red-bean.com/repos/test

Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25

Revision: 1

Node Kind: file

Schedule: normal
```

```
Last Changed Author: sally

Last Changed Rev: 42

Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)

Text Last Updated: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)

Checksum: d41d8cd98f00b204e9800998ecf8427e

Lock Token:
opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2

Lock Owner: harry

Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
```

名称

svn list — 列出版本库目录的条目。

概要

```
svn list [TARGET[@REV]...]
```

描述

列出每一个 *TARGET* 文件和 *TARGET* 目录的内容，如果 *TARGET* 是工作拷贝路径，会使用对应的版本库 URL。

缺省的 *TARGET* 是“.”，意味着当前工作拷贝的版本库 URL。

如果一个客户端连接到 **svnserve** 进程，如下事情会发生：

- 最后一次提交的修订版本号

- 最后一次提交的作者
- 如果锁定，字符为“O”（更多细节见 svn info）
- 大小（单位字节）
- 最后提交的日期时间

使用选项--xml，输出是 XML 格式（如果没有指定--incremental，会包括一个头和一个围绕的元素）。会展示所有的信息；不接受--verbose 选项。

别名

ls

改变

无 2

是否访问版本库

是

选项

```
--revision (-r) REV

--verbose (-v)

--recursive (-R)

--incremental

--xml
```

```
--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

如果你希望在没有下载工作拷贝时查看版本库有哪些文件，**svn list** 会非常有用：

```
$ svn list http://svn.red-bean.com/repos/test/support

README.txt

INSTALL

examples/

…
```

你也可以传递--verbose 选项来得到额外信息，非常类似 UNIX 的 **ls -l** 命令：

```
$ svn list --verbose file:///tmp/repos

    16 sally        28361 Jan 16 23:18 README.txt

    27 sally            0 Jan 18 15:27 INSTALL

    24 harry              Jan 18 11:27 examples/
```

更多细节见“ **svn list** ”一节。

名称

svn lock — 锁定版本库的工作拷贝路径或 URL，所以没有其他用户可以提交这些文件的修改。

概要

```
Synopsis
```

描述

svn lock TARGET...

别名

无

改变

锁定每个 *TARGET*。如果任何 *TARGET* 已经被另一个用户锁定，则会打印警告信息并且继续锁定剩下的 *TARGET*。可以使用--force 从其它用户来窃取锁定。

是否访问版本库

是

选项

```
--targets FILENAME
```

```
--message (-m) TEXT

--file (-F) FILE

--force-log

--encoding ENC

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR

--force
```

例子

在工作拷贝锁定两个文件：

```
$ svn lock tree.jpg house.jpg

'tree.jpg' locked by user 'harry'.

'house.jpg' locked by user 'harry'.
```

锁定工作拷贝的一个被其它用户锁定的文件：

```
$ svn lock tree.jpg

svn: warning: Path '/tree.jpg is already locked by user 'sally
in \

    filesystem '/svn/repos/db'
```

```
$ svn lock --force tree.jpg

'tree.jpg' locked by user 'harry'.
```

没有工作拷贝的情况下锁定文件：

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg

'tree.jpg' locked by user 'harry'.
```

更多细节见“锁定”一节。

名称

svn log — 显示提交日志信息。

概要

```
svn log [PATH]

svn log URL [PATH...]

svn log URL[@REV] [PATH...]
```

描述

缺省目标是你的当前目录的路径，如果没有提供参数，**svn log** 会显示当前目录下的所有文件和目录的日志信息，你可以通过指定路径来精炼结果，一个或多个修订版本，或者是任何两个的组合。对于本地路径的缺省修订版本范围 BASE:1。

如果你只是指定一个 URL，就会打印这个 URL 上所有的日志信息，如果添加部分路径，只有这条路径下的 URL 信息会被打印，URL 缺省的修订版本范围是 HEAD:1。

svn log 使用--verbose 选项也会打印所有影响路径的日志信息，使用--quiet 选项不会打印日志信息正文本身（这与--verbose 协调一致）。

每个日志信息只会打印一次，即使是那些明确请求不止一次的路径，日志会跟随在拷贝过程中，使用--stop-on-copy 可以关闭这个特性，可以用来监测分支点。

别名

无

改变

无 2

是否访问版本库

是

选项

```
--revision (-r) REV

--quiet (-q)

--verbose (-v)
```

```
--targets FILENAME

--stop-on-copy

--incremental

--limit NUM

--xml

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

你可以在顶级目录运行 **svn log** 看到工作拷贝中所有修改的路径的日志信息：

```
$ svn log

------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line



Tweak.

------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines

…
```

检验一个特定文件所有的日志信息：

```
$ svn log foo.c

------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line


Added defines.

------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines

…
```

如果你手边没有工作拷贝，你可以查看一个 URL 的日志：

```
$ svn log http://svn.red-bean.com/repos/test/foo.c

------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line


Added defines.

------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines

…
```

如果你希望查看某个 URL 下面不同的多个路径，你可以使用 URL [PATH...]语法。

```
$ svn log http://svn.red-bean.com/repos/test/ foo.c bar.c

------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line



Added defines.

------------------------------------------------------------------------
r31 | harry | 2003-01-10 12:25:08 -0600 (Fri, 10 Jan 2003) | 1 line



Added new file bar.c

------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines

…
```

当你想连接多个对日志命令的调用结果，你会希望使用--incremental 选项。**svn log** 通常会在日志信息的开头和每一小段间打印一行虚线，如果你对一段修订版本运行 **svn log**，你会得到下面的结果：

```
$ svn log -r 14:15

------------------------------------------------------------------------
```

```
r14 | ...



------------------------------------------------------------------------

r15 | ...



------------------------------------------------------------------------
```

然而，如果你希望收集两个不连续的日志信息到一个文件，你会这样做：

```
$ svn log -r 14 > mylog

$ svn log -r 19 >> mylog

$ svn log -r 27 >> mylog

$ cat mylog

------------------------------------------------------------------------

r14 | ...



------------------------------------------------------------------------

------------------------------------------------------------------------

r19 | ...



------------------------------------------------------------------------

------------------------------------------------------------------------
```

```
r27 | ...



------------------------------------------------------------------------
```

你可以使用 incremental 选项来避免两行虚线带来的混乱：

```
$ svn log --incremental -r 14 > mylog

$ svn log --incremental -r 19 >> mylog

$ svn log --incremental -r 27 >> mylog

$ cat mylog

------------------------------------------------------------------------

r14 | ...



------------------------------------------------------------------------

r19 | ...



------------------------------------------------------------------------

r27 | ...
```

--incremental 选项为--xml 提供了一个相似的输出控制。

如果你在特定路径和修订版本运行 **svn log**，输出结果为空

```
$ svn log -r 20
http://svn.red-bean.com/untouched.txt
```

```
------------------------------------------------------------------------
```

这只意味着这条路径在那个修订版本没有修改，如果从版本库的顶级目录运行这个命令，或者是你知道那个修订版本修改了那个文件，你可以明确的指定它：

```
$ svn log -r 20 touched.txt

------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line



Made a change.

------------------------------------------------------------------------
```

名称

svn merge — 应用两组源文件的差别到工作拷贝路径。

概要

```
svn merge [-c M | -r N:M] SOURCE[@REV] [WCPATH]

svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]

svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]
```

描述

第一种和第二种形式里，源路径（第一种是 URL，第二种是工作拷贝路径）用修订版本号 *N* 和 *M* 指定，这是要比较的两组源文件，如果省略修订版本号，缺省是 HEAD。

-c M 选项与-r N:M 等价，其中 *N = M-1*，使用-c -M 则相反：-r M:N，其中 *N = M-1*。

第三种形式，*SOURCE* 可以是 URL 或者工作拷贝项目，与之对应的 URL 会被使用。在修订版本号 *N* 和 *M* 的 URL 定义了要比较的两组源。

WCPATH 是接收变化的工作拷贝路径，如果省略 *WCPATH*，会假定缺省值“.”，除非源有相同基本名称与“.”中的某一文件名字匹配：在这种情况下，区别会应用到那个文件。

不像 **svn diff**，合并操作在执行时会考虑文件的祖先，当你从一个分支合并到另一个分支，而这两个分支有各自重命名的文件时，这一点会非常重要。

别名

无

改变

工作拷贝 2

是否访问版本库

只有在对 URL 操作时会

选项

```
--revision (-r) REV

--change (-c) REV

--non-recursive (-N)

--quiet (-q)

--force

--dry-run

--diff3-cmd CMD

--extensions (-x) ARG

--ignore-ancestry

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

将一个分支合并回主干（假定你有一份主干的工作拷贝，分支在修订版本 250 创建）：

```
$ svn merge -r 250:HEAD
http://svn.red-bean.com/repos/branches/my-branch

U  myproj/tiny.txt
```

```
U  myproj/thhgttg.txt

U  myproj/win.txt

U  myproj/flo.txt
```

如果你的分支在修订版本 23，你希望将主干的修改合并到分支，你可以在你的工作拷贝的分支上这样做：

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors

U  myproj/thhgttg.txt

…
```

合并一个单独文件的修改：

```
$ cd myproj

$ svn merge -r 30:31 thhgttg.txt

U  thhgttg.txt
```

名称

svn mkdir — 创建一个纳入版本控制的新目录。

概要

```
svn mkdir PATH...

svn mkdir URL...
```

描述

创建一个目录，名字是提供的 *PATH* 或者 URL 的最后一部分，工作拷贝 *PATH* 指定的目录会预定要添加，而通过 URL 指定的目录会作为一次立即提交在版本库建立。多个目录 URL 的提交是原子操作，在两种情况下，中介目录必须已经存在。

别名

无

改变

如果是对 URL 操作则会影响版本库，否则是工作拷贝

是否访问版本库

只有在对 URl 操作时会

选项

```
--message (-m) TEXT

--file (-F) FILE

--quiet (-q)

--username USER

--password PASS

--no-auth-cache

--non-interactive

--editor-cmd EDITOR

--encoding ENC
```

```
--force-log

--config-dir DIR
```

例子

在工作拷贝创建一个目录：

```
$ svn mkdir newdir

A         newdir
```

在版本库创建一个目录（立即提交，所以需要日志信息）：

```
$ svn mkdir -m "Making a new dir."
http://svn.red-bean.com/repos/newdir



Committed revision 26.
```

名称

svn move — 移动一个文件或目录。

概要

```
svn move SRC DST
```

描述

这个命令移动文件或目录到你的工作拷贝或者是版本库。

这个命令同 **`svn copy`** 加一个 **`svn delete`** 等同。

Subversion 不支持在工作拷贝和 URL 之间拷贝，此外，你只可以一个版本库内移动文件—Subversion 不支持跨版本库的移动。

WC -> WC

移动和预订一个文件或目录将要添加（包含历史）。

URL -> URL

完全服务器端的重命名。

别名

mv, rename, ren

改变

如果是对 URL 操作则会影响版本库，否则是工作拷贝

是否访问版本库

只有在对 URI 操作时会

选项

```
--message (-m) TEXT

--file (-F) FILE

--revision (-r) REV (废弃的)

--quiet (-q)
```

```
--force

--username USER

--password PASS

--no-auth-cache

--non-interactive

--editor-cmd EDITOR

--encoding ENC

--force-log

--config-dir DIR
```

例子

移动工作拷 bede 一个文件：

```
$ svn move foo.c bar.c

A         bar.c

D         foo.c
```

移动版本库中的一个文件（一个立即提交，所以需要提交信息）：

```
$ svn move -m "Move a file" http://svn.red-bean.com/repos/foo.c
\

                          http://svn.red-bean.com/repos/bar.c



Committed revision 27.
```

名称

svn propdel — 删除一个项目的一个属性。

概要

```
svn propdel PROPNAME [PATH...]

svn propdel PROPNAME --revprop -r REV [TARGET]
```

描述

这会删除文件、目录或修订版本的属性。第一种形式是在工作拷贝删除版本化属性，第二种是在一个版本库修订版本中删除未版本化的属性（*TARGET* 只是用来确定访问哪个版本库）。

别名

pdel, pd

改变

只有在对 URL 操作时会 2

是否访问版本库

只有在对 URl 操作时会

选项

```
--quiet (-q)

--recursive (-R)

--revision (-r) REV
```

```
--revprop

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

删除你的工作拷贝中一个文件的一个属性

```
$ svn propdel svn:mime-type  some-script

property 'svn:mime-type' deleted from 'some-script'.
```

删除一个修订版本的属性：

```
$ svn propdel --revprop -r 26 release-date

property 'release-date' deleted from repository revision '26'
```

名称

svn propedit — 修改一个或多个版本控制之下文件的属性。

概要

```
svn propedit PROPNAME PATH...

svn propedit PROPNAME --revprop -r REV [TARGET]
```

描述

使用喜欢的编辑器编辑一个或多个属性，第一种形式是在工作拷贝编辑版本化的属性，第二种形式是远程编辑未版本化的版本库修订版本属性（*TARGET* 只是用来确定访问哪个版本库）。

别名

pedit, pe

改变

只有在对 URL 操作时会 2

是否访问版本库

只有在对 URI 操作时会

选项

```
--revision (-r) REV

--revprop

--username USER

--password PASS

--no-auth-cache

--non-interactive

--encoding ENC

--editor-cmd EDITOR
```

```
--config-dir DIR
```

例子

svn propedit 对修改多个值的属性非常简单：

```
$ svn propedit svn:keywords  foo.c

   <svn will launch your favorite editor here, with a buffer
open

   containing the current contents of the svn:keywords
property.  You

   can add multiple values to a property easily here by entering
one

   value per line.>

Set new value for property 'svn:keywords' on 'foo.c'
```

名称

svn propget — 打印一个属性的值。

概要

```
svn propget PROPNAME [TARGET[@REV]...]

svn propget PROPNAME --revprop -r REV [URL]
```

描述

打印一个文件、目录或修订版本的一个属性的值，第一种形式是打印工作拷贝中一个或多个项目的版本化的属性，第二种形式是远程打印版本库修订版本的未版本化的属性。属性的详情见“属性”一节。

别名

pget, pg

改变

只有在对 URL 操作时会 2

是否访问版本库

只有在对 URl 操作时会

选项

```
--recursive (-R)

--revision (-r) REV

--revprop

--strict

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

检查工作拷贝的一个文件的一个属性：

```
$ svn propget svn:keywords foo.c

Author

Date

Rev
```

对于修订版本属性相同：

```
$ svn propget svn:log --revprop -r 20

Began journal.
```

名称

svn proplist — 列出所有的属性。

概要

```
svn proplist [TARGET[@REV]...]

svn proplist --revprop -r REV [TARGET]
```

