Subversion 权威指南

针对 Subversion 1.6

（编译自 r3478/r3461)

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

目录

插图清单

表格清单

范例清单

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}条目，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 对于 Subversion 邮件列表中常见问题的研究。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 的标准流程是怎样的？分支与标签同其它版本控制系统的工作方式一样吗？我怎样知道某处的修改是谁做的？

    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 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；你仍有可能偶尔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

> "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。"

<div align="right">—Greg Hudson，Subversion 开发者</div>

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，对网络化操作的支持，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，几乎所有新的开源项目都选择了 Subversion 替代CVS。

本书是为 Subversion 1.6 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。

1. 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**)，还有一些针对本书的辅助程序。

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** 程序也可以在如 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 （或其他系统)用户能够轻松的投入到Subversion 的使用当中，在侧栏不时会出现一些涉及CVS 的内容，此外，在附录 B 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器参数进行编译，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或实践。

2. 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，她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 为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。然后，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，那么第 1 章 基本概念介绍了版本控制的重要思想，第 2 章 基本使用是重要的入门教程。

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(第 3 章 高级主题)，像如何使用分支和执行合并(第 4 章 分支与合并)，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。

3. 本书约定

本节描述了本书中使用的各种约定。

等宽
　　用于用户的原文输入，命令输出和命令行选项

斜体
　　用于程序，Subversion 工具的子命令名称，文件和目录的名称，以及新术语

等宽斜体
用于代码和文本中的可替换部分

Also, we sprinkled especially helpful or important bits of information throughout the book (in contextually relevant locations), set off visually so they're easy to find. Look for the following icons as you read:

注意

此图标表示旁边的内容需特别注意。

提示

此图标表示旁边描述了一个有用的小技巧。

警告

此图标表示警告。请特别注意这些内容以免出现问题。

4. 本书的结构

以下是各个章节的内容介绍：

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，工作副本和修订版本的概念。

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，以及访问版本库的不同方式：`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 的内部信息，Subversion 的文件系统，以及程序员眼中的工作副本管理区，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

对于缺乏耐心的家伙，我们会立刻解释如何安装和使用 Subversion。我们已经告诫你了。

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 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, 版权

Creative Commons Attribution License 的副本，本书的许可证。

5. 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，它始终按自由许可证(参见 附录 D, 版权)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 — 它在自由许可证的控制之下。你的唯一限制是必须保留原始作者。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。

本书的在线主页和许多志愿者的翻译工作位于 http://svnbook.red-bean.com。在这个网站上，你可以找到本书最新快照和标签版本的链接的各种格式，以及访问本书的 Subversion 版本库(存放 DocBook XML 源文件)的指令。我们欢迎和鼓励反馈。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>。

6. 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]

---

[1]噢，还要感谢 Karl 为了本书所付出的辛勤工作。

感谢 O'Reilly 和我们的编辑：Chuck Toporek, Linda Mui, Tatiana Apandi, Mary Brady 和 Mary Treseler。他们的耐心和支持是巨大的。

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：Bhuvaneswaran A, David Alber, C. Scott Ananian, David Anderson, Ariel Arjona, Seth Arnold, Jani Averbach, Charles Bailey, Ryan Barrett, Francois Beausoleil, Brian R. Becker, Yves Bergeron, Karl Berry, Jennifer Bevan, Matt Blais, Jim Blandy, Phil Bordelon, Sietse Brouwer, Tom Brown, Zack Brown, Martin Buchholz, Paul Burba, Sean Callan-Hinsvark, Branko Cibej, Archie Cobbs, Jason Cohen, Ryan Cresawn, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Markus Dreyer, Nick Duffek, Boris Dusek, Ben Elliston, Justin Erenkrantz, Jens M. Felderhoff, Kyle Ferrio, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Vlad Georgescu, Peter Gervai, Dave Gilbert, Eric Gillespie, David Glasser, Marcel Gosselin, Lieven Govaerts, Steve Greenland, Matthew Gregan, Tom Gregory, Maverick Grey, Art Haas, Mark E. Hamilton, Eric Hanchrow, Liam Healy, Malte Helmert, Michael Henderson, Øyvind A. Holm, Greg Hudson, Alexis Huxley, Auke Jilderda, Toby Johnson, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Robert Kleemann, Erik Kline, Josh Knowles, Andreas J. Koenig, Axel Kollmorgen, Nuutti Kotivuori, Kalin Kozhuharov, Matt Kraai, Regis Kuckaertz, Stefan Kueng, Steve Kunkee, Scott Lamb, Wesley J. Landaker, Benjamin Landsteiner, Vincent Lefevre, Morten Ludvigsen, Dennis Lundberg, Paul Lussier, Bruce A. Mah, Jonathon Mah, Karl Heinz Marbaise, Philip Martin, Feliciano Matias, Neil Mayhew, Patrick Mayweg, Gareth McCaughan, Craig McElroy, Simon McKenna, Christophe Meresse, Jonathan Metillon, Jean-Francois Michaud, Jon Middleton, Robert Moerland, Marcel Molina Jr., Tim Moloney, Alexander Mueller, Tabish Mustufa, Christopher Ness, Roman Neuhauser, Mats Nilsson, Greg Noel, Joe Orton, Eric Paire, Dimitri Papadopoulos-Orfanos, Jerry Peek, Chris Pepper, Amy Lyn Pilato, Kevin Pilch-Bisson, Hans Polak, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, Srinivasa Ramanujan, Jack Repenning, Tobias Ringstrom, Jason Robbins, Garrett Rooney, Joel Rosdahl, Christian Sauer, Ryan Schmidt, Jochem Schulenklopper, Jens Seidel, Daniel Shahaf, Larry Shatzer, Danil Shopyrin, Erik Sjoelund, Joey Smith, W. Snyder, Stefan Sperling, Robert Spier, M. S. Sriram, Russell Steicke, David Steinbrunner, Sander Striker, David Summers, Johan Sundstroem, Ed Swierk, John Szakmeister, Arfrever Frehtes Taifersar Arahesis, Robert Tasarz, Michael W. Thelen, Mason Thomas, Erik van der Kolk, Joshua Varner, Eric Wadsworth, Chris Wagner, Colin Watson, Alex Waugh, Chad Whitacre, Andy Whitcroft, Josef Wolf, Luke Worth, Hyrum Wright, Blair Zajac, Florian Zumbiehl, 以及整个 Subversion 社区。

6.1. 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。

感谢我的家人和朋友对我诚挚的鼓励，尽管他们对我的课题不感兴趣。（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。

6.2. 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。

致 Mike 和 Ben：与你们一起完成本书的工作非常快乐。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。

6.3. 来自 C. Michael Pilato

特别感谢 Amy，我 10 年中难以置信的最好的朋友和妻子，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持久忍受 。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；很遗憾，它是提供关键概念的另一半。抱歉，Aidan - 我不能找到将 Disney/Pixar 字符转换成文本的方法。但是，爸爸爱你们，不能等待教你们编程知识。

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，让我在编程实践困境中知道怎样做；Brain Fitzpatrick，同我分享他的写作经验。所有我一直从你们那里获得新知识的人—尽管又不断忘记。

最后，感谢所有对我展现完美卓越创造力的人们。

7. 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。

7.1. Subversion 是正确的工具吗？

如果你是一个考虑如何使用 Subversion 的用户或管理员，你要问自己的第一件事就是：
"这是这项工作的正确工具吗？"，Subversion 是一个梦幻般的锤子，但要小心不要把任何问
题当作钉子。

如果你希望归档文件和目录旧版本，有可能要恢复或需要查看日志获得其修改的历史，
那么 Subversion 是你需要的工具。如果你需要和别人协作文档(通常通过网络)并跟踪所
做的修改，那么 Subversion 也适合。这是 Subversion 为什么使用在软件开发环境
—编程是一件天性的社会活动，Subversion 使得与其他程序员的交互变得简单。当然，使用
Subversion 也有代价：管理负担。你会需要管理一个存放所有历史的数据版本库，并需要经
常的备份。而在日常的工作中，你不能像往常一样拷贝，移动，重命名或删除文件，相反，你
需要通过 Subversion 完成这些工作。

假定你能够接受额外的工作流程，你一定要确定不要使用 Subversion 来解决其他工具能
够完成的很好的工作。例如，因为 Subversion 会复制所有的数据到参与者，一个常见的误用
是将其作为普通的分布式系统。问题是此类数据通常很少修改，在这种情况下，使用
Subversion 有点"过了"。[2]有一些可以复制数据更简单的工具，**没有必要**过度的跟踪变更，
例如 **rsync** 或 **unison**。

7.2. 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，
详细设计工作从五月开始，在 CollabNet 中的 Brian Behlendorf 和 Jason Robbins，以及 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 很快就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。

---

[2]或者像一个朋友说的，"用别克拍苍蝇"。

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。

7.3. 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

diagram by Brian W. Fitzpatrick <fitz@red-bean.com>
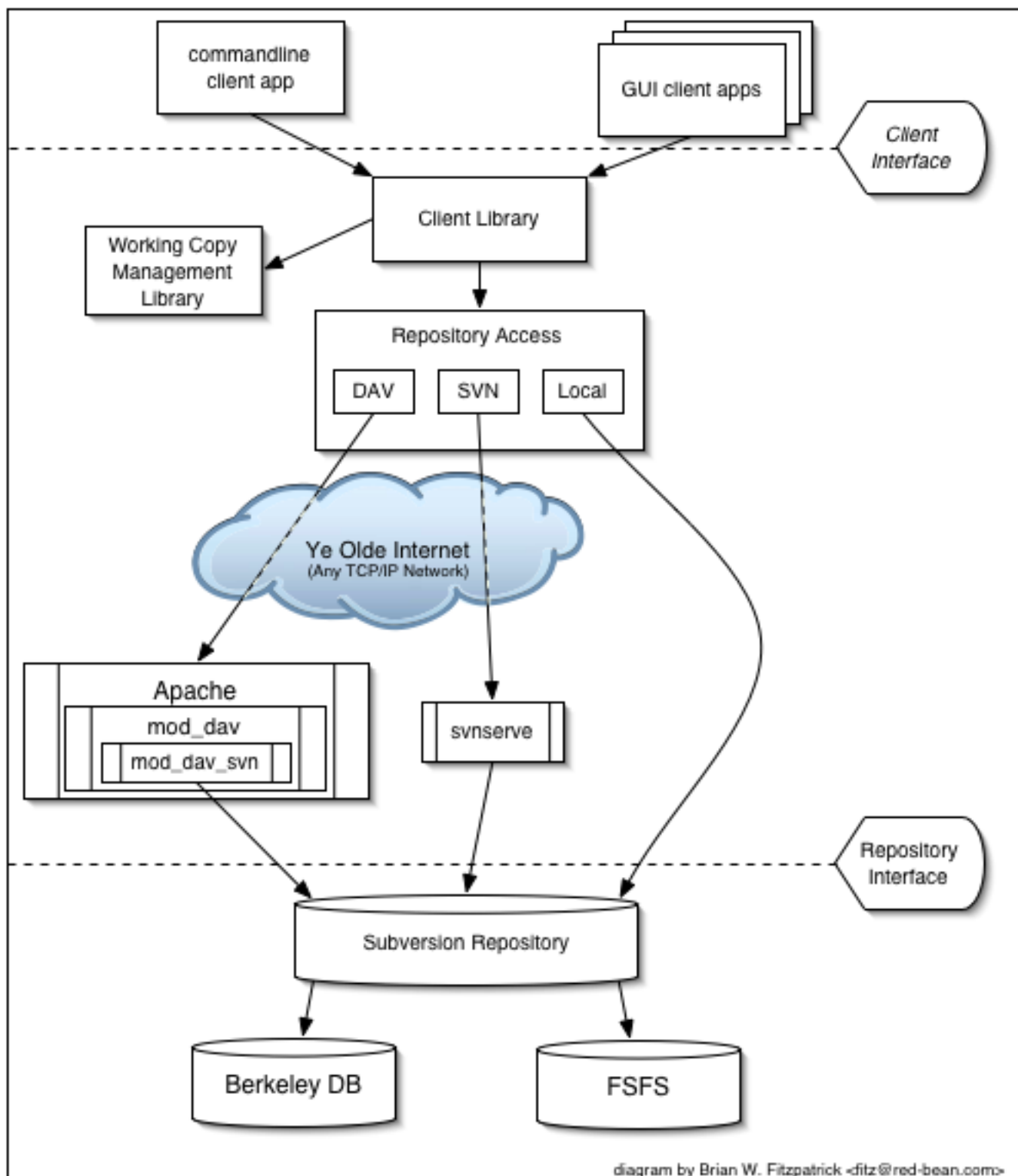
　　图中的一端是保存所有版本数据的 Subversion 版本库，另一端是 Subvesion 的客户程序，管理着所有版本数据的本地影射（称为"工作副本"），在这两极之间是各种各样的版本库访问(RA)层，某些使用电脑网络通过网络服务器访问版本库，某些则绕过网络服务器直接访问版本库。

7.4. 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副本的状态(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

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。

svndumpfilter
过滤 Subversion 版本库转储数据流的工具

svnsync
一个通过网络增量镜像版本库的程序

7.5. Subversion 有什么新东西？

此书的第一版在 2004 年发布，也就是 Subversion 1.0 发布后不久。经过 4 年，Subversion 发布了 5 个主要的新版本，修正了 bug，增加了主要的新特性。我们一直保持本书在线版本的更新，我们现在很兴奋的是 O'Reilly 将会发布包含 Subversion 1.5 版本的第二版，它是项目的一个主要里程碑。下面是从 Subversion 1.0 以来主要变更的总结。注意这不是完整的列表；详细信息可以访问 Subversion 的网站 http://subversion.tigris.org。

Subversion 1.1 (2004年9月)
1.1 版本引入了 FSFS，纯文件的版本库存储选项。虽然 Berkeley DB 后端被广泛的使用，但因为 FSFS 其较低的门槛和较小的管理需要，FSFS 还是成为新建版本库的缺省的选项。另外这个版本能够将符号链纳入版本控制，能够自动封装 URL，还有本地化的用户界面。

Subversion 1.2 (2005年5月)
1.2 版本引入了文件在服务器端锁定的功能，实现对特定资源的顺序访问。虽然 Subversion 一直基本上是一个并行版本控制系统，特定类型的的二进制文件(例如艺术作品)不能合并在一起，锁定特性填补了对此类资源的版本化保护。随着锁定也引入了一个完整的 WebDAV 自动版本实现，允许 Subversion 版本库作为网络文件夹加载。最后，Subversion 1.2 开始使用新的，更快的二进制差异算法来压缩和检索文件的旧版本。

Subversion 1.3 (2005年12月)

1.3 版本为 **svnserve** 服务器引入路径为基础的授权控制，与 Apache 服务器对应的特性匹配。Apache 服务器自己也获得了新的日志特性，Subversion 其它语言的 API 绑定也取得了巨大的进步。

Subversion 1.4(2006年9月)

1.4 版本引入了完全的新工具 — **svnsync** — 用来通过网络完成单向的版本库复制。一个重要的部分是工作副本元数据得到修补，不再使用 XML(获得客户端的速度改善)，而 Berkeley DB 版本库后端获得了在发生崩溃时自动恢复的能力。

Subversion 1.5 (2008年6月)

1.5 版本花费了比以前版本更长的时间，但是关键特性是巨大的：分支和合并的半自动跟踪。这是为用户带来的巨大便利，也使 Subversion 的能力远远超越 CVS，进入了商业竞争者 Perforce 和 Clearcase 的级别，Subversion 1.5 也引入了一些其它用户关注的特性，例如交互式的文件冲突解决，部分检出，变更列表的客户端管理，外部定义的强大的新语法，以及对 **svnserve** 服务器的 SASL 认证支持。

Subversion 1.6 (???)

???

第 1 章 基本概念

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。

1.1. 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
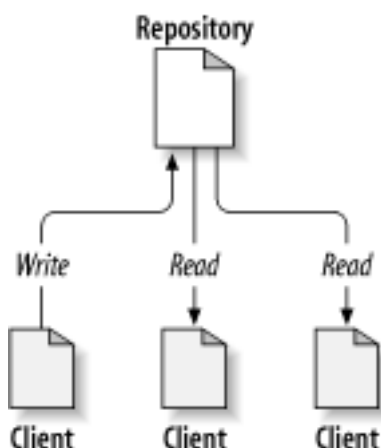
这有什么意义吗？说了这么多，Subversion 听起来和一般的文件服务器没什么不同。事实上，Subversion 的版本库的确**是**一种文件服务器，但不是一般的文件服务器。Subversion 版本库的特别之处在于，**它会记录每一次改变**：每个文件的改变，甚至是目录树本身的改变，例如文件和目录的添加, 删除和重新组织。

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。

1.2. 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，如果你遇到了其它类似

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。

1.2.1. 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。假设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 Harry 先保存它的修改，(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，因为她没有在开始的时候看到 Harry 的修改。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
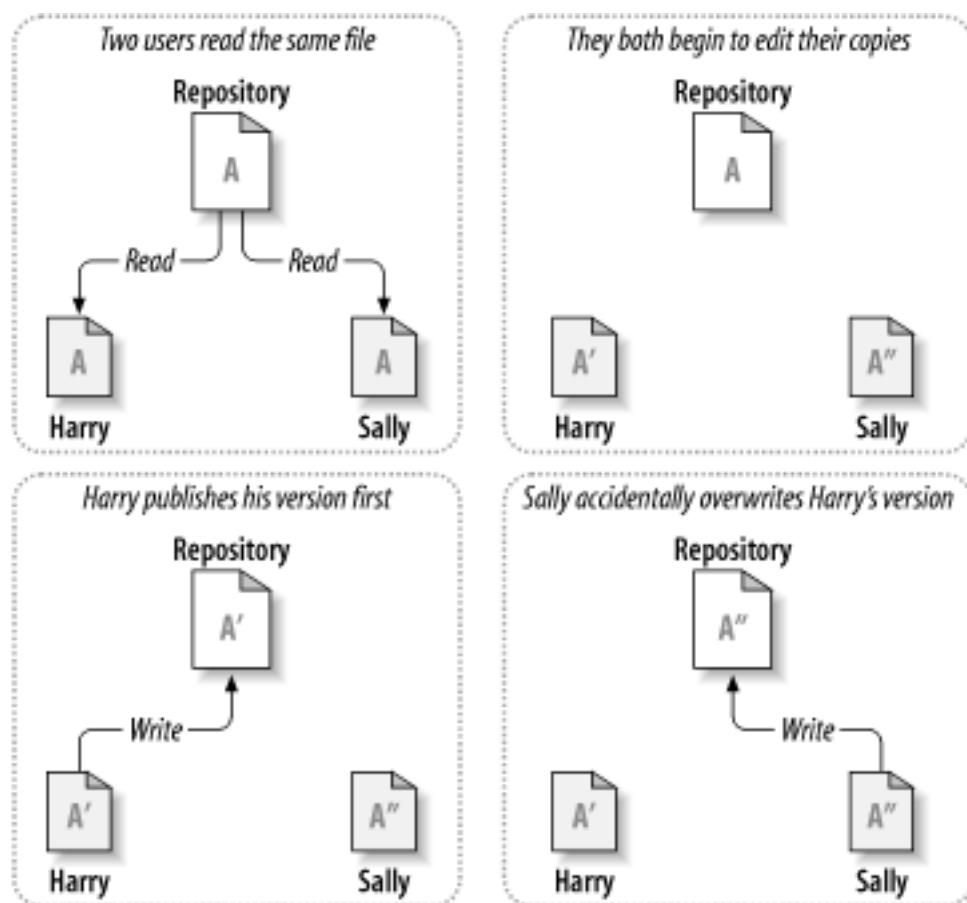
1.2.2. "锁定-修改-解锁"方案

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
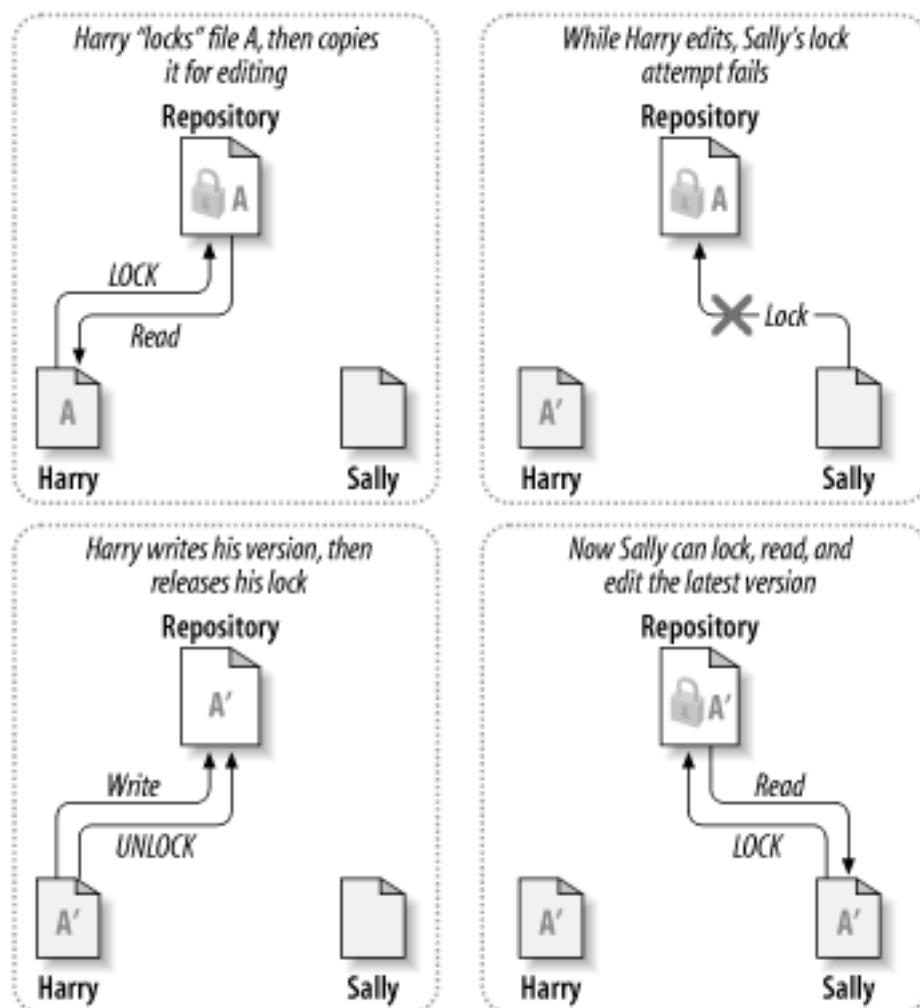
锁定-修改-解锁模型有一点问题就是限制太多，经常会成为用户的障碍：

- **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串行开发**。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。但是如果 A 和 B 互相依赖，修改导致它们不兼容会怎么样呢？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。

1.2.3. "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**工作副本**—版本库中文件和目录的本地映射。用户并行工作，修改各自的工作副本，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副本，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**版本库和他的工作副本之后，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
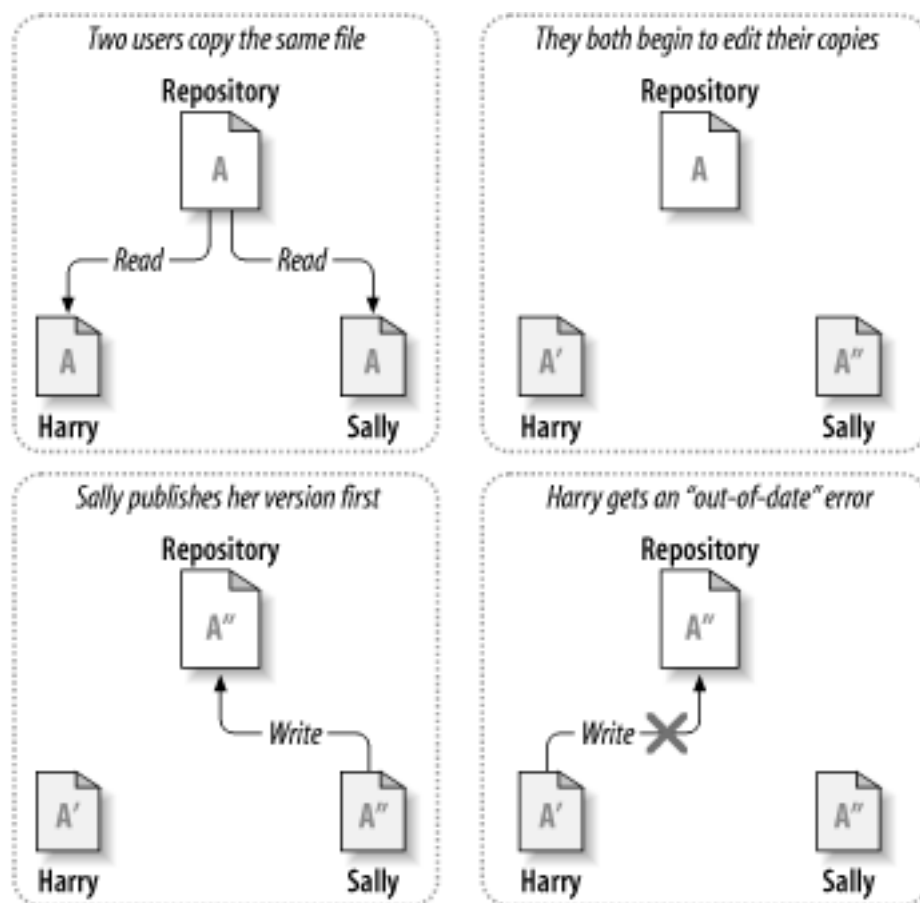
图 **1.5.** "拷贝-修改-合并"方案(续)
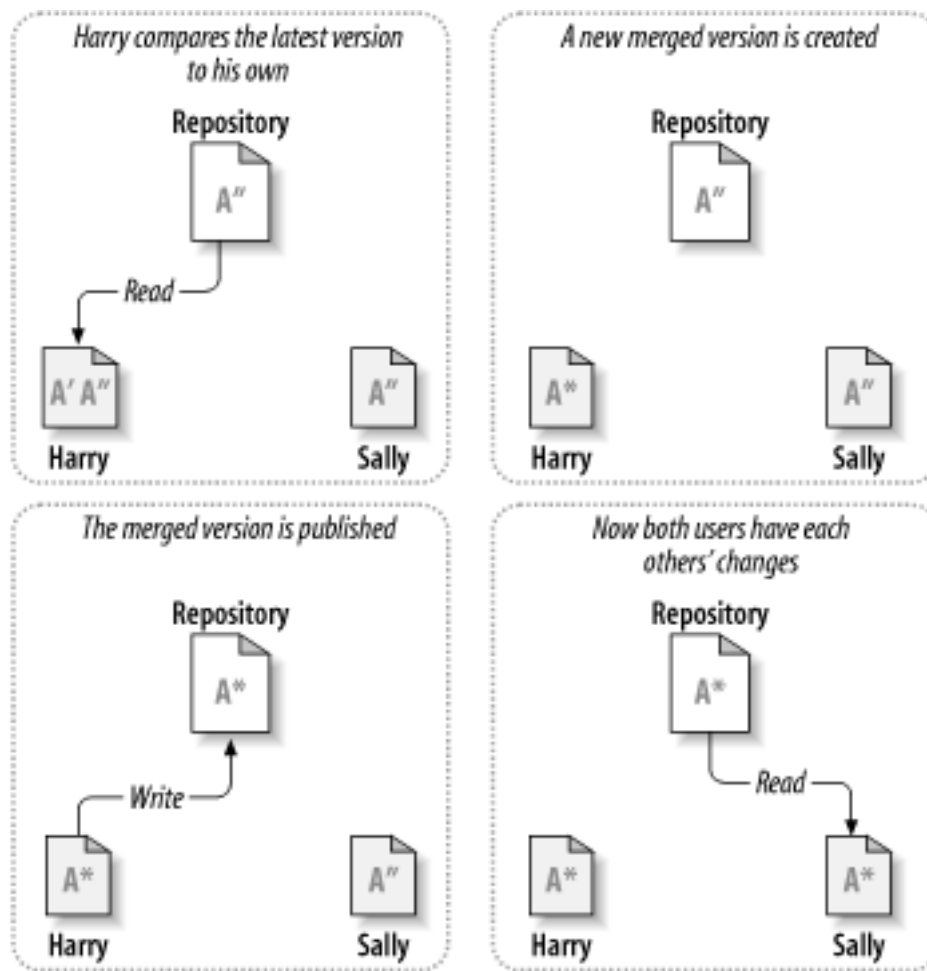
但是如果 Sally 和 Harry 的修改交迭了该怎么办？这种情况叫做**冲突**，这通常不是个大问题，当 Harry 告诉他的客户端去合并版本库的最新修改到自己的工作副本时，他的文件 A 就会处于冲突状态：他可以看到一对冲突的修改集，并手工的选择保留一组修改。需要注意的是软件不能自动的解决冲突，只有人可以理解并作出智能的选择，一旦 Harry 手工的解决了冲突—也许需要与 Sally 讨论—它可以安全的把合并的文件保存到版本库。

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第 3.7 节 "锁定"。

1.3. 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。

1.3.1. Subversion 版本库的 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:///var/svn/repos
…
$ svn checkout file://localhost/var/svn/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:/var/svn/repos
…
C:\> svn checkout -"file:///X|/var/svn/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。

注意

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://` 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 (见第 8.1.1 节 "版本库层")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会封装这些不安全的字符，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。

In Subversion 1.6, a new caret (^) notation was introduced as a shorthand for "the URL of the repository's root directory". For example:

```
$ svn list ^/tags/bigsandwich/
```

In this example, we're specifying a URL for the `/tags/bigsandwich` directory in the root of the repository. Note that this URL syntax *only* works when your current working directory is a working copy—the commandline client knows the repository's root URL by looking at the working copy's metadata.

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 "版本库访问 URL"描述了不同的 URL 模式对应的访问方法。

表 1.1. 版本库访问 URL

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://` 类似，但是包括 SSL 加密。
`svn://`	通过定制的协议访问 `svnserve` 服务器
`svn+ssh://`	与 `svn://` 类似，但通过 SSH 隧道。

关于 Subversion 解析 URL 的更多信息，见第 1.3.1 节 "Subversion 版本库的 URL"。关于不同的网络服务器类型，见第 6 章 服务配置。

1.3.2. 工作副本

你已经阅读过了关于工作副本的内容；现在我们要讲一讲客户端怎样建立和使用它。

一个 Subversion工 作副本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，你的工作副本是你的私有工作区，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副本。

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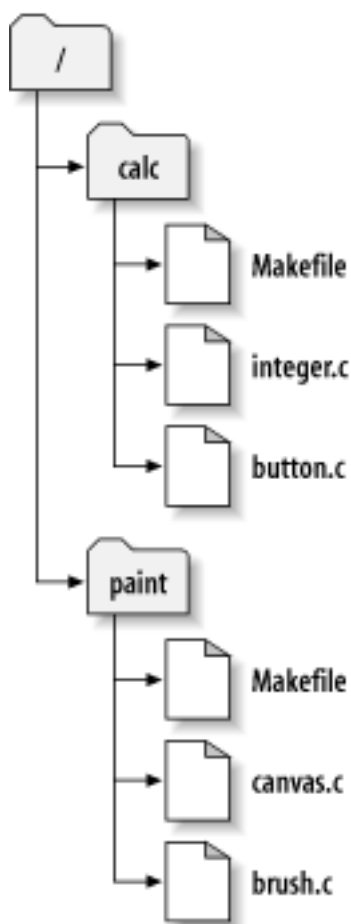
为了得到一个工作副本，你必须**检出**(*check out*)版本库的一个子树(术语"check out"听起来像是锁定或者保留资源，实际上不是，只是简单的得到一个项目的私有拷贝)，举个例子，检出 /calc 后，你可以得到这样的工作副本：

```
$ 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  button.c integer.c -.svn/
```

列表中的 A 表示 Subversion 增加了一些条目到工作副本，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**检入**)修改到版本库。

将你的修改发布给别人，你可以使用 Subversion 的 **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 的一个工作副本，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，你提交了对 button.c 的修改，Sally 的工作副本并没有改变，Subversion 只在用户要求的时候才改变工作副本。

要使项目最新，Sally 可以通过使用 **svn update** 命令，要求 Subversion **更新**她的工作副本。这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副本。

```
$ pwd
/home/sally/calc

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

$ 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。

1.3.3. 修订版本

svn commit 操作可以作为一个原子事务，发布任意数量文件和目录的修改。在你的工作副本里，你可以改变文件内容，删除，改名，以及复制文件和目录，然后作为一个原子事务提交。

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，网络问题和其它用户行为。

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
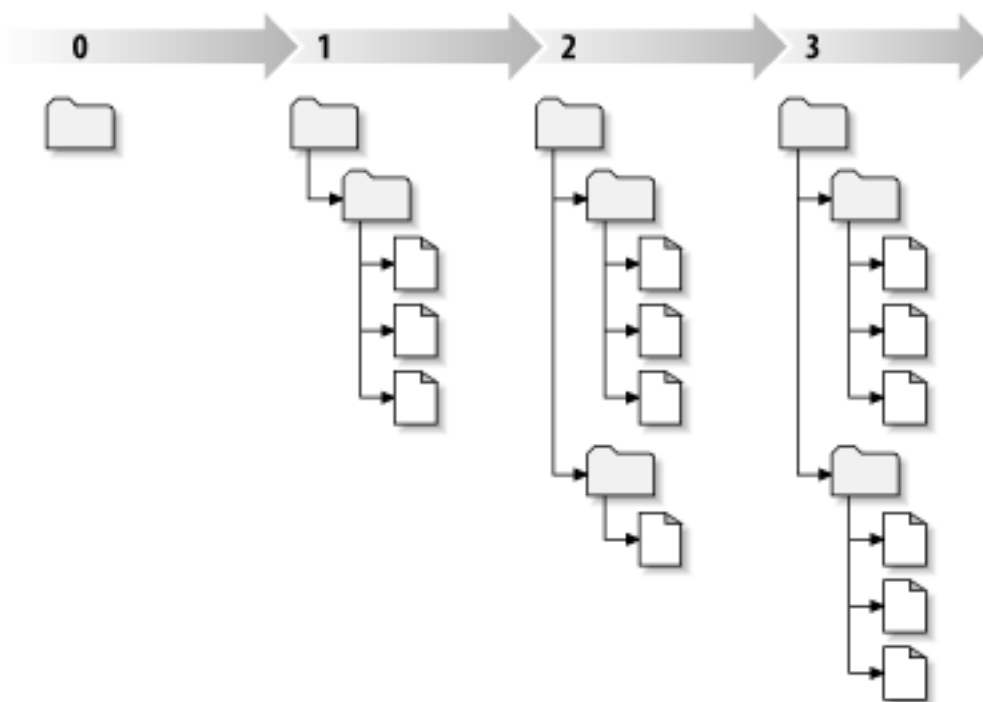
全局版本号

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，工作副本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副本，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，你的工作副本会是这个样子的：

```
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** 来更新你的工作副本，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副本，你对 `button.c` 的改变还在，在这个例子里，`Makefile` 在 4，5，6 的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，所以你在工作副本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。

1.3.4. 工作副本怎样跟踪版本库

对于工作副本的每一个文件，Subversion在管理区域`.svn/`记录两项关键的信息：

- 作为工作文件基准的版本(叫做文件的**工作版本**)

- 本地副本最近一次被版本库更新的时间戳。

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。

本地未修改，已过时
> 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副本。

本地已修改，已过时
> 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，请看第 2.4.3.1 节 "查看你的修改概况"。

1.3.5. 混合修订版本的工作副本

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。

1.3.5.1. 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副本，而不会强迫将修改发布。

这个规则的主要副作用就是，工作副本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，假定你有一个工作副本，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，许多用户希望工作副本完全变成修订版本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，可以使得整个工作副本成为修订版本15—但是，那样我们会打破"推"和"拉"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，工作副本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，整个工作副本被标记为修订版本15。

1.3.5.2. 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)，你的工作副本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(详细信息见第 2.4.3.1 节"查看你的修改概况")。

通常，新用户对于工作副本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(见第 2.5.1 节"产生历史修改列表")，当用户对一个工作副本对象调用这个命令，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。

1.3.5.3. 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副本的一部分"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—这个特性允许工作副本的任何一个部分在历史中前进或后退。

1.3.5.4. 混合版本有限制

无论你如何在工作副本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。

1.4. 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，客户工作副本和版本库修订树的概念。

- 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 跟踪和管理工作副本信息的方式。

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。

第 2 章 基本使用

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副本中获取别人的修改，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。

2.1. 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包含文档的工具—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.
  Unversionable items such as device files and pipes are ignored
  if ---force is specified.

Valid options:
  --q [--quiet]             -: print nothing, or only summary
information
  --N [--non-recursive]     -: obsolete; try ---depth=files or ---
depth=immediates
  ---depth ARG              -: limit operation by depth ARG
('empty', -'files',
                              -'immediates', or -'infinity')
…
```

> **选项(Options), 开关(Switches)和标志(Flags)，天呐！**
>
> Subversion 命令行客户端有许多命令行控制特性(我们叫做选项)，但是有两个不同类型的选项：短选项是一个短横线紧跟一个单独的字符，长选项包含两个短横线紧跟一组字符(例如 -s 和 --this-is-a-long-option 对应)。每个选项都有长格式，但是只有特定选项有附加的短格式(通常是经常使用的选项)。为了保持清晰，我们**通常**在代码实例中使用长形式，但是当要描述选项时，如果有一个短形式，我们会提供长形式(改进清晰性)和短形式(便于记忆)，你可以使用你最舒服的形式，但是不要两个都用。

2.2. 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**。

2.2.1. 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** 不需要一个工作副本，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 -/var/svn/newrepos
$ svn import mytree file:///var/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:///var/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**转化成工作副本，为了开始工作，你还是需要运行 **svn checkout** 导出一个工作副本。

2.2.2. 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:///var/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，可以看第 4.7.1 节 "版本库布局"和第 5.2.1 节 "规划你的版本库结构"中关于"项目根目录"的内容。

2.3. 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 顶点(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 努力不限制版本控制的数据类型。文件的内容和属性值都是按照二进制数据存储和传递，并且第 3.3.1 节 "文件内容类型"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（见第 1.2 节 "版本模型"），你可以在工作副本中开始修改的目录和文件，你的工作副本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。

警告

因为你的工作副本"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副本的一个文件，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。

目录 .svn 中有什么？

工作副本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 使用它来管理工作副本。

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副本，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.
```

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

2.3.1. 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。在某些系统中，Subversion 不能加密你的认证数据。你会被询问，是否缓存你的明文密码到磁盘。你可以选择缓存，这样做比较方便，在接下来的操作中你就可以不必输入密码。但如果你很在乎明文密码缓存，不想每次都被询问，你可以永久关闭明文缓存，或者针对每个服务器设置。

To permanently disable caching of passwords in plaintext, you can add the line `store-plaintext-passwords = no` to the global section in the `servers` configuration file on the local machine. To disable caching of plaintext passwords for a particular server, use the same setting in the appropriate group section in the `servers` configuration file. See 第 7.1.3 节 "配置选项" in 第 7 章 定制你的 *Subversion* 体验 for details.