描述

列出文件、目录或修订版本的属性，第一种形式是列出工作拷贝的所有版本化的属性，第二种形式是列出版本库修订版本的未版本化的属性（*TARGET* 只是用来确定访问哪个版本库）。

别名

plist, pl

改变

只有在对 URL 操作时会 2

是否访问版本库

只有在对 URl 操作时会

选项

```
--verbose (-v)

--recursive (-R)

--revision (-r) REV

--quiet (-q)

--revprop

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

你可以使用 proplist 察看你的工作拷贝的一个项目的属性：

```
$ svn proplist foo.c

Properties on 'foo.c':

  svn:mime-type

  svn:keywords

  owner
```

通过--verbose 选项，svn proplist 也可以非常便利的显示属性的值：

```
$ svn proplist --verbose foo.c

Properties on 'foo.c':

  svn:mime-type : text/plain

  svn:keywords : Author Date Rev

  owner : sally
```

名称

svn propset — 设置文件、目录或者修订版本的属性 PROPNAME 为 PROPVAL。

概要

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...
```

```
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE]
[TARGET]
```

描述

设置文件、目录或者修订版本的属性 *PROPNAME* 为 *PROPVAL*。第一个例子在工作拷贝创建了一个版本化的本地属性修改，第二个例子创建了一个未版本化的远程的对版本库修订版本的属性修改（*TARGET* 只是用来确定访问哪个版本库）。

Subversion 有一系列“特殊的”影响行为方式的属性，关于这些属性的详情请见“Subversion 属性”一节。

别名

pset, ps

改变

只有在对 URL 操作时会 2

是否访问版本库

只有在对 URI 操作时会

选项

```
--file (-F) FILE

--quiet (-q)

--revision (-r) REV
```

```
--targets FILENAME

--recursive (-R)

--revprop

--username USER

--password PASS

--no-auth-cache

--non-interactive

--encoding ENC

--force

--config-dir DIR
```

例子

设置文件的 mimetype：

```
$ svn propset svn:mime-type image/jpeg foo.jpg

property 'svn:mime-type' set on 'foo.jpg'
```

在 UNIX 系统，如果你希望一个文件设置执行权限：

```
$ svn propset svn:executable ON somescript

property 'svn:executable' set on 'somescript'
```

或许为了合作者的利益你有一个内部的属性设置：

```
$ svn propset owner sally foo.c

property 'owner' set on 'foo.c'
```

如果你在特定修订版本的日志信息里有一些错误，并且希望修改，可以使用--revprop 设置 svn:log 为新的日志信息：

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to
New York."

property 'svn:log' set on repository revision '25'
```

或者，你没有工作拷贝，你可以提供一个 URL。

```
$ svn propset --revprop -r 26 svn:log "Document nap."
http://svn.red-bean.com/repos

property 'svn:log' set on repository revision '25'
```

最后，你可以告诉 propset 从一个文件得到输入，你甚至可以使用这个方式来设置一个属性为二进制内容：

```
$ svn propset owner-pic -F sally.jpg moo.c

property 'owner-pic' set on 'moo.c'
```

缺省，你不可以在 Subversion 版本库修改修订版本属性，你的版本库管理员必须显示的通过创建一个名字为 pre-revprop-change 的钩子来允许修订版本属性修改，关于钩子脚本的详情请见“实现版本库钩子”一节。

名称

svn resolved — 删除工作拷贝文件或目录的“冲突”状态。

概要

```
svn resolved PATH...
```

描述

删除工作拷贝文件或目录的“conflicted”状态。这个程序不是语义上的改变冲突标志，它只是删除冲突相关的人造文件，从而重新允许 *PATH* 提交；也就是说，它告诉 Subversion 冲突已经“解决了”。关于解决冲突更深入的考虑可以查看“解决冲突（合并别人的修改）”一节。

别名

无

改变

工作拷贝 2

是否访问版本库

否

选项

```
--targets FILENAME

--recursive (-R)

--quiet (-q)
```

```
--config-dir DIR
```

例子

如果你在更新时得到冲突，你的工作拷贝会产生三个新的文件：

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

当你解决了 *foo.c* 的冲突，并且准备提交，运行 **svn resolved** 让你的工作拷贝知道你已经完成了所有事情。

你*可以*仅仅删除冲突的文件并且提交，但是 **svn resolved** 除了删除冲突文件，还修正了一些记录在工作拷贝管理区域的记录数据，所以我们推荐你使用这个命令。

名称

svn revert — 取消所有的本地编辑。

概要

```
svn revert PATH...
```

描述

恢复所有对文件和目录的修改，并且解决所有的冲突状态。**svn revert** 不会只是恢复工作拷贝中一个项目的内容，也包括了对属性修改的恢复。最终，你可以使用它来取消所有已经做过的预定操作（例如，文件预定要添加或删除可以“恢复”）。

别名

无

改变

工作拷贝 2

是否访问版本库

否

选项

```
--targets FILENAME

--recursive (-R)

--quiet (-q)
```

```
--config-dir DIR
```

例子

丢弃对一个文件的修改：

```
$ svn revert foo.c

Reverted foo.c
```

如果你希望恢复一整个目录的文件，可以使用--recursive 选项：

```
$ svn revert --recursive .

Reverted newdir/afile

Reverted foo.c

Reverted bar.txt
```

最后，你可以取消预定的操作：

```
$ svn add mistake.txt whoops

A         mistake.txt

A         whoops

A         whoops/oopsie.c



$ svn revert mistake.txt whoops

Reverted mistake.txt

Reverted whoops



$ svn status
```

```
?      mistake.txt

?      whoops
```

svn revert 本身有固有的危险，因为它的目的是放弃数据—未提交的修改。一旦你选择了恢复，Subversion *没有方法*找回未提交的修改。

如果你没有给 **svn revert** 提供了目标，它不会做任何事情—为了保护你不小心失去对工作拷贝的修改，**svn revert** 需要你提供至少一个目标。

名称

svn status — 打印工作拷贝文件和目录的状态。

概要

```
svn status [PATH...]
```

描述

打印工作拷贝文件和目录的状态。如果没有参数，只会打印本地修改的项目（不会访问版本库），使用--show-updates 选项，会添加工作修订版本和服务器过期信息。使用--verbose 会打印每个项目的完全修订版本信息。

输出的前六列都是一个字符宽，每一列给出了工作拷贝项目的每一方面的信息。

第一列指出一个项目的是添加、删除还是其它的修改。

' '

没有修改。

'A'

预定要添加的项目。

'D'

预定要删除的项目。

'M'

项目已经修改了。

'R'

项目在工作拷贝中已经被替换了。这意味着文件预定要删除，然后有一个同样名称的文件要在同一个位置替换它。

'C'

项目的内容（相对于属性）与更新得到的数据冲突了。

'X'

项目与外部定义相关。

'I'

项目被忽略（例如使用 svn:ignore 属性）。

'?'

项目不在版本控制之下。

'!'

项目已经丢失（例如，你使用 **svn** 移动或者删除了它）。这也说明了一个目录不是完整的（一个检出或更新中断）。

'~'

项目作为一种对象（文件、目录或链接）纳入版本控制，但是已经被另一种对象替代。

第二列告诉一个文件或目录的属性的状态。

' '

没有修改。

'M'

这个项目的属性已经修改。

'C'

这个项目的属性与从版本库得到的更新有冲突。

第三列只在工作拷贝锁定时才会出现。（见“有时你只需要清理”一节。）

' '

项目没有锁定。

'L'

项目已经锁定。

第四列只在预定包含历史添加的项目出现。

' '

没有历史预定要提交。

'+'

历史预定要伴随提交。

第五列只在项目跳转到相对于它的父目录时出现(见“使用分支”一节)。

' '

项目是它的父目录的孩子。

'S'

项目已经转换。

第六列显示锁定信息。

' '

当使用--show-updates,文件没有锁定。如果*不*使用--show-updates,这意味着文件在工作拷贝被锁定。

K

文件锁定在工作拷贝。

O

文件被另一个工作拷贝的另一个用户锁定，只有在使用--show-updates 时显示。

T

文件锁定在工作拷贝，但是锁定被“窃取”而不可用。文件当前锁定在版本库，只有在使用--show-updates 时显示。

B

文件锁定在工作拷贝，但是锁定被“破坏”而不可用。文件当前锁定在版本库，只有在使用--show-updates 时显示。

过期信息出现在第七列（只在使用--show-updates 选项时出现）。

' '

这个项目在工作拷贝是最新的。

'*'

在服务器这个项目有了新的修订版本。

余下的字段是可变得宽度且使用空格分隔，如果使用--show-updates 或--verbose 选项，工作修订版本是下一个字段。

如果传递--verbose 选项,最后提交的修订版本和最后的提交作者会在后面显示。

工作拷贝路径永远是最后一个字段，所以它可以包括空格。

别名

stat, st

改变

无 2

是否访问版本库

只有使用--show-updates 时会访问

选项

```
--show-updates (-u)

--verbose (-v)

--non-recursive (-N)

--quiet (-q)

--no-ignore

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR

--ignore-externals
```

例子

这是查看你在工作拷贝所做的修改的最简单的方法。

```
$ svn status wc

 M     wc/bar.c

A  +   wc/qax.c
```

如果你希望找出工作拷贝哪些文件是最新的，使用--show-updates 选项（这不会对工作拷贝有任何修改）。这里你会看到 *wc/foo.c* 在上次更新后有了修改：

```
$ svn status --show-updates wc

 M          965    wc/bar.c

       *    965    wc/foo.c

A  +        965    wc/qax.c

Status against revision:    981
```

--show-updates *只会*在过期的项目（如果你运行 **svn update**，就会更新的项目）旁边安置一个星号。--show-updates 不会导致状态列表反映项目的版本库版本(尽管你可以通过--verbose 选项查看版本库的修订版本号）。

最后，是你能从 status 子命令得到的所有信息：

```
$ svn status --show-updates --verbose wc

 M          965       938 sally       wc/bar.c

       *    965       922 harry       wc/foo.c

A  +        965       687 harry       wc/qax.c
```

```
            965       687 harry       wc/zig.c

Head revision:   981
```

关于 **svn status** 的更多例子可以见“查看你的修改概况”一节。

名称

svn switch — 把工作拷贝更新到别的 URL。

概要

```
svn switch URL [PATH]

switch --relocate FROM TO [PATH...]
```

描述

这个子命令（没有--relocate 选项）更新你的工作拷贝来反映新的 URL —通常是一个与你的工作拷贝分享共同祖先的 URL，尽管这不是必需的。这是 Subversion 移动工作拷贝到分支的方式。更深入的了解请见“使用分支”一节。

--relocate 选项导致 **svn switch** 做不同的事情：它更新你的工作拷贝指向到*同一个*版本库目录，但是不同的 URL（通常因为管理员将版本库转移了服务器，或到了同一个服务器的另一个 URL）。

别名

SW

改变

工作拷贝 2

是否访问版本库

是

选项

```
--revision (-r) REV

--non-recursive (-N)

--quiet (-q)

--diff3-cmd CMD

--relocate FROM TO

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR
```

例子

如果你目前所在目录 *vendors* 分支到 *vendors-with-fix*，你希望转移到那个分支：

```
$ svn switch
http://svn.red-bean.com/repos/branches/vendors-with-fix .

U  myproj/foo.txt

U  myproj/bar.txt

U  myproj/baz.c

U  myproj/qux.c

Updated to revision 31.
```

为了跳转回来，只需要提供最初取出工作拷贝的版本库 URL：

```
$ svn switch http://svn.red-bean.com/repos/trunk/vendors .

U  myproj/foo.txt

U  myproj/bar.txt

U  myproj/baz.c

U  myproj/qux.c

Updated to revision 31.
```

如果你不希望跳转所有的工作拷贝，你可以只跳转一部分。

有时候管理员会修改版本库的“基本位置”—换句话说，版本库的内容并不改变，但是访问根的主 URL 变了。举个例子，主机名变了、URL 模式变了或者是 URL 中的任何一部分改变了。我们不选择重新检出一个工作拷贝，你可以使用 **svn switch** 来重写版本库所有 URL 的开头。使用--relocate 来做这种替换，没有文件内容会改变，访问的版本库也不会改变。只是像在工作拷贝.*svn*/运行了一段 Perl 脚本 **s/OldRoot/NewRoot/**。

```
$ svn checkout file:///tmp/repos test

A  test/a

A  test/b

…


$ mv repos newlocation

$ cd test/


$ svn update

svn: Unable to open an ra local session to URL

svn: Unable to open repository 'file:///tmp/repos'


$ svn switch --relocate file:///tmp/repos
file:///tmp/newlocation .

$ svn update

At revision 3.
```

小心使用--relocate 选项，如果你输入了错误的选项，你会在工作拷贝创建无意义的 URL，会导致整个工作区不可用并且难于修复。理解何时应该使用--relocate 也是非常重要的，下面是一些规则：

- 如果工作拷贝需要反映一个版本库的新目录，只需要使用 **svn switch**。
- 如果你的工作拷贝还是反映相同的版本库目录，但是版本库本身的位置改变了，使用 **svn switch --relocate**。

名称

svn unlock — 解锁工作拷贝路径或 URL。

概要

```
svn unlock TARGET...
```

描述

解锁每个 *TARGET*。如果任何另一个用户锁定了 *TARGET*，或者没有正确工作拷贝的锁定令牌，打印警告并继续解锁余下的 *TARGET*。使用--force 可以打破其它用户或工作拷贝的锁定。

别名

无

改变

锁定每个 *TARGET*。如果任何 *TARGET* 已经被另一个用户锁定，则会打印警告信息并且继续锁定剩下的 *TARGET*。可以使用--force 从其它用户来窃取锁定。

是否访问版本库

是

选项

```
--targets FILENAME

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR

--force
```

例子

解锁工作拷贝中的两个文件：

```
$ svn unlock tree.jpg house.jpg

'tree.jpg' unlocked.