You can also disable caching of authentication credentials entirely, regardless of whether the credentials are stored in encrypted form or not.

在某个命令中关闭密码缓存，可以使用 --no-auth-cache 选项，如果希望永久关闭缓存，可以在本机 Subversion 的配置文件中添加 store-passwords = no 这一行，详情请见第 3.11.2 节 "客户端凭证缓存"。

2.3.2. 认证为其它用户

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副本的人非常方便。但是有时候会不好用—特别是如果你使用的是共享工作副本，例如系统配置目录，或者是 WEB 服务器文档目录。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。

2.4. 基本的工作循环

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：

1. 更新你的工作副本。

   • **svn update**

2. 做出修改

   • **svn add**

   • **svn delete**

- **svn copy**

- **svn move**

3. 检验修改

- **svn status**

- **svn diff**

4. 可能会取消一些修改

- **svn revert**

5. 解决冲突(合并别人的修改)

- **svn update**

- **svn resolve**

6. 提交你的修改

- **svn commit**

2.4.1. 更新你的工作副本

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

```
$ 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更新你的工作副本来引入这些更改。

当服务器通过 **svn update** 将修改传递到你的工作副本时，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副本作了哪些工作。关于这些字母的详细含义，可以执行 **svn help update**。

2.4.2. 修改你的工作副本

现在你可以开始工作，并且修改你的工作副本了。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**, **svn move** 和 **svn mkdir**。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。

　　你可以对工作副本做出两种修改：**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 会自动检测到文件的修改。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"标记"为调度删除，添加，拷贝或移动。这些动作会在工作副本上立刻发生效果，但只有提交后才会在版本库里生效。

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 "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，可以通过对符号链接操作，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接，也会知道这个符号链接指向的"对象"。当这个符号链接检出到另一个支持符号链接的操作系统上时，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 这类不支持符号链接的系统上的操作，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 系统上作为符号链接使用，所以它也会防止 Windows 用户作其他 Subversion 相关的操作。

　　下面是 Subversion 用来修改目录树结构的五个最常用的子命令。

svn add foo
　　调度将文件，目录或者符号链接 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** 中的内容也会调度增加。如果你只想添加 foo 本身，请使用 --non-recursive (-N) 参数。

svn delete foo
　　调度将文件，目录或者符号链接 foo 从版本库中删除，如果 foo 是文件或符号链接，它会马上从工作副本中删除。如果 foo 是目录，不会被删除，但是 Subversion 调度删除它。当你提交修改后，foo 就会在你的工作副本和版本库中被删除。[1]

svn copy foo bar
　　建立一个新条目 bar 作为 foo 的复制品，并且自动调度增加 bar，当在下次提交时会将 bar 添加到版本库，这种复制会记录下来历史(按照来自 foo 的方式记录)。如果不传递 --parents，**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 的拷贝调度添加，foo 已经调度删除。如果不传递 --parents，**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，并且调度增加到版本库。

---

[1]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(或者更新你的工作副本到)你做出删除操作的前一个修订版本，来找回所有的东西，详情参见第 4.3.5 节 "找回删除的项目"。

不通过工作副本修改版本库

有些情况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，而不是工作副本路径时。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)。

与指定 URL 的操作方式有些区别，因为在使用工作副本的运作方式时，工作副本成为一个"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。

2.4.3. 检查你的修改

当你完成修改，你需要提交它们到版本库，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，给了你一次撤销修改的机会。此外，这是一个在发布之前，复审和检查的好机会。你可通过命令 **svn status** 浏览所做的修改，通过命令 **svn diff** 检查修改的详细信息。

看！没有网络！

You can use the commands **svn status**, **svn diff**, and **svn revert** without any network access even if your repository *is* across the network. This makes it easy to manage your changes-in-progress when you are somewhere without a network connection, such as traveling on an airplane, riding a commuter train, or hacking on the beach. [2]

Subversion does this by keeping private caches of pristine versions of each versioned file inside the .svn administrative areas. This allows Subversion to report—and revert—local modifications to those files *without network access*. This cache (called the "text-base") also allows Subversion to send the user's local modifications during a commit to the server as a compressed *delta* (or "difference") against the pristine version. Having this cache is a tremendous benefit—even if you have a fast Internet connection, it's much faster to send only a file's changes rather than the whole file to the server.

Subversion has been optimized to help you with this task, and it is able to do many things without communicating with the repository. In particular, your working copy contains a hidden cached "pristine" copy of each version-controlled file within the .svn area. Because of this, Subversion can quickly show you how your working files have changed or even allow you to undo your changes without contacting the repository.

2.4.3.1. 查看你的修改概况

To get an overview of your changes, you'll use the **svn status** command. You'll probably use **svn status** more than any other Subversion command.

---

[2]而且你也没有无线上网卡。不要妄想连接到我们，哈！

> **CVS 用户：控制另类的更新！**
>
> You're probably used to using **cvs update** to see what changes you've made to your working copy. **svn status** will give you all the information you need regarding what has changed in your working copy—without accessing the repository or potentially incorporating new changes published by other users.
>
> In Subversion, **svn update** does just that—it updates your working copy with any changes committed to the repository since the last time you updated your working copy. You may have to break the habit of using the **update** command to see what local modifications you've made.

If you run **svn status** at the top of your working copy with no arguments, it will detect all file and tree changes you've made. Here are a few examples of the most common status codes that **svn status** can return. (Note that the text following # is not actually printed by **svn status**.)

```
?       scratch.c           # file is not under version control
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
```

In this output format, **svn status** prints six columns of characters, followed by several whitespace characters, followed by a file or directory name. The first column tells the status of a file or directory and/or its contents. The codes we listed are:

A item
 预定加入到版本库的文件，目录或符号链的item。

C item
 The file item is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy (and weren't resolved during the update). You must resolve this conflict before committing your changes to the repository.

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，它可以显示工作副本中的**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
```

This is the "long form" output of **svn status**. The letters in the first column mean the same as before, but the second column shows the working revision of the item. The third and fourth columns show the revision in which the item last changed, and who changed it.

None of the prior invocations to **svn status** contact the repository—instead, they compare the metadata in the .svn directory with the working copy. Finally, there is the --show-updates (-u) option, which contacts the repository and adds information about things that are out of date:

```
$ 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
```

Notice the two asterisks: if you were to run **svn update** at this point, you would receive changes to README and trout.c. This tells you some very useful information—you'll need to update and get the server changes on README before you commit, or the repository will reject your commit for being out of date (more on this subject later).

svn status can display much more information about the files and directories in your working copy than we've shown here—for an exhaustive description of **svn status** and its output, see svn status.

2.4.3.2. 检查你的本地修改的详情

Another way to examine your changes is with the **svn diff** command. You can find out *exactly* how you've modified things by running **svn diff** with no arguments, which prints out file changes in *unified diff format*:

```
$ 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.
```

The **svn diff** command produces this output by comparing your working files against the cached "pristine" copies within the .svn area. Files scheduled for addition are displayed as all added text, and files scheduled for deletion are displayed as all deleted text.

Output is displayed in unified diff format. That is, removed lines are prefaced with –, and added lines are prefaced with +. **svn diff** also prints filename and offset information useful to the **patch** program, so you can generate "patches" by redirecting the diff output to a file:

```
$ svn diff > patchfile
```

You could, for example, email the patch file to another developer for review or testing prior to a commit.

Subversion uses its internal diff engine, which produces unified diff format, by default. If you want diff output in a different format, specify an external diff program using `--diff-cmd` and pass any flags you'd like to it using the `--extensions` (`-x`) option. For example, to see local differences in file `foo.c` in context output format while ignoring case differences, you might run **svn diff --diff-cmd /usr/bin/diff --extensions '-i' foo.c**.

2.4.4. 取消本地修改

Suppose while viewing the output of **svn diff** you determine that all the changes you made to a particular file are mistakes. Maybe you shouldn't have changed the file at all, or perhaps it would be easier to make different changes starting from scratch.

这是使用**svn revert**的好机会：

```
$ svn revert README
Reverted -'README'
```

Subversion reverts the file to its premodified state by overwriting it with the cached "pristine" copy from the `.svn` area. But also note that **svn revert** can undo *any* scheduled operations—for example, you might decide that you don't want to add a new file after all:

```
$ svn status foo
?      foo

$ svn add foo
A        foo

$ svn revert foo
Reverted -'foo'

$ svn status foo
?      foo
```

> **注意**
>
> **svn revert** *item* has exactly the same effect as deleting *item* from your working copy and then running **svn update -r BASE** *item*. However, if you're reverting a file, **svn revert** has one very noticeable difference—it doesn't have to communicate with the repository to restore your file.

或许你不小心删除了一个文件：

```
$ svn status README
```

```
$ svn delete README
D         README

$ svn revert README
Reverted -'README'

$ svn status README
```

2.4.5. 解决冲突（合并别人的修改）

We've already seen how **svn status -u** can predict conflicts. Suppose you run **svn update** and some interesting things occur:

```
$ svn update
U  INSTALL
G  README
Conflict discovered in -'bar.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```

The U and G codes are no cause for concern; those files cleanly absorbed changes from the repository. The files marked with U contained no local changes but were Updated with changes from the repository. The G stands for merGed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap with the local changes.

But the next two lines are part of a feature (new in Subversion 1.5) called *interactive conflict resolution*. This means that the changes from the server overlapped with your own, and you have the opportunity to resolve this conflict. The most commonly used options are displayed, but you can see all of the options by typing *h*:

```
…
  (p)  postpone    -- mark the conflict to be resolved later
  (df) diff-full   -- show all changes made to merged file
  (e)  edit        -- change merged file in an editor
  (r)  resolved    -- accept merged version of file
  (mf) mine-full   -- accept my version of entire file (ignore
their changes)
  (tf) theirs-full -- accept their version of entire file (lose
my changes)
  (l)  launch      -- launch external tool to resolve conflict
  (h)  help        -- show this list
```

在我们详细查看每个选项含义之前，让我们简短的回顾一下所有这些选项。

(p)ostpone
    让文件在更新完成之后保持冲突状态。

(d)iff-(f)ull
> 使用标准区别格式显示base修订版本和冲突文件本身的区别。

(e)dit
> 用你喜欢的编辑器打开冲突的文件，编辑器是环境变量EDITOR设置的。

(r)esolved
> After editing a file, tell **svn** that you've resolved the conflicts in the file and that it should accept the current contents—basically that you've "resolved" the conflict.

(m)ine-(f)ull
> 丢弃新从服务器接收的变更，并只使用你查看文件的本地修改。

(t)heirs-(f)ull
> 丢弃你对查看文件的本地修改，只使用从服务器新接收的变更。

(l)aunch
> 启动外部程序来执行冲突解决，这需要一些预先的准备。

(h)elp
> 显示所有在冲突解决时可能使用的命令。

我们现在会更详细的覆盖这些命令，根据关联功能对其进行分组。

2.4.5.1. 交互式的查看冲突区别

Before deciding how to attack a conflict interactively, odds are that you'd like to see exactly what is in conflict, and the *diff-full* command (`df`) is what you'll use for this:

```
…
Select: (p) postpone, (df) diff-full, (e) edit,
        (h)elp for more options -: df
--- -.svn/text-base/sandwich.txt.svn-base      Tue Dec 11
21:33:57 2007
+++ -.svn/tmp/tempfile.32.tmp     Tue Dec 11 21:34:33 2007
@@ --1 +1,5 @@
-Just buy a sandwich.
+<<<<<<< -.mine
+Go pick up a cheesesteak.
+=======
+Bring me a taco!
+>>>>>>> -.r32
…
```

The first line of the diff content shows the previous contents of the working copy (the BASE revision), the next content line is your change, and the last content line is the change that was just received from the server (*usually* the HEAD revision). With this information in hand, you're ready to move on to the next action.

2.4.5.2. 交互式的解决冲突区别

There are four different ways to resolve conflicts interactively—two of which allow you to selectively merge and edit changes, and two of which allow you to simply pick a version of the file and move along.

If you wish to choose some combination of your local changes, you can use the "edit" command (`e`) to manually edit the file with conflict markers in a text editor (determined by the `EDITOR` environment variable). Editing the file by hand in your favorite text editor is a somewhat low-tech way of remedying conflicts (see 第 2.4.5.4 节 "手工合并冲突" for a walkthrough), so some people like to use fancy graphical merge tools instead.

To use a merge tool, you need to either set the `SVN_MERGE` environment variable or define the `merge-tool-cmd` option in your Subversion configuration file (see 第 7.1.3 节 "配置选项" for more details). Subversion will pass four arguments to the merge tool: the `BASE` revision of the file, the revision of the file received from the server as part of the update, the copy of the file containing your local edits, and the merged copy of the file (which contains conflict markers). If your merge tool is expecting arguments in a different order or format, you'll need to write a wrapper script for Subversion to invoke. After you've edited the file, if you're satisfied with the changes you've made, you can tell Subversion that the edited file is no longer in conflict by using the "resolve" command (`r`).

If you decide that you don't need to merge any changes, but just want to accept one version of the file or the other, you can either choose your changes (a.k.a. "mine") by using the "mine-full" command (`mf`) or choose theirs by using the "theirs-full" command (`tf`).

2.4.5.3. 延后解决冲突

This may sound like an appropriate section for avoiding marital disagreements, but it's actually still about Subversion, so read on. If you're doing an update and encounter a conflict that you're not prepared to review or resolve, you can type **p** to postpone resolving a conflict on a file-by-file basis when you run **svn update**. If you're running an update and don't want to resolve any conflicts, you can pass the `--non-interactive` option to **svn update**, and any file in conflict will be marked with a `C` automatically.

The `C` stands for conflict. This means that the changes from the server overlapped with your own, and now you have to manually choose between them after the update has completed. When you postpone a conflict resolution, **svn** typically does three things to assist you in noticing and resolving that conflict:

- Subversion在更新时打印C标记，并且标记这个文件已冲突。

- If Subversion considers the file to be mergeable, it places *conflict markers*—special strings of text that delimit the "sides" of the conflict—into the file to visibly demonstrate the overlapping areas. (Subversion uses the `svn:mime-type` property to decide whether a file is capable of contextual, line-based merging. See 第 3.3.1 节 "文件内容类型" to learn more.)

- 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副本：

filename.mine
> This is your file as it existed in your working copy before you updated your working copy—that is, without conflict markers. This file has only your latest changes in it. (If Subversion considers the file to be unmergeable, the .mine file isn't created, since it would be identical to the working file.)

filename.r*OLDREV*
> This is the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.

filename.r*NEWREV*
> This is the file that your Subversion client just received from the server when you updated your working copy. This file corresponds to the HEAD revision of the repository.

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。

For example, Sally makes changes to the file sandwich.txt, but does not yet commit those changes. Meanwhile, Harry commits changes to that same file. Sally updates her working copy before committing and she gets a conflict, which she postpones:

```
$ svn update
Conflict discovered in -'sandwich.txt'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h)elp for more options -: p
C  sandwich.txt
Updated to revision 2.
$ ls --1
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

At this point, Subversion will *not* allow Sally to commit the file sandwich.txt until the three temporary files are removed:

```
$ 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
```

If you've postponed a conflict, you need to resolve the conflict before Subversion will allow you to commit your changes. You'll do this with the **svn resolve** command and one of several arguments to the --accept option.

如果你希望选择上次检出后修改之前的文件版本，选择*base*参数。

如果你希望选择只包含你修改的版本，选择*mine-full*参数。

如果你希望选择最近从服务器更新的版本(因此会丢弃你的所以编辑)，选择*theirs-full*参数。

However, if you want to pick and choose from your changes and the changes that your update fetched from the server, merge the conflicted text "by hand" (by examining and editing the conflict markers within the file) and then choose the `working` argument.

svn resolve removes the three temporary files and accepts the version of the file that you specified with the `--accept` option, and Subversion no longer considers the file to be in a state of conflict:

```
$ svn resolve ---accept working sandwich.txt
Resolved conflicted state of -'sandwich.txt'
```

2.4.5.4. 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。

Here's an example. Due to a miscommunication, you and Sally, your collaborator, both edit the file `sandwich.txt` at the same time. Sally commits her changes, and when you go to update your working copy, you get a conflict and you're going to have to edit `sandwich.txt` to resolve the conflict. First, let's take a look at the file:

```
$ 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
```

The strings of less-than signs, equals signs, and greater-than signs are conflict markers and are not part of the actual data in conflict. You generally want to ensure that those are removed from the file before your next commit. The text between the first two sets of markers is composed of the changes you made in the conflicting area:

```
<<<<<<< -.mine
```

```
Salami
Mortadella
Prosciutto
=======
```

后两组之间的是Sally提交的修改冲突：

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

Usually you won't want to just delete the conflict markers and Sally's changes—she's going to be awfully surprised when the sandwich arrives and it's not what she wanted. This is where you pick up the phone or walk across the office and explain to Sally that you can't get sauerkraut from an Italian deli. [3] Once you've agreed on the changes you will commit, edit your file and remove the conflict markers:

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

Now use **svn resolve**, and you're ready to commit your changes:

```
$ svn resolve ---accept working sandwich.txt
Resolved conflicted state of -'sandwich.txt'
$ svn commit --m -"Go ahead and use my sandwich, discarding
Sally's edits."
```

Note that **svn resolve**, unlike most of the other commands we deal with in this chapter, requires that you explicitly list any filenames that you wish to resolve. In any case, you want to be careful and use **svn resolve** only when you're certain that you've fixed the conflict in your file—once the temporary files are removed, Subversion will let you commit the file even if it still contains conflict markers.

If you ever get confused while editing the conflicted file, you can always consult the three files that Subversion creates for you in your working copy—including your file as it was before you updated. You can even use a third-party interactive merging tool to examine those three files.

---

[3] 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。

2.4.5.5. 丢弃你的修改而接收新获取的修订版本

If you get a conflict and decide that you want to throw out your changes, you can run **svn resolve --accept theirs-full** *CONFLICTED-PATH* and Subversion will discard your edits and remove the temporary files:

```
$ svn update
Conflict discovered in -'sandwich.txt'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: p
C    sandwich.txt
Updated to revision 2.
$ ls sandwich.*
sandwich.txt  sandwich.txt.mine  sandwich.txt.r2  sandwich.txt.r1
$ svn resolve ---accept theirs-full sandwich.txt
Resolved conflicted state of -'sandwich.txt'
```

2.4.5.6. 撤销：使用 **svn revert**

If you decide that you want to throw out your changes and start your edits again (whether this occurs after a conflict or anytime), just revert your changes:

```
$ svn revert sandwich.txt
Reverted -'sandwich.txt'
$ ls sandwich.*
sandwich.txt
```

Note that when you revert a conflicted file, you don't have to use **svn resolve**.

2.4.6. 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。

The **svn commit** command sends all of your changes to the repository. When you commit a change, you need to supply a *log message* describing your change. Your log message will be attached to the new revision you create. If your log message is brief, you may wish to supply it on the command line using the --message (or -m) option:

```
$ 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(-F)选项：

```
$ svn commit --F logmsg
Sending        sandwich.txt
Transmitting file data -.
Committed revision 4.
```

If you fail to specify either the `--message` or `--file` option, Subversion will automatically launch your favorite editor (see the information on `editor-cmd` in 第 7.1.3.2 节 "配置") for composing a log message.

提示

If you're in your editor writing a commit message and decide that you want to cancel your commit, you can just quit your editor without saving changes. If you've already saved your commit message, simply delete the text, save again, and then abort:

```
$ svn commit
Waiting for Emacs...Done

Log message unchanged or not specified
(a)bort, (c)ontinue, (e)dit
a
$
```

The repository doesn't know or care whether your changes make any sense as a whole; it checks only to make sure nobody else has changed any of the same files that you did when you weren't looking. If somebody *has* done that, the entire commit will fail with a message informing you that one or more of your files are out of date:

```
$ svn commit --m -"Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: File -'/sandwich.txt' is 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。）

At this point, you need to run **svn update**, deal with any merges or conflicts that result, and attempt your commit again.

That covers the basic work cycle for using Subversion. Subversion offers many other features that you can use to manage your repository and working copy, but most of your day-to-day use of Subversion will involve only the commands that we've discussed so far in this chapter. We will, however, cover a few more commands that you'll use fairly often.

2.5. 检验历史

Your Subversion repository is like a time machine. It keeps a record of every change ever committed and allows you to explore this history by examining previous versions of files and directories as well as the metadata that accompanies them. With a single Subversion command, you can check out the repository (or restore an existing working copy) exactly as it was at any date or revision number in the past. However, sometimes you just want to *peer into* the past instead of *going into* it.

有多个命令可以从版本库为你提供历史数据：

svn log
Shows you broad information: log messages with date and author information attached to revisions and which paths changed in each revision

svn diff
显示特定修改的行级详细信息

svn cat
Retrieves a file as it existed in a particular revision number and displays it on your screen

svn list
显示指定版本的目录中的文件

2.5.1. 产生历史修改列表

To find information about the history of a file or directory, use the **svn log** command. **svn log** will provide you with a record of who made changes to a file or directory, at what revision it changed, the time and date of that revision, and—if it was provided—the log message that accompanied the commit:

```
$ svn log
------------------------------------------------------------------------
r3 -| sally -| 2008-05-15 23:09:28 --0500 (Thu, 15 May 2008) -| 1
line

Added include lines and corrected # of cheese slices.
------------------------------------------------------------------------
r2 -| harry -| 2008-05-14 18:43:15 --0500 (Wed, 14 May 2008) -| 1
line

Added main() methods.
------------------------------------------------------------------------
r1 -| sally -| 2008-05-10 19:50:31 --0500 (Sat, 10 May 2008) -| 1
line

Initial import
------------------------------------------------------------------------
```

Note that the log messages are printed in *reverse chronological order* by default. If you wish to see a different range of revisions in a particular order or just a single revision, pass the `--revision` (`-r`) option:

```
$ 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
```

You can also examine the log history of a single file or directory. For example:

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

These will display log messages *only* for those revisions in which the working file (or URL) changed.

> **为什么 svn log 不会显示我刚刚提交的内容？**
>
> If you make a commit and immediately type **`svn log`** with no arguments, you may notice that your most recent commit doesn't show up in the list of log messages. This is due to a combination of the behavior of **svn commit** and the default behavior of **svn log**. First, when you commit changes to the repository, **svn** bumps only the revision of files (and directories) that it commits, so usually the parent directory remains at the older revision (See 第 1.3.5.1 节 "更新和提交是分开的" for an explanation of why). **svn log** then defaults to fetching the history of the directory at its current revision, and thus you don't see the newly committed changes. The solution here is to either update your working copy or explicitly provide a revision number to **svn log** by using the `--revision` (`-r`) option.

If you want even more information about a file or directory, **svn log** also takes a `--verbose` (`-v`) option. Because Subversion allows you to move and copy files and directories, it is important to be able to track path changes in the filesystem. So, in verbose mode, **svn log** will include a list of changed paths in a revision in its output:

```
$ svn log --r 8 --v
------------------------------------------------------------------------
r8 -| sally -| 2008-05-21 13:19:25 --0500 (Wed, 21 May 2008) -| 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 also takes a `--quiet` (-q) option, which suppresses the body of the log message. When combined with `--verbose`, it gives just the names of the changed files.

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

At first glance, this seems like an error. But recall that while revisions are repository-wide, **svn log** operates on a path in the repository. If you supply no path, Subversion uses the current working directory as the default target. As a result, if you're operating in a subdirectory of your working copy and attempt to see the log of a revision in which neither that directory nor any of its children was changed, Subversion will show you an empty log. If you want to see what changed in that revision, try pointing **svn log** directly at the topmost URL of your repository, as in **svn log -r 2 http://svn.collab.net/repos/svn**.

2.5.2. 检查历史修改详情

We've already seen **svn diff** before—it displays file differences in unified diff format; we used it to show the local modifications made to our working copy before committing to the repository.

事实上，**svn diff**有三种不同的用法：

• 检查本地修改

• 比较工作副本与版本库

• 比较版本库中的版本

2.5.2.1. 检查本地修改

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"副本：

```
$ 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
$
```

2.5.2.2. 比较工作副本和版本库

如果传递一个 `--revision` (`-r`) 参数，你的工作副本会与版本库中的指定版本比较：

```
$ 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
$
```

2.5.2.3. 比较版本库中的版本

如果通过 `--revision` (`-r`) 传递两个通过冒号分开的版本号，这两个版本会直接比较：

```
$ 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
$
```

A more convenient way of comparing one revision to the previous revision is to use the `--change` (`-c`) option:

```
$ 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副本，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
…
$
```

2.5.3. 浏览版本库

Using **svn cat** and **svn list**, you can view various revisions of files and directories without changing the working revision of your working copy. In fact, you don't even need a working copy to use either one.

2.5.3.1. 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
$
```

2.5.3.2. 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。

警告

The **svn list** command with no arguments defaults to the *repository URL* of the current working directory, *not* the local working copy directory. After all, if you want a listing of your local directory, you could use just plain **ls** (or any reasonable non-Unixy equivalent).

2.5.4. 获得旧的版本库快照

In addition to all of the previous commands, you can use **svn update** and **svn checkout** with the --revision option to take an entire working copy "back in time": [4]

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

提示

Many Subversion newcomers attempt to use the preceding **svn update** example to "undo" committed changes, but this won't work as you can't commit changes that you obtain from backdating a working copy if the changed files have newer revisions. See 第 4.3.5 节 "找回删除的项目" for a description of how to "undo" a commit.

Lastly, if you're building a release and wish to bundle up your files from Subversion but don't want those pesky .svn directories in the way, you can use **svn export** to create a local copy of

---

[4]看到了吧？我们说过Subversion是一个时间机器。

all or part of your repository sans `.svn` directories. As with **svn update** and **svn checkout**, you can also pass the `--revision` option to **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
…
```

2.6. 有时你只需要清理

现在我们已经覆盖了使用Subversion的日常任务，我们会检阅一些工作副本相关的管理任务。

2.6.1. 处理你的工作副本

Subversion doesn't track either the state or the existence of working copies on the server, so there's no server overhead to keeping working copies around. Likewise, there's no need to let the server know that you're going to delete a working copy.

如果你还是喜欢使用工作副本，直到你再次使用它之前，把其保留在磁盘没有任何错误，任何时候一个**svn update**命令可以让使用的文件成为最新。

However, if you're definitely not going to use a working copy again, you can safely delete the entire thing, but you'd be well served to take a look through the working copy for unversioned files. To find these files, run **`svn status`** and review any files that are prefixed with a ? to make certain that they're not of importance. After you're done reviewing, you can safely delete your working copy.

2.6.2. 从中断中恢复

When Subversion modifies your working copy (or any information within `.svn`), it tries to do so as safely as possible. Before changing the working copy, Subversion writes its intentions to a logfile. Next, it executes the commands in the logfile to apply the requested change, holding a lock on the relevant part of the working copy while it works—to prevent other Subversion clients from accessing the working copy mid-change. Finally, Subversion removes the logfile. Architecturally, this is similar to a journaled filesystem. If a Subversion operation is interrupted (e.g, if the process is killed or if the machine crashes), the logfiles remain on disk. By reexecuting the logfiles, Subversion can complete the previously started operation, and your working copy can get itself back into a consistent state.

And this is exactly what **svn cleanup** does: it searches your working copy and runs any leftover logs, removing working copy locks in the process. If Subversion ever tells you that some part of your working copy is "locked," this is the command that you should run. Also, **svn status** will display an L next to locked items:

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

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

Don't confuse these working copy locks with the ordinary locks that Subversion users create when using the lock-modify-unlock model of concurrent version control; see the sidebar 锁定的三种含义 for clarification.

2.7. 处理结构性冲突

So far, we have only talked about conflicts at the level of file content. When you and your collaborators make overlapping changes within the same file, Subversion forces you to merge those changes before you can commit.[5]

But what happens if your collaborators move or delete a file that you are still working on? Maybe there was a miscommunication, and one person thinks the file should be deleted, while another person still wants to commit changes to the file. Or maybe your collaborators did some refactoring, renaming files and moving around directories in the process. If you were still working on these files, those modifications may need to be applied to the files at their new location. Such conflicts manifest themselves at the directory tree structure level rather than at the file content level, and are known as *tree conflicts*.

Subversion 1.6 之前的树冲突

Prior to Subversion 1.6, tree conflicts could yield rather unexpected results. For example, if a file was locally modified, but had been renamed in the repository, running **svn update** would make Subversion carry out the following steps:

- Check the file to be renamed for local modifications.

- Delete the file at its old location, and if it had local modifications, keep an on-disk copy of the file at the old location. This on-disk copy now appears as an unversioned file in the working copy.

- Add the file, as it exists in the repository, at its new location.

When this situation arises, there is the possibility that the user makes a commit without realizing that local modifications have been left in a now-unversioned file in the working copy, and have not reached the repository. This gets more and more likely (and tedious) if the number of files affected by this problem is large.

Since Subversion 1.6, this and other similar situations are flagged as conflicts in the working copy.

---

[5]Well, you could mark files containing conflict markers as resolved and commit them, if you really wanted to. But this is rarely done in practice.

As with textual conflicts, tree conflicts prevent a commit from being made from the conflicted state, giving the user the opportunity to examine the state of the working copy for potential problems arising from the tree conflict, and resolving any such problems before committing.

2.7.1. 树冲突示例

Suppose a software project you were working on currently looked like this:

```
$ svn ls --Rv svn://svn.example.com/trunk/
      4 harry                   Feb 06 14:34 -./
      4 harry            23 Feb 06 14:34 COPYING
      4 harry            41 Feb 06 14:34 Makefile
      4 harry            33 Feb 06 14:34 README
      4 harry               Feb 06 14:34 code/
      4 harry            51 Feb 06 14:34 code/bar.c
      4 harry           124 Feb 06 14:34 code/foo.c
```

Your collaborator Harry has renamed the file bar.c to baz.c. You are still working on bar.c in your working copy, but you don't know yet that the file has been renamed in the repository.

Harry 的提交日志是:

```
$ svn log --r5 svn://svn.example.com/trunk
------------------------------------------------------------------------
r5 -| harry -| 2009-02-06 14:42:59 +0000 (Fri, 06 Feb 2009) -| 2
lines
Changed paths:
   M -/trunk/Makefile
   D -/trunk/code/bar.c
   A -/trunk/code/baz.c (from -/trunk/code/bar.c:4)

Rename bar.c to baz.c, and adjust Makefile accordingly.
```

你的本地修改是:

```
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c   (revision 4)
+++ code/foo.c   (working copy)
@@ --3,5 +3,5 @@
 int main(int argc, char *argv[])
 {
        printf("I don't like being moved around!\n%s", bar());
```

```
-        return 0;
+        return 1;
 -}
Index: code/bar.c
===============================================================
--- code/bar.c   (revision 4)
+++ code/bar.c   (working copy)
@@ --1,4 +1,4 @@
 const char *bar(void)
 {
-        return -"Me neither!\n";
+        return -"Well, I do like being moved around!\n";
 -}
```

Your changes are all based on revision 4. They cannot be committed because Harry has already checked in revision 5:

```
$ svn commit --m -"Small fixes"
Sending        code/bar.c
Sending        code/foo.c
Transmitting file data -..
svn: Commit failed (details follow):
svn: File not found: transaction -'5-5', path -'/trunk/code/bar.c'
```

At this point, you need to run **svn update**. Besides bringing our working copy up to date so that you can see Harry's changes, this also flags a tree conflict so you have the opportunity to evaluate and properly resolve it.

```
$ svn update
   C code/bar.c
A    code/baz.c
U    Makefile
Updated to revision 5.
Summary of conflicts:
  Tree conflicts: 1
```

In its output, **svn update** signifies tree conflicts using a capital C in the fourth output column. **svn status** reveals additional details of the conflict:

```
$ svn status
M       code/foo.c
A  +  C code/bar.c
      >    local edit, incoming delete upon update
M       code/baz.c
```

Note how bar.c is automatically scheduled for re-addition in your working copy, which simplifies things in case you want to keep the file.

Because a move in Subversion is implemented as a copy operation followed by a delete operation, and these two operations cannot be easily related to one another during an update, all Subversion can warn you about is an incoming delete operation on a locally modified file. This delete operation *may* be part of a move, or it could be a genuine delete operation. Talking to your collaborators, or, as a last resort, **svn log**, is a good way to find out what has actually happened.

Both `foo.c` and `baz.c` are reported as locally modified in the output of **svn status**. You made the changes to `foo.c` yourself, so this should not be surprising. But why is `baz.c` reported as locally modified?

The answer is that despite the limitations of the move implementation, Subversion was smart enough to transfer your local edits in `bar.c` into `baz.c`:

```
$ svn diff code/baz.c
Index: code/baz.c
===================================================================
--- code/baz.c  (revision 5)
+++ code/baz.c  (working copy)
@@ --1,4 +1,4 @@
 const char *bar(void)
 {
-        return -"Me neither!\n";
+        return -"Well, I do like being moved around!\n";
 -}
```

🚫 **警告**

Local edits to the file `bar.c`, which is renamed during an update to `baz.c`, will only be applied to `bar.c` if your working copy of `bar.c` is based on the revision in which it was last modified before being moved in the repository. Otherwise, Subversion will resort to retreiving `baz.c` from the repository, and will not try to transfer your local modifications to it. You will have to do so manually.

`bar.c` is now said to be the *victim* of a tree conflict. It cannot be committed until the conflict is resolved:

```
$ svn commit --m -"Small fixes"
svn: Commit failed (details follow):
svn: Aborting commit: -'code/bar.c' remains in conflict
```

So how can this conflict be resolved? You can either agree or disagree with the move Harry made. In case you agree, you can delete `bar.c` and mark the tree conflict as resolved:

```
$ svn remove ---force code/bar.c
D         code/bar.c
$ svn resolve ---accept=working code/bar.c
Resolved conflicted state of -'code/bar.c'
```

```
$ svn status
M       code/foo.c
M       code/baz.c
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c  (revision 5)
+++ code/foo.c  (working copy)
@@ --3,5 +3,5 @@
 int main(int argc, char *argv[])
 {
         printf("I don't like being moved around!\n%s", bar());
-        return 0;
+        return 1;
 -}
Index: code/baz.c
===================================================================
--- code/baz.c  (revision 5)
+++ code/baz.c  (working copy)
@@ --1,4 +1,4 @@
 const char *bar(void)
 {
-        return -"Me neither!\n";
+        return -"Well, I do like being moved around!\n";
 -}
```

If you do not agree with the move, you can delete baz.c instead, after making sure any changes made to it after it was renamed are either preserved or not worth keeping. Do not forget to revert the changes Harry made to the Makefile. Since bar.c is already scheduled for re-addition, there is nothing else left to do, and the conflict can be marked resolved:

```
$ svn remove ---force code/baz.c
D         code/baz.c
$ svn resolve ---accept=working code/bar.c
Resolved conflicted state of -'code/bar.c'
$ svn status
M       code/foo.c
A  +    code/bar.c
D       code/baz.c
M       Makefile
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c (revision 5)
+++ code/foo.c (working copy)
@@ --3,5 +3,5 @@
 int main(int argc, char *argv[])
 {
```

```
  printf("I don't like being moved around!\n%s", bar());
- return 0;
+ return 1;
 -}
Index: code/bar.c
===================================================================
--- code/bar.c (revision 5)
+++ code/bar.c (working copy)
@@ --1,4 +1,4 @@
 const char *bar(void)
 {
- return -"Me neither!\n";
+ return -"Well, I do like being moved around!\n";
 -}
Index: code/baz.c
===================================================================
--- code/baz.c (revision 5)
+++ code/baz.c (working copy)
@@ --1,4 +0,0 @@
-const char *bar(void)
-{
- return -"Me neither!\n";
-}
Index: Makefile
===================================================================
--- Makefile (revision 5)
+++ Makefile (working copy)
@@ --1,2 +1,2 @@
 foo:
- $(CC) --o $@ code/foo.c code/baz.c
+ $(CC) --o $@ code/foo.c code/bar.c
```

In either case, you have now resolved your first tree conflict! You can commit your changes and tell Harry during tea break about all the extra work he caused for you.

2.8. 总结

Now we've covered most of the Subversion client commands. Notable exceptions are those dealing with branching and merging (see 第 4 章 分支与合并) and properties (see 第 3.2 节 "属性"). However, you may want to take a moment to skim through 第 9 章 *Subversion* 完全参考 to get an idea of all the different commands that Subversion has—and how you can use them to make your work easier.

第 **3** 章 高级主题

If you've been reading this book chapter by chapter, from start to finish, you should by now have acquired enough knowledge to use the Subversion client to perform the most common version control operations. You understand how to check out a working copy from a Subversion repository. You are comfortable with submitting and receiving changes using the **svn commit** and **svn update** operations. You've probably even developed a reflex that causes you to run the **svn status** command almost unconsciously. For all intents and purposes, you are ready to use Subversion in a typical environment.

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。

This chapter highlights some of Subversion's features that, while important, aren't part of the typical user's daily routine. It assumes that you are familiar with Subversion's basic file and directory versioning capabilities. If you aren't, you'll want to first read 第 1 章 基本概念 and 第 2 章 基本使用. Once you've mastered those basics and consumed this chapter, you'll be a Subversion power user!

3.1. 版本清单

As we described in 第 1.3.3 节 "修订版本", revision numbers in Subversion are pretty straightforward—integers that keep getting larger as you commit more changes to your versioned data. Still, it doesn't take long before you can no longer remember exactly what happened in each and every revision. Fortunately, the typical Subversion workflow doesn't often demand that you supply arbitrary revisions to the Subversion operations you perform. For operations that *do* require a revision specifier, you generally supply a revision number that you saw in a commit email, in the output of some other Subversion operation, or in some other context that would give meaning to that particular number.

But occasionally, you need to pinpoint a moment in time for which you don't already have a revision number memorized or handy. So besides the integer revision numbers, **svn** allows as input some additional forms of revision specifiers: *revision keywords* and revision dates.

> **注意**
>
> The various forms of Subversion revision specifiers can be mixed and matched when used to specify revision ranges. For example, you can use `-r REV1:REV2` where *REV1* is a revision keyword and *REV2* is a revision number, or where *REV1* is a date and *REV2* is a revision keyword, and so on. The individual revision specifiers are independently evaluated, so you can put whatever you want on the opposite sides of that colon.

3.1.1. 修订版本关键字

The Subversion client understands a number of revision keywords. These keywords can be used instead of integer arguments to the `--revision` (`-r`) option, and are resolved into specific revision numbers by Subversion:

HEAD
>版本库中最新的(或者是"最年轻的")版本。

BASE
>The revision number of an item in a working copy. If the item has been locally modified, this refers to the way the item appears without those local modifications.

COMMITTED
>项目最近修改的修订版本，与BASE相同或更早。

PREV
>The revision immediately *before* the last revision in which an item changed. Technically, this boils down to COMMITTED−1.

As can be derived from their descriptions, the PREV, BASE, and COMMITTED revision keywords are used only when referring to a working copy path—they don't apply to repository URLs. HEAD, on the other hand, can be used in conjunction with both of these path types.

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
```