'house.jpg' unlocked.
```

解锁工作拷贝的一个被其他用户锁定的文件：

```
$ svn unlock tree.jpg

svn: 'tree.jpg' is not locked in this working copy

$ svn unlock --force tree.jpg

'tree.jpg' unlocked.
```

没有工作拷贝时解锁一个文件：

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg

'tree.jpg unlocked.
```

更多细节见“锁定”一节。

名称

svn update — 更新你的工作拷贝。

概要

```
svn update [PATH...]
```

描述

svn update 会把版本库的修改带到工作拷贝，如果没有给定修订版本，它会把你的工作拷贝更新到 HEAD 修订版本，否则，它会把工作拷贝更新到你用--revision 指定的修订版本。为了保持同步，**svn update** 也会删除所有在工作拷贝发现的无效锁定（见“有时你只需要清理”一节）。

对于每一个更新的项目开头都有一个表示所做动作的字符，这些字符有下面的意思：

A

添加

D

删除

U

更新

C

冲突

G

合并

第一列的字符反映文件本身的更新，而第二列会反映文件属性的更新。

别名

up

改变

工作拷贝 2

是否访问版本库

是

选项

```
--revision (-r) REV

--non-recursive (-N)

--quiet (-q)
```

```
--no-ignore

--incremental

--diff3-cmd CMD

--username USER

--password PASS

--no-auth-cache

--non-interactive

--config-dir DIR

--ignore-externals
```

例子

获取你上次更新之后版本库的修改：

```
$ svn update

A  newdir/toggle.c

A  newdir/disclose.c

A  newdir/launch.c

D  newdir/README

Updated to revision 32.
```

你也可以将工作拷贝更新到旧的修订版本（Subversion 没有 CVS 的“sticky”文件的概念；见附录 B, *CVS 用户的 Subversion 指南*）：

```
$ svn update -r30

A  newdir/README

D  newdir/toggle.c
```

```
D  newdir/disclose.c

D  newdir/launch.c

U  foo.c

Updated to revision 30.
```

如果你希望检查单个文件的旧的修订版本，你会希望使用 **svn cat**。

svnadmin

svnadmin 是一个用来监控和修改 Subversion 版本库的管理工具，详情请见“svnadmin”一节。

因为 **svnadmin** 直接访问版本库（因此只可以在存放版本库的机器上使用），它通过路径访问版本库，而不是 URL。

svnadmin 选项

--bdb-log-keep

（Berkeley DB 特定）关闭数据库日志文件的自动删除，保留这些文件可以帮助你在灾难性版本库故障时更加便利。

--bdb-txn-nosync

（Berkeley DB 特定）在提交数据库事务时关闭 fsync。可以在 **svnadmin create** 命令创建 Berkeley DB 后端时开启 DB_TXN_NOSYNC（可以改进速度，但是有相关的风险）。

--bypass-hooks

绕过版本库钩子系统。

--clean-logs

删除不使用的 Berkeley DB 日志。

--force-uuid

缺省情况下，当版本库加载已经包含修订版本的数据时 **svnadmin** 会忽略流中的 UUID，这个选项会导致版本库的 UUID 设置为流的 UUID。

--ignore-uuid

缺省情况下，当加载空版本库时，**svnadmin** 会使用来自流中的 UUID，这个选项会导致忽略 UUID（如果你的配置文件已经设置了 --force-uuid，将会用于将其覆盖）。

--incremental

导出一个修订版本针对前一个修订版本的区别，而不是通常的完全结果。

--parent-dir *DIR*

当加载一个转储文件时，根路径为 *DIR* 而不是/。

--revision (-r) *ARG*

指定一个操作的修订版本。

--quiet

不显示通常的过程—只显示错误。

--use-post-commit-hook

当导入使用一个转储文件时，在每次新的修订版本产生时运行版本库 post-commit 钩子。

--use-pre-commit-hook

当加载一个转储文件时，每次新加修订版本之前运行版本库的 pre-commit 钩子。如果钩子失败，终止提交并中断加载进程。

svnadmin 子命令

名称

svnadmin create — 创建一个新的空的版本库。

概要

```
svnadmin create REPOS_PATH
```

描述

在提供的路径上创建一个新的空的版本库，如果提供的目录不存在，它会为你创建。[55]对于 Subversion 1.2，**svnadmin** 缺省使用 fsfs 文件系统后端创建版本库。

选项

```
--bdb-txn-nosync

--bdb-log-keep

--config-dir DIR

--fs-type TYPE
```

例子

创建一个版本库就是这样简单：

```
$ svnadmin create /usr/local/svn/repos
```

在 Subversion 1.0，一定会创建一个 Berkeley DB 版本库，在 Subversion 1.1，Berkeley DB 版本库是缺省类型，但是一个 FSFS 版本库也是可以创建，使用--fs-type 选项：

```
$ svnadmin create /usr/local/svn/repos --fs-type fsfs
```

---

[55] 记住 **svnadmin** 只工作在本地*路径*，而不是 *URL*。

名称

svnadmin deltify — 修订版本范围的路径的增量变化。

概要

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

描述

svnadmin deltify 因为历史原因而存在，这个命令已经废弃，不再需要。

它开始于当 Subversion 提供了管理员控制版本库压缩策略的能力，结果是复杂工作得到了*非常*小的收益，所以这个“特性”被废弃了。

选项

```
--revision (-r) REV

--quiet (-q)
```

名称

svnadmin dump — 将文件系统的内容转储到标准输出。

概要

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

描述

使用“dumpfile”可移植格式将文件系统的内容转储到标准输出，将反馈发送到标准错误，导出的修订版本从 *LOWER* 到 *UPPER*。如果没有提供修订版本，会导出所有的修订版本树，如果只提供 *LOWER*，导出一个修订版本树，通常的用法见“版本库数据的移植”一节。

缺省情况下，Subversion 的转储流包含了一个包括所有文件和目录的单独修订版本（请求的修订版本范围的第一个），后面是其它的只包含本修订所修改的文件和目录的修订版本（请求范围的其它版本）。对于修改的文件，转储文件包括所有的内容和属性，对于目录，包括所有的属性。

有一对有用的选项可以改变转储文件产生的方式，第一个是--incremental，使得第一个修订版本只显示其修改的文件和目录，而不是整个目录树，就像转储文件中其它的修订版本。这对产生一个准备导入到已经有数据的版本库时非常有用。

第二个有用的选项是--deltas，这个选项导致 **svnadmin dump** 不会保留修改文件的所有内容，而只是记录修改的部分。这样减少（有些情况下是非常大的）了 **svnadmin dump** 产生的转储文件的大小。然而，也有缺点—增量转储文件需要更多的 CPU 来创建，也不可以用 **svndumpfilter** 操作，也不如非增量文件容易被如 **gzip** 和 **bzip2** 等第三方工具压缩。

选项

```
--revision (-r) REV

--incremental

--quiet (-q)

--deltas
```

例子

转储整个版本库：

```
$ svnadmin dump /usr/local/svn/repos

SVN-fs-dump-format-version: 1

Revision-number: 0

* Dumped revision 0.

Prop-content-length: 56

Content-length: 56

…
```

从版本库增量转储一个单独的事务：

```
$ svnadmin dump /usr/local/svn/repos -r 21 --incremental

* Dumped revision 21.

SVN-fs-dump-format-version: 1

Revision-number: 21

Prop-content-length: 101

Content-length: 101

…
```

名称

svnadmin help — 求助！

概要

```
svnadmin help [SUBCOMMAND...]
```

描述

当你困于一个没有网络连接和本书的沙漠岛屿时，这个子命令非常有用。

别名

?, h

名称

svnadmin hotcopy — 制作一个版本库的热备份。

概要

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

描述

这个子命令会制作一个版本库的完全“热”拷贝，包括所有的钩子，配置文件，当然还有数据库文件。如果你传递--clean-logs选项，**svnadmin**会执行热拷贝操作，然后删除不用的 Berkeley DB 日志文件。你可以在任何时候运行这个命令得到一个版本库的安全拷贝，不管其它进程是否使用这个版本库。

选项

```
--clean-logs
```

就像“Berkeley DB”一节描述的，热拷贝的 Berkeley DB 版本库*不能*跨操作系统移植，也不能在不同“字节续”的主机上工作。

名称

svnadmin list-dblogs — 询问 Berkeley DB 在给定的 Subversion 版本库有哪些日志文件存在（只有在版本库使用 bdb 作为后端时使用）。

概要

```
svnadmin list-dblogs REPOS_PATH
```

描述

Berkeley DB 创建了记录所有版本库修改的日志，允许我们在面对大灾难时恢复。除非你开启了 DB_LOG_AUTOREMOVE，否则日志文件会累积，尽管大多数是不再使用可以从磁盘删除得到空间。详情见“管理磁盘空间”一节。

名称

svnadmin list-unused-dblogs — 询问 Berkeley DB 哪些日志文件可以安全的删除（只有在版本库使用 bdb 作为后端时使用）。

概要

```
svnadmin list-unused-dblogs REPOS_PATH
```

描述

Berkeley DB 创建了记录所有版本库修改的日志，允许我们在面对大灾难时恢复。除非你开启了 DB_LOG_AUTOREMOVE，否则日志文件会累积，尽管大多数是不再使用可以从磁盘删除得到空间。详情见“管理磁盘空间”一节。

例子

Berkeley DB 创建了记录所有版本库修改的日志，允许我们在面对大灾难时恢复。除非你开启了 DB_LOG_AUTOREMOVE，否则日志文件会累积，尽管大多数是不再使用，可以从磁盘删除得到空间。详情见“管理磁盘空间”一节。

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
```

```
## disk space reclaimed!
```

名称

svnadmin load — 从标准输入读进一个“svnadmin load”格式化的流。

概要

```
svnadmin load REPOS_PATH
```

描述

从标准输入读取格式化流“dumpfile”，提交新修订版本到版本库的文件系统，在标准输出返回进度。

选项

```
--quiet (-q)

--ignore-uuid

--force-uuid

--use-pre-commit-hook

--use-post-commit-hook

--parent-dir
```

例子

这里显示了加载一个备份文件到版本库(当然,使用 **svnadmin dump**):

```
$ svnadmin load /usr/local/svn/restored < repos-backup

<<< Started new txn, based on original revision 1

     * adding path : test ... done.

     * adding path : test/a ... done.

…
```

或者你希望加载到一个子目录：

```
$ svnadmin load --parent-dir new/subdir/for/project
/usr/local/svn/restored < repos-backup

<<< Started new txn, based on original revision 1

     * adding path : test ... done.

     * adding path : test/a ... done.

…
```

名称

svnadmin lslocks — 打印所有锁定的描述。

概要

```
svnadmin lslocks REPOS_PATH
```

描述

打印版本库所有锁定的描述。

选项

无

例子

显示了版本库*/svn/repos* 中一个锁定的文件：

```
$ svnadmin lslocks /svn/repos

Path: /tree.jpg

UUID Token:
opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753

Owner: harry

Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)

Expires:

Comment (1 line):

Rework the uppermost branches on the bald cypress in the
foreground.
```

名称

svnadmin lstxns — 打印所有未提交的事物名称。

概要

```
svnadmin lstxns REPOS_PATH
```

描述

打印所有未提交的事物名称。关于未提交事物是怎样创建和如何使用的信息见“删除终止的事务”一节。

例子

列出版本库所有突出的事物。

```
$ svnadmin lstxns /usr/local/svn/repos/

1w

1x
```

名称

svnadmin recover — 将版本库数据库恢复到稳定状态（只有在版本库使用 bdb 作为后端时使用），此外，如果 *repos/conf/passwd* 不存在，它会创建一个默认的密码文件。

概要

```
svnadmin recover REPOS_PATH
```

描述

在你得到的错误说明你需要恢复版本库时运行这个命令。

选项

```
--wait
```

例子

恢复挂起的版本库：

```
$ svnadmin recover /usr/local/svn/repos/

Repository lock acquired.

Please wait; recovering the repository may take some time...



Recovery completed.

The latest repos revision is 34.
```

恢复数据库需要一个版本库的独占锁（这是一个“数据库锁”；见“锁定”的三种含义），如果另一个进程访问版本库，**svnadmin recover** 会出错：

```
$ svnadmin recover /usr/local/svn/repos

svn: Failed to get exclusive repository access; perhaps another
process

such as httpd, svnserve or svn has it open?



$
```

--wait 选项可以导致 **svnadmin recover** 一直等待其它进程断开连接：

```
$ svnadmin recover /usr/local/svn/repos --wait

Waiting on repository lock; perhaps another process has it open?
```

```
### time goes by...

Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

名称

svnadmin rmlocks — 无条件的删除版本库的一个或多个锁定。

概要

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

描述

从 *LOCKED_PATH* 删除没个锁定。

选项

无

例子

这删除了版本库*/svn/repos* 里 *tree.jpg* 和 *house.jpg* 文件上的锁定：

```
$ svnadmin rmlocks /svn/repos tree.jpg house.jpg

Removed lock on '/tree.jpg.

Removed lock on '/house.jpg.
```

名称

svnadmin rmtxns — 从版本库删除事物。

概要

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

描述

删除版本库的事物，更多细节在“删除终止的事务”一节。

选项

```
--quiet (-q)
```

例子

删除命名的事物：

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

很幸运，**lstxns** 的输出作为 **rmtxns** 输入工作良好：

```
$ svnadmin rmtxns /usr/local/svn/repos/  `svnadmin lstxns
/usr/local/svn/repos/`
```

从版本库删除所有未提交的事务。

名称

svnadmin setlog — 设置某个修订版本的日志信息。

概要

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

描述

设置修订版本 REVISION 的日志信息为 FILE 的内容。

这与使用 **svn propset --revprop** 设置某一修订版本的 svn:log 属性效果一样，除了你也可以使用--bypass-hooks 选项绕过的所有 pre-或 post-commit 的钩子脚本，这在 pre-revprop-change 钩子脚本中禁止修改修订版本属性时非常有用。

修订版本属性不在版本控制之下的，所以这个命令会永久覆盖前一个日志信息。

选项

```
--revision (-r) REV
```

```
--bypass-hooks
```

例子

设置修订版本 19 的日志信息为文件 *msg* 的内容：

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

名称

svnadmin verify — 验证版本库保存的数据。

概要

```
svnadmin verify REPOS_PATH
```

描述

如果你希望验证版本库的完整性可以运行这个命令，这样会遍历版本库的所有的修订版本，导出修订版本并丢弃输出—有规律的执行这个命令来防止磁盘失败会是一个好方法，如果这个命令失败了—这是发生问题的第一个征兆—这表明你的版本库至少有一个损坏的修订版本，你必须从备份恢复损坏的修订版本（你需要备份，你没有吗？）。

例子

检验挂起的版本库：

```
$ svnadmin verify /usr/local/svn/repos/

* Verified revision 1729.
```

svnlook

svnlook 是检验 Subversion 版本库不同方面的命令行工具，它不会对版本库有任何修改—它只是用来“看”。**svnlook** 通常被版本库钩子使用，但是版本库管理也会发现它在诊断目的上也非常有用。

因为 **svnlook** 通过直接版本库访问（因此只可以在保存版本库的机器上工作）工作，所以他通过版本库的路径访问，而不是 URL。

如果没有指定修订版本或事物，**svnlook** 缺省的是版本库最年轻的（最新的）修订版本。

svnlook 选项

svnlook 的选项是全局的，就像 **svn** 和 **svnadmin**；然而，大多数选项只会应用到一个子命令，因为 **svnlook** 的功能是（有意的）限制在一定范围的。

--no-diff-deleted

> 防止 **svnlook** 打印删除文件的区别，缺省的行为方式是当一个文件在一次事物/修订版本中删除后，得到的结果与保留这个文件的内容变成空相同。

--revision (-r)

指定要进行检查的特定修订版本。

--revprop

操作针对修订版本属性，而不是 Subversion 文件或目录的属性。

这个选项需要你传递--revision（-r）参数。

--transaction (-t)

指定一个希望检查的特定事物 ID。

--show-ids

显示文件系统树中每条路径的文件系统节点修订版本 ID。

svnlook 子命令

名称

svnlook author — 打印作者。

概要

```
svnlook author REPOS_PATH
```

描述

打印版本库一个修订版本或者事物的作者。

选项

```
--revision (-r) REV
```

```
--transaction (-t)
```

例子

svnlook author 垂手可得，但是并不令人激动：

```
$ svnlook author -r 40 /usr/local/svn/repos

sally
```

名称

svnlook cat — 打印一个文件的内容。

概要

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

描述

打印一个文件的内容。

选项

```
--revision (-r) REV

--transaction (-t)
```

例子

这会显示事物 ax8 中一个文件的内容，位于*/trunk/README*：

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README


             Subversion, a version control system.

             ====================================


$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003)
$


Contents:


     I. A FEW POINTERS

    II. DOCUMENTATION

   III. PARTICIPATING IN THE SUBVERSION COMMUNITY

…
```

名称

svnlook changed — 打印修改的路径。

概要

```
svnlook changed REPOS PATH
```

描述

打印在特定修订版本或事物修改的路径，也是在前两列使用“svn update 样式的”状态字符：

'A '

条目添加到版本库。

'D '

条目从版本库删除。

'U '

文件内容改变了。

' U'

条目属性改变了，注意开头的空格。

'UU'

文件内容和属性修改了。

文件和目录可以区分，目录路径后面会显示字符'/'。

选项

```
--revision (-r) REV

--transaction (-t)
```

例子

这里显示了在测试版本库中修订版本 39 改变的文件和目录，注意修改的第一个项目是一个目录，证据就是结尾的/：

```
$ svnlook changed -r 39 /usr/local/svn/repos

A   trunk/vendors/deli/

A   trunk/vendors/deli/chips.txt

A   trunk/vendors/deli/sandwich.txt

A   trunk/vendors/deli/pickle.txt

U   trunk/vendors/baker/bagel.txt

 U  trunk/vendors/baker/croissant.txt

UU  trunk/vendors/baker/pretzel.txt

D   trunk/vendors/baker/baguette.txt
```

名称

svnlook date — 打印时间戳。

概要

```
svnlook date REPOS_PATH
```

描述

打印版本库一个修订版本或事物的时间戳。

选项

```
--revision (-r) REV

--transaction (-t)
```

例子

显示测试版本库修订版本 40 的日期：

```
$ svnlook date -r 40 /tmp/repos/

2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

名称

svnlook diff — 打印修改的文件和属性的区别。

概要

```
svnlook diff REPOS PATH
```

描述

打印版本库中 GNU 样式的文件和属性修改区别。

选项

```
--revision (-r) REV

--transaction (-t)

--no-diff-added
```

```
--no-diff-deleted
```

例子

这显示了一个新添加的（空的）文件，一个删除的文件和一个拷贝的文件：

```
$ svnlook diff -r 40 /usr/local/svn/repos/

Copied: egg.txt (from rev 39, trunk/vendors/deli/pickle.txt)



Added: trunk/vendors/deli/soda.txt

===================================================================



Modified: trunk/vendors/deli/sandwich.txt

===================================================================

--- trunk/vendors/deli/sandwich.txt      (original)

+++ trunk/vendors/deli/sandwich.txt      2003-02-22
17:45:04.000000000 -0600

@@ -0,0 +1 @@

+Don't forget the mayo!



Modified: trunk/vendors/deli/logo.jpg

===================================================================

(Binary files differ)
```

```
Deleted: trunk/vendors/deli/chips.txt

================================================================
==================


Deleted: trunk/vendors/deli/pickle.txt

================================================================
==================
```

如果一个文件有非文本的 svn:mime-type 属性，区别不会明确显示。

名称

svnlook dirs-changed — 打印本身修改的目录。

概要

```
svnlook dirs-changed REPOS_PATH
```

描述

打印本身修改（属性编辑）或子文件修改的目录。

选项

```
--revision (-r) REV

--transaction (-t)
```

例子

这显示了在我们的实例版本库中在修订版本 40 修改的目录：

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos

trunk/vendors/deli/
```

名称

svnlook help — 求助！

概要

```
也可以是 svnlook -h 和 svnlook -?。
```

描述

显示 svnlook 的帮助信息，这个命令如同 **svn help** 的兄弟，也是你的朋友，即使你从不调用它，并且忘掉了邀请它加入你的上一次聚会。

别名

?, h

名称

svnlook history — 打印版本库（如果没有路径，则是根目录）某一个路径的历史。

概要

```
svnlook history REPOS_PATH

          [PATH_IN_REPOS]
```

描述

打印版本库（如果没有路径，则是根目录）某一个路径的历史。

选项

```
--revision (-r) REV

--show-ids
```

例子

这显示了实例版本库中作为修订版本 20 的路径*/tags/1.0* 的历史输出。

```
$ svnlook history -r 20 /usr/local/svn/repos /tags/1.0
--show-ids

REVISION   PATH <ID>

--------   ---------

     19   /tags/1.0 <1.2.12>

     17   /branches/1.0-rc2 <1.1.10>

     16   /branches/1.0-rc2 <1.1.x>

     14   /trunk <1.0.q>

     13   /trunk <1.0.o>

     11   /trunk <1.0.k>
```

```
      9   /trunk <1.0.g>
      8   /trunk <1.0.e>
      7   /trunk <1.0.b>
      6   /trunk <1.0.9>
      5   /trunk <1.0.7>
      4   /trunk <1.0.6>
      2   /trunk <1.0.3>
      1   /trunk <1.0.2>
```

名称

svnlook info — 打印作者、时间戳、日志信息大小和日志信息。

概要

```
svnlook info REPOS_PATH
```

描述

打印作者、时间戳、日志信息大小和日志信息。

选项

```
--revision (-r) REV

--transaction (-t)
```

例子

显示了你的实例版本库在修订版本 40 的信息输出。

```
$ svnlook info -r 40 /usr/local/svn/repos

sally

2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)

15

Rearrange lunch.
```

名称

svnlook lock — 如果版本库路径已经被锁定，描述它。

概要

```
svnlook lock REPOS PATH PATH IN REPOS
```

描述

打印 *PATH_IN_REPOS* 锁定的所有信息，如果 *PATH_IN_REPOS* 没有锁定，则不打印任何内容。

选项

无

例子

这描述了文件 *tree.jpg* 的锁定。

```
$ svnlook lock /svn/repos tree.jpg

UUID Token:
opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753

Owner: harry

Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)

Expires:

Comment (1 line):

Rework the uppermost branches on the bald cypress in the
foreground.
```

名称

svnlook log — 打印日志信息。

概要

```
svnlook log REPOS_PATH
```

描述

打印日志信息。

选项

```
--revision (-r) REV

--transaction (-t)
```

例子

这显示了实例版本库在修订版本 40 的日志输出：

```
$ svnlook log /tmp/repos/

Rearrange lunch.
```

名称

svnlook propget — 打印版本库中一个路径一个属性的原始值。

概要

```
svnlook propget REPOS PATH PROPNAME [PATH IN REPOS]
```

描述

列出版本库中一个路径一个属性的值。

别名

pg, pget

选项

```
--revision (-r) REV

--transaction (-t)

--revprop
```

例子

这显示了 HEAD 修订版本中文件*/trunk/sandwich* 的“seasonings”属性的值：

```
$ svnlook pg /usr/local/svn/repos seasonings /trunk/sandwich
mustard
```

名称

svnlook proplist — 打印版本化的文件和目录的属性名称和值。

概要

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

描述

列出版本库中一个路径的属性，使用--verbose 选项也会显示所有的属性值。

别名

pl, plist

选项

```
--revision (-r) REV
--transaction (-t)
--verbose (-v)
```

```
--revprop
```

例子

这显示了 HEAD 修订版本中*/trunk/README* 的属性名称：

```
$ svnlook proplist /usr/local/svn/repos /trunk/README
  original-author
  svn:mime-type
```

这与前一个例子是同一个命令，但是同时显示了属性值：

```
$ svnlook --verbose proplist /usr/local/svn/repos /trunk/README
  original-author : fitz
  svn:mime-type : text/plain
```

名称

svnlook tree — 打印树。

概要

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

描述

打印树，从 *PATH_IN_REPOS*（如果提供，会作为树的根）开始，可以选择显示节点修订版本 ID。

选项

```
--revision (-r) REV

--transaction (-t)

--show-ids
```

例子

这会显示实例版本库中修订版本 40 的树输出（包括节点 ID）：

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids

/ <0.0.2j>

 trunk/ <p.0.2j>

  vendors/ <q.0.2j>

   deli/ <1g.0.2j>

    egg.txt <1i.e.2j>

    soda.txt <1k.0.2j>

    sandwich.txt <1j.0.2j>
```

名称

svnlook uuid — 打印版本库的 UUID。

概要

```
svnlook uuid REPOS_PATH
```

描述

打印版本库的 UUID，UUID 是版本库的 *u*niversal *u*nique *ID*entifier（全局唯一标示），Subversion 客户端可以使用这个标示区分不同的版本库。

例子

```
$ svnlook uuid /usr/local/svn/repos

e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

名称

svnlook youngest — 显示最年轻的修订版本号。

概要

```
svnlook youngest REPOS_PATH
```

描述

打印一个版本库最年轻的修订版本号。

例子

这显示了在实例版本库显示最年轻的修订版本：

```
$ svnlook youngest /tmp/repos/

42
```

svnsync

svnsync 是 Subversion 的远程版本库镜像工具，它允许你把一个版本库的内容录入到另一个。

在任何镜像场景中，有两个版本库：源版本库，镜像（或“sink”）版本库，源版本库就是 **svnsync** 获取修订版本的库，镜像版本库是源版本库修订版本的目标，两个版本库可以是在本地或远程—它们只是通过 URL 跟踪。

svnsync 进程只需要对源版本库有读权限；它不会尝试修改它。但是很明显，**svnsync** 可以读写访问镜像版本库。

svnsync 对于不能作为镜像操作一部分的修改非常敏感，为了防止发生这个情况，最好保证 **svnsync** 是唯一可以修改镜像版本库的进程。

svnsync 选项

`--config-dir` *DIR*

指导 Subversion 从指定目录而不是默认位置（用户主目录的.*subversion*）读取配置信息。

`--no-auth-cache`

阻止在 Subversion 管理区缓存认证信息（如用户名密码）。

`--non-interactive`

如果认证失败，或者是不充分的凭证时，防止出现要求凭证的提示（例如用户名和密码）。这在运行自动脚本时非常有用，只是让 Subversion 失败而不是提示更多的信息。

--password *PASS*

指出在命令行中提供你的密码—另外，如果它是需要的，Subversion 会提示你输入。

--username *NAME*

表示你要在命令行提供认证的用户名—否则如果需要，Subversion 会提示你这一点。

svnsync 子命令

下面是一些子命令：

名称

svnsync copy-revprops — 从源版本库拷贝所有的修订版本属性到镜像版本库。

概要

```
svnsync copy-revprops DEST_URL REV
```

描述

因为 Subversion 修订版本属性可以在任何时候修改，很有可能有一些修订版本的属性会在已经同步后改变，因为 **svnsync synchronize** 不会对没有同步的修订版本范围进行操作，而不会注意修改范围之外的属性修改。这样导致了修订版本属性在源版本库与目标版本库的偏离，**svnsync copy-revprops** 是这个问题的答案，用它可以同步特定修订版本的属性。

别名

无

选项

```
--non-interactive

--no-auth-cache

--username NAME

--password PASS

--config-dir DIR
```

例子

为单个修订版本重新同步修订版本属性：

```
$ svnsync copy-revprops file:///opt/svn/repos-mirror 6

Copied properties for revision 6.

$
```

名称

svnsync initialize — 为与另一个版本库的同步初而始化目标版本库。

概要

```
svnsync initialize DEST URL SOURCE URL
```

描述

svnsync initialize 检验版本库是否满足了新镜像版本库的需求—它必须没有存在的版本历史，并允许修订版本修改—记录镜像版本库与源版本库关联的初始管理信息，这是对即将镜像的版本库的第一个 **svnsync** 操作。

别名

init

选项

```
--non-interactive

--no-auth-cache

--username NAME

--password PASS

--config-dir DIR
```

例子

因为无法修改修订版本属性而初始化镜像版本库失败：

```
$ svnsync initialize file:///opt/svn/repos-mirror
http://svn.example.com/repos

svnsync: Repository has not been enabled to accept revision
propchanges;

ask the administrator to create a pre-revprop-change hook

$
```

以镜像初始化版本库，包含已创建允许所有修订版本属性修改的 pre-revprop-change 钩子：

```
$ svnsync initialize file:///opt/svn/repos-mirror
http://svn.example.com/repos

Copied properties for revision 0.

$
```

名称

svnsync synchronize — 将所有未完成的修订版本从源版本库转移到镜像版本库。

概要

```
svnsync synchronize DEST URL
```

描述

svnsync synchronize 命令做了版本库镜像工作的所有体力活，通过讯问镜像版本库来查看已经拷贝的修订版本，然后开始拷贝未镜像修订版本到镜像版本库。

svnsync synchronize 可以优雅的取消并重新开始。

别名

sync

选项

```
--non-interactive

--no-auth-cache

--username NAME

--password PASS

--config-dir DIR
```

例子

从源版本库拷贝未同步修订版本到镜像版本库：

```
$ svnsync synchronize file:///opt/svn/repos-mirror

Committed revision 1.