3.1.2. 版本日期

Revision numbers reveal nothing about the world outside the version control system, but sometimes you need to correlate a moment in real time with a moment in version history. To facilitate this, the `--revision` (`-r`) option can also accept as input date specifiers wrapped in curly braces (`{` and `}`). Subversion accepts the standard ISO-8601 date and time formats, plus a few others. Here are some examples. (Remember to use quotes around any date that contains spaces.)

```
$ 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？**
>
> If you specify a single date as a revision without specifying a time of day (for example `2006-11-27`), you may think that Subversion should give you the last revision that took place on the 27th of November. Instead, you'll get back a revision from the 26th, or even earlier. Remember that Subversion will find the *most recent revision of the repository* as of the date you give. If you give a date without a timestamp, such as `2006-11-27`, Subversion assumes a time of 00:00:00, so looking for the most recent revision won't return anything on the 27th.
>
> 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}`)。

You can also use a range of dates. Subversion will find all revisions between both dates, inclusive:

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

警告

Since the timestamp of a revision is stored as an unversioned, modifiable property of the revision (see 第 3.2 节 "属性"), revision timestamps can be changed to represent complete falsifications of true chronology, or even removed altogether. Subversion's ability to correctly convert revision dates into real revision numbers depends on revision datestamps maintaining a sequential ordering—the younger the revision, the younger its timestamp. If this ordering isn't maintained, you will likely find that trying to use dates to specify revision ranges in your repository doesn't always return the data you might have expected.

3.2. 属性

We've already covered in detail how Subversion stores and retrieves various versions of files and directories in its repository. Whole chapters have been devoted to this most fundamental piece of functionality provided by the tool. And if the versioning support stopped there, Subversion would still be complete from a version control perspective.

但不仅仅如此。

In addition to versioning your directories and files, Subversion provides interfaces for adding, modifying, and removing versioned metadata on each of your versioned directories and files. We refer to this metadata as *properties*, and they can be thought of as two-column tables that map property names to arbitrary values attached to each item in your working copy. Generally speaking, the names and values of the properties can be whatever you want them to be, with the constraint that the names must contain only ASCII characters. And the best part about these properties is that they, too, are versioned, just like the textual contents of your files. You can modify, commit, and revert property changes as easily as you can file content changes. And the sending and receiving of property changes occurs as part of your typical commit and update operations—you don't have to change your basic processes to accommodate them.

注意

Subversion has reserved the set of properties whose names begin with `svn:` as its own. While there are only a handful of such properties in use today, you should avoid creating custom properties for your own needs whose names begin with this prefix. Otherwise, you run the risk that a future release of Subversion will grow support for a feature or behavior driven by a property of the same name but with perhaps an entirely different interpretation.

Properties show up elsewhere in Subversion, too. Just as files and directories may have arbitrary property names and values attached to them, each revision as a whole may have arbitrary properties attached to it. The same constraints apply—human-readable names and anything-you-want binary values. The main difference is that revision properties are not

versioned. In other words, if you change the value of, or delete, a revision property, there's no way, within the scope of Subversion's functionality, to recover the previous value.

Subversion has no particular policy regarding the use of properties. It asks only that you not use property names that begin with the prefix `svn:`. That's the namespace that it sets aside for its own use. And Subversion does, in fact, use properties—both the versioned and unversioned variety. Certain versioned properties have special meaning or effects when found on files and directories, or they house a particular bit of information about the revisions on which they are found. Certain revision properties are automatically attached to revisions by Subversion's commit process, and they carry information about the revision. Most of these properties are mentioned elsewhere in this or other chapters as part of the more general topics to which they are related. For an exhaustive list of Subversion's predefined properties, see 第 9.10 节 "Subversion 属性".

In this section, we will examine the utility—both to users of Subversion and to Subversion itself—of property support. You'll learn about the property-related **svn** subcommands and how property modifications affect your normal Subversion workflow.

3.2.1. 为什么需要属性？

Just as Subversion uses properties to store extra information about the files, directories, and revisions that it contains, you might also find properties to be of similar use. You might find it useful to have a place close to your versioned data to hang custom metadata about that data.

Say you wish to design a web site that houses many digital photos and displays them with captions and a datestamp. Now, your set of photos is constantly changing, so you'd like to have as much of this site automated as possible. These photos can be quite large, so as is common with sites of this nature, you want to provide smaller thumbnail images to your site visitors.

Now, you can get this functionality using traditional files. That is, you can have your `image123.jpg` and an `image123-thumbnail.jpg` side by side in a directory. Or if you want to keep the filenames the same, you might have your thumbnails in a different directory, such as `thumbnails/image123.jpg`. You can also store your captions and datestamps in a similar fashion, again separated from the original image file. But the problem here is that your collection of files multiplies with each new photo added to the site.

Now consider the same web site deployed in a way that makes use of Subversion's file properties. Imagine having a single image file, `image123.jpg`, with properties set on that file that are named `caption`, `datestamp`, and even `thumbnail`. Now your working copy directory looks much more manageable—in fact, it looks to the casual browser like there are nothing but image files in it. But your automation scripts know better. They know that they can use **svn** (or better yet, they can use the Subversion language bindings—see 第 8.3 节 "使用 API") to dig out the extra information that your site needs to display without having to read an index file or play path manipulation games.

注意

While Subversion places few restrictions on the names and values you use for properties, it has not been designed to optimally carry large property values or large

sets of properties on a given file or directory. Subversion commonly holds all the property names and values associated with a single item in memory at the same time, which can cause detrimental performance or failed operations when extremely large property sets are used.

Custom revision properties are also frequently used. One common such use is a property whose value contains an issue tracker ID with which the revision is associated, perhaps because the change made in that revision fixes a bug filed in the tracker issue with that ID. Other uses include hanging more friendly names on the revision—it might be hard to remember that revision 1935 was a fully tested revision. But if there's, say, a `test-results` property on that revision with the value `all passing`, that's meaningful information to have.

可搜索性(或者，为什么不使用属性)

For all their utility, Subversion properties—or, more accurately, the available interfaces to them—have a major shortcoming: while it is a simple matter to *set* a custom property, *finding* that property later is a whole different ball of wax.

Trying to locate a custom revision property generally involves performing a linear walk across all the revisions of the repository, asking of each revision, "Do you have the property I'm looking for?" Trying to find a custom versioned property is painful, too, and often involves a recursive **svn propget** across an entire working copy. In your situation, that might not be as bad as a linear walk across all revisions. But it certainly leaves much to be desired in terms of both performance and likelihood of success, especially if the scope of your search would require a working copy from the root of your repository.

For this reason, you might choose—especially in the revision property use case—to simply add your metadata to the revision's log message using some policy-driven (and perhaps programmatically enforced) formatting that is designed to be quickly parsed from the output of **svn log**. It is quite common to see the following in Subversion log messages:

```
Issue(s): IZ2376, IZ1919
Reviewed by:  sally

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

But here again lies some misfortune. Subversion doesn't yet provide a log message templating mechanism, which would go a long way toward helping users be consistent with the formatting of their log-embedded revision metadata.

3.2.2. 操作属性

The **svn** program affords a few ways to add or modify file and directory properties. For properties with short, human-readable values, perhaps the simplest way to add a new property is to specify the property name and value on the command line of the **svn propset** subcommand:

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

But we've been touting the flexibility that Subversion offers for your property values. And if you are planning to have a multiline textual, or even binary, property value, you probably do not want to supply that value on the command line. So the **svn propset** subcommand takes a `--file` (`-F`) option for specifying the name of a file that contains the new property value.

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

There are some restrictions on the names you can use for properties. A property name must start with a letter, a colon (`:`), or an underscore (`_`); after that, you can also use digits, hyphens (`-`), and periods (`.`). [1]

In addition to the **propset** command, the **svn** program supplies the **propedit** command. This command uses the configured editor program (see 第 7.1.3.2 节 "配置") to add or modify properties. When you run the command, **svn** invokes your editor program on a temporary file that contains the current value of the property (or that is empty, if you are adding a new property). Then, you just modify that value in your editor program until it represents the new value you wish to store for the property, save the temporary file, and then exit the editor program. If Subversion detects that you've actually changed the existing value of the property, it will accept that as the new property value. If you exit your editor without making any changes, no property modification will occur:

```
$ svn propedit copyright calc/button.c  ### exit the editor
without changes
No changes to property -'copyright' on -'calc/button.c'
$
```

We should note that, as with other **svn** subcommands, those related to properties can act on multiple paths at once. This enables you to modify properties on whole sets of files with a single command. For example, we could have done the following:

```
$ 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'
…
$
```

All of this property adding and editing isn't really very useful if you can't easily get the stored property value. So the **svn** program supplies two subcommands for displaying the names and

---

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

values of properties stored on files and directories. The **svn proplist** command will list the names of properties that exist on a path. Once you know the names of the properties on the node, you can request their values individually using **svn propget**. This command will, given a property name and a path (or set of paths), print the value of the property to the standard output stream.

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

There's even a variation of the **proplist** command that will list both the name and the value for all of the properties. Simply supply the `--verbose` (`-v`) option.

```
$ 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.
...
```

The last property-related subcommand is **propdel**. Since Subversion allows you to store properties with empty values, you can't remove a property altogether using **svn propedit** or **svn propset**. For example, this command will *not* yield the desired effect:

```
$ 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 -:
$
```

You need to use the **propdel** subcommand to delete properties altogether. The syntax is similar to the other property commands:

```
$ 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
$
```

Remember those unversioned revision properties? You can modify those, too, using the same **svn** subcommands that we just described. Simply add the `--revprop` command-line parameter and specify the revision whose property you wish to modify. Since revisions are global, you don't need to specify a target path to these property-related commands so long as you are positioned in a working copy of the repository whose revision property you wish to modify. Otherwise, you can simply provide the URL of any path in the repository of interest (including the repository's root URL). For example, you might want to replace the commit log message of an existing revision. [2] If your current working directory is part of a working copy of your repository, you can simply run the **svn propset** command with no target path:

```
$ svn propset svn:log -'* button.c: Fix a compiler warning.' --r11 ---revprop
property -'svn:log' set on repository revision -'11'
$
```

But even if you haven't checked out a working copy from that repository, you can still effect the property change by providing the repository's root URL:

```
$ 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'
$
```

Note that the ability to modify these unversioned properties must be explicitly added by the repository administrator (see 第 5.4.2 节 "修正提交消息"). That's because the properties aren't versioned, so you run the risk of losing information if you aren't careful with your edits. The repository administrator can set up methods to protect against this loss, and by default, modification of unversioned properties is disabled.

提示

Users should, where possible, use **svn propedit** instead of **svn propset**. While the end result of the commands is identical, the former will allow them to see the current value of the property that they are about to change, which helps them to verify that they are, in fact, making the change they think they are making. This is especially true when modifying unversioned revision properties. Also, it is significantly easier to modify multiline property values in a text editor than at the command line.

---

[2] 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。

3.2.3. 属性和 **Subversion** 工作流程

Now that you are familiar with all of the property-related **svn** subcommands, let's see how property modifications affect the usual Subversion workflow. As we mentioned earlier, file and directory properties are versioned, just like your file contents. As a result, Subversion provides the same opportunities for merging—cleanly or with conflicts—someone else's modifications into your own.

As with file contents, your property changes are local modifications, made permanent only when you commit them to the repository with **svn commit**. Your property changes can be easily unmade, too—the **svn revert** command will restore your files and directories to their unedited states—contents, properties, and all. Also, you can receive interesting information about the state of your file and directory properties by using the **svn status** and **svn diff** commands.

```
$ 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

$
```

Notice how the **status** subcommand displays M in the second column instead of the first. That is because we have modified the properties on calc/button.c, but not its textual contents. Had we changed both, we would have seen M in the first column, too. (We cover **svn status** in 第 2.4.3.1 节 "查看你的修改概况").

> **属性冲突**
>
> As with file contents, local property modifications can conflict with changes committed by someone else. If you update your working copy directory and receive property changes on a versioned object that clash with your own, Subversion will report that the object is in a conflicted state.
>
> ```
> $ svn update calc
> M  calc/Makefile.in
> Conflict for property -'linecount' discovered on -'calc/
> button.c'.
> Select: (p) postpone, (df) diff-full, (e) edit,
>         (s) show all options: p
>  C calc/button.c
> Updated to revision 143.
> $
> ```
>
> Subversion will also create, in the same directory as the conflicted object, a file with a `.prej` extension that contains the details of the conflict. You should examine the contents of this file so you can decide how to resolve the conflict. Until the conflict is resolved, you will see a `C` in the second column of **svn status** output for that object, and attempts to commit your local modifications will fail.
>
> ```
> $ svn status calc
>  C     calc/button.c
> ?      calc/button.c.prej
> $ cat calc/button.c.prej
> Trying to change property -'linecount' from -'1267' to -'1301',
> but property has been locally changed from -'1267' to -'1256'.
> $
> ```
>
> 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。

You might also have noticed the nonstandard way that Subversion currently displays property differences. You can still use **svn diff** and redirect its output to create a usable patch file. The **patch** program will ignore property patches—as a rule, it ignores any noise it can't understand. This does, unfortunately, mean that to fully apply a patch generated by **svn diff**, any property modifications will need to be applied by hand.

3.2.4. 自动设置属性

Properties are a powerful feature of Subversion, acting as key components of many Subversion features discussed elsewhere in this and other chapters—textual diff and merge support, keyword substitution, newline translation, and so on. But to get the full benefit of properties, they must be set on the right files and directories. Unfortunately, that step can be

easily forgotten in the routine of things, especially since failing to set a property doesn't usually result in an obvious error (at least compared to, say, failing to add a file to version control). To help your properties get applied to the places that need them, Subversion provides a couple of simple but useful features.

Whenever you introduce a file to version control using the **svn add** or **svn import** commands, Subversion tries to assist by setting some common file properties automatically. First, on operating systems whose filesystems support an execute permission bit, Subversion will automatically set the `svn:executable` property on newly added or imported files whose execute bit is enabled. (See 第 3.3.2 节 "文件的可执行性" later in this chapter for more about this property.)

Second, Subversion tries to determine the file's MIME type. If you've configured a `mime-types-files` runtime configuration parameter, Subversion will try to find a MIME type mapping in that file for your file's extension. If it finds such a mapping, it will set your file's `svn:mime-type` property to the MIME type it found. If no mapping file is configured, or no mapping for your file's extension could be found, Subversion runs a very basic heuristic to determine whether the file contains nontextual content. If so, it automatically sets the `svn:mime-type` property on that file to `application/octet-stream` (the generic "this is a collection of bytes" MIME type). Of course, if Subversion guesses incorrectly, or if you wish to set the `svn:mime-type` property to something more precise—perhaps `image/png` or `application/x-shockwave-flash`—you can always remove or edit that property. (For more on Subversion's use of MIME types, see 第 3.3.1 节 "文件内容类型" later in this chapter.)

Subversion also provides, via its runtime configuration system (see 第 7.1 节 "运行配置区"), a more flexible automatic property setting feature that allows you to create mappings of filename patterns to property names and values. Once again, these mappings affect adds and imports, and can not only override the default MIME type decision made by Subversion during those operations, but can also set additional Subversion or custom properties, too. For example, you might create a mapping that says that anytime you add JPEG files—ones whose names match the pattern `*.jpg`—Subversion should automatically set the `svn:mime-type` property on those files to `image/jpeg`. Or perhaps any files that match `*.cpp` should have `svn:eol-style` set to `native`, and `svn:keywords` set to `Id`. Automatic property support is perhaps the handiest property-related tool in the Subversion toolbox. See 第 7.1.3.2 节 "配置" for more about configuring that support.

3.3. 文件移植性

Fortunately for Subversion users who routinely find themselves on different computers with different operating systems, Subversion's command-line program behaves almost identically on all those systems. If you know how to wield **svn** on one platform, you know how to wield it everywhere.

However, the same is not always true of other general classes of software or of the actual files you keep in Subversion. For example, on a Windows machine, the definition of a "text file" would be similar to that used on a Linux box, but with a key difference—the character sequences used to mark the ends of the lines of those files. There are other differences, too. Unix platforms have (and Subversion supports) symbolic links; Windows does not. Unix platforms use filesystem permission to determine executability; Windows uses filename extensions.

Because Subversion is in no position to unite the whole world in common definitions and implementations of all of these things, the best it can do is to try to help make your life simpler when you need to work with your versioned files and directories on multiple computers and operating systems. This section describes some of the ways Subversion does this.

3.3.1. 文件内容类型

Subversion joins the ranks of the many applications that recognize and make use of Multipurpose Internet Mail Extensions (MIME) content types. Besides being a general-purpose storage location for a file's content type, the value of the `svn:mime-type` file property determines some behavioral characteristics of Subversion itself.

识别文件类型

Various programs on most modern operating systems make assumptions about the type and format of the contents of a file by the file's name, specifically its file extension. For example, files whose names end in `.txt` are generally assumed to be human-readable; that is, able to be understood by simple perusal rather than requiring complex processing to decipher. Files whose names end in `.png`, on the other hand, are assumed to be of the Portable Network Graphics type—not human-readable at all, and sensible only when interpreted by software that understands the PNG format and can render the information in that format as a raster image.

Unfortunately, some of those extensions have changed their meanings over time. When personal computers first appeared, a file named `README.DOC` would have almost certainly been a plain-text file, just like today's `.txt` files. But by the mid-1990s, you could almost bet that a file of that name would not be a plain-text file at all, but instead a Microsoft Word document in a proprietary, non-human-readable format. But this change didn't occur overnight—there was certainly a period of confusion for computer users over what exactly they had in hand when they saw a `.DOC` file. [3]

The popularity of computer networking cast still more doubt on the mapping between a file's name and its content. With information being served across networks and generated dynamically by server-side scripts, there was often no real file per se, and therefore no filename. Web servers, for example, needed some other way to tell browsers what they were downloading so that the browser could do something intelligent with that information, whether that was to display the data using a program registered to handle that datatype or to prompt the user for where on the client machine to store the downloaded data.

Eventually, a standard emerged for, among other things, describing the contents of a data stream. In 1996, RFC 2045 was published. It was the first of five RFCs describing MIME. It describes the concept of media types and subtypes and recommends a syntax for the representation of those types. Today, MIME media types—or "MIME types"—are used almost universally across email applications, web servers, and other software as the de facto mechanism for clearing up the file content confusion.

---

[3] 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！

For example, one of the benefits that Subversion typically provides is contextual, line-based merging of changes received from the server during an update into your working file. But for files containing nontextual data, there is often no concept of a "line." So, for versioned files whose `svn:mime-type` property is set to a nontextual MIME type (generally, something that doesn't begin with `text/`, though there are exceptions), Subversion does not attempt to perform contextual merges during updates. Instead, any time you have locally modified a binary working copy file that is also being updated, your file is left untouched and Subversion creates two new files. One file has a `.oldrev` extension and contains the BASE revision of the file. The other file has a `.newrev` extension and contains the contents of the updated revision of the file. This behavior is really for the protection of the user against failed attempts at performing contextual merges on files that simply cannot be contextually merged.

警告

The `svn:mime-type` property, when set to a value that does not indicate textual file contents, can cause some unexpected behaviors with respect to other properties. For example, since the idea of line endings (and therefore, line-ending conversion) makes no sense when applied to nontextual files, Subversion will prevent you from setting the `svn:eol-style` property on such files. This is obvious when attempted on a single file target—**svn propset** will error out. But it might not be as clear if you perform a recursive property set, where Subversion will silently skip over files that it deems unsuitable for a given property.

Beginning in Subversion 1.5, users can configure a new `mime-types-file` runtime configuration parameter, which identifies the location of a MIME types mapping file. Subversion will consult this mapping file to determine the MIME type of newly added and imported files.

Also, if the `svn:mime-type` property is set, then the Subversion Apache module will use its value to populate the `Content-type:` HTTP header when responding to GET requests. This gives your web browser a crucial clue about how to display a file when you use it to peruse your Subversion repository's contents.

3.3.2. 文件的可执行性

On many operating systems, the ability to execute a file as a command is governed by the presence of an execute permission bit. This bit usually defaults to being disabled, and must be explicitly enabled by the user for each file that needs it. But it would be a monumental hassle to have to remember exactly which files in a freshly checked-out working copy were supposed to have their executable bits toggled on, and then to have to do that toggling. So, Subversion provides the `svn:executable` property as a way to specify that the executable bit for the file on which that property is set should be enabled, and Subversion honors that request when populating working copies with such files.

This property has no effect on filesystems that have no concept of an executable permission bit, such as FAT32 and NTFS. [4] Also, although it has no defined values, Subversion will force its value to * when setting this property. Finally, this property is valid only on files, not on directories.

---

[4]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。

3.3.3. 行结束字符序列

Unless otherwise noted using a versioned file's `svn:mime-type` property, Subversion assumes the file contains human-readable data. Generally speaking, Subversion uses this knowledge only to determine whether contextual difference reports for that file are possible. Otherwise, to Subversion, bytes are bytes.

This means that by default, Subversion doesn't pay any attention to the type of *end-of-line (EOL) markers* used in your files. Unfortunately, different operating systems have different conventions about which character sequences represent the end of a line of text in a file. For example, the usual line-ending token used by software on the Windows platform is a pair of ASCII control characters—a carriage return (`CR`) followed by a line feed (`LF`). Unix software, however, just uses the `LF` character to denote the end of a line.

Not all of the various tools on these operating systems understand files that contain line endings in a format that differs from the *native line-ending style* of the operating system on which they are running. So, typically, Unix programs treat the `CR` character present in Windows files as a regular character (usually rendered as `^M`), and Windows programs combine all of the lines of a Unix file into one giant line because no carriage return-linefeed (or `CRLF`) character combination was found to denote the ends of the lines.

This sensitivity to foreign EOL markers can be frustrating for folks who share a file across different operating systems. For example, consider a source code file, and developers that edit this file on both Windows and Unix systems. If all the developers always use tools that preserve the line-ending style of the file, no problems occur.

But in practice, many common tools either fail to properly read a file with foreign EOL markers, or convert the file's line endings to the native style when the file is saved. If the former is true for a developer, he has to use an external conversion utility (such as **dos2unix** or its companion, **unix2dos**) to prepare the file for editing. The latter case requires no extra preparation. But both cases result in a file that differs from the original quite literally on every line! Prior to committing his changes, the user has two choices. Either he can use a conversion utility to restore the modified file to the same line-ending style that it was in before his edits were made, or he can simply commit the file—new EOL markers and all.

The result of scenarios like these include wasted time and unnecessary modifications to committed files. Wasted time is painful enough. But when commits change every line in a file, this complicates the job of determining which of those lines were changed in a nontrivial way. Where was that bug really fixed? On what line was a syntax error introduced?

The solution to this problem is the `svn:eol-style` property. When this property is set to a valid value, Subversion uses it to determine what special processing to perform on the file so that the file's line-ending style isn't flip-flopping with every commit that comes from a different operating system. The valid values are:

`native`
> This causes the file to contain the EOL markers that are native to the operating system on which Subversion was run. In other words, if a user on a Windows machine checks out a working copy that contains a file with an `svn:eol-style` property set to `native`, that file

will contain `CRLF` EOL markers. A Unix user checking out a working copy that contains the same file will see `LF` EOL markers in his copy of the file.

Note that Subversion will actually store the file in the repository using normalized `LF` EOL markers regardless of the operating system. This is basically transparent to the user, though.

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
> This causes the file to contain `CR` characters for EOL markers, regardless of the operating system in use. This line-ending style is not very common.

3.4. 忽略未版本控制的条目

In any given working copy, there is a good chance that alongside all those versioned files and directories are other files and directories that are neither versioned nor intended to be. Text editors litter directories with backup files. Software compilers generate intermediate—or even final—files that you typically wouldn't bother to version. And users themselves drop various other files and directories wherever they see fit, often in version control working copies.

It's ludicrous to expect Subversion working copies to be somehow impervious to this kind of clutter and impurity. In fact, Subversion counts it as a *feature* that its working copies are just typical directories, just like unversioned trees. But these not-to-be-versioned files and directories can cause some annoyance for Subversion users. For example, because the **svn add** and **svn import** commands act recursively by default and don't know which files in a given tree you do and don't wish to version, it's easy to accidentally add stuff to version control that you didn't mean to. And because **svn status** reports, by default, every item of interest in a working copy—including unversioned files and directories—its output can get quite noisy where many of these things exist.

So Subversion provides two ways for telling it which files you would prefer that it simply disregard. One of the ways involves the use of Subversion's runtime configuration system (see 第 7.1 节 "运行配置区"), and therefore applies to all the Subversion operations that make use of that runtime configuration—generally those performed on a particular computer or by a particular user of a computer. The other way makes use of Subversion's directory property support and is more tightly bound to the versioned tree itself, and therefore affects everyone who has a working copy of that tree. Both of the mechanisms use *file patterns* (strings of literal and special wildcard characters used to match against filenames) to decide which files to ignore.

The Subversion runtime configuration system provides an option, `global-ignores`, whose value is a whitespace-delimited collection of file patterns. The Subversion client checks these patterns against the names of the files that are candidates for addition to version control, as well as to unversioned files that the **svn status** command notices. If any file's name matches one of the patterns, Subversion will basically act as if the file didn't exist at all. This is really useful

for the kinds of files that you almost never want to version, such as editor backup files such as Emacs' `*~` and `.*~` files.

Subversion 中的文件模式？

File patterns (also called *globs* or *shell wildcard patterns*) are strings of characters that are intended to be matched against filenames, typically for the purpose of quickly selecting some subset of similar files from a larger grouping without having to explicitly name each file. The patterns contain two types of characters: regular characters, which are compared explicitly against potential matches, and special wildcard characters, which are interpreted differently for matching purposes.

There are different types of file pattern syntaxes, but Subversion uses the one most commonly found in Unix systems implemented as the `fnmatch` system function. It supports the following wildcards, described here simply for your convenience:

?

    匹配任意单个字符

*

    匹配任意字符串，包括空字符串

[

    Begins a character class definition terminated by `]`, used for matching a subset of characters

You can see this same pattern matching behavior at a Unix shell prompt. The following are some examples of patterns being used for various things:

```
$ ls   ### the book sources
appa-quickstart.xml            ch06-server-configuration.xml
appb-svn-for-cvs-users.xml     ch07-customizing-svn.xml
appc-webdav.xml                ch08-embedding-svn.xml
book.xml                       ch09-reference.xml
ch00-preface.xml               ch10-world-peace-thru-svn.xml
ch01-fundamental-concepts.xml  copyright.xml
ch02-basic-usage.xml           foreword.xml
ch03-advanced-topics.xml       images/
ch04-branching-and-merging.xml index.xml
ch05-repository-admin.xml       styles.css
$ ls ch*   ### the book chapters
ch00-preface.xml               ch06-server-configuration.xml
ch01-fundamental-concepts.xml  ch07-customizing-svn.xml
ch02-basic-usage.xml           ch08-embedding-svn.xml
ch03-advanced-topics.xml       ch09-reference.xml
ch04-branching-and-merging.xml ch10-world-peace-thru-svn.xml
ch05-repository-admin.xml
$ ls ch?0-*   ### the book chapters whose numbers end in zero
ch00-preface.xml  ch10-world-peace-thru-svn.xml
$ ls ch0[3578]-*   ### the book chapters that Mike is
responsible for
ch03-advanced-topics.xml   ch07-customizing-svn.xml
ch05-repository-admin.xml   ch08-embedding-svn.xml
$
```

文件模式匹配可能比我们这里描述更复杂一点，但是基本的使用水平会适合大多数 Subversion的用户。

When found on a versioned directory, the `svn:ignore` property is expected to contain a list of newline-delimited file patterns that Subversion should use to determine ignorable objects in that same directory. These patterns do not override those found in the `global-ignores` runtime configuration option, but are instead appended to that list. And it's worth noting again that, unlike the `global-ignores` option, the patterns found in the `svn:ignore` property apply only to the directory on which that property is set, and not to any of its subdirectories. The `svn:ignore` property is a good way to tell Subversion to ignore files that are likely to be present in every user's working copy of that directory, such as compiler output or—to use an example more appropriate to this book—the HTML, PDF, or PostScript files generated as the result of a conversion of some source DocBook XML files to a more legible output format.

注意

Subversion's support for ignorable file patterns extends only to the one-time process of adding unversioned files and directories to version control. Once an object is under Subversion's control, the ignore pattern mechanisms no longer apply to it. In other words, don't expect Subversion to avoid committing changes you've made to a versioned file simply because that file's name matches an ignore pattern—Subversion *always* notices all of its versioned objects.

CVS 用户的忽略模式

The Subversion `svn:ignore` property is very similar in syntax and function to the CVS `.cvsignore` file. In fact, if you are migrating a CVS working copy to Subversion, you can directly migrate the ignore patterns by using the `.cvsignore` file as input file to the **svn propset** command:

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

There are, however, some differences in the ways that CVS and Subversion handle ignore patterns. The two systems use the ignore patterns at some different times, and there are slight discrepancies in what the ignore patterns apply to. Also, Subversion does not recognize the use of the `!` pattern as a reset back to having no ignore patterns at all.

The global list of ignore patterns tends to be more a matter of personal taste and ties more closely to a user's particular tool chain than to the details of any particular working copy's needs. So, the rest of this section will focus on the `svn:ignore` property and its uses.

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
```

In this example, you have made some property modifications to button.c, but in your working copy, you also have some unversioned files: the latest calculator program that you've compiled from your source code, a source file named data.c, and a set of debugging output logfiles. Now, you know that your build system always results in the calculator program being generated. [5] And you know that your test suite always leaves those debugging logfiles lying around. These facts are true for all working copies of this project, not just your own. And you know that you aren't interested in seeing those things every time you run **svn status**, and you are pretty sure that nobody else is interested in them either. So you use **svn propedit svn:ignore calc** to add some ignore patterns to the calc directory. For example, you might add this as the new value of the svn:ignore property:

```
calculator
debug_log*
```

After you've added this property, you will now have a local property modification on the calc directory. But notice what else is different about your **svn status** output:

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

Now, all that cruft is missing from the output! Your calculator compiled program and all those logfiles are still in your working copy; Subversion just isn't constantly reminding you that they are present and unversioned. And now with all the uninteresting noise removed from the display, you are left with more intriguing items—such as that source code file data.c that you probably forgot to add to version control.

Of course, this less-verbose report of your working copy status isn't the only one available. If you actually want to see the ignored files as part of the status report, you can pass the --no-ignore option to Subversion:

```
$ 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
```

---

[5]这不是编译系统的基本功能吗?

```
I      calc/debug_log.3.gz
```

As mentioned earlier, the list of file patterns to ignore is also used by **svn add** and **svn import**. Both of these operations involve asking Subversion to begin managing some set of files and directories. Rather than force the user to pick and choose which files in a tree she wishes to start versioning, Subversion uses the ignore patterns—both the global and the per-directory lists—to determine which files should not be swept into the version control system as part of a larger recursive addition or import operation. And here again, you can use the `--no-ignore` option to tell Subversion ignore its ignores list and operate on all the files and directories present.

> **提示**
>
> Even if `svn:ignore` is set, you may run into problems if you use shell wildcards in a command. Shell wildcards are expanded into an explicit list of targets before Subversion operates on them, so running **svn** ***SUBCOMMAND* *** is just like running **svn** ***SUBCOMMAND* file1 file2 file3** .... In the case of the **svn add** command, this has an effect similar to passing the `--no-ignore` option. So instead of using a wildcard, use **svn add --force .** to do a bulk scheduling of unversioned things for addition. The explicit target will ensure that the current directory isn't overlooked because of being already under version control, and the `--force` option will cause Subversion to crawl through that directory, adding unversioned files while still honoring the `svn:ignore` property and `global-ignores` runtime configuration variable. Be sure to also provide the `--depth files` option to the **svn add** command if you don't want a fully recursive crawl for things to add.

3.5. 关键字替换

Subversion has the ability to substitute *keywords*—pieces of useful, dynamic information about a versioned file—into the contents of the file itself. Keywords generally provide information about the last modification made to the file. Because this information changes each time the file changes, and more importantly, just *after* the file changes, it is a hassle for any process except the version control system to keep the data completely up to date. Left to human authors, the information would inevitably grow stale.

For example, say you have a document in which you would like to display the last date on which it was modified. You could burden every author of that document to, just before committing their changes, also tweak the part of the document that describes when it was last changed. But sooner or later, someone would forget to do that. Instead, simply ask Subversion to perform keyword substitution on the `LastChangedDate` keyword. You control where the keyword is inserted into your document by placing a *keyword anchor* at the desired location in the file. This anchor is just a string of text formatted as $KeywordName$.

All keywords are case-sensitive where they appear as anchors in files: you must use the correct capitalization for the keyword to be expanded. You should consider the value of the `svn:keywords` property to be case-sensitive, too—certain keyword names will be recognized regardless of case, but this behavior is deprecated.

Subversion defines the list of keywords available for substitution. That list contains the following five keywords, some of which have aliases that you can also use:

Date

This keyword describes the last time the file was known to have been changed in the repository, and is of the form `$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $`. It may also be specified as `LastChangedDate`. Unlike the `Id` keyword, which uses UTC, the `Date` keyword displays dates using the local time zone.

Revision

This keyword describes the last known revision in which this file changed in the repository, and looks something like `$Revision: 144 $`. It may also be specified as `LastChangedRevision` or `Rev`.

Author

This keyword describes the last known user to change this file in the repository, and looks something like `$Author: harry $`. It may also be specified as `LastChangedBy`.

HeadURL

This keyword describes the full URL to the latest version of the file in the repository, and looks something like `$HeadURL: http://svn.collab.net/repos/trunk/README $`. It may be abbreviated as `URL`.

Id

This keyword is a compressed combination of the other keywords. Its substitution looks something like `$Id: calc.c 148 2006-07-28 21:30:43Z sally $`, and is interpreted to mean that the file `calc.c` was last changed in revision 148 on the evening of July 28, 2006 by the user `sally`. The date displayed by this keyword is in UTC, unlike that of the `Date` keyword (which uses the local time zone).

Several of the preceding descriptions use the phrase "last known" or similar wording. Keep in mind that keyword expansion is a client-side operation, and your client "knows" only about changes that have occurred in the repository when you update your working copy to include those changes. If you never update your working copy, your keywords will never expand to different values even if those versioned files are being changed regularly in the repository.

Simply adding keyword anchor text to your file does nothing special. Subversion will never attempt to perform textual substitutions on your file contents unless explicitly asked to do so. After all, you might be writing a document [6] about how to use keywords, and you don't want Subversion to substitute your beautiful examples of unsubstituted keyword anchors!

To tell Subversion whether to substitute keywords on a particular file, we again turn to the property-related subcommands. The `svn:keywords` property, when set on a versioned file, controls which keywords will be substituted on that file. The value is a space-delimited list of keyword names or aliases.

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.
$LastChangedDate$
$Rev$
```

---

[6] ... 或者可能是本书的一节 ...

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

With no `svn:keywords` property set on that file, Subversion will do nothing special. Now, let's enable substitution of the `LastChangedDate` keyword.

```
$ svn propset svn:keywords -"Date Author" weather.txt
property -'svn:keywords' set on -'weather.txt'
$
```

Now you have made a local property modification on the `weather.txt` file. You will see no changes to the file's contents (unless you made some of your own prior to setting the property). Notice that the file contained a keyword anchor for the `Rev` keyword, yet we did not include that keyword in the property value we set. Subversion will happily ignore requests to substitute keywords that are not present in the file and will not substitute keywords that are not present in the `svn:keywords` property value.

Immediately after you commit this property change, Subversion will update your working file with the new substitute text. Instead of seeing your keyword anchor `$LastChangedDate$`, you'll see its substituted result. That result also contains the name of the keyword and continues to be delimited by the dollar sign (`$`) characters. And as we predicted, the `Rev` keyword was not substituted because we didn't ask for it to be.

Note also that we set the `svn:keywords` property to `Date  Author`, yet the keyword anchor used the alias `$LastChangedDate$` and still expanded correctly:

```
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.
```

If someone else now commits a change to `weather.txt`, your copy of that file will continue to display the same substituted keyword value as before—until you update your working copy. At that time, the keywords in your `weather.txt` file will be resubstituted with information that reflects the most recent known commit to that file.

> **$GlobalRev$ 是什么？**
>
> New users are often confused by how the `$Rev$` keyword works. Since the repository has a single, globally increasing revision number, many people assume that it is this number that is reflected by the `$Rev$` keyword's value. But `$Rev$` expands to show the last revision in which the file *changed*, not the last revision to which it was updated. Understanding this clears the confusion, but frustration often remains—without the support of a Subversion keyword to do so, how can you automatically get the global revision number into your files?
>
> To do this, you need external processing. Subversion ships with a tool called **svnversion**, which was designed for just this purpose. It crawls your working copy and generates as output the revision(s) it finds. You can use this program, plus some additional tooling, to embed that revision information into your files. For more information on **svnversion**, see

Subversion 1.2 introduced a new variant of the keyword syntax, which brought additional, useful—though perhaps atypical—functionality. You can now tell Subversion to maintain a fixed length (in terms of the number of bytes consumed) for the substituted keyword. By using a double colon (::) after the keyword name, followed by a number of space characters, you define that fixed width. When Subversion goes to substitute your keyword for the keyword and its value, it will essentially replace only those space characters, leaving the overall width of the keyword field unchanged. If the substituted value is shorter than the defined field width, there will be extra padding characters (spaces) at the end of the substituted field; if it is too long, it is truncated with a special hash (#) character just before the final dollar sign terminator.