Copied properties for revision 1.

Committed revision 2.

Copied properties for revision 2.

Committed revision 3.
```

```
Copied properties for revision 3.

…

Committed revision 45.

Copied properties for revision 45.

Committed revision 46.

Copied properties for revision 46.

Committed revision 47.

Copied properties for revision 47.

$
```

svnserve

当对远程源版本库使用 **svnsync** 时，使用 Subversion 的自定义网络协议。

svnserve 允许 Subversion 版本库使用 svn 网络协议，你可以作为独立服务器进程运行 svnserve，或者是使用其它进程，如 **inetd**、**xinetd**（也是 svn://）或使用 svn+ssh://访问方法的 **sshd** 为你启动进程。

一旦客户端已经选择了一个版本库来传递它的 URL，**svnserve** 会读取版本库目录的 *conf/svnserve.conf* 文件，来检测版本库特定的设置，如使用哪个认证数据库和应用怎样的授权策略。关于 *svnserve.conf* 文件的详情见“svnserve，一个自定义的服务器”一节。

svnserve 选项

不象前面描述的例子，**svnserve** 没有子命令—**svnserve** 完全通过选项控制。

--daemon (-d)

导致 **svnserve** 以守护进程方式运行，**svnserve** 维护本身并且接受和服务 svn 端口（缺省 3690）的 TCP/IP 连接。

--listen-port=*PORT*

在守护进程模式时导致 svnserve 监听 *PORT* 端口。（FreeBSD 守护进程缺省只监听 tcp6—这个选项告诉他们监听 tcp4。）

--listen-host=*HOST*

svnserve 监听的 *HOST*，可能是一个主机名或是一个 IP 地址。

--foreground

当与-d一起使用，会导致 **svnserve** 停留在前台，主要用来调试。

--inetd (-i)

导致 **svnserve** 使用标准输出/标准输入文件描述符，更准确的是使用 **inetd** 作为守护进程。

--help (-h)

显示有用的摘要和选项。

--version

显示版本信息，版本库后端存在和可用的模块列表。

--root=*ROOT* (-r=*ROOT*)

设置 **svnserve** 服务的版本库的虚拟根，客户端提供的 URL 中显示的路径会解释为这个根的相对路径，不会允许离开这个根。

--tunnel (-t)

导致 **svnserve** 以管道模式运行，很像 **inetd** 操作的模式（两种模式都维护标准输入/标准输出的连接），除了连接是用当前 uid 的用户名预先认证过的这一点。这个选项在客户端使用如 **ssh** 之类的管道时自动传递，这意味着你很少有必要再去传递这个参数给 **svnserve**，所以如果你发现在命令行输入了 svnserve --tunnel，并想知道接下来怎么做，可以看“SSH 隧道”一节。

--tunnel-user NAME

与--tunnel 选项结合使用；告诉 svnserve 假定 *NAME* 就是认证用户，而不是 svnserve 进程的 UID 用户，当希望多个用户通过 SSH 共享同一个系统帐户，但是维护各自的提交标示符时非常有用。

--threads (-T)

当以守护进程模式运行，导致 **svnserve** 为每个连接产生一个线程而不是一个进程，**svnserve** 进程本身在启动后会一直在后台。

--listen-once (-X)

导致 **svnserve** 在 svn 端口接受一个连接，维护完成它退出。这个选项主要用来调试。

svnversion

名称

svnversion — 总结工作拷贝的本地修订版本。

概要

```
svnversion [OPTIONS] [WC PATH [TRAIL URL]]
```

描述

svnversion 是用来总结工作拷贝修订版本混合的程序，结果修订版本号或范围会写到标准输出。

通常在构建过程中利用其输出定义程序的版本号码。

如果提供 *TRAIL_URL*，URL 的尾端部分用来监测是否 *WC_PATH* 本身已经跳转（监测 *WC_PATH* 的跳转不需要依赖 *TRAIL_URL*）。

当没有定义 *WC_PATH*，会使用当前路径作为工作拷贝路径，如果没有显式定义 *WC_PATH*，*TRAIL_URL* 将无法定义。

选项

像 **svnserve，svnversion** 没有子命令，只有选项。

--no-newline (-n)

忽略输出的尾端新行。.

--committed (-c)

使用最后修改修订版本而不是当前的（例如，本地存在的最高修订版本）修订版本。

--help (-h)

打印帮助摘要。

--version

打印 **svnversion**，如果没有错误则退出。

例子

如果工作拷贝都是一样的修订版本（例如，在更新后那一刻），会打印修订版本：

```
$ svnversion
4168
```

添加 *TRAIL_URL* 来展示工作拷贝不是从你希望的地方跳转过来的，注意这个命令需要 *WC_PATH*：

```
$ svnversion . /repos/svn/trunk

4168
```

对于混合修订版本的工作拷贝，修订版本的范围会被打印：

```
$ svnversion

4123:4168
```

如果工作拷贝包含修改，后面会紧跟一个"M"：

```
$ svnversion

4168M
```

如果工作拷贝已经跳转，后面会有一个"S"：

```
$ svnversion

4168S
```

因此，这里是一个混合修订版本，跳转的工作拷贝包含了一些本地修改：

```
$ svnversion

4212:4168MS
```

如果从一个目录而不是工作拷贝调用，**svnversion** 假定它是一个导出的工作拷贝并且打印"exported"：

```
$ svnversion

exported
```

mod_dav_svn

名称

mod_dav_svn Configuration Directives — Apache 通过 Apache HTTP 服务器用来维护 Subversion 版本库配置指示。

描述

这个小节主要描述了 Subversion Apache 配置的每个指示，关于 Apache 配置 Subversion 的更多信息见“httpd，Apache 的 HTTP 服务器”一节。

指示

DAV svn

> 这个指示必须包含在所有 Subversion 版本库的 Directory 或 Location 块中,它告诉 httpd 使用 Subversion 的后端,用 mod_dav 来处理所有的请求。

SVNAutoversioning On

> 这个指示允许 WebDAV 客户端的请求导致自动提交，每个修订版本会产生一个普通的日志信息。如果你开启了自动版本化，你很可能需要设置 ModMimeUsePathInfo On，这样 mod_mime 可以自动的（像 mod_mime 一样好，当然）将 svn:mime-type 设置为正确的 mime-type 值。更多信息见附录 C,*WebDAV 和自动版本*。

SVNPath

这个指示指定 Subversion 版本库文件文件系统的位置，在一个 Subversion 版本库的配置块里，必须提供这个指示或 SVNParentPath，但不能同时存在。

SVNSpecialURI

指定特定 Subversion 资源的 URI 部分(命名空间),缺省是“!svn”，大多数管理员不会用到这个指示。只有那些必须要在版本库中放一个名字为*!svn* 的文件时需要设置。如果你在一个已经使用中的服务器上这样修改，它会破坏所有的工作拷贝，你的用户会拿着叉子和火把追杀你。

SVNReposName

指定 Subversion 版本库在 HTTP GET 请求中使用的名字，这个值会作为所有目录列表（当你用 web 浏览器察看 Subversion 版本库时会看到）的标题，这个指示是可选的。

SVNIndexXSLT

目录列表所使用的 XSL 转化的 URI，这个指示可选。

SVNParentPath

指定子目录会是版本库的父目录在文件系统的位置，在一个 Subversion 版本库的配置块里，必须提供这个指示或 SVNPath，但不能同时存在。

SVNPathAuthz

控制开启和关闭路径为基础的授权，更多细节见“禁用基于路径的检查”一节。

Subversion 属性

Subversion 允许用户在文件或目录上发明任意名称的版本化属性和非版本化属性，唯一的限制就是“svn:”是 Subversion 本身的保留前缀，用户可以设置这些属性来改变 Subversion 的行为方式，用户不能发明新的“svn:”属性。

版本控制的属性

svn:executable

如果出现在一个文件上，客户端会将此文件在 Unix 工作拷贝中设置为可执行，见“文件的可执行性”一节。

svn:mime-type

如果出现在一个文件，这个值表示了文件的 mime-type，这允许客户端在执行更新时决定以行为依据的合并是否安全，同时也会影响使用浏览器浏览文件时的行为方式。见“文件内容类型”一节。

svn:ignore

如果出现在目录上，这是一组 **svn status** 可以忽略的未版本化文件的名称模式，见“忽略未版本控制的条目”一节。

svn:keywords

如果出现在一个文件上，这个值告诉客户端如何扩展文件的特定关键字，见“关键字替换”一节。

svn:eol-style

如果出现在一个文件上，这个值告诉客户端如何处理工作拷贝中的文件的行结束符，见“行结束字符串”一节和 svn export。

svn:externals

如果出现在一个目录上，则这个值就是客户端必须要检出的路径和 URL 列表。见“外部定义”一节。

svn:special

如果出现在一个文件上，表示了那个文件不是一个普通的文件，而是一个符号链或者是其他特殊的对象[56]。

svn:needs-lock

如果出现在一个文件上，告诉客户端在工作拷贝将文件置为只读，可以提醒我们在修改以前必须解锁。见“锁定交流”一节。

未版本控制的属性

svn:author

如果出现，则保存了创建这个修订版本的认证用户名。（如果没有出现，则修订版本是匿名提交的。）

svn:date

保存了 ISO 8601 格式的修订版本创建 UTC 时间，这个值来自服务器主机时钟，不是客户端的。

svn:log

保存了描述修订版本的日志信息。

svn:autoversioned

如果出现，则修订版本是通过自动版本化特性创建，见“自动版本化”一节。

版本库钩子

名称

start-commit — 开始提交的通知

描述

start-commit 在开始事务之前执行，通常是用来确定用户是否有提交权限。

如果 start-commit 钩子程序返回非零值，提交就会在创建之前停止，标准错误的任何输出都会返回到客户端。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 认证过的尝试提交的用户名

普通用户

访问控制

名称

pre-commit — 在提交结束之前提醒。

描述

pre-commit hook 在事务完成提交之前运行，通常这个钩子是用来保护因为内容或位置（例如，你要求所有到一个特定分支的提交必须包括一个 bug 追踪的 ticket 号，或者是要求日志信息不为空）而不允许的提交。

如果 pre-commit 钩子返回非零值，提交会退出，提交事务被删除，所有标准错误的输出返回到客户端。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 提交事务的名称

普通用户

修改确认和控制

名称

post-commit — 成功提交的通知。

描述

post-commit hook 在事务完成后运行，创建一个新的修订版本。大多数人用这个钩子来发送关于提交的描述性电子邮件，或者作为版本库的备份。

post-commit 钩子程序的返回值和输出被忽略。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 提交创建的版本号

普通用户

提交通知，工具集成

名称

pre-revprop-change — 修订版本属性修改的通知。

描述

pre-revprop-change 钩子在修订版本属性修改之前，正常提交范围之外被执行，不象其他钩子，这个钩子默认是拒绝所有的属性修改，钩子必须实际存在并且返回一个零值，这样属性修改才能实现。

如果 pre-revprop-change 钩子没有实现或返回一个非零值，对属性的修改就不会成功，所以的标准错误输出会返回到客户端。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 要修改属性的修订版本
3. 企图修改属性的认证用户名
4. 属性名称已修改
5. 变更描述：A (添加的)，D (删除的)或 M (修改的)

此外，Subversion 通过标准输入将属性值传递给钩子程序。

普通用户

访问控制，修改确认和控制

名称

post-revprop-change — 修订版本属性修改成功的通知

描述

post-revprop-change 钩子会在修订版本属性修改后立即执行，在提交范围之外。可以从其对应物 pre-revprop-change 知道，如果没有实现 pre-revprop-change 钩子就不会执行。它通常用来在属性修改后发送邮件通知。

post-revprop-change 的返回值和输出会被忽略。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 已经修改属性的修订版本
3. 做出修改的认证用户名
4. 属性名称已修改
5. 变更描述：A (添加的)，D (删除的)或 M (修改的)

此外，Subversion 通过标准输入将属性的前一个值传递给钩子。

普通用户

属性修改通知

名称

pre-lock — 路径锁定尝试的通知。

描述

这个钩子会在每次有人尝试锁定文件时执行，可以防止完全的锁定，或者用来指定控制哪些用户可以锁定特定路径的复杂策略，如果钩子发现已存在的钩子，也可以决定是否“窃取”这个钩子。

如果 pre-lock 钩子返回非零值，锁定动作会退出，并将标准错误返回到客户端。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 将要锁定的版本化路径
3. 尝试锁定的认证用户名

普通用户

访问控制

名称

post-lock — 成功锁定路径的通知。

描述

post-lock 在路径锁定后执行，通常用来发送锁定事件邮件通知。

post-unlock 钩子程序的输出和返回值会被忽略。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 锁定路径的认证用户名

此外，锁定路径通过标准输入传递给钩子程序，每行一个路径。

普通用户

加锁通知

名称

pre-unlock — 路径解锁尝试的通知。

描述

这个钩子在某人企图删除一个文件上的钩子时发生，可以用来制定哪些用户可以解除文件锁定的策略。制定破坏锁定的策略非常重要，如果一个用户 A 锁定了一个文件，允许用户 B 打开这个锁？如果这个锁已经一周了呢？这种事情可以通过钩子决定并执行。

如果 pre-unlock 返回非零值，解锁过程就会退出，标准错误返回到客户端。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 将要锁定的版本化路径
3. 尝试锁定的认证用户名

普通用户

访问控制

名称

post-unlock — 路径成功解锁的通知。

描述

post-unlock 在一个或多个路径已经被解锁后执行，通常用来发送解锁事件通知邮件。

post-unlock 钩子程序的输出和返回值会被忽略。

输入参数

传递给你钩子程序的命令行参数，按照顺序是：

1. 版本库路径
2. 路径解锁的认证用户名

此外，解锁路径通过标准输入传递给钩子程序，每行一个路径。

普通用户

解锁通知

---

[54] 是的，使用--version 选项不需要子命令，几分钟后我们会到达那个部分。

[55] 记住 **svnadmin** 只工作在本地*路径*，而不是 *URL*。

[56] 此时，符号链是唯一的“特别”对象，但是以后，也许 Subversion 会有更多的特别对象。

Subversion 快速入门指南

目录

如果你渴望快速配置 Subversion 并运行（而且你喜欢通过实验学习），本章会展示如何创建版本库，导入代码，然后以工作拷贝检出，继续我们会给出本书的相关章节的链接。

如果读者还不熟悉版本控制，以及在 Subversion 和 CVS 中使用的“拷贝-修改-合并”模型这些基础的概念，那么建议在进一步学习之前，首先阅读第 1 章 *基本概念*。

安装 Subversion

Subversion 是基于 APR 构建的。APR 全称为 Apache Portable Runtime library，是一个移植性很好的程序库。APR 库提供了全部与操作系统相关的操作接口，如磁盘访问、内存管理等等，这使得 Subversion 自身能够在不加修改的情况下运行于不同的操作系统之上。Subversion 对 APR 的依赖并不意味着必须使用 Apache 作为它的网络服务器程序，相反，Apache 只是 Subversion 支持的网络服务器程序之一。APR 是一个独立的程序库，任何应用程序都可以使用它（Apache 也是基于它开发的）。这就是说，Subversion 能够在所有可运行 Apache 服务器的操

作系统上运转，如 Windows、Linux、各种 BSD、Mac OS X、Netware 等等。

最简单的安装 Subversion 的方法就是下载与你的操作系统对应的二进制程序包。在 Subversion 的网站（http://subversion.tigris.org）上通常可以找到由志愿者提供下载的程序包。在这个网站上，会提供微软操作系统上的图形化应用程序安装包。而对于类 Unix 系统，则可以使用其自身的程序包系统（PRMs、DEBs、ports tree 等等）来获取 Subversion。

此外，还可以通过编译源代码包直接生成 Subversion 程序，尽管这不是一件简单的任务（如果你没有构建过开源软件包，你最好下载二进制发布版本）。首先，从 Subversion 网站下载最新的源代码包，然后解压缩。然后，根据 *INSTALL* 文件的指示进行编译。需要注意的是，正式发布的源代码包中可能没有包含构建命令行客户端工具所需的全部内容，从 Subversion1.4 开始，所有依赖的库（如 apr，apr-util 和 neno 库）以*-deps* 为名称单独发布，这些库应该可以满足你在你的系统上的安装，你需要将依赖库解压缩到 Subversion 源程序相同的目录。但是一些可选的组件则依赖于其它一些程序库，如 Berkeley DB 和 Apache httpd。因此，如果想要进行完整的编译，请根据 *INSTALL* 文件中的内容确认这些程序库是否可用。

如果你是一个喜欢使用最新软件的人，你可以从 Subversion 本身的版本库得到 Subversion 最新的源代码，显然，你首先需要一个 Subversion

客户端，有了之后，你就可以从 http://svn.collab.net/repos/svn/trunk/ 检出一个 Subversion 源代码的工作拷贝：[57]

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
subversion

A    subversion/HACKING

A    subversion/INSTALL

A    subversion/README

A    subversion/autogen.sh

A    subversion/build.conf

…
```

上面的命令会检出一个流血的，最新的 Subversion 源代码版本到你的叫做 *subversion* 的当前工作目录。很明显，你可以调整最后的参数改为你需要的。不管你怎么称呼你的新的工作拷贝目录，在操作之后，你现在已经有了 Subversion 的源代码。当然，你还是需要得到一些帮助库（apr，apr-util 等等）—见工作拷贝根目录的 *INSTALL* 来得到更多细节。

快速指南

“请确定你坐在了正确的位置，你的盘桌已经关闭，乘务员们，准备起飞…。”

这是一个非常高层次的教程，能够帮助你熟悉 Subversion 的基本配置和操作，在结束这个教程时，你一定能够对 Subversion 的典型使用有了一个基础的认识。

运行下面的例子需要首先正确安装 Subversion 客户端程序 **svn** 以及管理工具 **svnadmin**，并且必须为 1.2 或更新版本的 Subversion 程序（可以运行 **svn --version** 来检查 Subversion 的版本。）

Subversion 的所有版本化数据都储存在中心版本库中。因此首先，我们需要创建一个版本库：

```
$ svnadmin create /path/to/repos
$ ls /path/to/repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

这个命令创建了一个新目录*/path/to/repos*，并在其中创建了一个 Subversion 版本库。这个目录里主要保存了一些数据库文件（还有其它一些文件），而不像 CVS 那样保存着版本化的文件。需要更多版本库创建和维护方面的内容，参见第 5 章 *版本库管理*。

在 Subversion 没有“项目”的概念。Subversion 的版本库只是一个虚拟的版本化文件系统，可以存放你想要存放的任何文件。有些管理员喜欢为每个项目建立一个独立的版本库，而另外一些管理员则喜欢将多个项目存放到同一个版本库的不同目录里。这两种方式各有各的优点，关于这方面内容的叙述，参见“规划你的版本库结构”一节。不论是哪一种方式，版本库都只是负责管理文件和目录，而“项目”则是人为指定的概念。因此，尽管本书中遍布着项目这个词，但是请记住我们只不过是在谈论版本库中的某些特定目录（或者是一组目录）。

在这个例子中，我们假定已经有一些需要导入到 Subversion 版本库的条目（一组文件和目录）。接下来，我们需要把这些条目整理到一个名为 *myproject* 的目录（或者其它任意目录）里。在这个目录下，创建三个顶级子目录：*branches*、*tags* 和 *trunk*，这样做的原因将在后文中阐述。之后，将所有需版本化的数据保存到 *trunk* 目录下，同时保持 *branches* 和 *tags* 目录为空：

```
/tmp/myproject/branches/
/tmp/myproject/tags/
/tmp/myproject/trunk/
                    foo.c
                    bar.c
                    Makefile
                    …
```

branches、*tags* 和 *trunk* 这三个子目录不是 Subversion 必须的。但这样做是 Subversion 的习惯用法，我们还是遵守这个约定吧。

准备好了数据之后，就可以使用 **`svn import`** 命令（参见“导入数据到你的版本库”一节）将其导入到版本库中：

```
$ svn import /tmp/myproject file:///path/to/repos/myproject -m
"initial import"
Adding         /tmp/myproject/branches
Adding         /tmp/myproject/tags
Adding         /tmp/myproject/trunk
```

```
Adding         /tmp/myproject/trunk/foo.c
Adding         /tmp/myproject/trunk/bar.c
Adding         /tmp/myproject/trunk/Makefile
…
Committed revision 1.
$
```

现在版本库中已经保存了目录中的数据。如前所述，直接查看版本库是看不到文件和目录的；它们存放在数据库之中。但是版本库的虚拟文件系统中则包含了一个名为 *myproject* 的顶级目录，其中依此保存了所有的数据。

注意我们在一开始创建的那个*/tmp/myproject* 目录并没有改变，Subversion 并不在意它（事实上，完全可以删除这个目录）。要开始使用版本库数据，我们还要创建一个新的用于存储数据的“工作拷贝”，这是一个私有工作区。从 Subversion 版本库里“检出”一个 *myproject/trunk* 目录工作拷贝的操作如下：

```
$ svn checkout file:///path/to/repos/myproject/trunk myproject
A  myproject/foo.c
A  myproject/bar.c
A  myproject/Makefile
…
Checked out revision 1.
```

现在，在 *myproject* 目录下生成了一个版本库数据的独立拷贝。我们可以在这个工作拷贝中编辑文件，并将修改提交到版本库中。

- 进入工作拷贝目录，编辑某个文件的内容。
- 运行 **`svn diff`** 以标准差别格式查看修改的内容。
- 运行 **`svn commit`** 将更改提交到版本库中。
- 运行 **`svn update`** “更新”工作拷贝。

完整的工作拷贝操作指南，请参见第 2 章 *基本使用*。

现在，Subversion 版本库可以通过网络方式访问。参考第 6 章 *服务配置*，了解不同服务器软件的使用以及配置方法。

---

[57] 注意上面例子中检出的 URL 并不是以 svn 结尾，而是它的一个叫做 trunk 的子目录，可以看我们对 Subversion 的分支和标签模型的讨论来理解背后的原因。

CVS 用户的 Subversion 指南

目录

版本号现在不同了

目录的版本

更多离线操作

这个附录可以作为 CVS 用户开始使用 Subversion 的指南，实质上就是鸟瞰这两个系统之间的区别列表，在每一小节，我们会尽可能提供相关章节的引用。

尽管 Subversion 的目标是接管当前和未来的 CVS 用户基础，需要一些新的特性设计来修正一些 CVS“不好的”行为习惯，这意味着，作为一个 CVS 用户，你或许需要打破习惯—忘记一些奇怪的习惯来作为开始。

版本号现在不同了

在 CVS 中，修订版本号是每文件的，这是因为 CVS 使用 RCS 文件保存数据，每个文件都在版本库有一个对应的 RCS 文件，版本库几乎就是根据项目树的结构创建。

在 Subversion，版本库看起来像是一个单独的文件系统，每次提交导致一个新的文件系统；本质上，版本库是一堆树，每棵树都有一个单独的修订版本号。当有人谈论“修订版本 54”时，他们是在讨论一个特定的树（并且间接来说，文件系统在提交 54 次之后的样子）。

技术上讲，谈论“文件 *foo.c* 的修订版本 5”是不正确的，相反，一个人会说“*foo.c* 在修订版本 5 出现”。同样，我们在假定文件的进展时也要小心，在 CVS，文件 *foo.c* 的修订版本 5 和 6 一定是不同的，在 Subversion，*foo.c* 可能在修订版本 5 和 6 之间*没有*改变。

类似的，在 CVS 中标签或分支是文件的一种标注，或者是单个文件的版本信息，而在 Subversion 中，标签和分支是整个目录树的拷贝（为了方便，进入版本库顶级目录的*/branches* 或*/tags* 子目录，*/trunk* 旁边）。版本库作为一个整体，每个文件的许多版本可见：每个分支的最新版本，每个标签的最新版本以及 trunk 本身的最新版本。所以，我们再精炼一下术语，我们说“*foo.c* 在修订版本 5 出现在*/branches/REL1*。”

更多细节见“修订版本”一节.

目录的版本

Subversion 会记录目录树的结构，不仅仅是文件的内容。这是编写 Subversion 替代 CVS 最重要的一个原因。

以下是对你这意味着什么的说明，作为一个前 CVS 用户：

- **`svn add`** 和 **`svn delete`** 现在也工作在目录上了，就像在文件上一样，还有 **`svn copy`** 和 **`svn move`** 也一样。然而，这些命令*不*会导致版本库即时的变化，相反，工作的项目只是“预定要”添加和删除，在运行 **`svn commit`** 之前没有版本库的修改。
- 目录不再是哑容器了；它们也有文件一样的修订版本号。（更准确一点，谈论“修订版本 5 的目录 *foo/*”是正确的。）

让我们再讨论一下最后一点，目录版本化是一个困难的问题；因为我们希望允许混合修订版本的工作拷贝，有一些防止我们滥用这个模型的限制。

从理论观点，我们定义“目录 *foo* 的修订版本 5”意味着一组目录条目和属性。现在假定我们从 *foo* 开始添加和删除文件，然后提交。如果说我们还有 *foo* 的修订版本 5 就是一个谎言。然而，如果说我们在提交之后增加了一位 *foo* 的修订版本号码，这也是一个谎言；*foo* 还有一些修改我们没有得到，因为我们还没有更新。

Subversion 通过在*.svn* 区域偷偷的纪录添加和删除来处理这些问题，当你最后运行 **`svn update`**，所有的账目会到版本库结算，并且目录的新修订版本号会正确设置。*因此，只有在更新之后才可以真正安全地说我*

*们有了一个“完美的”修订版本目录。*在大多数时候，你的工作拷贝会保存“不完美的”目录修订版本。

同样的，如果你尝试提交目录的属性修改会有一个问题，通常情况下，提交应该会提高工作目录的本地修订版本号，但是再一次，这还是一个谎言，因为这个目录还没有添加和删除发生，因为还没有更新发生。*因此，在你的目录不是最新的时候不允许你提交属性修改。*

关于目录版本的更多讨论见“混合修订版本的工作拷贝”一节。

更多离线操作

近些年来，磁盘空间变得异常便宜和丰富，但是网络带宽还没有，因此 Subversion 工作拷贝为紧缺资源进行了优化。

.svn 管理目录维护者与 *CVS* 同样的功能，除了它还保存了只读的文件“原始”拷贝，这允许你做许多离线操作：

`svn status`

显示你所做的本地修改（见“查看你的修改概况”一节）

`svn diff`

显示修改的详细信息（见 see“检查你的本地修改的详情”一节）

`svn revert`

删除你的本地修改（见“取消本地修改”一节）

另外，原始文件的缓存允许 Subversion 客户端在提交时只提交区别，这是 CVS 做不到的。

列表中最后一个子命令是新的；它不仅仅删除本地修改，也会取消如增加和删除的预定操作，这是恢复文件推荐的方式；运行 **rm file; svn update** 还可以工作，但是这样侮辱了更新操作的作用，而且，我们在这个主题…

区分状态和更新

在 Subversion，我们已经设法抹去 **cvs status** 和 **cvs update** 之间的混乱。

cvs status 命令有两个目的：第一，显示用户在工作拷贝的所有本地修改，第二，显示给用户哪些文件是最新的。很不幸，因为 CVS 难以阅读的状态输出，许多 CVS 用户并没有充分利用这个命令的好处。相反，他们慢慢习惯运行 **cvs update** 或 **cvs -n update** 来快速查看区别，如果用户忘记使用-n 选项，副作用就是将还没有准备好处理的版本库修改合并到工作拷贝。

对于 Subversion，我们通过修改 **svn status** 的输出使之同时满足阅读和解析的需要来努力消除这种混乱，同样，**svn update** 只会打印将要更新的文件信息，而*不是*本地修改。

状态

svn status 打印所有本地修改的文件，缺省情况下，不会联系版本库，然而这个命令接受一些选项，如下是一些最常用的：

-u

访问版本库检测并显示过期的信息。

-v

显示*所有*版本控制下的文件。

-N

非递归方式运行（不会访问子目录）。

status 命令有两种输出格式，缺省是“短”格式，本地修改像这样：

```
$ svn status
M      foo.c
M      bar/baz.c
```

如果你指定--show-updates（-u）选项，就会使用较长的格式输出：