For example, say you have a document in which you have some section of tabular data reflecting the document's Subversion keywords. Using the original Subversion keyword substitution syntax, your file might look something like:

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

Now, that looks nice and tabular at the start of things. But when you then commit that file (with keyword substitution enabled, of course), you see:

```
$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
```

The result is not so beautiful. And you might be tempted to then adjust the file after the substitution so that it again looks tabular. But that holds only as long as the keyword values are the same width. If the last committed revision rolls into a new place value (say, from 99 to 100), or if another person with a longer username commits the file, stuff gets all crooked again. However, if you are using Subversion 1.2 or later, you can use the new fixed-length keyword syntax and define some field widths that seem sane, so your file might look like this:

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

You commit this change to your file. This time, Subversion notices the new fixed-length keyword syntax and maintains the width of the fields as defined by the padding you placed between the double colon and the trailing dollar sign. After substitution, the width of the fields is completely unchanged—the short values for Rev and Author are padded with spaces, and the long Date field is truncated by a hash character:

```
$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。

警告

Be aware that because the width of a keyword field is measured in bytes, the potential for corruption of multibyte values exists. For example, a username that contains some multibyte UTF-8 characters might suffer truncation in the middle of the string of bytes that make up one of those characters. The result will be a mere truncation when viewed at the byte level, but will likely appear as a string with an incorrect or garbled final character when viewed as UTF-8 text. It is conceivable that certain applications, when asked to load the file, would notice the broken UTF-8 text and deem the entire file corrupt, refusing to operate on the file altogether. So, when limiting keywords to a fixed size, choose a size that allows for this type of byte-wise expansion.

3.6. 稀疏目录

By default, most Subversion operations on directories act in a recursive manner. For example, **svn checkout** creates a working copy with every file and directory in the specified area of the repository, descending recursively through the repository tree until the entire structure is copied to your local disk. Subversion 1.5 introduces a feature called *sparse directories* (or *shallow checkouts*) that allows you to easily check out a working copy—or a portion of a working copy—more shallowly than full recursion, with the freedom to bring in previously ignored files and subdirectories at a later time.

For example, say we have a repository with a tree of files and directories with names of the members of a human family with pets. (It's an odd example, to be sure, but bear with us.) A regular **svn checkout** operation will give us a working copy of the whole tree:

```
$ svn checkout file:///var/svn/repos mom
A    mom/son
A    mom/son/grandson
A    mom/daughter
A    mom/daughter/granddaughter1
A    mom/daughter/granddaughter1/bunny1.txt
A    mom/daughter/granddaughter1/bunny2.txt
A    mom/daughter/granddaughter2
A    mom/daughter/fishie.txt
A    mom/kitty1.txt
A    mom/doggie1.txt
Checked out revision 1.
$
```

Now, let's check out the same tree again, but this time we'll ask Subversion to give us only the topmost directory with none of its children at all:

```
$ svn checkout file:///var/svn/repos mom-empty ---depth empty
```

```
Checked out revision 1
$
```

Notice that we added to our original **svn checkout** command line a new `--depth` option. This option is present on many of Subversion's subcommands and is similar to the `--non-recursive` (`-N`) and `--recursive` (`-R`) options. In fact, it combines, improves upon, supercedes, and ultimately obsoletes these two older options. For starters, it expands the supported degrees of depth specification available to users, adding some previously unsupported (or inconsistently supported) depths. Here are the depth values that you can request for a given Subversion operation:

`--depth empty`
Include only the immediate target of the operation, not any of its file or directory children.

`--depth files`
Include the immediate target of the operation and any of its immediate file children.

`--depth immediates`
Include the immediate target of the operation and any of its immediate file or directory children. The directory children will themselves be empty.

`--depth infinity`
Include the immediate target, its file and directory children, its children's children, and so on to full recursion.

Of course, merely combining two existing options into one hardly constitutes a new feature worthy of a whole section in our book. Fortunately, there is more to this story. This idea of depth extends not just to the operations you perform with your Subversion client, but also as a description of a working copy citizen's *ambient depth*, which is the depth persistently recorded by the working copy for that item. Its key strength is this very persistence—the fact that it is *sticky*. The working copy remembers the depth you've selected for each item in it until you later change that depth selection; by default, Subversion commands operate on the working copy citizens present, regardless of their selected depth settings.

> **提示**
>
> You can check the recorded ambient depth of a working copy using the **svn info** command. If the ambient depth is anything other than infinite recursion, **svn info** will display a line describing that depth value:
>
> ```
> $ svn info mom-immediates -| grep -'^Depth:'
> Depth: immediates
> $
> ```

Our previous examples demonstrated checkouts of infinite depth (the default for **svn checkout**) and empty depth. Let's look now at examples of the other depth values:

```
$ svn checkout file:///var/svn/repos mom-files ---depth files
```

```
A    mom-files/kitty1.txt
A    mom-files/doggie1.txt
Checked out revision 1.
$ svn checkout file:///var/svn/repos mom-immediates ---depth
immediates
A    mom-immediates/son
A    mom-immediates/daughter
A    mom-immediates/kitty1.txt
A    mom-immediates/doggie1.txt
Checked out revision 1.
$
```

As described, each of these depths is something more than only the target, but something less than full recursion.

We've used **svn checkout** as an example here, but you'll find the `--depth` option present on many other Subversion commands, too. In those other commands, depth specification is a way to limit the scope of an operation to some depth, much like the way the older `--non-recursive` (`-N`) and `--recursive` (`-R`) options behave. This means that when operating on a working copy of some depth, while requesting an operation of a shallower depth, the operation is limited to that shallower depth. In fact, we can make an even more general statement: given a working copy of any arbitrary—even mixed—ambient depth, and a Subversion command with some requested operational depth, the command will maintain the ambient depth of the working copy members while still limiting the scope of the operation to the requested (or default) operational depth.

In addition to the `--depth` option, the **svn update** and **svn switch** subcommands also accept a second depth-related option: `--set-depth`. It is with this option that you can change the sticky depth of a working copy item. Watch what happens as we take our empty-depth checkout and gradually telescope it deeper using **svn update --set-depth *NEW-DEPTH TARGET***:

```
$ svn update ---set-depth files mom-empty
A    mom-empty/kittie1.txt
A    mom-empty/doggie1.txt
Updated to revision 1.
$ svn update ---set-depth immediates mom-empty
A    mom-empty/son
A    mom-empty/daughter
Updated to revision 1.
$ svn update ---set-depth infinity mom-empty
A    mom-empty/son/grandson
A    mom-empty/daughter/granddaughter1
A    mom-empty/daughter/granddaughter1/bunny1.txt
A    mom-empty/daughter/granddaughter1/bunny2.txt
A    mom-empty/daughter/granddaughter2
A    mom-empty/daughter/fishie1.txt
Updated to revision 1.
```

```
$
```

随着我们逐渐的增加我们的depth选择，版本库给我们目录树的片段。

In our example, we operated only on the root of our working copy, changing its ambient depth value. But we can independently change the ambient depth value of *any* subdirectory inside the working copy, too. Careful use of this ability allows us to flesh out only certain portions of the working copy tree, leaving other portions absent altogether (hence the "sparse" bit of the feature's name). Here's an example of how we might build out a portion of one branch of our family's tree, enable full recursion on another branch, and keep still other pieces pruned (absent from disk).

```
$ rm --rf mom-empty
$ svn checkout file:///var/svn/repos mom-empty ---depth empty
Checked out revision 1.
$ svn update ---set-depth empty mom-empty/son
A    mom-empty/son
Updated to revision 1.
$ svn update ---set-depth empty mom-empty/daughter
A    mom-empty/daughter
Updated to revision 1.
$ svn update ---set-depth infinity mom-empty/daughter/
granddaughter1
A    mom-empty/daughter/granddaughter1
A    mom-empty/daughter/granddaughter1/bunny1.txt
A    mom-empty/daughter/granddaughter1/bunny2.txt
Updated to revision 1.
$
```

Fortunately, having a complex collection of ambient depths in a single working copy doesn't complicate the way you interact with that working copy. You can still make, revert, display, and commit local modifications in your working copy without providing any new options (including --depth and --set-depth) to the relevant subcommands. Even **svn update** works as it does elsewhere when no specific depth is provided—it updates the working copy targets that are present while honoring their sticky depths.

You might at this point be wondering, "So what? When would I use this?" One scenario where this feature finds utility is tied to a particular repository layout, specifically where you have many related or codependent projects or software modules living as siblings in a single repository location (trunk/project1, trunk/project2, trunk/project3, etc.). In such scenarios, it might be the case that you personally care about only a handful of those projects—maybe some primary project and a few other modules on which it depends. You can check out individual working copies of all of these things, but those working copies are disjoint and, as a result, it can be cumbersome to perform operations across several or all of them at the same time. The alternative is to use the sparse directories feature, building out a single working copy that contains only the modules you care about. You'd start with an empty-depth checkout of the common parent directory of the projects, and then update with infinite depth only the items you wish to have, like we demonstrated in the previous example. Think of it like an opt-in system for working copy citizens.

Subversion 1.5's implementation of shallow checkouts is good but does not support a couple of interesting behaviors. First, you cannot de-telescope a working copy item. Running **svn update --set-depth empty** in an infinite-depth working copy will not have the effect of discarding everything but the topmost directory—it will simply error out. Second, there is no depth value to indicate that you wish an item to be explicitly excluded. You have to do implicit exclusion of an item by including everything else.

3.7. 锁定

Subversion's copy-modify-merge version control model lives and dies on its data merging algorithms—specifically on how well those algorithms perform when trying to resolve conflicts caused by multiple users modifying the same file concurrently. Subversion itself provides only one such algorithm: a three-way differencing algorithm that is smart enough to handle data at a granularity of a single line of text. Subversion also allows you to supplement its content merge processing with external differencing utilities (as described in 第 7.4.2 节 "外置 diff3"), some of which may do an even better job, perhaps providing granularity of a word or a single character of text. But common among those algorithms is that they generally work only on text files. The landscape starts to look pretty grim when you start talking about content merges of nontextual file formats. And when you can't find a tool that can handle that type of merging, you begin to run into problems with the copy-modify-merge model.

Let's look at a real-life example of where this model runs aground. Harry and Sally are both graphic designers working on the same project, a bit of marketing collateral for an automobile mechanic. Central to the design of a particular poster is an image of a car in need of some bodywork, stored in a file using the PNG image format. The poster's layout is almost finished, and both Harry and Sally are pleased with the particular photo they chose for their damaged car—a baby blue 1967 Ford Mustang with an unfortunate bit of crumpling on the left front fender.

Now, as is common in graphic design work, there's a change in plans, which causes the car's color to be a concern. So Sally updates her working copy to HEAD, fires up her photo-editing software, and sets about tweaking the image so that the car is now cherry red. Meanwhile, Harry, feeling particularly inspired that day, decides that the image would have greater impact if the car also appears to have suffered greater impact. He, too, updates to HEAD, and then draws some cracks on the vehicle's windshield. He manages to finish his work before Sally finishes hers, and after admiring the fruits of his undeniable talent, he commits the modified image. Shortly thereafter, Sally is finished with the car's new finish and tries to commit her changes. But, as expected, Subversion fails the commit, informing Sally that her version of the image is now out of date.

Here's where the difficulty sets in. If Harry and Sally were making changes to a text file, Sally would simply update her working copy, receiving Harry's changes in the process. In the worst possible case, they would have modified the same region of the file, and Sally would have to work out by hand the proper resolution to the conflict. But these aren't text files—they are binary images. And while it's a simple matter to describe what one would expect the results of this content merge to be, there is precious little chance that any software exists that is smart enough to examine the common baseline image that each of these graphic artists worked against, the changes that Harry made, and the changes that Sally made, and then spit out an image of a busted-up red Mustang with a cracked windshield!

Of course, things would have gone more smoothly if Harry and Sally had serialized their modifications to the image—if, say, Harry had waited to draw his windshield cracks on Sally's now-red car, or if Sally had tweaked the color of a car whose windshield was already cracked. As is discussed in 第 1.2.3 节 ""拷贝–修改–合并"方案", most of these types of problems go away entirely where perfect communication between Harry and Sally exists. [7] But as one's version control system is, in fact, one form of communication, it follows that having that software facilitate the serialization of nonparallelizable editing efforts is no bad thing. This is where Subversion's implementation of the lock-modify-unlock model steps into the spotlight. This is where we talk about Subversion's *locking* feature, which is similar to the "reserved checkouts" mechanisms of other version control systems.

Subversion's locking feature exists ultimately to minimize wasted time and effort. By allowing a user to programmatically claim the exclusive right to change a file in the repository, that user can be reasonably confident that any energy he invests on unmergeable changes won't be wasted—his commit of those changes will succeed. Also, because Subversion communicates to other users that serialization is in effect for a particular versioned object, those users can reasonably expect that the object is about to be changed by someone else. They, too, can then avoid wasting their time and energy on unmergeable changes that won't be committable due to eventual out-of-dateness.

When referring to Subversion's locking feature, one is actually talking about a fairly diverse collection of behaviors, which include the ability to lock a versioned file [8] (claiming the exclusive right to modify the file), to unlock that file (yielding that exclusive right to modify), to see reports about which files are locked and by whom, to annotate files for which locking before editing is strongly advised, and so on. In this section, we'll cover all of these facets of the larger locking feature.

"锁定"的三种含义

In this section, and almost everywhere in this book, the words "lock" and "locking" describe a mechanism for mutual exclusion between users to avoid clashing commits. Unfortunately, there are two other sorts of "lock" with which Subversion, and therefore this book, sometimes needs to be concerned.

The second is *working copy locks*, used internally by Subversion to prevent clashes between multiple Subversion clients operating on the same working copy. This is the sort of lock indicated by an L in the third column of **svn status** output, and removed by the **svn cleanup** command, as described in 第 2.6 节 "有时你只需要清理".

Third, there are *database locks*, used internally by the Berkeley DB backend to prevent clashes between multiple programs trying to access the database. This is the sort of lock whose unwanted persistence after an error can cause a repository to be "wedged," as described in 第 5.4.4 节 "Berkeley DB 恢复".

You can generally forget about these other kinds of locks until something goes wrong that requires you to care about them. In this book, "lock" means the first sort unless the contrary is either clear from context or explicitly stated.

---

[7] 对于Harry和Sally的好莱坞同名人来说交流也不是那么差的药，也是关于那一点。
[8] Subversion目前不允许锁定目录。

3.7.1. 创建锁定

In the Subversion repository, a *lock* is a piece of metadata that grants exclusive access to one user to change a file. This user is said to be the *lock owner*. Each lock also has a unique identifier, typically a long string of characters, known as the *lock token*. The repository manages locks, ultimately handling their creation, enforcement, and removal. If any commit transaction attempts to modify or delete a locked file (or delete one of the parent directories of the file), the repository will demand two pieces of information—that the client performing the commit be authenticated as the lock owner, and that the lock token has been provided as part of the commit process as a form of proof that the client knows which lock it is using.

To demonstrate lock creation, let's refer back to our example of multiple graphic designers working on the same binary image files. Harry has decided to change a JPEG image. To prevent other people from committing changes to the file while he is modifying it (as well as alerting them that he is about to change it), he locks the file in the repository using the **svn lock** command.

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

The preceding example demonstrates a number of new things. First, notice that Harry passed the `--message` (`-m`) option to **svn lock**. Similar to **svn commit**, the **svn lock** command can take comments—via either `--message` (`-m`) or `--file` (`-F`)—to describe the reason for locking the file. Unlike **svn commit**, however, **svn lock** will not demand a message by launching your preferred text editor. Lock comments are optional, but still recommended to aid communication.

Second, the lock attempt succeeded. This means that the file wasn't already locked, and that Harry had the latest version of the file. If Harry's working copy of the file had been out of date, the repository would have rejected the request, forcing Harry to **svn update** and reattempt the locking command. The locking command would also have failed if the file had already been locked by someone else.

As you can see, the **svn lock** command prints confirmation of the successful lock. At this point, the fact that the file is locked becomes apparent in the output of the **svn status** and **svn info** reporting subcommands.

```
$ 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.

$
```

The fact that the **svn info** command, which does not contact the repository when run against working copy paths, can display the lock token reveals an important piece of information about those tokens: they are cached in the working copy. The presence of the lock token is critical. It gives the working copy authorization to make use of the lock later on. Also, the **svn status** command shows a K next to the file (short for locKed), indicating that the lock token is present.

> ### 关于锁定令牌
>
> A lock token isn't an authentication token, so much as an *authorization* token. The token isn't a protected secret. In fact, a lock's unique token is discoverable by anyone who runs **svn info URL**. A lock token is special only when it lives inside a working copy. It's proof that the lock was created in that particular working copy, and not somewhere else by some other client. Merely authenticating as the lock owner isn't enough to prevent accidents.
>
> For example, suppose you lock a file using a computer at your office, but leave work for the day before you finish your changes to that file. It should not be possible to accidentally commit changes to that same file from your home computer later that evening simply because you've authenticated as the lock's owner. In other words, the lock token prevents one piece of Subversion-related software from undermining the work of another. (In our example, if you really need to change the file from an alternative working copy, you would need to *break* the lock and relock the file.)

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: Server sent unexpected return value (423 Locked) in response
to DELETE\
```

```
 request for -'/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-
df4224ddc35d/\
banana.jpg'
$
```

But Harry, after touching up the banana's shade of yellow, is able to commit his changes to the file. That's because he authenticates as the lock owner and also because his working copy holds the correct lock token:

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

Notice that after the commit is finished, **svn status** shows that the lock token is no longer present in the working copy. This is the standard behavior of **svn commit**—it searches the working copy (or list of targets, if you provide such a list) for local modifications and sends all the lock tokens it encounters during this walk to the server as part of the commit transaction. After the commit completes successfully, all of the repository locks that were mentioned are released —*even on files that weren't committed*. This is meant to discourage users from being sloppy about locking or from holding locks for too long. If Harry haphazardly locks 30 files in a directory named `images` because he's unsure of which files he needs to change, yet changes only four of those files, when he runs **`svn commit images`**, the process will still release all 30 locks.

This behavior of automatically releasing locks can be overridden with the `--no-unlock` option to **svn commit**. This is best used for those times when you want to commit changes, but still plan to make more changes and thus need to retain existing locks. You can also make this your default behavior by setting the `no-unlock` runtime configuration option (see 第 7.1 节 "运行配置区").

Of course, locking a file doesn't oblige one to commit a change to it. The lock can be released at any time with a simple **svn unlock** command:

```
$ svn unlock banana.c
'banana.c' unlocked.
```

3.7.2. 发现锁定

When a commit fails due to someone else's locks, it's fairly easy to learn about them. The easiest way is to run **`svn status --show-updates`**:

```
$ svn status --u
```

```
M               23   bar.c
M    O          32   raisin.jpg
       *        72   foo.h
Status against revision:     105
$
```

In this example, Sally can see not only that her copy of `foo.h` is out of date, but also that one of the two modified files she plans to commit is locked in the repository. The `O` symbol stands for "Other," meaning that a lock exists on the file and was created by somebody else. If she were to attempt a commit, the lock on `raisin.jpg` would prevent it. Sally is left wondering who made the lock, when, and why. Once again, **svn info** has the answers:

```
$ 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.
$
```

Just as you can use **svn info** to examine objects in the working copy, you can also use it to examine objects in the repository. If the main argument to **svn info** is a working copy path, then all of the working copy's cached information is displayed; any mention of a lock means that the working copy is holding a lock token (if a file is locked by another user or in another working copy, **svn info** on a working copy path will show no lock information at all). If the main argument to **svn info** is a URL, the information reflects the latest version of an object in the repository, and any mention of a lock describes the current lock on the object.

So in this particular example, Sally can see that Harry locked the file on February 16 to "make a quick tweak." It being June, she suspects that he probably forgot all about the lock. She might phone Harry to complain and ask him to release the lock. If he's unavailable, she might try to forcibly break the lock herself or ask an administrator to do so.

3.7.3. 解除和偷窃锁定

A repository lock isn't sacred—in Subversion's default configuration state, locks can be released not only by the person who created them, but by anyone. When somebody other than the original lock creator destroys a lock, we refer to this as *breaking the lock*.

From the administrator's chair, it's simple to break locks. The **svnlook** and **svnadmin** programs have the ability to display and remove locks directly from the repository. (For more information about these tools, see 第 5.4.1 节 "管理员的工具箱".)

```
$ svnadmin lslocks -/var/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 -/var/svn/repos -/project/raisin.jpg
Removed lock on -'/project/raisin.jpg'.
$
```

The more interesting option is to allow users to break each other's locks over the network. To do this, Sally simply needs to pass the `--force` to the **svn unlock** command:

```
$ 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.
$
```

Now, Sally's initial attempt to unlock failed because she ran **svn unlock** directly on her working copy of the file, and no lock token was present. To remove the lock directly from the repository, she needs to pass a URL to **svn unlock**. Her first attempt to unlock the URL fails,

82

because she can't authenticate as the lock owner (nor does she have the lock token). But when she passes `--force`, the authentication and authorization requirements are ignored, and the remote lock is broken.

Simply breaking a lock may not be enough. In the running example, Sally may not only want to break Harry's long-forgotten lock, but relock the file for her own use. She can accomplish this by using **svn unlock** with `--force` and then **svn lock** back-to-back, but there's a small chance that somebody else might lock the file between the two commands. The simpler thing to do is to *steal* the lock, which involves breaking and relocking the file all in one atomic step. To do this, Sally passes the `--force` option to **svn lock**:

```
$ 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'.
$
```

In any case, whether the lock is broken or stolen, Harry may be in for a surprise. Harry's working copy still contains the original lock token, but that lock no longer exists. The lock token is said to be *defunct*. The lock represented by the lock token has either been broken (no longer in the repository) or stolen (replaced with a different lock). Either way, Harry can see this by asking **svn status** to contact the repository:

```
$ svn status
    K raisin.jpg
$ svn status --u
    B        32   raisin.jpg
$ svn update
  B  raisin.jpg
$ svn status
$
```

If the repository lock was broken, then **svn status --show-updates** displays a B (Broken) symbol next to the file. If a new lock exists in place of the old one, then a T (sTolen) symbol is shown. Finally, **svn update** notices any defunct lock tokens and removes them from the working copy.

> **锁定策略**
>
> Different systems have different notions of how strict a lock should be. Some folks argue that locks must be strictly enforced at all costs, releasable only by the original creator or administrator. They argue that if anyone can break a lock, chaos runs rampant and the whole point of locking is defeated. The other side argues that locks are first and foremost a communication tool. If users are constantly breaking each other's locks, it represents a cultural failure within the team and the problem falls outside the scope of software enforcement.
>
> Subversion defaults to the "softer" approach, but still allows administrators to create stricter enforcement policies through the use of hook scripts. In particular, the `pre-lock` and `pre-unlock` hooks allow administrators to decide when lock creation and lock releases are allowed to happen. Depending on whether a lock already exists, these two hooks can decide whether to allow a certain user to break or steal a lock. The `post-lock` and `post-unlock` hooks are also available, and can be used to send email after locking actions. To learn more about repository hooks, see 第 5.3.2 节 "实现版本库钩子".

3.7.4. 锁定交流

We've seen how **svn lock** and **svn unlock** can be used to create, release, break, and steal locks. This satisfies the goal of serializing commit access to a file. But what about the larger problem of preventing wasted time?

For example, suppose Harry locks an image file and then begins editing it. Meanwhile, miles away, Sally wants to do the same thing. She doesn't think to run **svn status --show-updates**, so she has no idea that Harry has already locked the file. She spends hours editing the file, and when she tries to commit her change, she discovers that either the file is locked or that she's out of date. Regardless, her changes aren't mergeable with Harry's. One of these two people has to throw away his or her work, and a lot of time has been wasted.

Subversion's solution to this problem is to provide a mechanism to remind users that a file ought to be locked *before* the editing begins. The mechanism is a special property: `svn:needs-lock`. If that property is attached to a file (regardless of its value, which is irrelevant), Subversion will try to use filesystem-level permissions to make the file read-only—unless, of course, the user has explicitly locked the file. When a lock token is present (as a result of using **svn lock**), the file becomes read/write. When the lock is released, the file becomes read-only again.

The theory, then, is that if the image file has this property attached, Sally would immediately notice something is strange when she opens the file for editing: many applications alert users immediately when a read-only file is opened for editing, and nearly all would prevent her from saving changes to the file. This reminds her to lock the file before editing, whereby she discovers the preexisting lock:

```
$ -/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.
$
```

提示

Users and administrators alike are encouraged to attach the `svn:needs-lock` property to any file that cannot be contextually merged. This is the primary technique for encouraging good locking habits and preventing wasted effort.

Note that this property is a communication tool that works independently from the locking system. In other words, any file can be locked, whether or not this property is present. And conversely, the presence of this property doesn't make the repository require a lock when committing.

Unfortunately, the system isn't flawless. It's possible that even when a file has the property, the read-only reminder won't always work. Sometimes applications misbehave and "hijack" the read-only file, silently allowing users to edit and save the file anyway. There's not much that Subversion can do in this situation—at the end of the day, there's simply no substitution for good interpersonal communication. [9]

3.8. 外部定义

Sometimes it is useful to construct a working copy that is made out of a number of different checkouts. For example, you may want different subdirectories to come from different locations in a repository or perhaps from different repositories altogether. You could certainly set up such a scenario by hand—using **svn checkout** to create the sort of nested working copy structure you are trying to achieve. But if this layout is important for everyone who uses your repository, every other user will need to perform the same checkout operations that you did.

Fortunately, Subversion provides support for *externals definitions*. An externals definition is a mapping of a local directory to the URL—and ideally a particular revision—of a versioned directory. In Subversion, you declare externals definitions in groups using the `svn:externals` property. You can create or modify this property using **svn propset** or **svn propedit** (see 第 3.2.2 节 "操作属性"). It can be set on any versioned directory, and its value describes both the external repository location and the client-side directory to which that location should be checked out.

---

[9] 除非是，或许一个经典的火神精神融合。

The convenience of the `svn:externals` property is that once it is set on a versioned directory, everyone who checks out a working copy with that directory also gets the benefit of the externals definition. In other words, once one person has made the effort to define the nested working copy structure, no one else has to bother—Subversion will, after checking out the original working copy, automatically also check out the external working copies.

警告

The relative target subdirectories of externals definitions *must not* already exist on your or other users' systems—Subversion will create them when it checks out the external working copy.

You also get in the externals definition design all the regular benefits of Subversion properties. The definitions are versioned. If you need to change an externals definition, you can do so using the regular property modification subcommands. When you commit a change to the `svn:externals` property, Subversion will synchronize the checked-out items against the changed externals definition when you next run **svn update**. The same thing will happen when others update their working copies and receive your changes to the externals definition.

提示

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**。

Subversion releases prior to 1.5 honor an externals definition format that is a multiline table of subdirectories (relative to the versioned directory on which the property is set), optional revision flags, and fully qualified, absolute Subversion repository URLs. An example of this might look as follows:

```
$ svn propget svn:externals calc
third-party/sounds                 http://svn.example.com/repos/sounds
third-party/skins --r148      http://svn.example.com/skinproj
third-party/skins/toolkit --r21 http://svn.example.com/skin-maker
```

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
…
```

As of Subversion 1.5, though, a new format of the `svn:externals` property is supported. Externals definitions are still multiline, but the order and format of the various pieces of information have changed. The new syntax more closely mimics the order of arguments you might pass to **svn checkout**: the optional revision flags come first, then the external Subversion repository URL, and finally the relative local subdirectory. Notice, though, that this time we didn't say "fully qualified, absolute Subversion repository URLs." That's because the new format supports relative URLs and URLs that carry peg revisions. The previous example of an externals definition might, in Subversion 1.5, look like the following:

```
$ svn propget svn:externals calc
        http://svn.example.com/repos/sounds third-party/sounds
-r148 http://svn.example.com/skinproj third-party/skins
-r21  http://svn.example.com/skin-maker third-party/skins/toolkit
```

Or, making use of the peg revision syntax (which we describe in detail in), it might appear as:

```
$ svn propget svn:externals calc
http://svn.example.com/repos/sounds third-party/sounds
http://svn.example.com/skinproj@148 third-party/skins
http://svn.example.com/skin-maker@21 third-party/skins/toolkit
```

提示

You should seriously consider using explicit revision numbers in all of your externals definitions. Doing so means that you get to decide when to pull down a different snapshot of external information, and exactly which snapshot to pull. Besides avoiding the surprise of getting changes to third-party repositories that you might not have any control over, using explicit revision numbers also means that as you backdate your working copy to a previous revision, your externals definitions will also revert to the way they looked in that previous revision, which in turn means that the external working copies will be updated to match the way *they* looked back when your repository was at that previous revision. For software projects, this could be the difference between a successful and a failed build of an older snapshot of your complex codebase.

For most repositories, these three ways of formatting the externals definitions have the same ultimate effect. They all bring the same benefits. Unfortunately, they all bring the same

annoyances, too. Since the definitions shown use absolute URLs, moving or copying a directory to which they are attached will not affect what gets checked out as an external (though the relative local target subdirectory will, of course, move with the renamed directory). This can be confusing —even frustrating—in certain situations. For example, say you have a top-level directory named `my-project`, and you've created an externals definition on one of its subdirectories (`my-project/some-dir`) that tracks the latest revision of another of its subdirectories (`my-project/external-dir`).

```
$ 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

$
```

Now you use **svn move** to rename the `my-project` directory. At this point, your externals definition will still refer to a path under the `my-project` directory, even though that directory no longer exists.

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

Committed revision 12.
$ svn update

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

Also, absolute URLs can cause problems with repositories that are available via multiple URL schemes. For example, if your Subversion server is configured to allow everyone to check out the repository over `http://` or `https://`, but only allow commits to come in via `https://`, you have an interesting problem on your hands. If your externals definitions use the `http://` form of the repository URLs, you won't be able to commit anything from the working copies created by those externals. On the other hand, if they use the `https://` form of the URLs, anyone who might be checking out via `http://` because his client doesn't support `https://` will be unable to fetch the external items. Be aware, too, that if you need to reparent your working copy (using **svn switch** with the `--relocate` option), externals definitions will *not* also be reparented.

Subversion 1.5 takes a huge step in relieving these frustrations. As mentioned earlier, the URLs used in the new externals definition format can be relative, and Subversion provides syntax magic for specifying multiple flavors of URL relativity.

`../`
Relative to the URL of the directory on which the `svn:externals` property is set

`^/`
Relative to the root of the repository in which the `svn:externals` property is versioned

`//`
Relative to the scheme of the URL of the directory on which the `svn:externals` property is set

`/`
Relative to the root URL of the server on which the `svn:externals` property is versioned

So, looking a fourth time at our previous externals definition example, and making use of the new absolute URL syntax in various ways, we might now see:

```
$ svn propget svn:externals calc
^/sounds third-party/sounds
/skinproj@148 third-party/skins
//svn.example.com/skin-maker@21 third-party/skins/toolkit
```

The support that exists for externals definitions in Subversion remains less than ideal, though. An externals definition can point only to directories, not to files. Also, the local subdirectory part of the definition cannot contain `..` parent directory indicators (such as `../../skins/myskin`). Perhaps most disappointingly, the working copies created via the externals definition support are still disconnected from the primary working copy (on whose versioned directories the `svn:externals` property was actually set). And Subversion still truly operates only on nondisjoint working copies. So, for example, if you want to commit changes that you've made in one or more of those external working copies, you must run **svn commit** explicitly on those working copies—committing on the primary working copy will not recurse into any external ones.

We've already mentioned some of the additional shortcomings of the old `svn:externals` format and how the new Subversion 1.5 format improves upon it. But be careful when making use of the new format that you don't inadvertently cause problems for other folks accessing your repository who are using older Subversion clients. While Subversion 1.5 clients will continue to recognize and support the original externals definition format, older clients will *not* be able to correctly parse the new format.

Besides the **svn checkout**, **svn update**, **svn switch**, and **svn export** commands which actually manage the *disjoint* (or disconnected) subdirectories into which externals are checked out, the **svn status** command also recognizes externals definitions. It displays a status code of `X` for the disjoint external subdirectories, and then recurses into those subdirectories to display the status of the external items themselves. You can pass the `--ignore-externals` option to any of these subcommands to disable externals definition processing.

3.9. Peg 和实施修订版本

We copy, move, rename, and completely replace files and directories on our computers all the time. And your version control system shouldn't get in the way of your doing these things with your version-controlled files and directories, either. Subversion's file management support is quite liberating, affording almost as much flexibility for versioned files as you'd expect when manipulating your unversioned ones. But that flexibility means that across the lifetime of your repository, a given versioned object might have many paths, and a given path might represent several entirely different versioned objects. This introduces a certain level of complexity to your interactions with those paths and objects.

Subversion is pretty smart about noticing when an object's version history includes such "changes of address." For example, if you ask for the revision history log of a particular file that was renamed last week, Subversion happily provides all those logs—the revision in which the rename itself happened, plus the logs of relevant revisions both before and after that rename. So, most of the time, you don't even have to think about such things. But occasionally, Subversion needs your help to clear up ambiguities.

The simplest example of this occurs when a directory or file is deleted from version control, and then a new directory or file is created with the same name and added to version control. The thing you deleted and the thing you later added aren't the same thing. They merely happen to have had the same path—`/trunk/object`, for example. What, then, does it mean to ask Subversion about the history of `/trunk/object`? Are you asking about the thing currently at that location, or the old thing you deleted from that location? Are you asking about the operations that have happened to *all* the objects that have ever lived at that path? Subversion needs a hint about what you really want.

And thanks to moves, versioned object history can get far more twisted than even that. For example, you might have a directory named `concept`, containing some nascent software project you've been toying with. Eventually, though, that project matures to the point that the idea seems to actually have some wings, so you do the unthinkable and decide to give the project a name. [10] Let's say you called your software Frabnaggilywort. At this point, it makes sense to rename the directory to reflect the project's new name, so `concept` is renamed to `frabnaggilywort`. Life goes on, Frabnaggilywort releases a 1.0 version and is downloaded and used daily by hordes of people aiming to improve their lives.

It's a nice story, really, but it doesn't end there. Entrepreneur that you are, you've already got another think in the tank. So you make a new directory, `concept`, and the cycle begins again. In fact, the cycle begins again many times over the years, each time starting with that old `concept` directory, then sometimes seeing that directory renamed as the idea cures, sometimes seeing it deleted when you scrap the idea. Or, to get really sick, maybe you rename `concept` to something else for a while, but later rename the thing back to `concept` for some reason.

In scenarios like these, attempting to instruct Subversion to work with these reused paths can be a little like instructing a motorist in Chicago's West Suburbs to drive east down Roosevelt Road and turn left onto Main Street. In a mere 20 minutes, you can cross "Main Street" in Wheaton,

---

[10]"You're not supposed to name it. Once you name it, you start getting attached to it."—Mike Wazowski

Glen Ellyn, and Lombard. And no, they aren't the same street. Our motorist—and our Subversion —need a little more detail to do the right thing.

In version 1.1, Subversion introduced a way for you to tell it exactly which Main Street you meant. It's called the *peg revision*, and it is provided to Subversion for the sole purpose of identifying a unique line of history. Because at most, one versioned object may occupy a path at any given time—or, more precisely, in any one revision—the combination of a path and a peg revision is all that is needed to refer to a specific line of history. Peg revisions are specified to the Subversion command-line client using *at syntax*, so called because the syntax involves appending an "at sign" (`@`) and the peg revision to the end of the path with which the revision is associated.

But what of the `--revision` (`-r`) of which we've spoken so much in this book? That revision (or set of revisions) is called the *operative revision* (or *operative revision range*). Once a particular line of history has been identified using a path and peg revision, Subversion performs the requested operation using the operative revision(s). To map this to our Chicagoland streets analogy, if we are told to go to 606 N. Main Street in Wheaton, [11] we can think of "Main Street" as our path and "Wheaton" as our peg revision. These two pieces of information identify a unique path that can be traveled (north or south on Main Street), and they keep us from traveling up and down the wrong Main Street in search of our destination. Now we throw in "606 N." as our operative revision of sorts, and we know *exactly* where to go.

---

[11]606 N. Main Street, Wheaton, Illinois, is the home of the Wheaton *History* Center. It seemed appropriate....

Peg 修订版本算法

The Subversion command-line client performs the peg revision algorithm any time it needs to resolve possible ambiguities in the paths and revisions provided to it. Here's an example of such an invocation:

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

If *OPERATIVE-REV* is older than *PEG-REV*, the algorithm is as follows:

1. Locate `item` in the revision identified by *PEG-REV*. There can be only one such object.

2. 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。

3. Perform the requested action on that ancestor, wherever it is located, or whatever its name might be or might have been at that time.

But what if *OPERATIVE-REV* is *younger* than *PEG-REV*? Well, that adds some complexity to the theoretical problem of locating the path in *OPERATIVE-REV*, because the path's history could have forked multiple times (thanks to copy operations) between *PEG-REV* and *OPERATIVE-REV*. And that's not all—Subversion doesn't store enough information to performantly trace an object's history forward, anyway. So the algorithm is a little different:

1. Locate `item` in the revision identified by *OPERATIVE-REV*. There can be only one such object.

2. Trace the object's history backward (through any possible renames) to its ancestor in the revision *PEG-REV*.

3. Verify that the object's location (path-wise) in *PEG-REV* is the same as it is in *OPERATIVE-REV*. If that's the case, at least the two locations are known to be directly related, so perform the requested action on the location in *OPERATIVE-REV*. Otherwise, relatedness was not established, so error out with a loud complaint that no viable location was found. (Someday, we expect that Subversion will be able to handle this usage scenario with more flexibility and grace.)

Note that even when you don't explicitly supply a peg revision or operative revision, they are still present. For your convenience, the default peg revision is `BASE` for working copy items and `HEAD` for repository URLs. And when no operative revision is provided, it defaults to being the same revision as the peg revision.

Say that long ago we created our repository, and in revision 1 we added our first `concept` directory, plus an `IDEA` file in that directory talking about the concept. After several revisions in which real code was added and tweaked, we, in revision 20, renamed this directory to `frabnaggilywort`. By revision 27, we had a new concept, a new `concept` directory to hold it, and a new `IDEA` file to describe it. And then five years and thousands of revisions flew by, just like they would in any good romance story.

Now, years later, we wonder what the `IDEA` file looked like back in revision 1. But Subversion needs to know whether we are asking about how the *current* file looked back in revision 1, or

whether we are asking for the contents of whatever file lived at `concepts/IDEA` in revision 1. Certainly those questions have different answers, and because of peg revisions, you can ask those questions. To find out how the current `IDEA` file looked in that old revision, you run:

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

Of course, in this example, the current `IDEA` file didn't exist yet in revision 1, so Subversion gives an error. The previous command is shorthand for a longer notation which explicitly lists a peg revision. The expanded notation is:

```
$ 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。

The perceptive reader is probably wondering at this point whether the peg revision syntax causes problems for working copy paths or URLs that actually have at signs in them. After all, how does **svn** know whether `news@11` is the name of a directory in my tree or just a syntax for "revision 11 of `news`"? Thankfully, while **svn** will always assume the latter, there is a trivial workaround. You need only append an at sign to the end of the path, such as `news@11@`. **svn** cares only about the last at sign in the argument, and it is not considered illegal to omit a literal peg revision specifier after that at sign. This workaround even applies to paths that end in an at sign—you would use `filename@@` to talk about a file named `filename@`.