```
$ svn status -u
M           1047   foo.c
      *     1045   faces.html
      *           bloo.png
M           1050   bar/baz.c
Status against revision:   1066
```

在这个例子里，出现了两列，第二列的星号表示了文件或目录是否过期，第三列显示了工作拷贝修订版本号，在上面的例子里，星号表示如果进行更新，*faces.html* 会被合并，而 *bloo.png* 则是版本库新加的文件。（*bloo.png* 前面的修订版本号为空表示了这个文件在工作拷贝已经不存在了。）

此刻，你必须赶快看一下 svn status 中所说的可能属性代码，下面是一些你会看到的常用状态代码：

```
A    Resource is scheduled for Addition

D    Resource is scheduled for Deletion

M    Resource has local Modifications

C    Resource has Conflicts (changes have not been completely
merged

       between the repository and working copy version)

X    Resource is eXternal to this working copy (may come from
another

       repository).  See “外部定义”一节

?    Resource is not under version control

!    Resource is missing or incomplete (removed by another tool
than

       Subversion)
```

关于 **svn status** 的详细讨论，见“查看你的修改概况”一节。

更新

svn update 会更新你的工作拷贝，只打印这次更新的文件。

Subversion 将 CVS 的 P 和 U 合并为 U，当合并或冲突发生时，Subversion 会简单的打印 G 或 C，而不是大段相关内容。

关于 **svn update** 的详细讨论，见“更新你的工作拷贝”一节。

分支和标签

Subversion 不区分文件系统空间和“分支”空间；分支和标签都是普通的文件系统目录，这恐怕是 CVS 用户需要逾越的最大心理障碍，所有信息在第 4 章 *分支与合并*。

因为 Subversion 把分支和标签看作普通目录看待，一直要记住检出项目的 trunk（http://svn.example.com/repos/calc/trunk/），而不是项目本身的（http://svn.example.com/repos/calc/）。如果你错误的检出了项目本身，你会紧张的发现你的项目拷贝包含了所有的分支和标签。[58]

元数据属性

Subversion 的一个新特性就是你可以对文件和目录任意附加元数据（或者是“属性”），属性是关联在工作拷贝文件或目录的任意名称/值对。

为了设置或得到一个属性名称，使用 **svn propset** 和 **svn propget** 子命令，列出对象所有的属性，使用 **svn proplist**。

更多信息见“属性”一节。

解决冲突

CVS 使用内联“冲突标志”来标记冲突，并且在更新时打印 C。历史上讲，这导致了许多问题，因为 CVS 做得还不够。许多用户在它们快速闪过终端时忘记（或没有看到）C，即使出现了冲突标记，他们也经常忘记，然后提交了带有冲突标记的文件。

Subversion 通过让冲突更明显来解决这个问题，它记住一个文件是处于冲突状态，在你运行 **`svn resolved`** 之前不会允许你提交修改，详情见“解决冲突（合并别人的修改）”一节。

二进制文件和行结束标记转换

在大多数情况下，Subversion 比 CVS 更好的处理二进制文件，因为 CVS 使用 RCS，它只可以存储二进制文件的完整拷贝，但是，Subversion 使用二进制区别算法来表示文件的区别，而不管文件是文本文件还是二进制文件。这意味着所有的文件是以微分的（压缩的）形式存放在版本库。

CVS 用户需要使用-kb 选项来标记二进制文件，防止数据的混淆（因为关键字解释和行结束转化），他们有时候会忘记这样做。

Subversion 使用更加异想天开的方法—第一，如果你不明确的告诉它（详情见“关键字替换”一节和“行结束字符串”一节）这样做，它不

会做任何关键字或行结束转化的操作，缺省情况下 Subversion 会把所有的数据看作字节串，所有的储存在版本库的文件都处于未转化的状态。

第二，Subversion 维护了一个内部的概念来区别一个文件是“文本”还是“二进制”文件，但这个概念*只*在工作拷贝非常重要，在 **svn update**，Subversion 会对本地修改的文本文件执行上下文的合并，但是对二进制文件不会。

为了检测一个上下文的合并是可能的，Subversion 检测 svn:mime-type 属性，如果没有 svn:mime-type 属性，或者这个属性是文本的（例如 text/*），Subversion 会假定它是文本的，否则 Subversion 认为它是二进制文件。Subversion 也会在 **svn import** 和 **svn add** 命令时通过运行一个二进制检测算法来帮助用户。这些命令会做出很好的猜测，然后（如果可能）设置添加文件的 svn:mime-type 属性。（如果 Subversion 猜测错误，用户可以删除或手工编辑这个属性。）

版本化的模块

不像 CVS，Subversion 工作拷贝会意识到它检出了一个模块，这意味着如果有人修改了模块的定义（例如添加和删除组件），然后一个对 **svn update** 的调用会适当的更新工作拷贝，添加或删除组件。

Subversion 定义了模块作为一个目录属性的目录列表：见“外部定义”一节。

认证

通过 CVS 的 pserver，你需要在读写操作之前“登陆”到服务器—即使是匿名操作。Subversion 版本库使用 Apache 的 **httpd** 或 **svnserve** 作为服务器，你不需要开始时提供认证凭证—如果一个操作需要认证，服务器会要求你的凭证（不管这凭证是用户名与密码，客户证书还是两个都有）。所以如果你的工作拷贝是全局可读的，在所有的读操作中不需要任何认证。

相对于 CVS，Subversion 会一直在磁盘（在你的*~/.subversion/auth/*目录）缓存凭证，除非你通过--no-auth-cache 选项告诉它不这样做。

这个行为也有例外，当使用 SSH 管道的 **svnserve** 服务器时，使用 svn+ssh://的 URL 模式这种情况下，**ssh** 会在通道刚开始时无条件的要求认证。

迁移 CVS 版本库到 Subversion

或许让 CVS 用户熟悉 Subversion 最好的办法就是让他们的项目继续在新系统下工作，这可以简单得通过平淡的把 CVS 版本库的导出数据导入到 Subversion 完成，或者是更加完全的方案，不仅仅包括最新数据快照，还包括所有的历史，从一个系统到另一个系统。这是一个非常困难的问题，包括推导保持原子性的修改集，转化两个系统完全不同的分支政策。但是我们还是有许多工具声称至少部分具备了的转化已存在的 CVS 版本库为 Subversion 版本库的能力。

最流行的（好像是最成熟的）转化工具是 cvs2svn（http://cvs2svn.tigris.org/），它是最初由 Subversion 自己的开发社区成员开发的一个 Python 脚本：它会多次扫描你的 CVS 版本库，并尽可能尝试推断提交，分支和标签，当它结束时，结果是可以代表代码历史的 Subversion 版本库或可移植的 Subversion 转储文件，关于指令和警告的详细信息可以看网站。

---

[58] 如果在检出完成之前没有消耗完磁盘空间的话。

WebDAV 和自动版本

目录

WebDAV 是 HTTP 的一个扩展，作为一个文件共享的标准不断发展。当今的操作系统变得极端的 web 化，许多内置了对装配 WebDAV 服务器导出的“共享”的支持。

如果你使用 Apache/mod_dav_svn 作为你的 Subversion 网络服务器，某种程度上，你也是在运行一个 WebDAV 服务器。这个附录提供了这种协议一些背景知识，Subversion 如何使用它，Subversion 如何和认识 WebDAV 的软件交互工作。

什么是 WebDAV ？

DAV 的意思是“Distributed Authoring and Versioning”。RFC 2518 为 HTTP 1.1 定义了一组概念和附加扩展方法来把 web 变成一个更加普遍的读/写媒体，基本思想是一个 WebDAV 兼容的 web 服务器可以像普通的文件服务器一样工作；客户端可以通过 HTTP 装配类似于 NFS 或 SMB 的 WebDAV 共享文件夹。

悲惨的是，RFC 规范并没有提供任何版本控制模型。基本的 DAV 客户端和服务器只是假定每个文件或目录只有一个版本存在，可以重复的覆盖。

因为 RFC 2518 漏下了版本概念，几年之后，另一个委员会留下来负责撰写 RFC 3253 来添加 WebDAV 的版本化，也就是“DeltaV”。WebDAV/DeltaV 客户端和服务器经常叫做“DeltaV”客户端和服务器，因为 DeltaV 暗含了基本的 WebDAV。

最初的 WebDAV 标准得到了广泛的成功，所有的现代操作系统拥有内置的（后面有详细资料）对普通 WebDAV 的支持，许多流行的应用程序也可以使用 WebDAV—Microsoft Office，Dreamweaver 和

Photoshop。在服务器方面，Apache 从 1998 年就开始支持 WebDAV，并被认为是一个事实上的开源标准，也有许多商业的 WebDAV 服务器，例如 Microsoft 的 IIS。

不幸的是，DeltaV 没有这样的成功，很难寻找到任何 DeltaV 客户端和服务器。只有一些不太出名的商业产品，因此很难测试交互性，不清楚为什么 DeltaV 还这样停滞，一些人说规范太复杂了，还有些人认为尽管 DeltaV 的特性有很大的吸引力（即使最新的技术用户也喜欢使用网络文件共享），版本控制特性对大多数用户还不是这样有趣和必须。最后，有些人认为 DeltaV 还这样不流行主要是因为一直没有开源的服务器产品实现它。

当 Subversion 还在设计阶段时，使用 Apache 的 httpd 作为主要网络服务器就是一个很好的想法，已经有了支持 WebDAV 服务的模块（**mod_dav_svn**）。DeltaV 有一个很新的规范，希望就是 Subversion 服务器模块最终能够成为一个开源的 DeltaV 参考实现，但非常不幸，DeltaV 得版本模型过于详细，与 Subversion 的模型并不匹配，虽然有些概念可以对应起来，但有些则不能。

这是什么意思呢？

首先，Subversion 客户端不是一个完全实现的 DeltaV 客户端，它需要从服务器得到 DeltaV 不能提供的东西，因此非常依赖于只有 **mod_dav_svn** 理解的 Subversion 特定的 REPORT 请求。

其次，**mod_dav_svn** 不是一个完全的 DeltaV 服务，许多与 Subversion 不相关的 DeltaV 规范还没有实现。

在开发者社区一直有这样的讨论，是否值得弥补这种形势。改变 Subversion 的设计来匹配 DeltaV 看起来并不现实，所以可能没有办法让客户端从普通的 DeltaV 服务器上得到所有的东西。另一方面，mod_dav_svn *可以*继续开发来实现所有的 DeltaV，但缺乏这样做的动力—几乎没有能与之交户的 DeltaV 客户端。

自动版本化

因为 Subversion 客户端不是完整的 DeltaV 客户端，Subversion 服务器也不是完整的 DeltaV 服务器，但仍有值得高兴的交互特性：叫做自动版本化。

自动版本化是 DeltaV 标准中的可选特性，一个典型的 DeltaV 服务器会拒绝一个对版本控制之下文件的 PUT 操作，为了修改一个版本控制下的文件，服务器只会接受一系列正确的版本请求：例如 MKACTIVITY、CHECKOUT、PUT 和 CHECKIN。但是如果 DeltaV 服务器支持自动版本化，服务器可以在后台假装客户端执行了一些列正确的版本请求，也就是说，DeltaV 服务器可以与一个对版本化一无所知的普通 WebDAV 客户端交互。

因为有许多操作系统已经集成了 WebDAV 客户端，这个特性的用例可能是这样的：假设一个办公室有许多使用 Microsoft Windows 或 Mac OS 的普通用户，每个用户“装载”了一个 Subversion 版本库，看起来

就是普通的网络共享文件夹。他们像普通目录一样的操作这个目录：打开文件、编辑它们，保存它们。同时，服务器自动的版本化所有的东西，任何管理员（或有知识的用户）可以一直使用 Subversion 客户端来查询历史来检索旧版本的数据。

这个场景不是小说：对于 Subversion 1.2 来说，是真实的和有效的。为了激活 **mod_dav_svn** 的自动版本化，需要使用 *httpd.conf* 中 Location 区块的 SVNAutoversioning 指示，例如：

```
<Location /repos>
  DAV svn
  SVNPath /path/to/repository
  SVNAutoversioning on
</Location>
```

当激活了 SVNAutoversioning，来自 WebDAV 的客户端请求会导致自动提交，每个修订版本会自动附加一个原始的日志信息。

然而，在激活这个特性之前，需要理解你做的事情。WebDAV 会做许多写请求，导致了产生数量*非常多*的自动提交修订版本。例如，当保存数据，许多客户端会使用一个 PUT 一个 0 字节的文件，然后紧跟一个 PUT 真实的文件数据。一个单独的文件写操作产生了两个不同的提交。考虑到许多应用程序隔几分钟的自动保存，会产生更多的提交。

如果你有发送邮件的 post-commit 钩子程序，例如，你会根据是否有价值来开启和关闭邮件通知，另外，一个聪明的 post-commit 钩子也应该能够区分自动版本化和 **svn commit** 产生的事务。技巧就是检查修订版本的 svn:autoversioned 属性，如果有，则提交来自一个原始的 WebDAV 客户端。

另一个作为 SVNAutoversioning 特性补充的特性来自 Apache 的 mod_mime 模块，如果一个原始的 WebDAV 客户端在版本库添加了一个新文件，用户就没有机会设置 svn:mime-type 属性，这会导致使用 WebDAV 共享目录查看时会看到原始的图标，而没有关联到任何应用。一个补救办法就是让系统管理员（或其他理解 Subversion）的人检出一份工作拷贝，然后为需要的文件手动设置 svn:mime-type 属性,但是这个整理工作永远不会结束，作为替代，你可以在你的 Subversion<Location>区使用 ModMimeUsePathInfo 指示：

```
<Location /repos>
  DAV svn
  SVNPath /path/to/repository
  SVNAutoversioning on