Let's ask the other question, then—in revision 1, what were the contents of whatever file occupied the address `concepts/IDEA` at the time? We'll use an explicit peg revision to help us out.

```
$ 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.
```

Notice that we didn't provide an operative revision this time. That's because when no operative revision is specified, Subversion assumes a default operative revision that's the same as the peg revision.

As you can see, the output from our operation appears to be correct. The text even mentions frabbing naggily worts, so this is almost certainly the file that describes the software now called Frabnaggilywort. In fact, we can verify this using the combination of an explicit peg revision and explicit operative revision. We know that in `HEAD`, the Frabnaggilywort project is located

in the `frabnaggilywort` directory. So we specify that we want to see how the line of history identified in `HEAD` as the path `frabnaggilywort/IDEA` looked in revision 1.

```
$ 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.
```

And the peg and operative revisions need not be so trivial, either. For example, say `frabnaggilywort` had been deleted from `HEAD`, but we know it existed in revision 20, and we want to see the diffs for its `IDEA` file between revisions 4 and 10. We can use the peg revision 20 in conjunction with the URL that would have held Frabnaggilywort's `IDEA` file in revision 20, and then use 4 and 10 as our operative revision range.

```
$ 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.
```

Fortunately, most folks aren't faced with such complex situations. But when you are, remember that peg revisions are that extra hint Subversion needs to clear up ambiguity.

3.10. 修改列表

It is commonplace for a developer to find himself working at any given time on multiple different, distinct changes to a particular bit of source code. This isn't necessarily due to poor

planning or some form of digital masochism. A software engineer often spots bugs in his peripheral vision while working on some nearby chunk of source code. Or perhaps he's halfway through some large change when he realizes the solution he's working on is best committed as several smaller logical units. Often, these logical units aren't nicely contained in some module, safely separated from other changes. The units might overlap, modifying different files in the same module, or even modifying different lines in the same file.

Developers can employ various work methodologies to keep these logical changes organized. Some use separate working copies of the same repository to hold each individual change in progress. Others might choose to create short-lived feature branches in the repository and use a single working copy that is constantly switched to point to one such branch or another. Still others use **diff** and **patch** tools to back up and restore uncommitted changes to and from patch files associated with each change. Each of these methods has its pros and cons, and to a large degree, the details of the changes being made heavily influence the methodology used to distinguish them.

Subversion 1.5 brings a new *changelists* feature that adds yet another method to the mix. Changelists are basically arbitrary labels (currently at most one per file) applied to working copy files for the express purpose of associating multiple files together. Users of many of Google's software offerings are familiar with this concept already. For example, Gmail [http://mail.google.com/] doesn't provide the traditional folders-based email organization mechanism. In Gmail, you apply arbitrary labels to emails, and multiple emails can be said to be part of the same group if they happen to share a particular label. Viewing only a group of similarly labeled emails then becomes a simple user interface trick. Many other Web 2.0 sites have similar mechanisms—consider the "tags" used by sites such as YouTube [http://www.youtube.com/] and Flickr [http://www.flickr.com/], "categories" applied to blog posts, and so on. Folks understand today that organization of data is critical, but that how that data is organized needs to be a flexible concept. The old files-and-folders paradigm is too rigid for some applications.

Subversion's changelist support allows you to create changelists by applying labels to files you want to be associated with that changelist, remove those labels, and limit the scope of the files on which its subcommands operate to only those bearing a particular label. In this section, we'll look in detail at how to do these things.

3.10.1. 创建和更新修改列表

You can create, modify, and delete changelists using the **svn changelist** command. More accurately, you use this command to set or unset the changelist association of a particular working copy file. A changelist is effectively created the first time you label a file with that changelist; it is deleted when you remove that label from the last file that had it. Let's examine a usage scenario that demonstrates these concepts.

Harry is fixing some bugs in the calculator application's mathematics logic. His work leads him to change a couple of files:

```
$ svn status
M       integer.c
M       mathops.c
```

```
$
```

While testing his bug fix, Harry notices that his changes bring to light a tangentially related bug in the user interface logic found in `button.c`. Harry decides that he'll go ahead and fix that bug, too, as a separate commit from his math fixes. Now, in a small working copy with only a handful of files and few logical changes, Harry can probably keep his two logical change groupings mentally organized without any problem. But today he's going to use Subversion's changelists feature as a special favor to the authors of this book.

Harry first creates a changelist and associates with it the two files he's already changed. He does this by using the **svn changelist** command to assign the same arbitrary changelist name to those files:

```
$ svn changelist math-fixes integer.c mathops.c
Path -'integer.c' is now a member of changelist -'math-fixes'.
Path -'mathops.c' is now a member of changelist -'math-fixes'.
$ svn status

--- Changelist -'math-fixes':
M       integer.c
M       mathops.c
$
```

就像你看到的，你的**svn status**反映了新的分组。

Harry now sets off to fix the secondary UI problem. Since he knows which file he'll be changing, he assigns that path to a changelist, too. Unfortunately, Harry carelessly assigns this third file to the same changelist as the previous two files:

```
$ svn changelist math-fixes button.c
Path -'button.c' is now a member of changelist -'math-fixes'.
$ svn status

--- Changelist -'math-fixes':
        button.c
M       integer.c
M       mathops.c
$
```

Fortunately, Harry catches his mistake. At this point, he has two options. He can remove the changelist association from `button.c`, and then assign a different changelist name:

```
$ svn changelist ---remove button.c
Path -'button.c' is no longer a member of a changelist.
$ svn changelist ui-fix button.c
Path -'button.c' is now a member of changelist -'ui-fix'.
$
```

Or, he can skip the removal and just assign a new changelist name. In this case, Subversion will first warn Harry that `button.c` is being removed from the first changelist:

```
$ svn changelist ui-fix button.c
svn: warning: Removing -'button.c' from changelist -'math-fixes'.
Path -'button.c' is now a member of changelist -'ui-fix'.
$ svn status

--- Changelist -'ui-fix':
        button.c

--- Changelist -'math-fixes':
M       integer.c
M       mathops.c
$
```

Harry now has two distinct changelists present in his working copy, and **svn status** will group its output according to these changelist determinations. Notice that even though Harry hasn't yet modified `button.c`, it still shows up in the output of **svn status** as interesting because it has a changelist assignment. Changelists can be added to and removed from files at any time, regardless of whether they contain local modifications.

现在 Harry 开始修正 `button.c` 中的用户界面问题。

```
$ svn status

--- Changelist -'ui-fix':
M      button.c

--- Changelist -'math-fixes':
M       integer.c
M       mathops.c
$
```

3.10.2. 用修改列表作为操作过滤器

The visual grouping that Harry sees in the output of **svn status** as shown in our previous section is nice, but not entirely useful. The **status** command is but one of many operations that he might wish to perform on his working copy. Fortunately, many of Subversion's other operations understand how to operate on changelists via the use of the `--changelist` option.

When provided with a `--changelist` option, Subversion commands will limit the scope of their operation to only those files to which a particular changelist name is assigned. If Harry now wants to see the actual changes he's made to the files in his `math-fixes` changelist, he *could* explicitly list only the files that make up that changelist on the **svn diff** command line.

```
$ svn diff integer.c mathops.c
Index: integer.c
===================================================================
--- integer.c (revision 1157)
+++ integer.c (working copy)
…
Index: mathops.c
===================================================================
--- mathops.c (revision 1157)
+++ mathops.c (working copy)
…
$
```

That works okay for a few files, but what if Harry's change touched 20 or 30 files? That would be an annoyingly long list of explicitly named files. Now that he's using changelists, though, Harry can avoid explicitly listing the set of files in his changelist from now on, and instead provide just the changelist name:

```
$ svn diff ---changelist math-fixes
Index: integer.c
===================================================================
--- integer.c (revision 1157)
+++ integer.c (working copy)
…
Index: mathops.c
===================================================================
--- mathops.c (revision 1157)
+++ mathops.c (working copy)
…
$
```

And when it's time to commit, Harry can again use the `--changelist` option to limit the scope of the commit to files in a certain changelist. He might commit his user interface fix by doing the following:

```
$ svn ci --m -"Fix a UI bug found while working on math logic." \
      ---changelist ui-fix
Sending        button.c
Transmitting file data -.
Committed revision 1158.
$
```

In fact, the **svn commit** command provides a second changelists-related option: `--keep-changelists`. Normally, changelist assignments are removed from files after they are committed. But if `--keep-changelists` is provided, Subversion will leave the changelist assignment on the committed (and now unmodified) files. In any case, committing files assigned to one changelist leaves other changelists undisturbed.

```
$ svn status

--- Changelist -'math-fixes':
M       integer.c
M       mathops.c
$
```

注意

The `--changelist` option acts only as a filter for Subversion command targets, and will not add targets to an operation. For example, on a commit operation specified as **svn commit /path/to/dir**, the target is the directory `/path/to/dir` and its children (to infinite depth). If you then add a changelist specifier to that command, only those files in and under `/path/to/dir` that are assigned that changelist name will be considered as targets of the commit—the commit will not include files located elsewhere (such is in `/path/to/another-dir`), regardless of their changelist assignment, even if they are part of the same working copy as the operation's target(s).

Even the **svn changelist** command accepts the `--changelist` option. This allows you to quickly and easily rename or remove a changelist:

```
$ svn changelist math-bugs ---changelist math-fixes ---depth
infinity -.
svn: warning: Removing -'integer.c' from changelist -'math-fixes'.
Path -'integer.c' is now a member of changelist -'math-bugs'.
svn: warning: Removing -'mathops.c' from changelist -'math-fixes'.
Path -'mathops.c' is now a member of changelist -'math-bugs'.
$ svn changelist ---remove ---changelist math-bugs ---depth
infinity -.
Path -'integer.c' is no longer a member of a changelist.
Path -'mathops.c' is no longer a member of a changelist.
$
```

Finally, you can specify multiple instances of the `--changelist` option on a single command line. Doing so limits the operation you are performing to files found in any of the specified changesets.

3.10.3. 修改列表的限制

Subversion's changelist feature is a handy tool for grouping working copy files, but it does have a few limitations. Changelists are artifacts of a particular working copy, which means that changelist assignments cannot be propagated to the repository or otherwise shared with other users. Changelists can be assigned only to files—Subversion doesn't currently support the use of changelists with directories. Finally, you can have at most one changelist assignment on a given working copy file. Here is where the blog post category and photo service tag analogies break down—if you find yourself needing to assign a file to multiple changelists, you're out of luck.

3.11. 网络模型

At some point, you're going to need to understand how your Subversion client communicates with its server. Subversion's networking layer is abstracted, meaning that Subversion clients exhibit the same general behaviors no matter what sort of server they are operating against. Whether speaking the HTTP protocol (`http://`) with the Apache HTTP Server or speaking the custom Subversion protocol (`svn://`) with **svnserve**, the basic network model is the same. In this section, we'll explain the basics of that network model, including how Subversion manages authentication and authorization matters.

3.11.1. 请求和响应

The Subversion client spends most of its time managing working copies. When it needs information from a remote repository, however, it makes a network request, and the server responds with an appropriate answer. The details of the network protocol are hidden from the user—the client attempts to access a URL, and depending on the URL scheme, a particular protocol is used to contact the server (see the sidebar 版本库的 URL).

> **提示**
>
> Run `svn --version` to see which URL schemes and protocols the client knows how to use.

When the server process receives a client request, it often demands that the client identify itself. It issues an authentication challenge to the client, and the client responds by providing *credentials* back to the server. Once authentication is complete, the server responds with the original information that the client asked for. Notice that this system is different from systems such as CVS, where the client preemptively offers credentials ("logs in") to the server before ever making a request. In Subversion, the server "pulls" credentials by challenging the client at the appropriate moment, rather than the client "pushing" them. This makes certain operations more elegant. For example, if a server is configured to allow anyone in the world to read a repository, the server will never issue an authentication challenge when a client attempts to **svn checkout**.

If the particular network requests issued by the client result in a new revision being created in the repository (e.g., **svn commit**), Subversion uses the authenticated username associated with those requests as the author of the revision. That is, the authenticated user's name is stored as the value of the `svn:author` property on the new revision (see 第 9.10 节 "Subversion 属性"). If the client was not authenticated (i.e., if the server never issued an authentication challenge), the revision's `svn:author` property is empty.

3.11.2. 客户端凭证缓存

Many servers are configured to require authentication on every request. This would be a big annoyance to users if they were forced to type their passwords over and over again. Fortunately, the Subversion client has a remedy for this—a built-in system for caching authentication credentials on disk. By default, whenever the command-line client successfully responds to a server's authentication challenge, credentials are cached on disk and keyed on a combination of the server's hostname, port, and authentication realm.

高级主题

When the client receives an authentication challenge, it first looks for the appropriate credentials in the user's disk cache. If seemingly suitable credentials are not present, or if the cached credentials ultimately fail to authenticate, the client will, by default, fall back to prompting the user for the necessary information.

The security-conscious reader will suspect immediately that there is reason for concern here. "Caching passwords on disk? That's terrible! You should never do that!"

The Subversion developers recognize the legitimacy of such concerns, and so Subversion works with available mechanisms provided by the operating system and environment to try to minimize the risk of leaking this information. Here's a breakdown of what this means for users on the most common platforms:

- On Windows, the Subversion client stores passwords in the `%APPDATA%/Subversion/auth/` directory. On Windows 2000 and later, the standard Windows cryptography services are used to encrypt the password on disk. Because the encryption key is managed by Windows and is tied to the user's own login credentials, only the user can decrypt the cached password. (Note that if the user's Windows account password is reset by an administrator, all of the cached passwords become undecipherable. The Subversion client will behave as though they don't exist, prompting for passwords when required.)

- Similarly, on Mac OS X, the Subversion client stores all repository passwords in the login keyring (managed by the Keychain service), which is protected by the user's account password. User preference settings can impose additional policies, such as requiring that the user's account password be entered each time the Subversion password is used.

- For other Unix-like operating systems, no standard "keychain" services exist. However, the Subversion client knows how to store password securely using the "GNOME Keyring" and "KDE Wallet" services. Also, before storing unencrypted passwords in the `~/.subversion/auth/` caching area, the Subversion client will ask the user for permission to do so. Note that the `auth/` caching area is still permission-protected so that only the user (owner) can read data from it, not the world at large. The operating system's own file permissions protect the passwords from other non-administrative users on the same system, provided they have no direct physical access to the storage media of the home directory, or backups thereof.

Of course, for the truly paranoid, none of these mechanisms meets the test of perfection. So for those folks willing to sacrifice convenience for the ultimate in security, Subversion provides various ways of disabling its credentials caching system altogether.

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
…
```

Or, if you want to disable credential caching permanently, you can edit the `config` file in your runtime configuration area and set the `store-auth-creds` option to `no`. This will prevent the storing of credentials used in any Subversion interactions you perform on the affected computer. This can be extended to cover all users on the computer, too, by modifying the system-wide runtime configuration area (described in 第 7.1.1 节 "配置区布局").

```
[auth]
store-auth-creds = no
```

Sometimes users will want to remove specific credentials from the disk cache. To do this, you need to navigate into the `auth/` area and manually delete the appropriate cache file. Credentials are cached in individual files; if you look inside each file, you will see keys and values. The `svn:realmstring` key describes the particular server realm that the file is associated with:

```
$ ls ~/.subversion/auth/svn.simple/
5671adf2865e267db74f09ba6f872c28
3893ed123b39500bca8a0b382839198e
5c3c22968347b390f349ff340196ed39

$ cat ~/.subversion/auth/
svn.simple/5671adf2865e267db74f09ba6f872c28

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。

One last word about **svn**'s authentication behavior, specifically regarding the `--username` and `--password` options. Many client subcommands accept these options, but it is important to understand that using these options does *not* automatically send credentials to the server. As discussed earlier, the server "pulls" credentials from the client when it deems necessary; the client cannot "push" them at will. If a username and/or password are passed as options, they will be presented to the server only if the server requests them. These options are typically used to authenticate as a different user than Subversion would have chosen by default (such as your system login name) or when trying to avoid interactive prompting (such as when calling **svn** from a script).

注意

A common mistake is to misconfigure a server so that it never issues an authentication challenge. When users pass `--username` and `--password` options to the client, they're surprised to see that they're never used; that is, new revisions still appear to have been committed anonymously!

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. First, the client checks whether the user specified any credentials as command-line options (`--username` and/or `--password`). If so, the client will try to use those credentials to authenticate against the server.

2. If no command-line credentials were provided, or the provided ones were invalid, the client looks up the server's hostname, port, and realm in the runtime configuration's `auth/` area, to see whether appropriate credentials are cached there. If so, it attempts to use those credentials to authenticate.

3. 最终，如果前一种机制未能够为服务器成功认证用户，客户端返回并提示用户输入正确的凭证(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。)

3.12. 总结

After reading this chapter, you should have a firm grasp on some of Subversion's features that, while perhaps not used *every* time you interact with your version control system, are certainly handy to know about. But don't stop here! Read on to the following chapter, where you'll learn about branches, tags, and merging. Then you'll have nearly full mastery of the Subversion client. Though our lawyers won't allow us to promise you anything, this additional knowledge could make you measurably more cool. [12]

---

[12]No purchase necessary. Certains terms and conditions apply. No guarantee of coolness—implicit or otherwise—exists. Mileage may vary.

第 **4** 章 分支与合并

> "君子务本"
>
> —孔子

Branching, tagging, and merging are concepts common to almost all version control systems. If you're not familiar with these ideas, we provide a good introduction in this chapter. If you are familiar, hopefully you'll find it interesting to see how Subversion implements them.

Branching is a fundamental part of version control. If you're going to allow Subversion to manage your data, this is a feature you'll eventually come to depend on. This chapter assumes that you're already familiar with Subversion's basic concepts (第 1 章 基本概念).

4.1. 什么是分支？

Suppose it's your job to maintain a document for a division in your company—a handbook of some sort. One day a different division asks you for the same handbook, but with a few parts "tweaked" for them, since they do things slightly differently.

What do you do in this situation? You do the obvious: make a second copy of your document and begin maintaining the two copies separately. As each department asks you to make small changes, you incorporate them into one copy or the other.

You often want to make the same change to both copies. For example, if you discover a typo in the first copy, it's very likely that the same typo exists in the second copy. The two documents are almost the same, after all; they differ only in small, specific ways.

This is the basic concept of a *branch*—namely, a line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of something, and moves on from there, generating its own history (see 图 4.1 "分支与开发").

图 **4.1.** 分支与开发
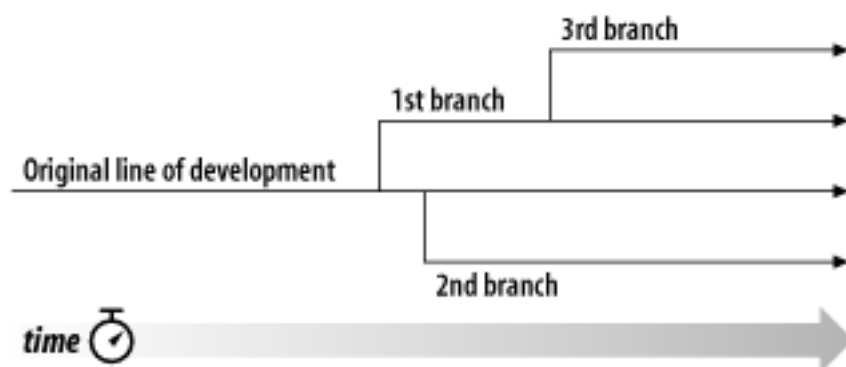
Subversion has commands to help you maintain parallel branches of your files and directories. It allows you to create branches by copying your data, and remembers that the copies are related to one another. It also helps you duplicate changes from one branch to another. Finally, it can make portions of your working copy reflect different branches so that you can "mix and match" different lines of development in your daily work.

4.2. 使用分支

At this point, you should understand how each commit creates an entirely new filesystem tree (called a "revision") in the repository. If you don't, go back and read about revisions in 第 1.3.3 节 "修订版本".

For this chapter, we'll go back to the same example from 第 1 章 基本概念. Remember that you and your collaborator, Sally, are sharing a repository that contains two projects, `paint` and `calc`. Notice that in 图 4.2 "开始规划版本库", however, each project directory now contains subdirectories named `trunk` and `branches`. The reason for this will soon become clear.

图 **4.2.** 开始规划版本库
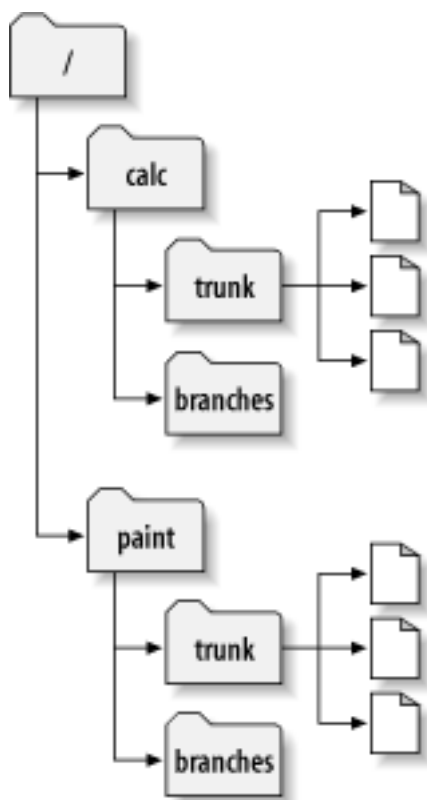
As before, assume that Sally and you both have working copies of the "calc" project. Specifically, you each have a working copy of `/calc/trunk`. All the files for the project are in this subdirectory rather than in `/calc` itself, because your team has decided that `/calc/trunk` is where the "main line" of development is going to take place.

Let's say that you've been given the task of implementing a large software feature. It will take a long time to write, and will affect all the files in the project. The immediate problem is that you don't want to interfere with Sally, who is in the process of fixing small bugs here and there. She's depending on the fact that the latest version of the project (in `/calc/trunk`) is always usable. If you start committing your changes bit by bit, you'll surely break things for Sally (and other team members as well).

One strategy is to crawl into a hole: you and Sally can stop sharing information for a week or two. That is, start gutting and reorganizing all the files in your working copy, but don't commit or

update until you're completely finished with the task. There are a number of problems with this, though. First, it's not very safe. Most people like to save their work to the repository frequently, should something bad accidentally happen to their working copy. Second, it's not very flexible. If you do your work on different computers (perhaps you have a working copy of `/calc/ trunk` on two different machines), you'll need to manually copy your changes back and forth or just do all the work on a single computer. By that same token, it's difficult to share your changes in progress with anyone else. A common software development "best practice" is to allow your peers to review your work as you go. If nobody sees your intermediate commits, you lose potential feedback and may end up going down the wrong path for weeks before another person on your team notices. Finally, when you're finished with all your changes, you might find it very difficult to remerge your final work with the rest of the company's main body of code. Sally (or others) may have made many other changes in the repository that are difficult to incorporate into your working copy—especially if you run **svn update** after weeks of isolation.

The better solution is to create your own branch, or line of development, in the repository. This allows you to save your half-broken work frequently without interfering with others, yet you can still selectively share information with your collaborators. You'll see exactly how this works as we go.

4.2.1. 创建分支

Creating a branch is very simple—you make a copy of the project in the repository using the **svn copy** command. Subversion is able to copy not only single files, but whole directories as well. In this case, you want to make a copy of the `/calc/trunk` directory. Where should the new copy live? Wherever you wish—it's a matter of project policy. Let's say that your team has a policy of creating branches in the `/calc/branches` area of the repository, and you want to name your branch `my-calc-branch`. You'll want to create a new directory, `/calc/branches/my-calc-branch`, which begins its life as a copy of `/calc/trunk`.

You may already have seen **svn copy** used to copy one file to another within a working copy. But it can also be used to do a "remote" copy entirely within the repository. Just copy one URL to another:

```
$ 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.
```

This command causes a near-instantaneous commit in the repository, creating a new directory in revision 341. The new directory is a copy of `/calc/trunk`. This is shown in 图 4.3 "版本库与复制". [1] While it's also possible to create a branch by using **svn copy** to duplicate a directory within the working copy, this technique isn't recommended. It can be quite slow, in fact! Copying a directory on the client side is a linear-time operation, in that it actually has to

---

[1]Subversion does not support copying between different repositories. When using URLs with **svn copy** or **svn move**, you can only copy items within the same repository.

duplicate every file and subdirectory on the local disk. Copying a directory on the server, however, is a constant-time operation, and it's the way most people create branches.

图 4.3. 版本库与复制
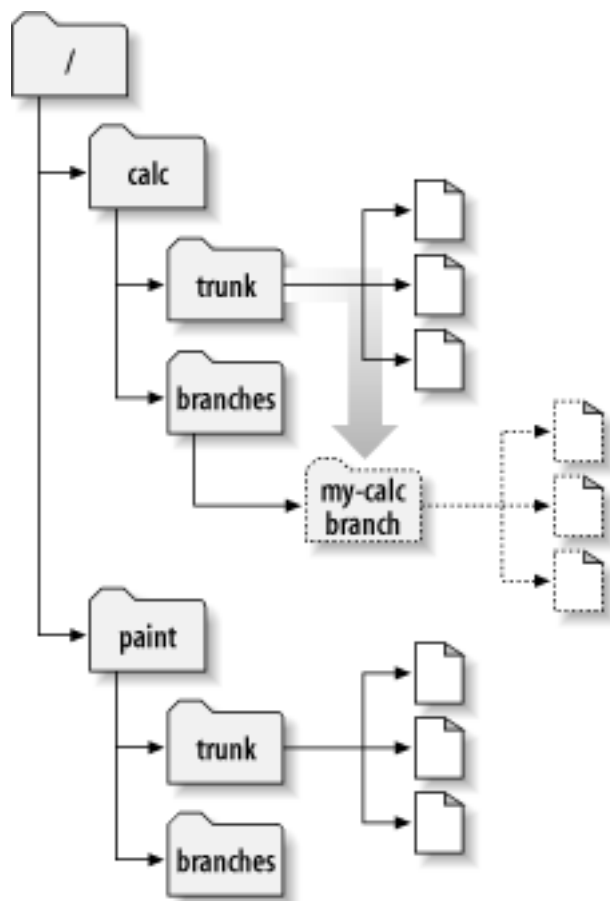
> **廉价复制**
>
> Subversion's repository has a special design. When you copy a directory, you don't need to worry about the repository growing huge—Subversion doesn't actually duplicate any data. Instead, it creates a new directory entry that points to an *existing* tree. If you're an experienced Unix user, you'll recognize this as the same concept behind a hard link. As further changes are made to files and directories beneath the copied directory, Subversion continues to employ this hard link concept where it can. It duplicates data only when it is necessary to disambiguate different versions of objects.
>
> This is why you'll often hear Subversion users talk about "cheap copies." It doesn't matter how large the directory is—it takes a very tiny, constant amount of time and space to make a copy of it. In fact, this feature is the basis of how commits work in Subversion: each revision is a "cheap copy" of the previous revision, with a few items lazily changed within. (To read more about this, visit Subversion's web site and read about the "bubble up" method in Subversion's design documents.)
>
> Of course, these internal mechanics of copying and sharing data are hidden from the user, who simply sees copies of trees. The main point here is that copies are cheap, both in time and in space. If you create a branch entirely within the repository (by running **svn copy URL1 URL2**), it's a quick, constant-time operation. Make branches as often as you want.

4.2.2. 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副本来开始使用：

```
$ 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.
```

There's nothing special about this working copy; it simply mirrors a different directory in the repository. When you commit changes, however, Sally won't see them when she updates, because her working copy is of `/calc/trunk`. (Be sure to read 第 4.5 节 "使用分支" later in this chapter: the **svn switch** command is an alternative way of creating a working copy of a branch.)

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。

Now two independent lines of development (shown in 图 4.4 "一个文件的分支历史") are happening on `integer.c`.

图 4.4. 一个文件的分支历史
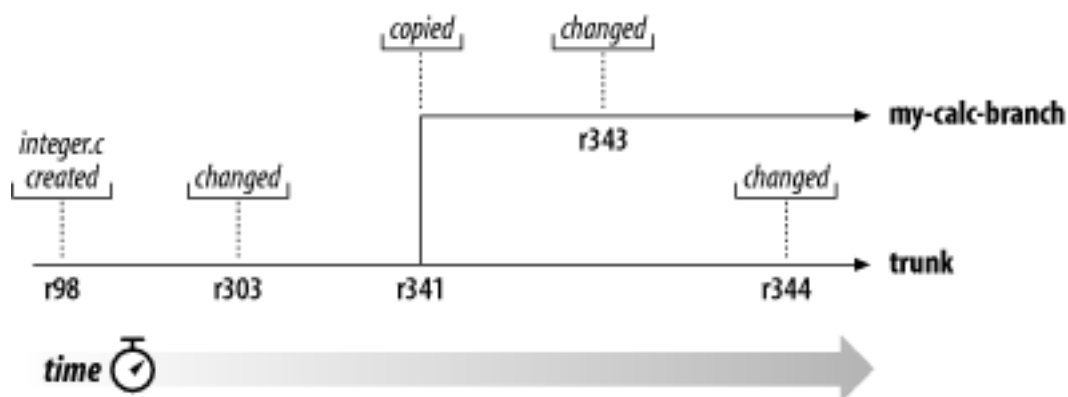
当你看到integer.c的改变时，你会发现很有趣：

```
$ 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:
```

```
    A -/calc/trunk/integer.c

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


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

Notice that Subversion is tracing the history of your branch's `integer.c` all the way back through time, even traversing the point where it was copied. It shows the creation of the branch as an event in the history, because `integer.c` was implicitly copied when all of `/calc/trunk/` was copied. Now look at what happens when Sally runs the same command on her copy of the file:

```
$ 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:
   A -/calc/trunk/integer.c

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


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

Sally sees her own revision 344 change, but not the change you made in revision 343. As far as Subversion is concerned, these two commits affected different files in different repository locations. However, Subversion *does* show that the two files share a common history. Before the branch copy was made in revision 341, the files used to be the same file. That's why you and Sally both see the changes made in revisions 303 and 98.

4.2.3. 分支背后的关键概念

You should remember two important lessons from this section. First, Subversion has no internal concept of a branch—it knows only how to make copies. When you copy a directory, the resultant directory is only a "branch" because *you* attach that meaning to it. You may think of the directory differently, or treat it differently, but to Subversion it's just an ordinary directory that happens to carry some extra historical information.

Second, because of this copy mechanism, Subversion's branches exist as *normal filesystem directories* in the repository. This is different from other version control systems, where branches are typically defined by adding extra-dimensional "labels" to collections of files. The location of your branch directory doesn't matter to Subversion. Most teams follow a convention of putting all branches into a /branches directory, but you're free to invent any policy you wish.

4.3. 基本合并

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。

For projects that have a large number of contributors, it's common for most people to have working copies of the trunk. Whenever someone needs to make a long-running change that is likely to disrupt the trunk, a standard procedure is to create a private branch and commit changes there until all the work is complete.

So, the good news is that you and Sally aren't interfering with each other. The bad news is that it's very easy to drift *too* far apart. Remember that one of the problems with the "crawl in a hole" strategy is that by the time you're finished with your branch, it may be near-impossible to merge your changes back into the trunk without a huge number of conflicts.

Instead, you and Sally might continue to share changes as you work. It's up to you to decide which changes are worth sharing; Subversion gives you the ability to selectively "copy" changes between branches. And when you're completely finished with your branch, your entire set of branch changes can be copied back into the trunk. In Subversion terminology, the general act of replicating changes from one branch to another is called *merging*, and it is performed using various invocations of the **svn merge** command.

In the examples that follow, we're assuming that both your Subversion client and server are running Subversion 1.5 (or later). If either client or server is older than version 1.5, things are more complicated: the system won't track changes automatically, and you'll have to use painful manual methods to achieve similar results. That is, you'll always need to use the detailed merge syntax to specify specific ranges of revisions to replicate (see later in this chapter), and take special care to keep track of what's already been merged and what hasn't. For this reason, we *strongly* recommend that you make sure your client and server are at least at version 1.5.

4.3.1. 变更集

Before we proceed further, we should warn you that there's going to be a lot of discussion of "changes" in the pages ahead. A lot of people experienced with version control systems use

the terms "change" and "changeset" interchangeably, and we should clarify what Subversion understands as a *changeset*.

Everyone seems to have a slightly different definition of changeset, or at least a different expectation of what it means for a version control system to have one. For our purposes, let's say that a changeset is just a collection of changes with a unique name. The changes might include textual edits to file contents, modifications to tree structure, or tweaks to metadata. In more common speak, a changeset is just a patch with a name you can refer to.

In Subversion, a global revision number N names a tree in the repository: it's the way the repository looked after the Nth commit. It's also the name of an implicit changeset: if you compare tree N with tree N−1, you can derive the exact patch that was committed. For this reason, it's easy to think of revision N as not just a tree, but a changeset as well. If you use an issue tracker to manage bugs, you can use the revision numbers to refer to particular patches that fix bugs—for example, "this issue was fixed by r9238." Somebody can then run **svn log -r 9238** to read about the exact changeset that fixed the bug, and run **svn diff -c 9238** to see the patch itself. And (as you'll see shortly) Subversion's **svn merge** command is able to use revision numbers. You can merge specific changesets from one branch to another by naming them in the merge arguments: passing **-c 9238** to **svn merge** would merge changeset r9238 into your working copy.

4.3.2. 保持分支同步

Continuing with our running example, let's suppose that a week has passed since you started working on your private branch. Your new feature isn't finished yet, but at the same time you know that other people on your team have continued to make important changes in the project's /trunk. It's in your best interest to replicate those changes to your own branch, just to make sure they mesh well with your changes. In fact, this is a best practice: frequently keeping your branch in sync with the main development line helps prevent "surprise" conflicts when it comes time for you to fold your changes back into the trunk.

Subversion is aware of the history of your branch and knows when it divided away from the trunk. To replicate the latest, greatest trunk changes to your branch, first make sure your working copy of the branch is "clean"—that it has no local modifications reported by **svn status**. Then simply run:

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

$ svn merge ^/trunk
--- Merging r345 through r356 into -'.':
U    button.c
U    integer.c
```

This basic syntax—**svn merge URL**—tells Subversion to merge all recent changes from the URL to the current working directory (which is typically the root of your working copy). Also notice that we're using the caret (^) syntax[2] to avoid having to type out the entire /trunk URL.

---

[2]这是 svn 1.6 新增的。

After running the prior example, your branch working copy now contains new local modifications, and these edits are duplications of all of the changes that have happened on the trunk since you first created your branch:

```
$ svn status
 M      -.
M       button.c
M       integer.c
```

At this point, the wise thing to do is look at the changes carefully with **svn diff**, and then build and test your branch. Notice that the current working directory (".") has also been modified; the **svn diff** will show that its svn:mergeinfo property has been either created or modified. This is important merge-related metadata that you should *not* touch, since it will be needed by future **svn merge** commands. (We'll learn more about this metadata later in the chapter.)

After performing the merge, you might also need to resolve some conflicts (just as you do with **svn update**) or possibly make some small edits to get things working properly. (Remember, just because there are no *syntactic* conflicts doesn't mean there aren't any *semantic* conflicts!) If you encounter serious problems, you can always abort the local changes by running **svn revert . -R** (which will undo all local modifications) and start a long "what's going on?" discussion with your collaborators. If things look good, however, you can submit these changes into the repository:

```
$ svn commit --m -"Merged latest trunk changes to my-calc-branch."
Sending         -.
Sending         button.c
Sending         integer.c
Transmitting file data -..
Committed revision 357.
```

At this point, your private branch is now "in sync" with the trunk, so you can rest easier knowing that as you continue to work in isolation, you're not drifting too far away from what everyone else is doing.

> **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：
>
> ```
> $ cd my-calc-branch
> $ svn diff --r 341:HEAD ^/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
> ```
>
> In this particular example, there really isn't much difference. But **svn merge** has special abilities that surpass the **patch** program. The file format used by **patch** is quite limited; it's able to tweak file contents only. There's no way to represent changes to *trees*, such as the addition, removal, or renaming of files and directories. Nor can the **patch** program notice changes to properties. If Sally's change had, say, added a new directory, the output of **svn diff** wouldn't have mentioned it at all. **svn diff** outputs only the limited patch format, so there are some ideas it simply can't express.
>
> The **svn merge** command, however, can express changes in tree structure and properties by directly applying them to your working copy. Even more important, this command records the changes that have been duplicated to your branch so that Subversion is aware of exactly which changes exist in each location (see 第 4.3.3 节 "合并信息和预览".) This is a critical feature that makes branch management usable; without it, users would have to manually keep notes on which sets of changes have or haven't been merged yet.

Suppose that another week has passed. You've committed more changes to your branch, and your comrades have continued to improve the trunk as well. Once again, you'd like to replicate the latest trunk changes to your branch and bring yourself in sync. Just run the same merge command again!

```
$ svn merge ^/trunk
--- Merging r357 through r380 into -'.':
U    integer.c
U    Makefile
A    README
```

Subversion knows which trunk changes you've already replicated to your branch, so it carefully replicates only those changes you don't yet have. Once again, you'll have to build, test, and **svn commit** the local modifications to your branch.

What happens when you finally finish your work, though? Your new feature is done, and you're ready to merge your branch changes back to the trunk (so your team can enjoy the bounty

of your labor). The process is simple. First, bring your branch in sync with the trunk again, just as you've been doing all along:

```
$ svn merge ^/trunk
--- Merging r381 through r385 into -'.':
U    button.c
U    README

$ # build, test, -...

$ svn commit --m -"Final merge of trunk changes to my-calc-branch."
Sending        -.
Sending        button.c
Sending        README
Transmitting file data -..
Committed revision 390.
```

Now, you use **svn merge** to replicate your branch changes back into the trunk. You'll need an up-to-date working copy of /trunk. You can do this by either doing an **svn checkout**, dredging up an old trunk working copy from somewhere on your disk, or using **svn switch** (see 第 4.5 节 "使用分支".) However you get a trunk working copy, remember that it's a best practice to do your merge into a working copy that has *no* local edits and has been recently updated (i.e., is not a mixture of local revisions). If your working copy isn't "clean" in these ways, you can run into some unnecessary conflict-related headaches and **svn merge** will likely return an error.

Once you have a clean working copy of the trunk, you're ready to merge your branch back into it:

```
$ pwd
/home/user/calc-trunk

$ svn update  # (make sure the working copy is up to date)
At revision 390.

$ svn merge ---reintegrate ^/branches/my-calc-branch
--- Merging differences between repository URLs into -'.':
U    button.c
U    integer.c
U    Makefile
 U   -.

$ # build, test, verify, -...

$ svn commit --m -"Merge my-calc-branch back into trunk!"
Sending        -.
Sending        button.c
Sending        integer.c
```

```
Sending        Makefile
Transmitting file data -..
Committed revision 391.
```

Congratulations, your branch has now been remerged back into the main line of development. Notice our use of the `--reintegrate` option this time around. The option is critical for reintegrating changes from a branch back into its original line of development—don't forget it! It's needed because this sort of "merge back" is a different sort of work than what you've been doing up until now. Previously, we had been asking **svn merge** to grab the "next set" of changes from one line of development (the trunk) and duplicate them to another (your branch). This is fairly straightforward, and each time Subversion knows how to pick up where it left off. In our prior examples, you can see that first it merges the ranges 345:356 from trunk to branch; later on, it continues by merging the next contiguously available range, 356:380. When doing the final sync, it merges the range 380:385.

When merging your branch back to the trunk, however, the underlying mathematics is quite different. Your feature branch is now a mishmosh of both duplicated trunk changes and private branch changes, so there's no simple contiguous range of revisions to copy over. By specifying the `--reintegrate` option, you're asking Subversion to carefully replicate *only* those changes unique to your branch. (And in fact, it does this by comparing the latest trunk tree with the latest branch tree: the resulting difference is exactly your branch changes!)

Now that your private branch is merged to trunk, you may wish to remove it from the repository:

```
$ svn delete ^/branches/my-calc-branch \
      --m -"Remove my-calc-branch."
Committed revision 392.
```

But wait! Isn't the history of that branch valuable? What if somebody wants to audit the evolution of your feature someday and look at all of your branch changes? No need to worry. Remember that even though your branch is no longer visible in the `/branches` directory, its existence is still an immutable part of the repository's history. A simple **svn log** command on the `/branches` URL will show the entire history of your branch. Your branch can even be resurrected at some point, should you desire (see 第 4.3.5 节 "找回删除的项目").

In Subversion 1.5, once a `--reintegrate` merge is done from branch to trunk, the branch is no longer usable for further work. It's not able to correctly absorb new trunk changes, nor can it be properly reintegrated to trunk again. For this reason, if you want to keep working on your feature branch, we recommend destroying it and then re-creating it from the trunk:

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-branch \
      --m -"Remove my-calc-branch."
Committed revision 392.

$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/new-branch
```

```
     --m -"Create a new branch from trunk."
Committed revision 393.

$ cd my-calc-branch

$ svn switch ^/branches/new-branch
Updated to revision 393.
```

The final command in the prior example—**svn switch**—is a way of updating an existing working copy to reflect a different repository directory. We'll discuss this more in 第 4.5 节 "使用分支".

4.3.3. 合并信息和预览

The basic mechanism Subversion uses to track changesets—that is, which changes have been merged to which branches—is by recording data in properties. Specifically, merge data is tracked in the `svn:mergeinfo` property attached to files and directories. (If you're not familiar with Subversion properties, now is the time to skim 第 3.2 节 "属性".)

你可以用与其它属性一样的方法，检查合并信息属性：

```
$ cd my-calc-branch
$ svn propget svn:mergeinfo -.
/trunk:341-390
```

It is *not* recommended that you change the value of this property yourself, unless you really know what you're doing. This property is automatically maintained by Subversion whenever you run **svn merge**. Its value indicates which changes (at a given path) have been replicated into the directory in question. In this case, the path is `/trunk` and the directory which has received the specific changes is `/branches/my-calc-branch`.

There's also a subcommand, **svn mergeinfo**, which can be helpful in seeing not only which changesets a directory has absorbed, but also which changesets it's still eligible to receive. This gives a sort of preview of the next set of changes that **svn merge** will replicate to your branch.

```
$ cd my-calc-branch

# Which changes have already been merged from trunk to branch?
$ svn mergeinfo ^/trunk
r341
r342
r343
…
r388
r389
r390

# Which changes are still eligible to merge from trunk to branch?
```

```
$ svn mergeinfo ^/trunk ---show-revs eligible
r391
r392
r393
r394
r395
```

The **svn mergeinfo** command requires a "source" URL (where the changes would be coming from), and takes an optional "target" URL (where the changes would be merged to). If no target URL is given, it assumes that the current working directory is the target. In the prior example, because we're querying our branch working copy, the command assumes we're interested in receiving changes to /branches/mybranch from the specified trunk URL.

Another way to get a more precise preview of a merge operation is to use the --dry-run option:

```
$ svn merge ^/trunk ---dry-run
U    integer.c

$ svn status
#  nothing printed, working copy is still unchanged.
```

The --dry-run option doesn't actually apply any local changes to the working copy. It shows only status codes that *would* be printed in a real merge. It's useful for getting a "high-level" preview of the potential merge, for those times when running **svn diff** gives too much detail.

> **提示**
>
> After performing a merge operation, but before committing the results of the merge, you can use **svn diff --depth=empty** */path/to/merge/target* to see only the changes to the immediate target of your merge. If your merge target was a directory, only property differences will be displayed. This is a handy way to see the changes to the svn:mergeinfo property recorded by the merge operation, which will remind you about what you've just merged.

Of course, the best way to preview a merge operation is to just do it! Remember, running **svn merge** isn't an inherently risky thing (unless you've made local modifications to your working copy—but we've already stressed that you shouldn't be merging into such an environment). If you don't like the results of the merge, simply run **svn revert . -R** to revert the changes from your working copy and retry the command with different options. The merge isn't final until you actually **svn commit** the results.

> **提示**
>
> While it's perfectly fine to experiment with merges by running **svn merge** and **svn revert** over and over, you may run into some annoying (but easily bypassed) roadblocks. For example, if the merge operation adds a new file (i.e., schedules it for addition), **svn revert** won't actually remove the file; it simply unschedules the addition. You're left with an unversioned file. If you then attempt to run the

merge again, you may get conflicts due to the unversioned file "being in the way." Solution? After performing a revert, be sure to clean up the working copy and remove unversioned files and directories. The output of **svn status** should be as clean as possible, ideally showing no output.

4.3.4. 取消修改

An extremely common use for **svn merge** is to roll back a change that has already been committed. Suppose you're working away happily on a working copy of /calc/trunk, and you discover that the change made way back in revision 303, which changed integer.c, is completely wrong. It never should have been committed. You can use **svn merge** to "undo" the change in your working copy, and then commit the local modification to the repository. All you need to do is to specify a *reverse* difference. (You can do this by specifying --revision 303:302, or by an equivalent --change -303.)

```
$ svn merge --c --303 ^/trunk
--- Reverse-merging r303 into -'integer.c':
U    integer.c

$ svn status
 M     -.
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.
```

As we mentioned earlier, one way to think about a repository revision is as a specific changeset. By using the -r option, you can ask **svn merge** to apply a changeset, or a whole range of changesets, to your working copy. In our case of undoing a change, we're asking **svn merge** to apply changeset #303 to our working copy *backward*.

Keep in mind that rolling back a change like this is just like any other **svn merge** operation, so you should use **svn status** and **svn diff** to confirm that your work is in the state you want it to be in, and then use **svn commit** to send the final version to the repository. After committing, this particular changeset is no longer reflected in the HEAD revision.

Again, you may be thinking: well, that really didn't undo the commit, did it? The change still exists in revision 303. If somebody checks out a version of the calc project between revisions 303 and 349, she'll still see the bad change, right?

Yes, that's true. When we talk about "removing" a change, we're really talking about removing it from the HEAD revision. The original change still exists in the repository's history.

For most situations, this is good enough. Most people are only interested in tracking the HEAD of a project anyway. There are special cases, however, where you really might want to destroy all evidence of the commit. (Perhaps somebody accidentally committed a confidential document.) This isn't so easy, it turns out, because Subversion was deliberately designed to never lose information. Revisions are immutable trees that build upon one another. Removing a revision from history would cause a domino effect, creating chaos in all subsequent revisions and possibly invalidating all working copies. [3]

4.3.5. 找回删除的项目

The great thing about version control systems is that information is never lost. Even when you delete a file or directory, it may be gone from the HEAD revision, but the object still exists in earlier revisions. One of the most common questions new users ask is, "How do I get my old file or directory back?"

The first step is to define exactly *which* item you're trying to resurrect. Here's a useful metaphor: you can think of every object in the repository as existing in a sort of two-dimensional coordinate system. The first coordinate is a particular revision tree, and the second coordinate is a path within that tree. So every version of your file or directory can be defined by a specific coordinate pair. (Remember the "peg revision" syntax—foo.c@224—mentioned back in 第 3.9 节 "Peg 和实施修订版本".)

First, you might need to use **svn log** to discover the exact coordinate pair you wish to resurrect. A good strategy is to run **`svn log --verbose`** in a directory that used to contain your deleted item. The `--verbose` (`-v`) option shows a list of all changed items in each revision; all you need to do is find the revision in which you deleted the file or directory. You can do this visually, or by using another tool to examine the log output (via **grep**, or perhaps via an incremental search in an editor).

```
$ 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.
…
```

In the example, we're assuming that you're looking for a deleted file real.c. By looking through the logs of a parent directory, you've spotted that this file was deleted in revision 808.

---

[3]The Subversion project has plans, however, to someday implement a command that would accomplish the task of permanently deleting information. In the meantime, see 第 5.4.1.3 节 "svndumpfilter" for a possible workaround.

Therefore, the last version of the file to exist was in the revision right before that. Conclusion: you want to resurrect the path `/calc/trunk/real.c` from revision 807.

That was the hard part—the research. Now that you know what you want to restore, you have two different choices.

One option is to use **svn merge** to apply revision 808 "in reverse." (We already discussed how to undo changes in 第 4.3.4 节 "取消修改".) This would have the effect of re-adding `real.c` as a local modification. The file would be scheduled for addition, and after a commit, the file would again exist in `HEAD`.

In this particular example, however, this is probably not the best strategy. Reverse-applying revision 808 would not only schedule `real.c` for addition, but the log message indicates that it would also undo certain changes to `integer.c`, which you don't want. Certainly, you could reverse-merge revision 808 and then **svn revert** the local modifications to `integer.c`, but this technique doesn't scale well. What if 90 files were changed in revision 808?

A second, more targeted strategy is not to use **svn merge** at all, but rather to use the **svn copy** command. Simply copy the exact revision and path "coordinate pair" from the repository to your working copy:

```
$ svn copy ^/trunk/real.c@807 -./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.
```

The plus sign in the status output indicates that the item isn't merely scheduled for addition, but scheduled for addition "with history." Subversion remembers where it was copied from. In the future, running **svn log** on this file will traverse back through the file's resurrection and through all the history it had prior to revision 807. In other words, this new `real.c` isn't really new; it's a direct descendant of the original, deleted file. This is usually considered a good and useful thing. If, however, you wanted to resurrect the file *without* maintaining a historical link to the old file, this technique works just as well:

```
$ svn cat ^/trunk/real.c@807 > -./real.c

$ svn add real.c
A         real.c

$ svn commit --m -"Re-created real.c from revision 807."
Adding          real.c
```

```
Transmitting file data -.
Committed revision 1390.
```

Although our example shows us resurrecting a file, note that these same techniques work just as well for resurrecting deleted directories. Also note that a resurrection doesn't have to happen in your working copy—it can happen entirely in the repository:

```
$ svn copy ^/trunk/real.c@807 ^/trunk/ \
      --m -"Resurrect real.c from revision 807."
Committed revision 1390.

$ svn update
A    real.c
Updated to revision 1390.
```

4.4. 高级合并

Here ends the automated magic. Sooner or later, once you get the hang of branching and merging, you're going to have to ask Subversion to merge *specific* changes from one place to another. To do this, you're going to have to start passing more complicated arguments to **svn merge**. The next section describes the fully expanded syntax of the command and discusses a number of common scenarios that require it.

4.4.1. 摘录合并

Just as the term "changeset" is often used in version control systems, so is the term *cherrypicking*. This word refers to the act of choosing *one* specific changeset from a branch and replicating it to another. Cherrypicking may also refer to the act of duplicating a particular set of (not necessarily contiguous!) changesets from one branch to another. This is in contrast to more typical merging scenarios, where the "next" contiguous range of revisions is duplicated automatically.

Why would people want to replicate just a single change? It comes up more often than you'd think. For example, let's go back in time and imagine that you haven't yet merged your private feature branch back to the trunk. At the water cooler, you get word that Sally made an interesting change to integer.c on the trunk. Looking over the history of commits to the trunk, you see that in revision 355 she fixed a critical bug that directly impacts the feature you're working on. You might not be ready to merge all the trunk changes to your branch just yet, but you certainly need that particular bug fix in order to continue your work.

```
$ svn diff --c 355 ^/trunk

Index: integer.c
===================================================================
--- integer.c (revision 354)
+++ integer.c (revision 355)
```

```
@@ --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, -"CP/MM");
+    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;
```

Just as you used **svn diff** in the prior example to examine revision 355, you can pass the same option to **svn merge**:

```
$ svn merge --c 355 ^/trunk
U    integer.c

$ svn status
M     integer.c
```

You can now go through the usual testing procedures before committing this change to your branch. After the commit, Subversion marks r355 as having been merged to the branch so that future "magic" merges that synchronize your branch with the trunk know to skip over r355. (Merging the same change to the same branch almost always results in a conflict!)

```
$ cd my-calc-branch

$ svn propget svn:mergeinfo -.
/trunk:341-349,355

# Notice that r355 isn't listed as -"eligible" to merge, because
# it's already been merged.
$ svn mergeinfo ^/trunk ---show-revs eligible
r350
r351
r352
r353
r354
r356
r357
r358
r359
r360

$ svn merge ^/trunk
--- Merging r350 through r354 into -'.':
```

```
 U   -.
U    integer.c
U    Makefile
--- Merging r356 through r360 into -'.':
 U   -.
U    integer.c
U    button.c
```

This use case of replicating (or *backporting*) bug fixes from one branch to another is perhaps the most popular reason for cherrypicking changes; it comes up all the time, for example, when a team is maintaining a "release branch" of software. (We discuss this pattern in 第 4.8.1 节 "发布分支".)

警告

Did you notice how, in the last example, the merge invocation caused two distinct ranges of merges to be applied? The **svn merge** command applied two independent patches to your working copy to skip over changeset 355, which your branch already contained. There's nothing inherently wrong with this, except that it has the potential to make conflict resolution trickier. If the first range of changes creates conflicts, you *must* resolve them interactively for the merge process to continue and apply the second range of changes. If you postpone a conflict from the first wave of changes, the whole merge command will bail out with an error message. [4]

A word of warning: while **svn diff** and **svn merge** are very similar in concept, they do have different syntax in many cases. Be sure to read about them in 第 9 章 *Subversion* 完全参考 for details, or ask **svn help**. For example, **svn merge** requires a working copy path as a target, that is, a place where it should apply the generated patch. If the target isn't specified, it assumes you are trying to perform one of the following common operations:

- 你希望合并目录修改到工作副本的当前目录。

- 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。

If you are merging a directory and haven't specified a target path, **svn merge** assumes the first case and tries to apply the changes into your current directory. If you are merging a file, and that file (or a file by the same name) exists in your current working directory, **svn merge** assumes the second case and tries to apply the changes to a local file with the same name.

4.4.2. 合并的语法：完整的描述

You've now seen some examples of the **svn merge** command, and you're about to see several more. If you're feeling confused about exactly how merging works, you're not alone. Many users (especially those new to version control) are initially perplexed about the proper syntax of the command and about how and when the feature should be used. But fear not, this command is actually much simpler than you think! There's a very easy technique for understanding exactly how **svn merge** behaves.

---

[4]At least, this is true in Subversion 1.5 at the time of this writing. This behavior may improve in future versions of Subversion.

The main source of confusion is the *name* of the command. The term "merge" somehow denotes that branches are combined together, or that some sort of mysterious blending of data is going on. That's not the case. A better name for the command might have been **svn diff-and-apply**, because that's all that happens: two repository trees are compared, and the differences are applied to a working copy.

If you're using **svn merge** to do basic copying of changes between branches, it will generally do the right thing automatically. For example, a command such as the following:

```
$ svn merge ^/branches/some-branch
```

will attempt to duplicate any changes made on `some-branch` into your current working directory, which is presumably a working copy that shares some historical connection to the branch. The command is smart enough to only duplicate changes that your working copy doesn't yet have. If you repeat this command once a week, it will only duplicate the "newest" branch changes that happened since you last merged.

If you choose to use the **svn merge** command in all its full glory by giving it specific revision ranges to duplicate, the command takes three main arguments:

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副本(通常叫做合并的**目标**)。

Once these three arguments are specified, the two trees are compared, and the differences are applied to the target working copy as local modifications. When the command is done, the results are no different than if you had hand-edited the files or run various **svn add** or **svn delete** commands yourself. If you like the results, you can commit them. If you don't like the results, you can simply **svn revert** all of the changes.

The syntax of **svn merge** allows you to specify the three necessary arguments rather flexibly. Here are some examples:

```
$ 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
```

The first syntax lays out all three arguments explicitly, naming each tree in the form *URL@REV* and naming the working copy target. The second syntax can be used as a shorthand for situations when you're comparing two different revisions of the same URL. The last syntax shows how the working copy argument is optional; if omitted, it defaults to the current directory.

While the first example shows the "full" syntax of **svn merge**, it needs to be used very carefully; it can result in merges which do not record any `svn:mergeinfo` metadata at all. The next section talks a bit more about this.

4.4.3. 不使用合并信息的合并

Subversion tries to generate merge metadata whenever it can, to make future invocations of **svn merge** smarter. There are still situations, however, where `svn:mergeinfo` data is not created or changed. Remember to be a bit wary of these scenarios:

合并无关的源
> If you ask **svn merge** to compare two URLs that aren't related to each other, a patch will still be generated and applied to your working copy, but no merging metadata will be created. There's no common history between the two sources, and future "smart" merges depend on that common history.

从外部版本库合并
> While it's possible to run a command such as **svn merge -r 100:200 *http:// svn.foreignproject.com/repos/trunk***, the resultant patch will also lack any historical merge metadata. At time of this writing, Subversion has no way of representing different repository URLs within the `svn:mergeinfo` property.

使用 `--ignore-ancestry`
> If this option is passed to **svn merge**, it causes the merging logic to mindlessly generate differences the same way that **svn diff** does, ignoring any historical relationships. We discuss this later in the chapter in 第 4.4.7 节 "关注还是忽视祖先".

应用反向合并到目标的自然历史
> Earlier in this chapter (第 4.3.4 节 "取消修改") we discussed how to use **svn merge** to apply a "reverse patch" as a way of rolling back changes. If this technique is used to undo a change to an object's personal history (e.g., commit r5 to the trunk, then immediately roll back r5 using **svn merge . -c -5**), this sort of merge doesn't affect the recorded mergeinfo. [5]

4.4.4. 合并冲突

Just like the **svn update** command, **svn merge** applies changes to your working copy. And therefore it's also capable of creating conflicts. The conflicts produced by **svn merge**, however, are sometimes different, and this section explains those differences.

To begin with, assume that your working copy has no local edits. When you **svn update** to a particular revision, the changes sent by the server will always apply "cleanly" to your working copy. The server produces the delta by comparing two trees: a virtual snapshot of your working copy, and the revision tree you're interested in. Because the left hand side of the comparison is exactly equal to what you already have, the delta is guaranteed to correctly convert your working copy into the right hand tree.

---

[5]Interestingly, after rolling back a revision like this, we wouldn't be able to reapply the revision using **svn merge . -c 5**, since the mergeinfo would already list r5 as being applied. We would have to use the `--ignore-ancestry` option to make the merge command ignore the existing mergeinfo!

But **svn merge** has no such guarantees and can be much more chaotic: the advanced user can ask the server to compare *any* two trees at all, even ones that are unrelated to the working copy! This means there's large potential for human error. Users will sometimes compare the wrong two trees, creating a delta that doesn't apply cleanly. **svn merge** will do its best to apply as much of the delta as possible, but some parts may be impossible. Just as the Unix **patch** command sometimes complains about "failed hunks," **svn merge** will similarly complain about "skipped targets":

```
$ svn merge --r 1288:1351 ^/branches/mybranch
U    foo.c
U    bar.c
Skipped missing target: -'baz.c'
U    glub.c
U    sputter.h

Conflict discovered in -'glorb.h'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```

In the previous example, it might be the case that `baz.c` exists in both snapshots of the branch being compared, and the resultant delta wants to change the file's contents, but the file doesn't exist in the working copy. Whatever the case, the "skipped" message means that the user is most likely comparing the wrong two trees; it's the classic sign of user error. When this happens, it's easy to recursively revert all the changes created by the merge (**svn revert . -- recursive**), delete any unversioned files or directories left behind after the revert, and rerun **svn merge** with different arguments.

Also notice that the preceding example shows a conflict happening on `glorb.h`. We already stated that the working copy has no local edits: how can a conflict possibly happen? Again, because the user can use **svn merge** to define and apply any old delta to the working copy, that delta may contain textual changes that don't cleanly apply to a working file, even if the file has no local modifications.

Another small difference between **svn update** and **svn merge** is the names of the full-text files created when a conflict happens. In 第 2.4.5 节 "解决冲突（合并别人的修改）", we saw that an update produces files named `filename.mine`, `filename.rOLDREV`, and `filename.rNEWREV`. When **svn merge** produces a conflict, though, it creates three files named `filename.working`, `filename.left`, and `filename.right`. In this case, the terms "left" and "right" are describing which side of the double-tree comparison the file came from. In any case, these differing names will help you distinguish between conflicts that happened as a result of an update and ones that happened as a result of a merge.

4.4.5. 阻塞修改

Sometimes there's a particular changeset that you don't want to be automatically merged. For example, perhaps your team's policy is to do new development work on `/trunk`, but to be more conservative about backporting changes to a stable branch you use for releasing to the public. On one extreme, you can manually cherrypick single changesets from the trunk to the branch—just the changes that are stable enough to pass muster. Maybe things aren't quite that

strict, though; perhaps most of the time you'd like to just let **svn merge** automatically merge most changes from trunk to branch. In this case, you'd like a way to mask a few specific changes out, that is, prevent them from ever being automatically merged.

In Subversion 1.5, the only way to block a changeset is to make the system believe that the change has *already* been merged. To do this, one can invoke a merge command with the `--record-only` option:

```
$ cd my-calc-branch

$ svn propget svn:mergeinfo -.
/trunk:1680-3305

# Let's make the metadata list r3328 as already merged.
$ svn merge --c 3328 ---record-only ^/trunk

$ svn status
M      -.

$ svn propget svn:mergeinfo -.
/trunk:1680-3305,3328

$ svn commit --m -"Block r3328 from being merged to the branch."
…
```

This technique works, but it's also a little bit dangerous. The main problem is that we're not clearly differentiating between the ideas of "I already have this change" and "I don't have this change." We're effectively lying to the system, making it think that the change was previously merged. This puts the responsibility on you—the user—to remember that the change wasn't actually merged, it just wasn't wanted. There's no way to ask Subversion for a list of "blocked changelists." If you want to track them (so that you can unblock them someday). you'll need to record them in a text file somewhere, or perhaps in an invented property. In Subversion 1.5, unfortunately, this is the only way to manage blocked revisions; the plans are to make a better interface for this in future versions.

4.4.6. 感知合并的日志和注解

One of the main features of any version control system is to keep track of who changed what, and when they did it. The **svn log** and **svn blame** commands are just the tools for this: when invoked on individual files, they show not only the history of changesets that affected the file, but also exactly which user wrote which line of code, and when she did it.

When changes start getting replicated between branches, however, things start to get complicated. For example, if you were to ask **svn log** about the history of your feature branch, it would show exactly every revision that ever affected the branch:

```
$ cd my-calc-branch
```

```
$ svn log --q
------------------------------------------------------------------------
r390 -| user -| 2002-11-22 11:01:57 --0600 (Fri, 22 Nov 2002) -|
1 line
------------------------------------------------------------------------
r388 -| user -| 2002-11-21 05:20:00 --0600 (Thu, 21 Nov 2002) -|
2 lines
------------------------------------------------------------------------
r381 -| user -| 2002-11-20 15:07:06 --0600 (Wed, 20 Nov 2002) -|
2 lines
------------------------------------------------------------------------
r359 -| user -| 2002-11-19 19:19:20 --0600 (Tue, 19 Nov 2002) -|
2 lines
------------------------------------------------------------------------
r357 -| user -| 2002-11-15 14:29:52 --0600 (Fri, 15 Nov 2002) -|
2 lines
------------------------------------------------------------------------
r343 -| user -| 2002-11-07 13:50:10 --0600 (Thu, 07 Nov 2002) -|
2 lines
------------------------------------------------------------------------
r341 -| user -| 2002-11-03 07:17:16 --0600 (Sun, 03 Nov 2002) -|
2 lines
------------------------------------------------------------------------
r303 -| sally -| 2002-10-29 21:14:35 --0600 (Tue, 29 Oct 2002) -|
2 lines
------------------------------------------------------------------------
r98 -| sally -| 2002-02-22 15:35:29 --0600 (Fri, 22 Feb 2002) -|
2 lines
------------------------------------------------------------------------
```

But is this really an accurate picture of all the changes that happened on the branch? What's being left out here is the fact that revisions 390, 381, and 357 were actually the results of merging changes from the trunk. If you look at one of these logs in detail, the multiple trunk changesets that comprised the branch change are nowhere to be seen:

```
$ svn log --v --r 390
------------------------------------------------------------------------
r390 -| user -| 2002-11-22 11:01:57 --0600 (Fri, 22 Nov 2002) -|
1 line
Changed paths:
   M -/branches/my-calc-branch/button.c
   M -/branches/my-calc-branch/README

Final merge of trunk changes to my-calc-branch.
```

We happen to know that this merge to the branch was nothing but a merge of trunk changes. How can we see those trunk changes as well? The answer is to use the --use-merge-history (-g) option. This option expands those "child" changes that were part of the merge.

```
$ svn log --v --r 390 --g
------------------------------------------------------------------------
r390 -| user -| 2002-11-22 11:01:57 --0600 (Fri, 22 Nov 2002) -|
1 line
Changed paths:
   M -/branches/my-calc-branch/button.c
   M -/branches/my-calc-branch/README

Final merge of trunk changes to my-calc-branch.
------------------------------------------------------------------------
r383 -| sally -| 2002-11-21 03:19:00 --0600 (Thu, 21 Nov 2002) -|
2 lines
Changed paths:
   M -/branches/my-calc-branch/button.c
Merged via: r390

Fix inverse graphic error on button.
------------------------------------------------------------------------
r382 -| sally -| 2002-11-20 16:57:06 --0600 (Wed, 20 Nov 2002) -|
2 lines
Changed paths:
   M -/branches/my-calc-branch/README
Merged via: r390

Document my last fix in README.
```

By making the log operation use merge history, we see not just the revision we queried (r390), but also the two revisions that came along on the ride with it—a couple of changes made by Sally to the trunk. This is a much more complete picture of history!

The **svn blame** command also takes the --use-merge-history (-g) option. If this option is neglected, somebody looking at a line-by-line annotation of button.c may get the mistaken impression that you were responsible for the lines that fixed a certain error:

```
$ svn blame button.c
…
   390    user    retval = inverse_func(button, path);
   390    user    return retval;
   390    user    -}
…
```

And while it's true that you did actually commit those three lines in revision 390, two of them were actually written by Sally back in revision 383:

```
$ svn blame button.c --g
…
```

```
G    383    sally    retval = inverse_func(button, path);
G    383    sally    return retval;
     390    user     -}
…
```

Now we know who to *really* blame for those two lines of code!

4.4.7. 关注还是忽视祖先

When conversing with a Subversion developer, you might very likely hear reference to the term *ancestry*. This word is used to describe the relationship between two objects in a repository: if they're related to each other, one object is said to be an ancestor of the other.

For example, suppose you commit revision 100, which includes a change to a file `foo.c`. Then `foo.c@99` is an "ancestor" of `foo.c@100`. On the other hand, suppose you commit the deletion of `foo.c` in revision 101, and then add a new file by the same name in revision 102. In this case, `foo.c@99` and `foo.c@102` may appear to be related (they have the same path), but in fact are completely different objects in the repository. They share no history or "ancestry."

The reason for bringing this up is to point out an important difference between **svn diff** and **svn merge**. The former command ignores ancestry, while the latter command is quite sensitive to it. For example, if you asked **svn diff** to compare revisions 99 and 102 of `foo.c`, you would see line-based diffs; the **diff** command is blindly comparing two paths. But if you asked **svn merge** to compare the same two objects, it would notice that they're unrelated and first attempt to delete the old file, then add the new file; the output would indicate a deletion followed by an add:

```
D    foo.c
A    foo.c
```

Most merges involve comparing trees that are ancestrally related to one another; therefore, **svn merge** defaults to this behavior. Occasionally, however, you may want the **merge** command to compare two unrelated trees. For example, you may have imported two source-code trees representing different vendor releases of a software project (see 第 4.9 节 "供方分支"). If you ask **svn merge** to compare the two trees, you'd see the entire first tree being deleted, followed by an add of the entire second tree! In these situations, you'll want **svn merge** to do a path-based comparison only, ignoring any relations between files and directories. Add the `--ignore-ancestry` option to your **merge** command, and it will behave just like **svn diff**. (And conversely, the `--notice-ancestry` option will cause **svn diff** to behave like the **svn merge** command.)

4.4.8. 合并和移动

A common desire is to refactor source code, especially in Java-based software projects. Files and directories are shuffled around and renamed, often causing great disruption to everyone working on the project. Sounds like a perfect case to use a branch, doesn't it? Just create a branch, shuffle things around, and then merge the branch back to the trunk, right?

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** 还不是足够强壮，特别是针对拷贝和移动操作。

When you use **svn copy** to duplicate a file, the repository remembers where the new file came from, but it fails to transmit that information to the client which is running **svn update** or **svn merge**. Instead of telling the client, "Copy that file you already have to this new location," it sends down an entirely new file. This can lead to problems, especially because the same thing happens with renamed files. A lesser-known fact about Subversion is that it lacks "true renames"—the **svn move** command is nothing more than an aggregation of **svn copy** and **svn delete**.

For example, suppose that while working on your private branch, you rename `integer.c` to `whole.c`. Effectively you've created a new file in your branch that is a copy of the original file, and deleted the original file. Meanwhile, back on `trunk`, Sally has committed some improvements to `integer.c`. Now you decide to merge your branch to the trunk:

```
$ cd calc/trunk

$ svn merge ---reintegrate ^/branches/my-calc-branch
--- Merging differences between repository URLs into -'.':
D   integer.c
A   whole.c
U   -.
```

This doesn't look so bad at first glance, but it's also probably not what you or Sally expected. The merge operation has deleted the latest version of the `integer.c` file (the one containing Sally's latest changes), and blindly added your new `whole.c` file—which is a duplicate of the *older* version of `integer.c`. The net effect is that merging your "rename" to the branch has removed Sally's recent changes from the latest revision!

This isn't true data loss. Sally's changes are still in the repository's history, but it may not be immediately obvious that this has happened. The moral of this story is that until Subversion improves, be very careful about merging copies and renames from one branch to another.

4.4.9. 阻塞不知道合并的客户端

If you've just upgraded your server to Subversion 1.5 or later, there's a significant risk that pre-1.5 Subversion clients can mess up your automated merge tracking. Why is this? When a pre-1.5 Subversion client performs **svn merge**, it doesn't modify the value of the `svn:mergeinfo` property at all. So the subsequent commit, despite being the result of a merge, doesn't tell the repository about the duplicated changes—that information is lost. Later on, when "merge-aware" clients attempt automatic merging, they're likely to run into all sorts of conflicts resulting from repeated merges.

If you and your team are relying on the merge-tracking features of Subversion, you may want to configure your repository to prevent older clients from committing changes. The easy way to do this is by inspecting the "capabilities" parameter in the `start-commit` hook script. If the client reports itself as having `mergeinfo` capabilities, the hook script can allow the commit to start. If the client doesn't report that capability, have the hook deny the commit. We'll learn more about hook scripts in the next chapter; see 第 5.3.2 节 "实现版本库钩子" and start-commit for details.

4.4.10. 合并跟踪的最终信息

The bottom line is that Subversion's merge-tracking feature has an extremely complex internal implementation, and the `svn:mergeinfo` property is the only window the user has into the machinery. Because the feature is relatively new, a numbers of edge cases and possible unexpected behaviors may pop up.

For example, sometimes mergeinfo will be generated when running a simple **svn copy** or **svn move** command. Sometimes mergeinfo will appear on files that you didn't expect to be touched by an operation. Sometimes mergeinfo won't be generated at all, when you expect it to. Furthermore, the management of mergeinfo metadata has a whole set of taxonomies and behaviors around it, such as "explicit" versus "implicit" mergeinfo, "operative" versus "inoperative" revisions, specific mechanisms of mergeinfo "elision," and even "inheritance" from parent to child directories.

We've chosen not to cover these detailed topics in this book for a couple of reasons. First, the level of detail is absolutely overwhelming for a typical user. Second, as Subversion continues to improve, we feel that a typical user *shouldn't* have to understand these concepts; they'll eventually fade into the background as pesky implementation details. All that said, if you enjoy this sort of thing, you can get a fantastic overview in a paper posted at CollabNet's website: http://www.collab.net/community/subversion/articles/merge-info.html.

For now, if you want to steer clear of bugs and odd behaviors in automatic merging, the CollabNet article recommends that you stick to these simple best practices:

- For short-term feature branches, follow the simple procedure described throughout 第 4.3 节 "基本合并".

- For long-lived release branches (as described in 第 4.8 节 "常用分支模式"), perform merges only on the root of the branch, not on subdirectories.

- Never merge into working copies with a mixture of working revision numbers, or with "switched" subdirectories (as described next in 第 4.5 节 "使用分支"). A merge target should be a working copy which represents a *single* location in the repository at a single point in time.

- Don't ever edit the `svn:mergeinfo` property directly; use **svn merge** with the `--record-only` option to effect a desired change to the metadata (as demonstrated in 第 4.4.5 节 "阻塞修改").

- Always make sure you have complete read access to all of your merge sources, and that your target working copy has no sparse directories.

4.5. 使用分支

The **svn switch** command transforms an existing working copy to reflect a different branch. While this command isn't strictly necessary for working with branches, it provides a nice shortcut. In our earlier example, after creating your private branch, you checked out a fresh working copy of the new repository directory. Instead, you can simply ask Subversion to change your working copy of `/calc/trunk` to mirror the new branch location:

```
$ cd calc

$ svn info -| grep URL
URL: http://svn.example.com/repos/calc/trunk

$ svn switch ^/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
```

"Switching" a working copy that has no local modifications to a different branch results in the working copy looking just as it would if you'd done a fresh checkout of the directory. It's usually more efficient to use this command, because often branches differ by only a small degree. The server sends only the minimal set of changes necessary to make your working copy reflect the branch directory.

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副本到分支的HEAD。

Of course, most projects are more complicated than our calc example, and contain multiple subdirectories. Subversion users often follow a specific algorithm when using branches:

1. 拷贝整个项目的"trunk"目录到一个新的分支目录。

2. 只是转换工作副本的**部分**目录到分支。

In other words, if a user knows that the branch work needs to happen on only a specific subdirectory, she uses **svn switch** to move only that subdirectory to the branch. (Or sometimes users will switch just a single working file to the branch!) That way, the user can continue to receive normal "trunk" updates to most of her working copy, but the switched portions will remain immune (unless someone commits a change to her branch). This feature adds a whole new dimension to the concept of a "mixed working copy"—not only can working copies contain a mixture of working revisions, but they can also contain a mixture of repository locations as well.

If your working copy contains a number of switched subtrees from different repository locations, it continues to function as normal. When you update, you'll receive patches to each subtree as appropriate. When you commit, your local changes will still be applied as a single, atomic change to the repository.

Note that while it's okay for your working copy to reflect a mixture of repository locations, these locations must all be within the *same* repository. Subversion repositories aren't yet able to communicate with one another; that feature is planned for the future. [6]

---

[6]You *can*, however, use **svn switch** with the --relocate option if the URL of your server changes and you don't want to abandon an existing working copy. See svn switch for more information and an example.

> **切换和更新**
>
> Have you noticed that the output of **svn switch** and **svn update** looks the same? The switch command is actually a superset of the update command.
>
> When you run **svn update**, you're asking the repository to compare two trees. The repository does so, and then sends a description of the differences back to the client. The only difference between **svn switch** and **svn update** is that the latter command always compares two identical repository paths.
>
> That is, if your working copy is a mirror of `/calc/trunk`, **svn update** will automatically compare your working copy of `/calc/trunk` to `/calc/trunk` in the `HEAD` revision. If you're switching your working copy to a branch, **svn switch** will compare your working copy of `/calc/trunk` to some *other* branch directory in the `HEAD` revision.
>
> In other words, an update moves your working copy through time. A switch moves your working copy through time *and* space.

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，任何工作副本的已经完成的本地修改会被保存。

提示

你是否发现你做出了复杂的修改(在/trunk的工作副本)，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/newbranch \
       --m -"Create branch -'newbranch'."
Committed revision 353.
$ svn switch ^/branches/newbranch
At revision 353.
```

The **svn switch** command, like **svn update**, preserves your local edits. At this point, your working copy is now a reflection of the newly created branch, and your next **svn commit** invocation will send your changes there.

4.6. 标签

Another common version control concept is a *tag*. A tag is just a "snapshot" of a project in time. In Subversion, this idea already seems to be everywhere. Each repository revision is exactly that—a snapshot of the filesystem after each commit.

However, people often want to give more human-friendly names to tags, such as `release-1.0`. And they want to make snapshots of smaller subdirectories of the filesystem. After all, it's not so easy to remember that release 1.0 of a piece of software is a particular subdirectory of revision 4822.