  ModMimeUsePathInfo on

</Location>
```

这个指示允许 mod_mime 在使用自动版本化添加新文件时尝试自动检测新文件的 mime-type，这个模块查看文件的扩展名，有可能的话还包括检查内容；如果文件符合某个常用模式，就会自动设置文件的 svn:mime-type。

客户端交互性

所有的 WebDAV 客户端分为三类—独立应用程序，文件浏览器扩展或文件系统实现,这些分类定义了 WebDAV 用户可用的功能性。表 C.1“”给 WebDAV 常见软件进行了分类，并提供了的简短描述。

表 C.1.

软件	类型	Windows	Mac	Linux	描述
Adobe Photoshop	独立的 WebDAV 应用程序	X			图像编辑软件，允许直接从 WebDAV 的 URL 打开文件和修改。
Cadaver	独立的 WebDAV 应用程序		X	X	命令行的 WebDAV 客户端，支持文件传输，目录树显示和锁定操作
DAV Explorer	独立的 WebDAV 应用程序	X	X	X	浏览 WebDAV 共享的 Java GUI 工具
Macromedia Dreamweaver	独立的 WebDAV 应用程序	X			Web 制作软件，可以直接读写 WebDAV 的 URL
Microsoft Office	独立的 WebDAV 应用程序	X			Office 上产套件,可以直接读写 WebDAV 的 URL
Microsoft Web 文件夹	文件浏览器 WebDAV 扩展	X			Novell NetDrive
GNOME Nautilus	文件浏览器 WebDAV			X	GUI 文件浏览器，可以对 WebDAV 共享执行目录树

软件	类型	Windows	Mac	Linux	描述
	扩展				操作
KDE Konqueror	文件浏览器 WebDAV 扩展			X	GUI 文件浏览器，可以对 WebDAV 共享执行目录树操作
Mac OS X	WebDAV 文件系统实现		X		内置支持加载 WebDAV 到本地功能的操作系统
驱动器映射程序，可以将 Windows 驱动器加载为远程的 WebDAV 共享	WebDAV 文件系统实现	X			SRT WebDrive
文件传输软件，可以将 Windows 驱动器加载为远程的 WebDAV 共享	WebDAV 文件系统实现	X			一个 WebDAV 应用就是一个内置 WebDAV 协议的程序，我们会覆盖大多数支持 WebDAV 的流行程序。
davfs2	WebDAV 文件系统实现			X	Linux 文件系统驱动允许加载 WebDAV 共享

独立的 WebDAV 应用程序

WebDAV 应用使用 WebDAV 协议与 WebDAV 服务器通讯，我们将会介绍一些支持 WebDAV 的流行程序。

Microsoft Office，Dreamweaver，Photoshop

在 Windows 下，有许多已知的应用程序支持 WebDAV 客户端功能，例如微软 Office，[59]Adobe 的 Photoshop 和 Macromedia 的 Dreamweaver 程序，他们可以直接打开和保存 URL，并且在编辑文件时经常使用 WebDAV 的锁。

需要注意尽管这些程序也存在于 Mac OS X，但是在这个平台上并不是直接支持 WebDAV。实际上在 Mac OS X，File->Open 会离开对应的程序，因为 OS X 已经实现了底层的文件系统级 WebDAV 支持。

Cadaver，DAV 浏览器

Cadaver 是一个简单的 Unix 命令行的 WebDAV 共享浏览程序，就像 Subversion 客户端，它使用 neon 的 HTTP 库，毫不奇怪，因为其作者就是 neon 的作者，Cadaver 是一个自由软件（是用 GPL 许可证），可以通过 http://www.webdav.org/cadaver/访问。

使用 cadaver 与命令行 FTP 程序类似，因此它在基本的 WebDAV 调试中非常有用，它可以用来在紧急情况下上传或下载文件，也可以用来验证属性，并拷贝、移动、锁定或解锁文件：

```
$ cadaver http://host/repos

dav:/repos/> ls

Listing collection `/repos/': succeeded.

Coll: > foobar                                0  May 10 16:19

      > playwright.el                      2864  May  4 16:18

      > proofbypoem.txt                    1461  May  5 15:09

      > westcoast.jpg                     66737  May  5 15:09



dav:/repos/> put README

Uploading README to `/repos/README':

Progress: [============================>] 100.0% of 357 bytes
succeeded.
```

```
dav:/repos/> get proofbypoem.txt

Downloading `/repos/proofbypoem.txt' to proofbypoem.txt:

Progress: [============================>] 100.0% of 1461 bytes
succeeded.
```

DAV Explorer 是另一个独立运行的 WebDAV 客户端，使用 Java 编写，有一个类 Apache 的许可证，网站是 http://www.ics.uci.edu/~webdav/。DAV Explorer 与 cadaver 功能差不多，优点可移植，并有一个用户友好的 GUI 程序。它也是最早的支持 WebDAV 访问控制协议（RFC 3744）的客户端之一。

当然，在这个情况下 DAV Explorer 的 ACL 支持没有任何用处，因为 **mod_dav_svn** 不支持它，事实上，Cadaver 和 DAV Explorer 支持的一些有限的 DeltaV 命令也并不有效，因为他们不允许 MKACTIVITY 请求，但是这都不相干；我们假定这些客户端都是针对自动版本化版本库工作。

文件浏览器 WebDAV 扩展

一些流行的文件浏览器 GUI 程序支持 WebDAV 扩展，允许用户将 DAV 共享当作本地文件夹访问，例如 Windows 浏览器可以以“network place”方式浏览 WebDAV 服务器。用户可以拖入和拖出文件，或者是改名、拷贝或删除其中的文件。但是因为它只是文件浏览器的一个特性，DAV 对普通应用不可见，所有的 DAV 交互必须通过浏览器界面。

Microsoft Web 文件夹

Microsoft 是 WebDAV 规范最早的支持者，最早在 Windows 98 配置客户端，被称作“网络文件夹”，这个客户端在 Windows NT4 和 2000 上也存在。

最早的Webfolders客户端是浏览器的扩展，主要的浏览文件系统的GUI程序，工作良好。在 Windows 98，如果“我的电脑”里没有网络文件夹，这个特性需要明确安装。在 Windows 2000，只需要添加一个新的“网络位置”，输入 URL，WebDAV 共享就会弹出让你浏览。

伴随着 Windows XP，Microsoft 开始了另一种网络文件夹的实现，叫做“WebDAV mini-redirector”，这个新的实现是文件系统级的客户端，允许 WebDAV 转载到驱动器盘符上。不幸的是，这个实现充满难以相信的 bug。客户端经常会尝试把 http 的 URL（http://host/repos）转化为 UNC 共享符号（\\host\repos），它也经常使用 Windows 域认证来回应基本的 HTTP 认证，按照 HOST\username 发送用户名。这类互动性问题在网络上大量传播，使大量用户受挫。即使是 ApacheWebDAV 的作者 Greg Stein 也建议不要对 Apache 服务器使用 XP 的网络文件夹。

结果是原始的网络文件夹并没有在 XP 中死掉，只是要被埋葬了。还是有办法适用这个技术：

1. 到网络位置。
2. 添加一个新的网络位置。

3. 当要求输入，输入版本库的 URL，但 URL 中*要包含端口号*。例如 http://host/repos 的输入是 http://host:80/repos。
4. 回应所有的认证请求。

有各种解决问题的方法，但好像没有一种能够在各版本和各级别的 Windows XP 中有效。在我们的测试里，只有上面这种策略在各种系统中有效。WebDAV 社区一致认为避免使用新的网络文件夹实现，而使用旧的，如果你希望在 Windows XP 使用真实的文件系统级的客户端，请使用第三方的程序，例如 WebDrive 或 NetDrive。

最后一个提示：如果你尝试使用 XP 的网络文件夹，确定你有 Microsoft 最新的版本，Microsoft 在 2005 年 1 月发布了一个问题修正，在 http://support.microsoft.com/?kbid=892211，特别的，这个发布是用来修正在访问 DAV 时发生无限递归的问题。

Nautilus，Konqueror

Nautilus 是 GNOME 桌面（http://www.gnome.orghttp://www.kde.org

GNOME 的 Nautilus 里，从 File menu 选择 Open location，并且输入 URL。版本库就会显示出来，就像其他文件系统。

KDE 的 Konqueror 里你需要在地址栏使用 webdav://模式来输入 URL，如果你输入 http://的 URL，Konqueror 会像普通的 web 浏览器。你会看到 mod_dav_svn 输出的普通 HTML 目录列表。通过输入

webdav://host/repos 代替 http://host/repos，Konqueror 就成为了一个 WebDAV 客户端，并且按照文件系统的方式显示版本库。

WebDAV 文件系统实现

WebDAV 文件系统实现被认为是最佳的 WebDAV 客户端，它通过低级的文件系统模块实现，通常在操作系统的核心。这意味着 DAV 共享像网络的其他文件系统一样装载，就像在 Unix 下面装载 NFS，或者是在 Windows 下装载一个 SMB 共享。结果就是这种客户端为所有程序提供了对 WebDAV 得透明访问。

WebDrive，NetDrive

WebDrive 和 NetDrive 都是完美的商业产品，允许将 WebDAV 绑定到 Windows 的盘符，当我们写作的时候，WebDrive 可以从 South River Technologies（http://www.southrivertech.com）购买。NetDrive 由 Netware 装运，通过查找“netdrive.exe”就会找到。尽管它可以自由得到，用户还是需要一个 Netware 许可证。（如果着听起来有点奇怪，你并不孤单，看 Novell 网站的这个页面：http://www.novell.com/coolsolutions/qna/999.html）

Mac OS X

Apple 的 OS X 操作系统是集成的文件系统级的 WebDAV 客户端，通过 Finder，选择 Go menu 的 Connect to Server 条目，输入 WebDAV

的 URL，会在桌面显示一个磁盘，就像其他装载的卷。你也可以从 Darwin 终端通过 **mount** 类型为 webdav 的文件系统实现。

```
$ mount -t webdav http://svn.example.com/repos/project
/some/mountpoint

$
```

注意如果 **mod_dav_svn** 是 1.2 之前的版本，OS X 不能按照可读写装载，而是会成为只读。这是因为，OS X 坚持要读写共享支持锁定，而锁定文件出现在 Subversion 1.2。

警告一句话：OS X 的 WebDAV 客户端有时候对 HTTP 重定向很敏感，如果 OS X 不能装载版本库，你或许需要开启 Apache 服务器 *httpd.conf* 的 BrowserMatch 指示：

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Linux davfs2

Linux davfs2 是一个 Linux 核心的文件系统模块，开发坐落在 http://dav.sourceforge.net/。一旦安装，一个 WebDAV 网络共享可以使用 mount 命令装载：

```
$ mount.davfs http://host/repos /mnt/dav
```

[59] 在 Windows 下，有一些有名的集成 WebDAV 客户端功能的软件，例如 Microsoft's Office、Adobe 的 Photoshop 和 Macromedia 的 Dreamweaver。它们都可以直接打开和保存 URL，也可以在编辑时大量的使用 WebDAV 的锁定。

第三方工具

Subversion 的模块设计（在“分层的库设计”一节讨论过）和语言绑定的能力（在“使用 C 和 C++ 以外的语言”一节描述过）使的我们可以作为扩展和后端支持来替代软件的某些部分，在这个附录里，我们会简略介绍一些使用 Subversion 功能的第三方的工具。关于更新的信息，可以在 Subversion 的网站（http://subversion.tigris.org/project_links.html）查看。

Copyright

the author.

Your fair use and other rights are in no way affected by the above.

The above is a summary of the full license below.

==========

Creative Commons Legal Code

Attribution 2.0

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE

LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN

ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS

INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES

REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR

DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS

CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS

PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE

WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS

PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND

AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS

YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF

SUCH TERMS AND CONDITIONS.

1. Definitions

a. "Collective Work" means a work, such as a periodical issue,

anthology or encyclopedia, in which the Work in its entirety in

unmodified form, along with a number of other contributions,

constituting separate and independent works in themselves, are

assembled into a collective whole. A work that constitutes a

Collective Work will not be considered a Derivative Work (as

defined below) for the purposes of this License.

b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

d. "Original Author" means the individual or entity who created the Work.

e. "Work" means the copyrightable work of authorship offered under

the terms of this License.

f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now

known or hereafter devised. The above rights include the right to make

such modifications as are technically necessary to exercise the rights

in other media and formats. All rights not expressly granted by

Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly

made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or

publicly digitally perform the Work only under the terms of this

License, and You must include a copy of, or the Uniform Resource

Identifier for, this License with every copy or phonorecord of

the Work You distribute, publicly display, publicly perform, or

publicly digitally perform. You may not offer or impose any

terms on the Work that alter or restrict the terms of this

License or the recipients' exercise of the rights granted

hereunder. You may not sublicense the Work. You must keep intact

all notices that refer to this License and to the disclaimer of

warranties. You may not distribute, publicly display, publicly

perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or

pseudonym if applicable) of the Original Author if supplied; the

title of the Work if supplied; to the extent reasonably

practicable, the Uniform Resource Identifier, if any, that

Licensor specifies to be associated with the Work, unless such

URI does not refer to the copyright notice or licensing

information for the Work; and in the case of a Derivative Work,

a credit identifying the use of the Work in the Derivative Work

(e.g., "French translation of the Work by Original Author," or

"Screenplay based on original Work by Original Author"). Such

credit may be implemented in any reasonable manner; provided,

however, that in the case of a Derivative Work or Collective

Work, at a minimum such credit will appear where any other

comparable authorship credit appears and in a manner at least as

prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING,

LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR

WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED,

STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF

TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE,

NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY,

OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT

DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED

WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY

APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY

LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE

OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE

WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH

DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate

automatically upon any breach by You of the terms of this

License. Individuals or entities who have received Derivative

Works or Collective Works from You under this License, however,

will not have their licenses terminated provided such

individuals or entities remain in full compliance with those

licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any

termination of this License.

b. Subject to the above terms and conditions, the license granted

here is perpetual (for the duration of the applicable copyright

in the Work). Notwithstanding the above, Licensor reserves the

right to release the Work under different license terms or to

stop distributing the Work at any time; provided, however that

any such election will not serve to withdraw this License (or

any other license that has been, or is required to be, granted

under the terms of this License), and this License will continue

in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You distribute or publicly digitally perform the Work

or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the

索引

- import, svn import
- info, svn info
- list, svn list
- lock, svn lock
- log, svn log
- merge, svn merge
- mkdir, svn mkdir
- move, svn move
- propdel, svn propdel
- propedit, svn propedit
- propget, svn propget
- proplist, svn proplist
- propset, svn propset
- resolved, svn resolved
- revert, svn revert
- status, svn status
- switch, svn switch
- unlock, svn unlock
- update, svn update

svnadmin

- 子命令

- create, svnadmin create
- deltify, svnadmin deltify
- dump, svnadmin dump
- help, svnadmin help
- hotcopy, svnadmin hotcopy
- list-dblogs, svnadmin list-dblogs
- list-unused-dblogs, svnadmin list-unused-dblogs
- load, svnadmin load
- lslocks, svnadmin lslocks
- lstxns, svnadmin lstxns
- recover, svnadmin recover
- rmlocks, svnadmin rmlocks
- rmtxns, svnadmin rmtxns
- setlog, svnadmin setlog
- verify, svnadmin verify

svnlook

- 子命令

- author, svnlook author
- cat, svnlook cat
- changed, svnlook changed
- date, svnlook date