4.6.1. 建立简单标签

Once again, **svn copy** comes to the rescue. If you want to create a snapshot of `/calc/trunk` exactly as it looks in the `HEAD` revision, make a copy of it:

```
$ 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 902.
```

This example assumes that a `/calc/tags` directory already exists. (If it doesn't, you can create it using **svn mkdir**.) After the copy completes, the new `release-1.0` directory is forever a snapshot of how the `/trunk` directory looked in the `HEAD` revision at the time you made the copy. Of course, you might want to be more precise about exactly which revision you copy, in case somebody else may have committed changes to the project when you weren't looking. So if you know that revision 901 of `/calc/trunk` is exactly the snapshot you want, you can specify it by passing `-r 901` to the **svn copy** command.

But wait a moment: isn't this tag creation procedure the same procedure we used to create a branch? Yes, in fact, it is. In Subversion, there's no difference between a tag and a branch. Both are just ordinary directories that are created by copying. Just as with branches, the only reason a copied directory is a "tag" is because *humans* have decided to treat it that way: as long as nobody ever commits to the directory, it forever remains a snapshot. If people start committing to it, it becomes a branch.

If you are administering a repository, there are two approaches you can take to managing tags. The first approach is "hands off": as a matter of project policy, decide where your tags will live, and make sure all users know how to treat the directories they copy. (That is, make sure they know not to commit to them.) The second approach is more paranoid: you can use one of the access control scripts provided with Subversion to prevent anyone from doing anything but creating new copies in the tags area (see 第 6 章 *服务配置*). The paranoid approach, however, isn't usually necessary. If a user accidentally commits a change to a tag directory, you can simply undo the change as discussed in the previous section. This is version control, after all!

4.6.2. 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。

For example, pretend your project is much larger than our `calc` example: suppose it contains a number of subdirectories and many more files. In the course of your work, you may decide that you need to create a working copy that is designed to have specific features and bug fixes. You can accomplish this by selectively backdating files or directories to particular revisions (using **svn update** with the `-r` option liberally), by switching files and directories to particular branches (making use of **svn switch**), or even just by making a bunch of local changes. When you're done, your working copy is a hodgepodge of repository locations from different revisions. But after testing, you know it's the precise combination of data you need to tag.

Time to make a snapshot. Copying one URL to another won't work here. In this case, you want to make a snapshot of your exact working copy arrangement and store it in the repository. Luckily, **svn copy** actually has four different uses (which you can read about in 第 9 章 *Subversion 完全参考*), including the ability to copy a working copy tree to the repository:

```
$ ls
my-working-copy/

$ svn copy my-working-copy \
          http://svn.example.com/repos/calc/tags/mytag \
          --m -"Tag my existing working copy state."

Committed revision 940.
```

Now there is a new directory in the repository, `/calc/tags/mytag`, which is an exact snapshot of your working copy—mixed revisions, URLs, local changes, and all.

Other users have found interesting uses for this feature. Sometimes there are situations where you have a bunch of local changes made to your working copy, and you'd like a collaborator to see them. Instead of running **svn diff** and sending a patch file (which won't capture directory, symlink, or property changes), you can use **svn copy** to "upload" your working copy to a private area of the repository. Your collaborator can then either check out a verbatim copy of your working copy or use **svn merge** to receive your exact changes.

While this is a nice method for uploading a quick snapshot of your working copy, note that this is *not* a good way to initially create a branch. Branch creation should be an event unto itself, and this method conflates the creation of a branch with extra changes to files, all within a single revision. This makes it very difficult (later on) to identify a single revision number as a branch point.

4.7. 维护分支

You may have noticed by now that Subversion is extremely flexible. Because it implements branches and tags with the same underlying mechanism (directory copies), and because branches and tags appear in normal filesystem space, many people find Subversion intimidating. It's almost *too* flexible. In this section, we'll offer some suggestions for arranging and managing your data over time.

4.7.1. 版本库布局

There are some standard, recommended ways to organize a repository. Most people create a `trunk` directory to hold the "main line" of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, often people create these top-level directories:

```
/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(见第 5.2.1 节 "规划你的版本库结构" 关于 "项目根目录")：

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, the **svn move** command can move or rename them however you wish. Switching from one layout to another is just a matter of issuing a series of server-side moves; if you don't like the way things are organized in the repository, just juggle the directories around.

Remember, though, that while moving directories may be easy to do, you need to be considerate of your users as well. Your juggling can be disorienting to users with existing working copies. If a user has a working copy of a particular repository directory, your **svn move** operation might remove the path from the latest revision. When the user next runs **svn update**, she will be told that her working copy represents a path that no longer exists, and the user will be forced to **svn switch** to the new location.

4.7.2. 数据的生命周期

Another nice feature of Subversion's model is that branches and tags can have finite lifetimes, just like any other versioned item. For example, suppose you eventually finish all your work on your personal branch of the `calc` project. After merging all of your changes back into `/calc/trunk`, there's no need for your private branch directory to stick around anymore:

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-
branch \
          --m -"Removing obsolete branch of calc project."

Committed revision 375.
```

And now your branch is gone. Of course, it's not really gone: the directory is simply missing from the `HEAD` revision, no longer distracting anyone. If you use **svn checkout**, **svn switch**, or **svn list** to examine an earlier revision, you'll still be able to see your old branch.

If browsing your deleted directory isn't enough, you can always bring it back. Resurrecting data is very easy in Subversion. If there's a deleted directory (or file) that you'd like to bring back into `HEAD`, simply use **svn copy** to copy it from the old revision:

```
$ svn copy http://svn.example.com/repos/calc/branches/my-calc-
branch@374 \
          http://svn.example.com/repos/calc/branches/my-calc-
branch \
          --m -"Restore my-calc-branch."

Committed revision 376.
```

In our example, your personal branch had a relatively short lifetime: you may have created it to fix a bug or implement a new feature. When your task is done, so is the branch. In software development, though, it's also common to have two "main" branches running side by side for very long periods. For example, suppose it's time to release a stable version of the calc project to the public, and you know it's going to take a couple of months to shake bugs out of the software. You don't want people to add new features to the project, but you don't want to tell all developers to stop programming either. So instead, you create a "stable" branch of the software that won't change much:

```
$ 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.
```

And now developers are free to continue adding cutting-edge (or experimental) features to /calc/trunk, and you can declare a project policy that only bug fixes are to be committed to /calc/branches/stable-1.0. That is, as people continue to work on the trunk, a human selectively ports bug fixes over to the stable branch. Even after the stable branch has shipped, you'll probably continue to maintain the branch for a long time—that is, as long as you continue to support that release for customers. We'll discuss this more in the next section.

4.8. 常用分支模式

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。

Version control is most often used for software development, so here's a quick peek at two of the most common branching/merging patterns used by teams of programmers. If you're not using Subversion for software development, feel free to skip this section. If you're a software developer using version control for the first time, pay close attention, as these patterns are often considered best practices by experienced folk. These processes aren't specific to Subversion; they're applicable to any version control system. Still, it may help to see them described in Subversion terms.

4.8.1. 发布分支

Most software has a typical life cycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality assurance teams take time to test supposedly stable versions of the software. New work cannot halt while

the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bug fix without having to wait for a major new release.

这是版本控制可以帮助你的。典型的过程如下：

1. *Developers commit all new work to the trunk.* Day-to-day changes are committed to `/trunk`: new features, bug fixes, and so on.

2. *The trunk is copied to a "release" branch.* When the team thinks the software is ready for release (say, a 1.0 release), `/trunk` might be copied to `/branches/1.0`.

3. *Teams continue to work in parallel.* One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on `/trunk`. If bugs are discovered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is "frozen" for final testing right before a release.

4. *The branch is tagged and released.* When testing is complete, `/branches/1.0` is copied to `/tags/1.0.0` as a reference snapshot. The tag is packaged and released to customers.

5. *The branch is maintained over time.* While work continues on `/trunk` for version 2.0, bug fixes continue to be ported from `/trunk` to `/branches/1.0`. When enough bug fixes have accumulated, management may decide to do a 1.0.1 release: `/branches/1.0` is copied to `/tags/1.0.1`, and the tag is packaged and released.

This entire process repeats as the software matures: when the 2.0 work is complete, a new 2.0 release branch is created, tested, tagged, and eventually released. After some years, the repository ends up with a number of release branches in "maintenance" mode, and a number of tags representing final shipped versions.

4.8.2. 特性分支

A *feature branch* is the sort of branch that's been the dominant example in this chapter (the one you've been working on while Sally continues to work on `/trunk`). It's a temporary branch created to work on a complex change without interfering with the stability of `/trunk`. Unlike release branches (which may need to be supported forever), feature branches are born, used for a while, merged back to the trunk, and then ultimately deleted. They have a finite span of usefulness.

Again, project policies vary widely concerning exactly when it's appropriate to create a feature branch. Some projects never use feature branches at all: commits to `/trunk` are a free-for-all. The advantage to this system is that it's simple—nobody needs to learn about branching or merging. The disadvantage is that the trunk code is often unstable or unusable. Other projects use branches to an extreme: no change is *ever* committed to the trunk directly. Even the most trivial changes are created on a short-lived branch, carefully reviewed, and merged to the trunk. Then the branch is deleted. This system guarantees an exceptionally stable and usable trunk at all times, but at the cost of tremendous process overhead.

Most projects take a middle-of-the-road approach. They commonly insist that `/trunk` compile and pass regression tests at all times. A feature branch is required only when a change requires a large number of destabilizing commits. A good rule of thumb is to ask this question: if

the developer worked for days in isolation and then committed the large change all at once (so that /trunk were never destabilized), would it be too large a change to review? If the answer to that question is "yes," the change should be developed on a feature branch. As the developer commits incremental changes to the branch, they can be easily reviewed by peers.

Finally, there's the issue of how to best keep a feature branch in "sync" with the trunk as work progresses. As we mentioned earlier, there's a great risk to working on a branch for weeks or months; trunk changes may continue to pour in, to the point where the two lines of development differ so greatly that it may become a nightmare trying to merge the branch back to the trunk.

This situation is best avoided by regularly merging trunk changes to the branch. Make up a policy: once a week, merge the last week's worth of trunk changes to the branch.

At some point, you'll be ready to merge the "synchronized" feature branch back to the trunk. To do this, begin by doing a final merge of the latest trunk changes to the branch. When that's done, the latest versions of branch and trunk will be absolutely identical except for your branch changes. You would then merge back with the `--reintegrate` option:

```
$ cd trunk-working-copy

$ svn update
At revision 1910.

$ svn merge ---reintegrate ^/branches/mybranch
--- Merging differences between repository URLs into -'.':
U    real.c
U    integer.c
A    newdirectory
A    newdirectory/newfile
 U   -.
…
```

Another way of thinking about this pattern is that your weekly sync of trunk to branch is analogous to running **svn update** in a working copy, while the final merge step is analogous to running **svn commit** from a working copy. After all, what else *is* a working copy but a very shallow private branch? It's a branch that's capable of storing only one change at a time.

4.9. 供方分支

As is especially the case when developing software, the data that you maintain under version control is often closely related to, or perhaps dependent upon, someone else's data. Generally, the needs of your project will dictate that you stay as up to date as possible with the data provided by that external entity without sacrificing the stability of your own project. This scenario plays itself out all the time—anywhere that the information generated by one group of people has a direct effect on that which is generated by another group.

For example, software developers might be working on an application that makes use of a third-party library. Subversion has just such a relationship with the Apache Portable Runtime

(APR) library (see 第 8.3.1 节 "Apache 可移植运行库"). The Subversion source code depends on the APR library for all its portability needs. In earlier stages of Subversion's development, the project closely tracked APR's changing API, always sticking to the "bleeding edge" of the library's code churn. Now that both APR and Subversion have matured, Subversion attempts to synchronize with APR's library API only at well-tested, stable release points.

Now, if your project depends on someone else's information, you could attempt to synchronize that information with your own in several ways. Most painfully, you could issue oral or written instructions to all the contributors of your project, telling them to make sure they have the specific versions of that third-party information that your project needs. If the third-party information is maintained in a Subversion repository, you could also use Subversion's externals definitions to effectively "pin down" specific versions of that information to some location in your own working copy directory (see 第 3.8 节 "外部定义").

But sometimes you want to maintain custom modifications to third-party code in your own version control system. Returning to the software development example, programmers might need to make modifications to that third-party library for their own purposes. These modifications might include new functionality or bug fixes, maintained internally only until they become part of an official release of the third-party library. Or the changes might never be relayed back to the library maintainers, existing solely as custom tweaks to make the library further suit the needs of the software developers.

Now you face an interesting situation. Your project could house its custom modifications to the third-party data in some disjointed fashion, such as using patch files or full-fledged alternative versions of files and directories. But these quickly become maintenance headaches, requiring some mechanism by which to apply your custom changes to the third-party code and necessitating regeneration of those changes with each successive version of the third-party code that you track.

The solution to this problem is to use *vendor branches*. A vendor branch is a directory tree in your own version control system that contains information provided by a third-party entity, or vendor. Each version of the vendor's data that you decide to absorb into your project is called a *vendor drop*.

Vendor branches provide two benefits. First, by storing the currently supported vendor drop in your own version control system, you ensure that the members of your project never need to question whether they have the right version of the vendor's data. They simply receive that correct version as part of their regular working copy updates. Second, because the data lives in your own Subversion repository, you can store your custom changes to it in-place—you have no more need of an automated (or worse, manual) method for swapping in your customizations.

4.9.1. 常规的供方分支管理过程

Managing vendor branches generally works like this: first, you create a top-level directory (such as `/vendor`) to hold the vendor branches. Then you import the third-party code into a subdirectory of that top-level directory. You then copy that subdirectory into your main development branch (e.g., `/trunk`) at the appropriate location. You always make your local changes in the main development branch. With each new release of the code you are tracking, you

bring it into the vendor branch and merge the changes into /trunk, resolving whatever conflicts occur between your local changes and the upstream changes.

An example will help to clarify this algorithm. We'll use a scenario where your development team is creating a calculator program that links against a third-party complex number arithmetic library, libcomplex. We'll begin with the initial creation of the vendor branch and the import of the first vendor drop. We'll call our vendor branch directory libcomplex, and our code drops will go into a subdirectory of our vendor branch called current. And since **svn import** creates all the intermediate parent directories it needs, we can actually accomplish both of these steps with a single command:

```
$ svn import -/path/to/libcomplex-1.0 \
             http://svn.example.com/repos/vendor/libcomplex/
current \
             --m -'importing initial 1.0 vendor drop'
…
```

We now have the current version of the libcomplex source code in /vendor/libcomplex/current. Now, we tag that version (see 第 4.6 节 "标签") and then copy it into the main development branch. Our copy will create a new directory called libcomplex in our existing calc project directory. It is in this copied version of the vendor data that we will make our customizations:

```
$ 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'
…
```

We check out our project's main branch—which now includes a copy of the first vendor drop—and we get to work customizing the libcomplex code. Before we know it, our modified version of libcomplex is now completely integrated into our calculator program. [7]

A few weeks later, the developers of libcomplex release a new version of their library—version 1.1—which contains some features and functionality that we really want. We'd like to upgrade to this new version, but without losing the customizations we made to the existing version. What we essentially would like to do is to replace our current baseline version of libcomplex 1.0 with a copy of libcomplex 1.1, and then re-apply the custom modifications we previously made to that library to the new version. But we actually approach the problem from the other direction, applying the changes made to libcomplex between versions 1.0 and 1.1 to our modified copy of it.

[7]And is entirely bug-free, of course!

To perform this upgrade, we check out a copy of our vendor branch and replace the code in the `current` directory with the new libcomplex 1.1 source code. We quite literally copy new files on top of existing files, perhaps exploding the libcomplex 1.1 release tarball atop our existing files and directories. The goal here is to make our `current` directory contain only the libcomplex 1.1 code and to ensure that all that code is under version control. Oh, and we want to do this with as little version control history disturbance as possible.

After replacing the 1.0 code with 1.1 code, **svn status** will show files with local modifications as well as, perhaps, some unversioned files. If we did what we were supposed to do, the unversioned files are only those new files introduced in the 1.1 release of libcomplex—we run **svn add** on those to get them under version control. If the 1.1 code no longer has certain files that were in the 1.0 tree, it may be hard to notice them; you'd have to compare the two trees with some external tool and then **svn delete** any files present in 1.0 but not in 1.1. (Although it might also be just fine to let these same files live on in unused obscurity!) Finally, once our `current` working copy contains only the libcomplex 1.1 code, we commit the changes we made to get it looking that way.

Our `current` branch now contains the new vendor drop. We tag the new version as 1.1 (in the same way we previously tagged the version 1.0 vendor drop), and then merge the differences between the tag of the previous version and the new current version into our main development branch:

```
$ cd working-copies/calc
$ svn merge ^/vendor/libcomplex/1.0       \
            ^/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'
…
```

In the trivial use case, the new version of our third-party tool would look, from a files-and-directories point of view, just like the previous version. None of the libcomplex source files would have been deleted, renamed, or moved to different locations—the new version would contain only textual modifications against the previous one. In a perfect world, our modifications would apply cleanly to the new version of the library, with absolutely no complications or conflicts.

But things aren't always that simple, and in fact it is quite common for source files to get moved around between releases of software. This complicates the process of ensuring that our modifications are still valid for the new version of code, and things can quickly degrade into a situation where we have to manually re-create our customizations in the new version. Once Subversion knows about the history of a given source file—including all its previous locations—the process of merging in the new version of the library is pretty simple. But we are responsible for telling Subversion how the source file layout changed from vendor drop to vendor drop.

4.9.2. svn_load_dirs.pl

Vendor drops that contain more than a few deletes, additions, and moves complicate the process of upgrading to each successive version of the third-party data. So Subversion supplies

the **svn_load_dirs.pl** script to assist with this process. This script automates the importing steps we mentioned in the general vendor branch management procedure to make sure mistakes are minimized. You will still be responsible for using the merge commands to merge the new versions of the third-party data into your main development branch, but **svn_load_dirs.pl** can help you more quickly and easily arrive at that stage.

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。

- It takes care of complicated series of operations between which Subversion requires an intermediate commit—such as before renaming a file or directory twice.

- 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 takes three mandatory arguments. The first argument is the URL to the base Subversion directory to work in. This argument is followed by the URL—relative to the first argument—into which the current vendor drop will be imported. Finally, the third argument is the local directory to import. Using our previous example, a typical run of **svn_load_dirs.pl** might look like this:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                         \
                   -/path/to/libcomplex-1.1
…
```

You can indicate that you'd like **svn_load_dirs.pl** to tag the new vendor drop by passing the -t command-line option and specifying a tag name. This tag is another URL relative to the first program argument.

```
$ svn_load_dirs.pl --t
libcomplex-1.1                                                     \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                         \
                   -/path/to/libcomplex-1.1
…
```

When you run **svn_load_dirs.pl**, it examines the contents of your existing "current" vendor drop and compares them with the proposed new vendor drop. In the trivial case, no files will be in one version and not the other, and the script will perform the new import without incident. If, however, there are discrepancies in the file layouts between versions, **svn_load_dirs.pl** will ask you how to resolve those differences. For example, you will have the opportunity to tell the script that you know that the file math.c in version 1.0 of libcomplex was renamed to arithmetic.c in libcomplex 1.1. Any discrepancies not explained by moves are treated as regular additions and deletions.

The script also accepts a separate configuration file for setting properties on files and directories matching a regular expression that are *added* to the repository. This configuration file is specified to **svn_load_dirs.pl** using the `-p` command-line option. Each line of the configuration file is a whitespace-delimited set of two or four values: a Perl-style regular expression against which to match the added path, a control keyword (either `break` or `cont`), and then optionally a property name and value.

```
\.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
```

For each added path, the configured property changes whose regular expression matches the path are applied in order, unless the control specification is `break` (which means that no more property changes should be applied to that path). If the control specification is `cont`—an abbreviation for `continue`—matching will continue with the next line of the configuration file.

Any whitespace in the regular expression, property name, or property value must be surrounded by either single or double quotes. You can escape quotes that are not used for wrapping whitespace by preceding them with a backslash (`\`) character. The backslash escapes only quotes when parsing the configuration file, so do not protect any other characters beyond what is necessary for the regular expression.

4.10. 总结

We covered a lot of ground in this chapter. We discussed the concepts of tags and branches and demonstrated how Subversion implements these concepts by copying directories with the **svn copy** command. We showed how to use **svn merge** to copy changes from one branch to another or roll back bad changes. We went over the use of **svn switch** to create mixed-location working copies. And we talked about how one might manage the organization and lifetimes of branches in a repository.

Remember the Subversion mantra: branches and tags are cheap. So don't be afraid to use them when needed!

As a helpful reminder of all the operations we discussed, here is handy reference table you can consult as you begin to make use of branches.

表 **4.1.** 分支与合并命令

动作	Command
创建一个分支或标签	**svn copy *URL1 URL2***
切换工作副本到分支或标签	**svn switch *URL***
将分支与主干同步	**svn merge *trunkURL*; svn commit**
参见合并历史或适当的修改集	**svn mergeinfo target --from-source=*URL***

动作	Command
合并分支到主干	**svn merge --reintegrate** *branchURL*; **svn commit**
复制特定的修改	**svn merge -c** *REV URL*; **svn commit**
合并一个范围的修改	**svn merge -r** *REV1:REV2 URL*; **svn commit**
让自动合并跳过一个修改	**svn merge -c** *REV* **--record-only** *URL*; **svn commit**
预览合并	**svn merge** *URL* **--dry-run**
丢弃合并结果	**svn revert -R .**
从历史复活某些事物	**svn copy** *URL@REV localPATH*
撤销已经提交的修改	**svn merge -c** *-REV URL*; **svn commit**
感知合并的检查历史	**svn log -g; svn blame -g**
从工作副本创建一个标签	**svn copy .** *tagURL*
重新整理分支或标签	**svn mv** *URL1 URL2*
删除分支或标签	**svn rm** *URL*

第 5 章 版本库管理

The Subversion repository is the central storehouse of all your versioned data. As such, it becomes an obvious candidate for all the love and attention an administrator can offer. While the repository is generally a low-maintenance item, it is important to understand how to properly configure and care for it so that potential problems are avoided, and so actual problems are safely resolved.

In this chapter, we'll discuss how to create and configure a Subversion repository. We'll also talk about repository maintenance, providing examples of how and when to use the **svnlook** and **svnadmin** tools provided with Subversion. We'll address some common questions and mistakes and give some suggestions on how to arrange the data in the repository.

If you plan to access a Subversion repository only in the role of a user whose data is under version control (i.e., via a Subversion client), you can skip this chapter altogether. However, if you are, or wish to become, a Subversion repository administrator, [1] this chapter is for you.

5.1. Subversion 版本库的定义

Before jumping into the broader topic of repository administration, let's further define what a repository is. How does it look? How does it feel? Does it take its tea hot or iced, sweetened, and with lemon? As an administrator, you'll be expected to understand the composition of a repository both from a literal, OS-level perspective—how a repository looks and acts with respect to non-Subversion tools—and from a logical perspective—dealing with how data is represented *inside* the repository.

Seen through the eyes of a typical file browser application (such as Windows Explorer) or command-line based filesystem navigation tools, the Subversion repository is just another directory full of stuff. There are some subdirectories with human-readable configuration files in them, some subdirectories with some not-so-human-readable data files, and so on. As in other areas of the Subversion design, modularity is given high regard, and hierarchical organization is preferred to cluttered chaos. So a shallow glance into a typical repository from a nuts-and-bolts perspective is sufficient to reveal the basic components of the repository:

```
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

Here's a quick fly-by overview of what exactly you're seeing in this directory listing. (Don't get bogged down in the terminology—detailed coverage of these components exists elsewhere in this and other chapters.)

conf
　　保存配置文件的目录

dav
　　A directory provided to `mod_dav_svn` for its private housekeeping data

---

[1] 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副本数据之外的神秘领域感兴趣的人。

db
   The data store for all of your versioned data

format
   A file that contains a single integer that indicates the version number of the repository layout

hooks
   A directory full of hook script templates (and hook scripts themselves, once you've installed some)

locks
   A directory for Subversion's repository lock files, used for tracking accessors to the repository

README.txt
   A file whose contents merely inform its readers that they are looking at a Subversion repository

Of course, when accessed via the Subversion libraries, this otherwise unremarkable collection of files and directories suddenly becomes an implementation of a virtual, versioned filesystem, complete with customizable event triggers. This filesystem has its own notions of directories and files, very similar to the notions of such things held by real filesystems (such as NTFS, FAT32, ext3, etc.). But this is a special filesystem—it hangs these directories and files from revisions, keeping all the changes you've ever made to them safely stored and forever accessible. This is where the entirety of your versioned data lives.

5.2. 版本库开发策略

Due largely to the simplicity of the overall design of the Subversion repository and the technologies on which it relies, creating and configuring a repository are fairly straightforward tasks. There are a few preliminary decisions you'll want to make, but the actual work involved in any given setup of a Subversion repository is pretty basic, tending toward mindless repetition if you find yourself setting up multiples of these things.

Some things you'll want to consider beforehand, though, are:

• 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。

5.2.1. 规划你的版本库结构

While Subversion allows you to move around versioned files and directories without any loss of information, and even provides ways of moving whole sets of versioned history from one repository to another, doing so can greatly disrupt the workflow of those who access the

repository often and come to expect things to be at certain locations. So before creating a new repository, try to peer into the future a bit; plan ahead before placing your data under version control. By conscientiously "laying out" your repository or repositories and their versioned contents ahead of time, you can prevent many future headaches.

Let's assume that as repository administrator, you will be responsible for supporting the version control system for several projects. Your first decision is whether to use a single repository for multiple projects, or to give each project its own repository, or some compromise of these two.

There are benefits to using a single repository for multiple projects, most obviously the lack of duplicated maintenance. A single repository means that there is one set of hook programs, one thing to routinely back up, one thing to dump and load if Subversion releases an incompatible new version, and so on. Also, you can move data between projects easily, without losing any historical versioning information.

The downside of using a single repository is that different projects may have different requirements in terms of the repository event triggers, such as needing to send commit notification emails to different mailing lists, or having different definitions about what does and does not constitute a legitimate commit. These aren't insurmountable problems, of course—it just means that all of your hook scripts have to be sensitive to the layout of your repository rather than assuming that the whole repository is associated with a single group of people. Also, remember that Subversion uses repository-global revision numbers. While those numbers don't have any particular magical powers, some folks still don't like the fact that even though no changes have been made to their project lately, the youngest revision number for the repository keeps climbing because other projects are actively adding new revisions. [2]

A middle-ground approach can be taken, too. For example, projects can be grouped by how well they relate to each other. You might have a few repositories with a handful of projects in each repository. That way, projects that are likely to want to share data can do so easily, and as new revisions are added to the repository, at least the developers know that those new revisions are at least remotely related to everyone who uses that repository.

After deciding how to organize your projects with respect to repositories, you'll probably want to think about directory hierarchies within the repositories themselves. Because Subversion uses regular directory copies for branching and tagging (see 第 4 章 分支与合并), the Subversion community recommends that you choose a repository location for each *project root*—the "topmost" directory that contains data related to that project—and then create three subdirectories beneath that root: `trunk`, meaning the directory under which the main project development occurs; `branches`, which is a directory in which to create various named branches of the main development line; and `tags`, which is a collection of tree snapshots that are created, and perhaps destroyed, but never changed. [3]

For example, your repository might look like this:

```
/
    calc/
        trunk/
        tags/
        branches/
    calendar/
        trunk/
        tags/
        branches/
    spreadsheet/
        trunk/
        tags/
        branches/
    …
```

Note that it doesn't matter where in your repository each project root is. If you have only one project per repository, the logical place to put each project root is at the root of that project's respective repository. If you have multiple projects, you might want to arrange them in groups inside the repository, perhaps putting projects with similar goals or shared code in the same subdirectory, or maybe just grouping them alphabetically. Such an arrangement might look like this:

```
/
    utils/
        calc/
            trunk/
            tags/
            branches/
        calendar/
            trunk/
            tags/
            branches/
        …
    office/
        spreadsheet/
            trunk/
            tags/
            branches/
        …
```

Lay out your repository in whatever way you see fit. Subversion does not expect or enforce a particular layout—in its eyes, a directory is a directory is a directory. Ultimately, you should choose the repository arrangement that meets the needs of the people who work on the projects that live there.

In the name of full disclosure, though, we'll mention another very common layout. In this layout, the `trunk`, `tags`, and `branches` directories live in the root directory of your repository, and your projects are in subdirectories beneath those, like so:

```
/
    trunk/
        calc/
        calendar/
        spreadsheet/
        …
    tags/
        calc/
        calendar/
        spreadsheet/
        …
    branches/
        calc/
        calendar/
        spreadsheet/
        …
```

There's nothing particularly incorrect about such a layout, but it may or may not seem as intuitive for your users. Especially in large, multiproject situations with many users, those users may tend to be familiar with only one or two of the projects in the repository. But the projects-as-branch-siblings approach tends to deemphasize project individuality and focus on the entire set of projects as a single entity. That's a social issue, though. We like our originally suggested arrangement for purely practical reasons—it's easier to ask about (or modify, or migrate elsewhere) the entire history of a single project when there's a single repository path that holds the entire history—past, present, tagged, and branched—for that project and that project alone.

5.2.2. 决定在哪里与如何部署你的版本库

Before creating your Subversion repository, an obvious question you'll need to answer is where the thing is going to live. This is strongly connected to myriad other questions involving how the repository will be accessed (via a Subversion server or directly), by whom (users behind your corporate firewall or the whole world out on the open Internet), what other services you'll be providing around Subversion (repository browsing interfaces, email-based commit notification, etc.), your data backup strategy, and so on.

We cover server choice and configuration in 第 6 章 服务配置, but the point we'd like to briefly make here is simply that the answers to some of these other questions might have implications that force your hand when deciding where your repository will live. For example, certain deployment scenarios might require accessing the repository via a remote filesystem from multiple computers, in which case (as you'll read in the next section) your choice of a repository backend data store turns out not to be a choice at all because only one of the available backends will work in this scenario.

Addressing each possible way to deploy Subversion is both impossible and outside the scope of this book. We simply encourage you to evaluate your options using these pages and other sources as your reference material and to plan ahead.

5.2.3. 选择数据存储格式

Subversion provides two options for the type of underlying data store—often referred to as "the backend" or, somewhat confusingly, "the (versioned) filesystem"—that each repository uses. One type of data store keeps everything in a Berkeley DB (or BDB) database environment; repositories that use this type are often referred to as being "BDB-backed." The other type stores data in ordinary flat files, using a custom format. Subversion developers have adopted the habit of referring to this latter data storage mechanism as *FSFS* [4] —a versioned filesystem implementation that uses the native OS filesystem directly—rather than via a database library or some other abstraction layer—to store data.

表 5.1 "版本库数据存储对照表" gives a comparative overview of Berkeley DB and FSFS repositories.

表 5.1. 版本库数据存储对照表

分类	特性	Berkeley DB	FSFS
可靠性	数据完整性	When properly deployed, extremely reliable; Berkeley DB 4.4 brings auto-recovery	Older versions had some rarely demonstrated, but data-destroying bugs
	对操作中断的敏感	Very; crashes and permission problems can leave the database "wedged," requiring journaled recovery procedures	十分迟钝
可用性	可只读加载	否	是
	存储平台无关	否	是
	可从网络文件系统访问	通常，不	是
	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	umask 问题的解决方案
伸缩性	版本库磁盘使用情况	Larger (especially if logfiles aren't purged)	较小
	修订版本树的数量	数据库；没有问题	Some older native filesystems don't scale well with thousands of entries in a single directory
	有很多文件的目录	较慢	较快

---

[4]Often pronounced "fuzz-fuzz," if Jack Repenning has anything to say about it. (This book, however, assumes that the reader is thinking "eff-ess-eff-ess.")

分类	特性	Berkeley DB	FSFS
性能	检出最新的代码	没有任何有意义的差异	没有任何有意义的差异
	大的提交	Slower overall, but cost is amortized across the lifetime of the commit	Faster overall, but finalization delay may cause client timeouts

There are advantages and disadvantages to each of these two backend types. Neither of them is more "official" than the other, though the newer FSFS is the default data store as of Subversion 1.2. Both are reliable enough to trust with your versioned data. But as you can see in 表 5.1 "版本库数据存储对照表", the FSFS backend provides quite a bit more flexibility in terms of its supported deployment scenarios. More flexibility means you have to work a little harder to find ways to deploy it incorrectly. Those reasons—plus the fact that not using Berkeley DB means there's one fewer component in the system—largely explain why today almost everyone uses the FSFS backend when creating new repositories.

Fortunately, most programs that access Subversion repositories are blissfully ignorant of which backend data store is in use. And you aren't even necessarily stuck with your first choice of a data store—in the event that you change your mind later, Subversion provides ways of migrating your repository's data into another repository that uses a different backend data store. We talk more about that later in this chapter.

下面的小节提供了数据存储类型更加详细的介绍。

5.2.3.1. Berkeley DB

When the initial design phase of Subversion was in progress, the developers decided to use Berkeley DB for a variety of reasons, including its open source license, transaction support, reliability, performance, API simplicity, thread safety, support for cursors, and so on.

Berkeley DB provides real transaction support—perhaps its most powerful feature. Multiple processes accessing your Subversion repositories don't have to worry about accidentally clobbering each other's data. The isolation provided by the transaction system is such that for any given operation, the Subversion repository code sees a static view of the database—not a database that is constantly changing at the hand of some other process—and can make decisions based on that view. If the decision made happens to conflict with what another process is doing, the entire operation is rolled back as though it never happened, and Subversion gracefully retries the operation against a new, updated (and yet still static) view of the database.

Another great feature of Berkeley DB is *hot backups*—the ability to back up the database environment without taking it "offline." We'll discuss how to back up your repository later in this chapter (in 第 5.4.8 节 "版本库备份"), but the benefits of being able to make fully functional copies of your repositories without any downtime should be obvious.

Berkeley DB is also a very reliable database system when properly used. Subversion uses Berkeley DB's logging facilities, which means that the database first writes to on-disk logfiles a description of any modifications it is about to make, and then makes the modification itself. This is to ensure that if anything goes wrong, the database system can back up to a previous

checkpoint—a location in the logfiles known not to be corrupt—and replay transactions until the data is restored to a usable state. See 第 5.4.3 节 "管理磁盘空间" later in this chapter for more about Berkeley DB logfiles.

But every rose has its thorn, and so we must note some known limitations of Berkeley DB. First, Berkeley DB environments are not portable. You cannot simply copy a Subversion repository that was created on a Unix system onto a Windows system and expect it to work. While much of the Berkeley DB database format is architecture-independent, other aspects of the environment are not. Second, Subversion uses Berkeley DB in a way that will not operate on Windows 95/98 systems—if you need to house a BDB-backed repository on a Windows machine, stick with Windows 2000 or later.

While Berkeley DB promises to behave correctly on network shares that meet a particular set of specifications, [5] most networked filesystem types and appliances do *not* actually meet those requirements. And in no case can you allow a BDB-backed repository that resides on a network share to be accessed by multiple clients of that share at once (which quite often is the whole point of having the repository live on a network share in the first place).

警告

If you attempt to use Berkeley DB on a noncompliant remote filesystem, the results are unpredictable—you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted. You should strongly consider using the FSFS data store for repositories that need to live on a network share.

Finally, because Berkeley DB is a library linked directly into Subversion, it's more sensitive to interruptions than a typical relational database system. Most SQL systems, for example, have a dedicated server process that mediates all access to tables. If a program accessing the database crashes for some reason, the database daemon notices the lost connection and cleans up any mess left behind. And because the database daemon is the only process accessing the tables, applications don't need to worry about permission conflicts. These things are not the case with Berkeley DB, however. Subversion (and programs using Subversion libraries) access the database tables directly, which means that a program crash can leave the database in a temporarily inconsistent, inaccessible state. When this happens, an administrator needs to ask Berkeley DB to restore to a checkpoint, which is a bit of an annoyance. Other things can cause a repository to "wedge" besides crashed processes, such as programs conflicting over ownership and permissions on the database files.

注意

Berkeley DB 4.4 brings (to Subversion 1.4 and later) the ability for Subversion to automatically and transparently recover Berkeley DB environments in need of such recovery. When a Subversion process attaches to a repository's Berkeley DB environment, it uses some process accounting mechanisms to detect any unclean disconnections by previous processes, performs any necessary recovery, and then continues on as though nothing happened. This doesn't completely eliminate instances of repository wedging, but it does drastically reduce the amount of human interaction required to recover from them.

---

[5]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。

So while a Berkeley DB repository is quite fast and scalable, it's best used by a single server process running as one user—such as Apache's **httpd** or **svnserve** (see 第 6 章 服务配置)—rather than accessing it as many different users via `file://` or `svn+ssh://` URLs. If you're accessing a Berkeley DB repository directly as multiple users, be sure to read 第 6.6 节 "支持多种版本库访问方法" later in this chapter.

5.2.3.2. FSFS

In mid-2004, a second type of repository storage system—one that doesn't use a database at all—came into being. An FSFS repository stores the changes associated with a revision in a single file, and so all of a repository's revisions can be found in a single subdirectory full of numbered files. Transactions are created in separate subdirectories as individual files. When complete, the transaction file is renamed and moved into the revisions directory, thus guaranteeing that commits are atomic. And because a revision file is permanent and unchanging, the repository also can be backed up while "hot," just like a BDB-backed repository.

Revision files and shards

FSFS repositories contain files that describe the changes made in a single revision, and files that contain the revision properties associated with a single revision. Repositories created in versions of Subversion prior to 1.5 keep these files in two directories—one for each type of file. As new revisions are committed to the repository, Subversion drops more files into these two directories—over time, the number of these files in each directory can grow to be quite large. This has been observed to cause performance problems on certain network-based filesystems.

Subversion 1.5 creates FSFS-backed repositories using a slightly modified layout in which the contents of these two directories are *sharded*, or scattered across several subdirectories. This can greatly reduce the time it takes the system to locate any one of these files, and therefore increases the overall performance of Subversion when reading from the repository.

Subversion 1.6 takes the sharded layout one step further, allowing administrators to optionally *pack* each of their repository shards up into a single multi-revision file. This can have both performance and disk usage benefits. See 第 5.4.3.4 节 "打包 FSFS 文件系统" for more information.

The FSFS revision files describe a revision's directory structure, file contents, and deltas against files in other revision trees. Unlike a Berkeley DB database, this storage format is portable across different operating systems and isn't sensitive to CPU architecture. Because no journaling or shared-memory files are being used, the repository can be safely accessed over a network filesystem and examined in a read-only environment. The lack of database overhead also means the overall repository size is a bit smaller.

FSFS has different performance characteristics, too. When committing a directory with a huge number of files, FSFS is able to more quickly append directory entries. On the other hand, FSFS writes the latest version of a file as a delta against an earlier version, which means that checking out the latest tree is a bit slower than fetching the full-texts stored in a Berkeley DB HEAD

revision. FSFS also has a longer delay when finalizing a commit, which could in extreme cases cause clients to time out while waiting for a response.

The most important distinction, however, is FSFS's imperviousness to wedging when something goes wrong. If a process using a Berkeley DB database runs into a permissions problem or suddenly crashes, the database can be left in an unusable state until an administrator recovers it. If the same scenarios happen to a process using an FSFS repository, the repository isn't affected at all. At worst, some transaction data is left behind.

The only real argument against FSFS is its relative immaturity compared to Berkeley DB. Unlike Berkeley DB, which has years of history, its own dedicated development team, and, now, Oracle's mighty name attached to it, [6] FSFS is a newer bit of engineering. Prior to Subversion 1.4, it was still shaking out some pretty serious data integrity bugs, which, while triggered in only very rare cases, nonetheless did occur. That said, FSFS has quickly become the backend of choice for some of the largest public and private Subversion repositories, and it promises a lower barrier to entry for Subversion across the board.

5.3. 创建和配置你的版本库

Earlier in this chapter (in 第 5.2 节 "版本库开发策略"), we looked at some of the important decisions that should be made before creating and configuring your Subversion repository. Now, we finally get to get our hands dirty! In this section, we'll see how to actually create a Subversion repository and configure it to perform custom actions when special repository events occur.

5.3.1. 创建版本库

Subversion repository creation is an incredibly simple task. The **svnadmin** utility that comes with Subversion provides a subcommand (**svnadmin create**) for doing just that.

```
$ # Create a repository
$ svnadmin create -/var/svn/repos
$
```

This creates a new repository in the directory `/var/svn/repos`, and with the default filesystem data store. Prior to Subversion 1.2, the default was to use Berkeley DB; the default is now FSFS. You can explicitly choose the filesystem type using the `--fs-type` argument, which accepts as a parameter either `fsfs` or `bdb`.

```
$ # Create an FSFS-backed repository
$ svnadmin create ---fs-type fsfs -/var/svn/repos
$

# Create a Berkeley-DB-backed repository
```

---

[6] Oracle在2006情人节购买了Sleepycat和它的旗舰软件Berkeley DB。

```
$ svnadmin create ---fs-type bdb -/var/svn/repos
$
```

运行这个命令之后，你有了一个Subversion版本库。

提示

The path argument to **svnadmin** is just a regular filesystem path and not a URL like the **svn** client program uses when referring to repositories. Both **svnadmin** and **svnlook** are considered server-side utilities—they are used on the machine where the repository resides to examine or modify aspects of the repository, and are in fact unable to perform tasks across a network. A common mistake made by Subversion newcomers is trying to pass URLs (even "local" `file://` ones) to these two programs.

Present in the `db/` subdirectory of your repository is the implementation of the versioned filesystem. Your new repository's versioned filesystem begins life at revision 0, which is defined to consist of nothing but the top-level root (/) directory. Initially, revision 0 also has a single revision property, `svn:date`, set to the time at which the repository was created.

现在你有了一个版本库，可以用户化了。

警告

While some parts of a Subversion repository—such as the configuration files and hook scripts—are meant to be examined and modified manually, you shouldn't (and shouldn't need to) tamper with the other parts of the repository "by hand." The **svnadmin** tool should be sufficient for any changes necessary to your repository, or you can look to third-party tools (such as Berkeley DB's tool suite) for tweaking relevant subsections of the repository. Do *not* attempt manual manipulation of your version control history by poking and prodding around in your repository's data store files!

5.3.2. 实现版本库钩子

A *hook* is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Some hooks (the so-called "pre hooks") run in advance of a repository operation and provide a means by which to both report what is about to happen and prevent it from happening at all. Other hooks (the "post hooks") run after the completion of a repository event and are useful for performing tasks that examine—but don't modify—the repository. Each hook is handed enough information to tell what that event is (or was), the specific repository changes proposed (or completed), and the username of the person who triggered the event.

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
```

```
$
```

There is one template for each hook that the Subversion repository supports; by examining the contents of those template scripts, you can see what triggers each script to run and what data is passed to that script. Also present in many of these templates are examples of how one might use that script, in conjunction with other Subversion-supplied programs, to perform common useful tasks. To actually install a working hook, you need only place some executable program or script into the `repos/hooks` directory, which can be executed as the name (such as **start-commit** or **post-commit**) of the hook.

On Unix platforms, this means supplying a script or program (which could be a shell script, a Python program, a compiled C binary, or any number of other things) named exactly like the name of the hook. Of course, the template files are present for more than just informational purposes —the easiest way to install a hook on Unix platforms is to simply copy the appropriate template file to a new file that lacks the `.tmpl` extension, customize the hook's contents, and ensure that the script is executable. Windows, however, uses file extensions to determine whether a program is executable, so you would need to supply a program whose basename is the name of the hook and whose extension is one of the special extensions recognized by Windows for executable programs, such as `.exe` for programs and `.bat` for batch files.

提示

For security reasons, the Subversion repository executes hook programs with an empty environment—that is, no environment variables are set at all, not even `$PATH` (or `%PATH%`, under Windows). Because of this, many administrators are baffled when their hook program runs fine by hand, but doesn't work when run by Subversion. Be sure to explicitly set any necessary environment variables in your hook program and/or use absolute paths to programs.

Subversion executes hooks as the same user who owns the process that is accessing the Subversion repository. In most cases, the repository is being accessed via a Subversion server, so this user is the same user as whom the server runs on the system. The hooks themselves will need to be configured with OS-level permissions that allow that user to execute them. Also, this means that any programs or files (including the Subversion repository) accessed directly or indirectly by the hook will be accessed as the same user. In other words, be alert to potential permission-related problems that could prevent the hook from performing the tasks it is designed to perform.

There are serveral hooks implemented by the Subversion repository, and you can get details about each of them in 第 9.11 节 "版本库钩子". As a repository administrator, you'll need to decide which hooks you wish to implement (by way of providing an appropriately named and permissioned hook program), and how. When you make this decision, keep in mind the big picture of how your repository is deployed. For example, if you are using server configuration to determine which users are permitted to commit changes to your repository, you don't need to do this sort of access control via the hook system.

There is no shortage of Subversion hook programs and scripts that are freely available either from the Subversion community itself or elsewhere. These scripts cover a wide range of utility— basic access control, policy adherence checking, issue tracker integration, email- or syndication-based commit notification, and beyond. Or, if you wish to write your own, see 第 8 章 嵌入 *Subversion*.

> **警告**
>
> While hook scripts can do almost anything, there is one dimension in which hook script authors should show restraint: do *not* modify a commit transaction using hook scripts. While it might be tempting to use hook scripts to automatically correct errors, shortcomings, or policy violations present in the files being committed, doing so can cause problems. Subversion keeps client-side caches of certain bits of repository data, and if you change a commit transaction in this way, those caches become indetectably stale. This inconsistency can lead to surprising and unexpected behavior. Instead of modifying the transaction, you should simply *validate* the transaction in the `pre-commit` hook and reject the commit if it does not meet the desired requirements. As a bonus, your users will learn the value of careful, compliance-minded work habits.

5.3.3. Berkeley DB 配置

A Berkeley DB environment is an encapsulation of one or more databases, logfiles, region files, and configuration files. The Berkeley DB environment has its own set of default configuration values for things such as the number of database locks allowed to be taken out at any given time, the maximum size of the journaling logfiles, and so on. Subversion's filesystem logic additionally chooses default values for some of the Berkeley DB configuration options. However, sometimes your particular repository, with its unique collection of data and access patterns, might require a different set of configuration option values.

The producers of Berkeley DB understand that different applications and database environments have different requirements, so they have provided a mechanism for overriding at runtime many of the configuration values for the Berkeley DB environment. BDB checks for the presence of a file named `DB_CONFIG` in the environment directory (namely, the repository's `db` subdirectory), and parses the options found in that file. Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so that you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file and makes no direct use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is at odds with what Subversion might expect. Also, changes made to `DB_CONFIG` won't take effect until you recover the database environment (using **svnadmin recover**).

5.3.4. FSFS Configuration

As of Subversion 1.6, FSFS filesystems have several configurable parameters which an administrator can use to fine-tune the performance or disk usage of their repositories. You can find these options—and the documentation for them—in the `db/fsfs.conf` file in the repository.

5.4. 版本库维护

Maintaining a Subversion repository can be daunting, mostly due to the complexities inherent in systems that have a database backend. Doing the task well is all about knowing

the tools—what they are, when to use them, and how. This section will introduce you to the repository administration tools provided by Subversion and discuss how to wield them to accomplish tasks such as repository data migration, upgrades, backups, and cleanups.

5.4.1. 管理员的工具箱

Subversion provides a handful of utilities useful for creating, inspecting, modifying, and repairing your repository. Let's look more closely at each of those tools. Afterward, we'll briefly examine some of the utilities included in the Berkeley DB distribution that provide functionality specific to your repository's database backend not otherwise provided by Subversion's own tools.

5.4.1.1. svnadmin

The **svnadmin** program is the repository administrator's best friend. Besides providing the ability to create Subversion repositories, this program allows you to perform several maintenance operations on those repositories. The syntax of **svnadmin** is similar to that of other Subversion command-line programs:

```
$ 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
…
```

Previously in this chapter (in 第 5.3.1 节 "创建版本库"), we were introduced to the **svnadmin create** subcommand. Most of the other **svnadmin** subcommands we will cover later in this chapter. And you can consult 第 9.2 节 "svnadmin" for a full rundown of subcommands and what each of them offers.

5.4.1.2. svnlook

svnlook is a tool provided by Subversion for examining the various revisions and *transactions* (which are revisions in the making) in a repository. No part of this program attempts to change the repository. **svnlook** is typically used by the repository hooks for reporting the changes that are about to be committed (in the case of the **pre-commit** hook) or that were just committed (in the case of the **post-commit** hook) to the repository. A repository administrator may use this tool for diagnostic purposes.

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.
…
```

Most of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision` (`-r`) and `--transaction` (`-t`) options to specify which revision or transaction, respectively, to examine. In the absence of both the `--revision` (`-r`) and `--transaction` (`-t`) options, **svnlook** will examine the youngest (or `HEAD`) revision in the repository. So the following two commands do exactly the same thing when 19 is the youngest revision in the repository located at `/var/svn/repos`:

```
$ svnlook info -/var/svn/repos
$ svnlook info -/var/svn/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 -/var/svn/repos
19
$
```

注意

Keep in mind that the only transactions you can browse are uncommitted ones. Most repositories will have no such transactions because transactions are usually either committed (in which case, you should access them as revision with the `--revision` (`-r`) option) or aborted and removed.

Output from **svnlook** is designed to be both human- and machine-parsable. Take, as an example, the output of the **svnlook info** subcommand:

```
$ svnlook info -/var/svn/repos
sally
2002-11-04 09:29:13 --0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
```

```
$
```

svnlook info的输出包含如下的内容，按照给定的顺序：

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

This output is human-readable, meaning items such as the datestamp are displayed using a textual representation instead of something more obscure (such as the number of nanoseconds since the Tastee Freez guy drove by). But the output is also machine-parsable—because the log message can contain multiple lines and be unbounded in length, **svnlook** provides the length of that message before the message itself. This allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

svnlook can perform a variety of other queries: displaying subsets of bits of information we've mentioned previously, recursively listing versioned directory trees, reporting which paths were modified in a given revision or transaction, showing textual and property differences made to files and directories, and so on. See 第 9.3 节 "svnlook" for a full reference of **svnlook**'s features.

5.4.1.3. svndumpfilter

While it won't be the most commonly used tool at the administrator's disposal, **svndumpfilter** provides a very particular brand of useful functionality—the ability to quickly and easily modify streams of Subversion repository history data by acting as a path-based filter.

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)
```

There are only two interesting subcommands: **svndumpfilter exclude** and **svndumpfilter include**. They allow you to make the choice between implicit or explicit inclusion of paths in the stream. You can learn more about these subcommands and **svndumpfilter**'s unique purpose later in this chapter, in 第 5.4.6 节 "过滤版本库历史".

5.4.1.4. svnsync

The **svnsync** program, which is new to the 1.4 release of Subversion, provides all the functionality required for maintaining a read-only mirror of a Subversion repository. The program really has one job—to transfer one repository's versioned history into another repository. And while there are few ways to do that, its primary strength is that it can operate remotely—the "source" and "sink" [7] repositories may be on different computers from each other and from **svnsync** itself.

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)
$
```

We talk more about replicating repositories with **svnsync** later in this chapter (see 第 5.4.7 节 "版本库复制").

5.4.1.5. fsfs-reshard.py

While not an official member of the Subversion toolchain, the **fsfs-reshard.py** script (found in the `tools/server-side` directory of the Subversion source distribution) is a useful performance tuning tool for administrators of FSFS-backed Subversion repositories. As described in the sidebar Revision files and shards, FSFS repositories use individual files to house information about each revision. Sometimes these files all live in a single directory; sometimes they are sharded across many directories. But the neat thing is that the number of directories used used to house these files is configurable. That's where **fsfs-reshard.py** comes in.

fsfs-reshard.py reshuffles the repository's file structure into a new arrangement that reflects the requested number of sharding subdirectories and updates the repository configuration to preserve this change. This is especially useful for converting an older Subversion repository into the new Subversion 1.5 sharded layout (which Subversion will not automatically do for you) or for fine-tuning an already sharded repository.

5.4.1.6. Berkeley DB 工具

If you're using a Berkeley DB repository, all of your versioned filesystem's structure and data live in a set of database tables within the `db/` subdirectory of your repository. This subdirectory is

---

[7]或者是，"sync"？

a regular Berkeley DB environment directory and can therefore be used in conjunction with any of the Berkeley database tools, typically provided as part of the Berkeley DB distribution.

For day-to-day Subversion use, these tools are unnecessary. Most of the functionality typically needed for Subversion repositories has been duplicated in the **svnadmin** tool. For example, **svnadmin list-unused-dblogs** and **svnadmin list-dblogs** perform a subset of what is provided by the Berkeley **db_archive** utility, and **svnadmin recover** reflects the common use cases of the **db_recover** utility.

However, there are still a few Berkeley DB utilities that you might find useful. The **db_dump** and **db_load** programs write and read, respectively, a custom file format that describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system. As we describe later in this chapter, you can also use **svnadmin dump** and **svnadmin load** for similar purposes, but **db_dump** and **db_load** can do certain jobs just as well and much faster. They can also be useful if the experienced Berkeley DB hacker needs to do in-place tweaking of the data in a BDB-backed repository for some reason, which is something Subversion's utilities won't allow. Also, the **db_stat** utility can provide useful information about the status of your Berkeley DB environment, including detailed statistics about the locking and storage subsystems.

For more information on the Berkeley DB tool chain, visit the documentation section of the Berkeley DB section of Oracle's web site, located at http://www.oracle.com/technology/documentation/berkeley-db/db/.

5.4.2. 修正提交消息

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` hook; see 第 5.3.2 节 "实现版本库钩子") to accept changes to this log message after the commit is finished, the user can "fix" her log message remotely using **svn propset** (see svn propset). However, because of the potential to lose information forever, Subversion repositories are not, by default, configured to allow changes to unversioned properties—except by an administrator.

If a log message needs to be changed by an administrator, this can be done using **svnadmin setlog**. This command changes the log message (the `svn:log` property) on a given revision of a repository, reading the new value from a provided file.

```
$ echo -"Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt --r 388
```

The **svnadmin setlog** command, by default, is still bound by the same protections against modifying unversioned properties as a remote client is—the `pre-` and `post-revprop-change` hooks are still triggered, and therefore must be set up to accept changes of this nature. But an administrator can get around these protections by passing the `--bypass-hooks` option to the **svnadmin setlog** command.

警告

Remember, though, that by bypassing the hooks, you are likely avoiding such things as email notifications of property changes, backup systems that track unversioned property changes, and so on. In other words, be very careful about what you are changing, and how you change it.

5.4.3. 管理磁盘空间

While the cost of storage has dropped incredibly in the past few years, disk usage is still a valid concern for administrators seeking to version large amounts of data. Every bit of version history information stored in the live repository needs to be backed up elsewhere, perhaps multiple times as part of rotating backup schedules. It is useful to know what pieces of Subversion's repository data need to remain on the live site, which need to be backed up, and which can be safely removed.

5.4.3.1. 让 Subversion 节约磁盘空间

To keep the repository small, Subversion uses *deltification* (or deltified storage) within the repository itself. Deltification involves encoding the representation of a chunk of data as a collection of differences against some other chunk of data. If the two pieces of data are very similar, this deltification results in storage savings for the deltified chunk—rather than taking up space equal to the size of the original data, it takes up only enough space to say, "I look just like this other piece of data over here, except for the following couple of changes." The result is that most of the repository data that tends to be bulky—namely, the contents of versioned files—is stored at a much smaller size than the original full-text representation of that data. And for repositories created with Subversion 1.4 or later, the space savings are even better—now those full-text representations of file contents are themselves compressed.

注意

Because all of the data that is subject to deltification in a BDB-backed repository is stored in a single Berkeley DB database file, reducing the size of the stored values will not immediately reduce the size of the database file itself. Berkeley DB will, however, keep internal records of unused areas of the database file and consume those areas first before growing the size of the database file. So while deltification doesn't produce immediate space savings, it can drastically slow future growth of the database.

5.4.3.2. 删除终止的事务

Though they are uncommon, there are circumstances in which a Subversion commit process might fail, leaving behind in the repository the remnants of the revision-to-be that wasn't—an uncommitted transaction and all the file and directory changes associated with it. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure occurred in the middle of an operation. Regardless of the reason, dead transactions can happen. They don't do any real harm, other than consuming disk space. A fastidious administrator may nonetheless wish to remove them.

可以使用**svnadmin lstxns**命令列出当前的事务名。

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

Each item in the resultant output can then be used with **svnlook** (and its `--transaction` (`-t`) option) to determine who created the transaction, when it was created, what types of changes were made in the transaction—information that is helpful in determining whether the transaction is a safe candidate for removal! If you do indeed want to remove a transaction, its name can be passed to **svnadmin rmtxns**, which will perform the cleanup of the transaction. In fact, **svnadmin rmtxns** can take its input directly from the output of **svnadmin lstxns**!

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`
$
```

If you use these two subcommands like this, you should consider making your repository temporarily inaccessible to clients. That way, no one can begin a legitimate transaction before you start your cleanup. 例 5.1 "txn-info.sh (报告异常事务)" contains a bit of shell-scripting that can quickly generate information about each outstanding transaction in your repository.

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
```

The output of the script is basically a concatenation of several chunks of **svnlook info** output (see 第 5.4.1.2 节 "svnlook") and will look something like this:

```
$ 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
$
```

A long-abandoned transaction usually represents some sort of failed or interrupted commit. A transaction's datestamp can provide interesting information—for example, how likely is it that an operation begun nine months ago is still active?

In short, transaction cleanup decisions need not be made unwisely. Various sources of information—including Apache's error and access logs, Subversion's operational logs, Subversion revision history, and so on—can be employed in the decision-making process. And of course, an administrator can often simply communicate with a seemingly dead transaction's owner (via email, e.g.) to verify that the transaction is, in fact, in a zombie state.

5.4.3.3. 删除不使用的 Berkeley DB 日志文件

Until recently, the largest offender of disk space usage with respect to BDB-backed Subversion repositories were the logfiles in which Berkeley DB performs its prewrites before modifying the actual database files. These files capture all the actions taken along the route of changing the database from one state to another—while the database files, at any given time, reflect a particular state, the logfiles contain all of the many changes along the way *between* states. Thus, they can grow and accumulate quite rapidly.

Fortunately, beginning with the 4.2 release of Berkeley DB, the database environment has the ability to remove its own unused logfiles automatically. Any repositories created using **svnadmin** when compiled against Berkeley DB version 4.2 or later will be configured for this automatic logfile removal. If you don't want this feature enabled, simply pass the `--bdb-log-keep` option to the **svnadmin create** command. If you forget to do this or change your mind at a later time, simply edit the `DB_CONFIG` file found in your repository's `db` directory, comment out the line that contains the `set_flags DB_LOG_AUTOREMOVE` directive, and then run **svnadmin recover** on your repository to force the configuration changes to take effect. See 第 5.3.3 节 "Berkeley DB 配置" for more information about database configuration.

Without some sort of automatic logfile removal in place, logfiles will accumulate as you use your repository. This is actually somewhat of a feature of the database system—you should be able to recreate your entire database using nothing but the logfiles, so these files can be useful for catastrophic database recovery. But typically, you'll want to archive the logfiles that are no longer

in use by Berkeley DB, and then remove them from disk to conserve space. Use the **svnadmin list-unused-dblogs** command to list the unused logfiles:

```
$ svnadmin list-unused-dblogs -/var/svn/repos
/var/svn/repos/log.0000000031
/var/svn/repos/log.0000000032
/var/svn/repos/log.0000000033
…
$ rm `svnadmin list-unused-dblogs -/var/svn/repos`
## disk space reclaimed!
```

> **警告**
>
> BDB-backed repositories whose logfiles are used as part of a backup or disaster recovery plan should *not* make use of the logfile autoremoval feature. Reconstruction of a repository's data from logfiles can only be accomplished only when *all* the logfiles are available. If some of the logfiles are removed from disk before the backup system has a chance to copy them elsewhere, the incomplete set of backed-up logfiles is essentially useless.

5.4.3.4. 打包 **FSFS** 文件系统

As described in the sidebar Revision files and shards, FSFS-backed Subversion repositories create, by default, a new on-disk file for each revision added to the repository. Having thousands of these files present on your Subversion server—even when housed in separate shard directories—can lead to inefficiencies.

The first problem is that the operating system has to reference many different files over a short period of time. This leads to inefficient use of disk caches and, as a result, more time spent seeking across large disks. Because of this, Subversion pays a performance penalty when accessing your versioned data.

The second problem is a bit more subtle. Because of the ways that most filesystems allocate disk space, each file claims more space on the disk than it actually uses. The amount of extra space required to house a single file can average anywhere from 2 to 16 kilobytes *per file*, depending on the underlying operating system filesystem in use. This penalty translates directly into a per-revision disk usage penalty for FSFS-backed repositories. The effect is most pronounced in repositories which have many small revisions, since the overhead involved in storing the revision file quickly outgrows the size of the actual data being stored.

To solve these problems, Subversion 1.6 introduced the **svnadmin pack** command. By concatenating all the files of a completed shard into a single "pack" file and then removing the original per-revision files, **svnadmin pack** reduces the file count within a given shard down to just a single file. In doing so, it aids filesystem caches and reduces (to one) the number of times a file storage overhead penalty is paid.

Subversion can pack existing sharded repositories which have been upgraded to the 1.6 filesystem format (see svnadmin upgrade). To do so, just run **svnadmin pack** on the repository:

```
$ svnadmin pack -/var/svn/repos
Packing shard 0...done.
Packing shard 1...done.
Packing shard 2...done.
…
Packing shard 34...done.
Packing shard 35...done.
Packing shard 36...done.
$
```

Because the packing process obtains the required locks before doing its work, you can run it on live repositories, or even as part of a post-commit hook. Repacking packed shards is legal, but will have no effect on the disk usage of the repository.

svnadmin pack has no effect on BDB-backed Subversion repositories.

5.4.4. Berkeley DB 恢复

As mentioned in 第 5.2.3.1 节 "Berkeley DB", a Berkeley DB repository can sometimes be left in a frozen state if not closed properly. When this happens, an administrator needs to rewind the database back into a consistent state. This is unique to BDB-backed repositories, though— if you are using FSFS-backed ones instead, this won't apply to you. And for those of you using Subversion 1.4 with Berkeley DB 4.4 or later, you should find that Subversion has become much more resilient in these types of situations. Still, wedged Berkeley DB repositories do occur, and an administrator needs to know how to safely deal with this circumstance.

To protect the data in your repository, Berkeley DB uses a locking mechanism. This mechanism ensures that portions of the database are not simultaneously modified by multiple database accessors, and that each process sees the data in the correct state when that data is being read from the database. When a process needs to change something in the database, it first checks for the existence of a lock on the target data. If the data is not locked, the process locks the data, makes the change it wants to make, and then unlocks the data. Other processes are forced to wait until that lock is removed before they are permitted to continue accessing that section of the database. (This has nothing to do with the locks that you, as a user, can apply to versioned files within the repository; we try to clear up the confusion caused by this terminology collision in the sidebar 锁定的三种含义.)

In the course of using your Subversion repository, fatal errors or interruptions can prevent a process from having the chance to remove the locks it has placed in the database. The result is that the backend database system gets "wedged." When this happens, any attempts to access the repository hang indefinitely (since each new accessor is waiting for a lock to go away—which isn't going to happen).

If this happens to your repository, don't panic. The Berkeley DB filesystem takes advantage of database transactions, checkpoints, and prewrite journaling to ensure that only the most catastrophic of events [8] can permanently destroy a database environment. A sufficiently paranoid

---

[8]For example, hard drive + huge electromagnet = disaster.

repository administrator will have made off-site backups of the repository data in some fashion, but don't head off to the tape backup storage closet just yet.

然后，使用下面的方法试着"恢复"你的版本库：

1. Make sure no processes are accessing (or attempting to access) the repository. For networked repositories, this also means shutting down the Apache HTTP Server or svnserve daemon.

2. Become the user who owns and manages the repository. This is important, as recovering a repository while running as the wrong user can tweak the permissions of the repository's files in such a way that your repository will still be inaccessible even after it is "unwedged."

3. Run the command **svnadmin recover /var/svn/repos**. You should see output such as this:

```
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 19.
```

此命令可能需要数分钟才能完成。

4. 重新启动服务进程。

This procedure fixes almost every case of repository wedging. Make sure that you run this command as the user that owns and manages the database, not just as root. Part of the recovery process might involve re-creating from scratch various database files (shared memory regions, e.g.). Recovering as root will create those files such that they are owned by root, which means that even after you restore connectivity to your repository, regular users will be unable to access it.

If the previous procedure, for some reason, does not successfully unwedge your repository, you should do two things. First, move your broken repository directory aside (perhaps by renaming it to something like repos.BROKEN) and then restore your latest backup of it. Then, send an email to the Subversion users mailing list (at <users@subversion.tigris.org>) describing your problem in detail. Data integrity is an extremely high priority to the Subversion developers.

5.4.5. 版本库数据的移植

A Subversion filesystem has its data spread throughout files in the repository, in a fashion generally understood by (and of interest to) only the Subversion developers themselves. However, circumstances may arise that call for all, or some subset, of that data to be copied or moved into another repository.

Subversion provides such functionality by way of *repository dump streams*. A repository dump stream (often referred to as a "dump file" when stored as a file on disk) is a portable, flat

file format that describes the various revisions in your repository—what was changed, by whom, when, and so on. This dump stream is the primary mechanism used to marshal versioned history —in whole or in part, with or without modification—between repositories. And Subversion provides the tools necessary for creating and loading these dump streams: the **svnadmin dump** and **svnadmin load** subcommands, respectively.

警告

While the Subversion repository dump format contains human-readable portions and a familiar structure (it resembles an RFC 822 format, the same type of format used for most email), it is *not* a plain-text file format. It is a binary file format, highly sensitive to meddling. For example, many text editors will corrupt the file by automatically converting line endings.

There are many reasons for dumping and loading Subversion repository data. Early in Subversion's life, the most common reason was due to the evolution of Subversion itself. As Subversion matured, there were times when changes made to the backend database schema caused compatibility issues with previous versions of the repository, so users had to dump their repository data using the previous version of Subversion and load it into a freshly created repository with the new version of Subversion. Now, these types of schema changes haven't occurred since Subversion's 1.0 release, and the Subversion developers promise not to force users to dump and load their repositories when upgrading between minor versions (such as from 1.3 to 1.4) of Subversion. But there are still other reasons for dumping and loading, including re-deploying a Berkeley DB repository on a new OS or CPU architecture, switching between the Berkeley DB and FSFS backends, or (as we'll cover later in this chapter in 第 5.4.6 节 "过滤版本库历史") purging versioned data from repository history.

注意

The Subversion repository dump format describes versioned repository changes only. It will not carry any information about uncommitted transactions, user locks on filesystem paths, repository or server configuration customizations (including hook scripts), and so on.

Whatever your reason for migrating repository history, using the **svnadmin dump** and **svnadmin load** subcommands is straightforward. **svnadmin dump** will output a range of repository revisions that are formatted using Subversion's custom filesystem dump format. The dump format is printed to the standard output stream, while informative messages are printed to the standard error stream. This allows you to redirect the output stream to a file while watching the status output in your terminal window. For example:

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
```

```
* Dumped revision 25.
* Dumped revision 26.
```

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions. Note that **svnadmin dump** is reading revision trees from the repository just like any other "reader" process would (e.g., **svn checkout**), so it's safe to run this command at any time.

The other subcommand in the pair, **svnadmin load**, parses the standard input stream as a Subversion repository dump file and effectively replays those dumped revisions into the target repository for that operation. It also gives informative feedback, this time using the standard output stream:

```
$ 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) >>>
```

The result of a load is new revisions added to a repository—the same thing you get by making commits against that repository from a regular Subversion client. Just as in a commit, you can use hook programs to perform actions before and after each of the commits made during a load process. By passing the `--use-pre-commit-hook` and `--use-post-commit-hook` options to **svnadmin load**, you can instruct Subversion to execute the pre-commit and post-commit hook programs, respectively, for each loaded revision. You might use these, for example, to ensure that loaded revisions pass through the same validation steps that regular commits pass through. Of course, you should use these options with care—if your post-commit hook sends

emails to a mailing list for each new commit, you might not want to spew hundreds or thousands of commit emails in rapid succession at that list! You can read more about the use of hook scripts in 第 5.3.2 节 "实现版本库钩子".

Note that because **svnadmin** uses standard input and output streams for the repository dump and load processes, people who are feeling especially saucy can try things such as this (perhaps even using different versions of **svnadmin** on each side of the pipe):

```
$ svnadmin create newrepos
$ svnadmin dump oldrepos -| svnadmin load newrepos
```

By default, the dump file will be quite large—much larger than the repository itself. That's because by default every version of every file is expressed as a full text in the dump file. This is the fastest and simplest behavior, and it's nice if you're piping the dump data directly into some other process (such as a compression program, filtering program, or loading process). But if you're creating a dump file for longer-term storage, you'll likely want to save disk space by using the `--deltas` option. With this option, successive revisions of files will be output as compressed, binary differences—just as file revisions are stored in a repository. This option is slower, but it results in a dump file much closer in size to the original repository.

We mentioned previously that **svnadmin dump** outputs a range of revisions. Use the `--revision` (`-r`) option to specify a single revision, or a range of revisions, to dump. If you omit this option, all the existing repository revisions will be dumped.

```
$ svnadmin dump myrepos --r 23 > rev-23.dumpfile
$ svnadmin dump myrepos --r 100:200 > revs-100-200.dumpfile
```

As Subversion dumps each new revision, it outputs only enough information to allow a future loader to re-create that revision based on the previous one. In other words, for any given revision in the dump file, only the items that were changed in that revision will appear in the dump. The only exception to this rule is the first revision that is dumped with the current **svnadmin dump** command.

By default, Subversion will not express the first dumped revision as merely differences to be applied to the previous revision. For one thing, there is no previous revision in the dump file! And second, Subversion cannot know the state of the repository into which the dump data will be loaded (if it ever is). To ensure that the output of each execution of **svnadmin dump** is self-sufficient, the first dumped revision is, by default, a full representation of every directory, file, and property in that revision of the repository.

However, you can change this default behavior. If you add the `--incremental` option when you dump your repository, **svnadmin** will compare the first dumped revision against the previous revision in the repository—the same way it treats every other revision that gets dumped. It will then output the first revision exactly as it does the rest of the revisions in the dump range—mentioning only the changes that occurred in that revision. The benefit of this is that you can create several small dump files that can be loaded in succession, instead of one large one, like so:

```
$ 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
```

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran. Used like this, **svnadmin dump** can be one way to back up changes to your repository over time in case of a system crash or some other catastrophic event.

The dump format can also be used to merge the contents of several different repositories into a single repository. By using the `--parent-dir` option of **svnadmin load**, you can specify a new virtual root directory for the load process. That means if you have dump files for three repositories—say `calc-dumpfile`, `cal-dumpfile`, and `ss-dumpfile`—you can first create a new repository to hold them all:

```
$ svnadmin create -/var/svn/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:///var/svn/projects/calc \
      file:///var/svn/projects/calendar \
      file:///var/svn/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 -/var/svn/projects ---parent-dir calc < calc-
dumpfile
…
$ svnadmin load -/var/svn/projects ---parent-dir calendar < cal-
dumpfile
…
```

```
$ svnadmin load -/var/svn/projects ---parent-dir spreadsheet < ss-
dumpfile
…
$
```

We'll mention one final way to use the Subversion repository dump format—conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human-readable, it should be relatively easy to describe generic sets of changes—each of which should be treated as a new revision—using this file format. In fact, the **cvs2svn** utility (see 第 B.11 节 "迁移 CVS 版本库到 Subversion") uses the dump format to represent the contents of a CVS repository so that those contents can be copied into a Subversion repository.

5.4.6. 过滤版本库历史

Since Subversion stores your versioned history using, at the very least, binary differencing algorithms and data compression (optionally in a completely opaque database system), attempting manual tweaks is unwise if not quite difficult, and at any rate strongly discouraged. And once data has been stored in your repository, Subversion generally doesn't provide an easy way to remove that data. [9] But inevitably, there will be times when you would like to manipulate the history of your repository. You might need to strip out all instances of a file that was accidentally added to the repository (and shouldn't be there for whatever reason). [10] Or, perhaps you have multiple projects sharing a single repository, and you decide to split them up into their own repositories. To accomplish tasks such as these, administrators need a more manageable and malleable representation of the data in their repositories—the Subversion repository dump format.

As we described earlier in 第 5.4.5 节 "版本库数据的移植", the Subversion repository dump format is a human-readable representation of the changes that you've made to your versioned data over time. Use the **svnadmin dump** command to generate the dump data, and **svnadmin load** to populate a new repository with it. The great thing about the human-readability aspect of the dump format is that, if you aren't careless about it, you can manually inspect and modify it. Of course, the downside is that if you have three years' worth of repository activity encapsulated in what is likely to be a very large dump file, it could take you a long, long time to manually inspect and modify it.

That's where **svndumpfilter** becomes useful. This program acts as a path-based filter for repository dump streams. Simply give it either a list of paths you wish to keep or a list of paths you wish to not keep, and then pipe your repository dump data through this filter. The result will be a modified stream of dump data that contains only the versioned paths you (explicitly or implicitly) requested.

Let's look at a realistic example of how you might use this program. Earlier in this chapter (see 第 5.2.1 节 "规划你的版本库结构"), we discussed the process of deciding how to choose

---

[9]那就是你是用版本控制的原因，对吗？

[10]Conscious, cautious removal of certain bits of versioned data is actually supported by real use cases. That's why an "obliterate" feature has been one of the most highly requested Subversion features, and one which the Subversion developers hope to soon provide.

a layout for the data in your repositories—using one repository per project or combining them, arranging stuff within your repository, and so on. But sometimes after new revisions start flying in, you rethink your layout and would like to make some changes. A common change is the decision to move multiple projects that are sharing a single repository into separate repositories for each project.

Our imaginary repository contains three projects: `calc`, `calendar`, and `spreadsheet`. They have been living side-by-side in a layout like this:

```
/
    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 -/var/svn/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

Next, run that dump file through the filter, each time including only one of our top-level directories. This results in three new dump files:

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile > ss-dumpfile
…
$
```

At this point, you have to make a decision. Each of your dump files will create a valid repository, but will preserve the paths exactly as they were in the original repository. This means that even though you would have a repository solely for your `calc` project, that repository would

still have a top-level directory named `calc`. If you want your `trunk`, `tags`, and `branches` directories to live in the root of your repository, you might wish to edit your dump files, tweaking the `Node-path` and `Node-copyfrom-path` headers so that they no longer have that first `calc/` path component. Also, you'll want to remove the section of dump data that creates the `calc` directory. It will look something like the following:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```

> ## 警告
>
> If you do plan on manually editing the dump file to remove a top-level directory, make sure your editor is not set to automatically convert end-of-line characters to the native format (e.g., `\r\n` to `\n`), as the content will then not agree with the metadata. This will render the dump file useless.

All that remains now is to create your three new repositories, and load each dump file into the right repository, ignoring the UUID found in the dump stream:

```
$ svnadmin create calc
$ svnadmin load ---ignore-uuid 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 ---ignore-uuid 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 ---ignore-uuid spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path -: Makefile -... done.
     * adding path -: ss.c -... done.
…
$
```

Both of **svndumpfilter**'s subcommands accept options for deciding how to deal with "empty" revisions. If a given revision contains only changes to paths that were filtered out, that now-empty revision could be considered uninteresting or even unwanted. So to give the user control over what to do with those revisions, **svndumpfilter** provides the following command-line options:

`--drop-empty-revs`
> Do not generate empty revisions at all—just omit them.

`--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`
> If empty revisions are not dropped, preserve the revision properties (log message, author, date, custom properties, etc.) for those empty revisions. Otherwise, empty revisions will contain only the original datestamp, and a generated log message that indicates that this revision was emptied by **svndumpfilter**.

While **svndumpfilter** can be very useful and a huge timesaver, there are unfortunately a couple of gotchas. First, this utility is overly sensitive to path semantics. Pay attention to whether paths in your dump file are specified with or without leading slashes. You'll want to look at the `Node-path` and `Node-copyfrom-path` headers.

```
…
Node-path: spreadsheet/Makefile
…
```

If the paths have leading slashes, you should include leading slashes in the paths you pass to **svndumpfilter include** and **svndumpfilter exclude** (and if they don't, you shouldn't). Further, if your dump file has an inconsistent usage of leading slashes for some reason, [11] you should probably normalize those paths so that they all have, or all lack, leading slashes.

Also, copied paths can give you some trouble. Subversion supports copy operations in the repository, where a new path is created by copying some already existing path. It is possible that at some point in the lifetime of your repository, you might have copied a file or directory from some location that **svndumpfilter** is excluding, to a location that it is including. To make the dump data self-sufficient, **svndumpfilter** needs to still show the addition of the new path—including the contents of any files created by the copy—and not represent that addition as a copy from a source that won't exist in your filtered dump data stream. But because the Subversion repository dump format shows only what was changed in each revision, the contents of the copy source might not be readily available. If you suspect that you have any copies of this sort in your repository, you might want to rethink your set of included/excluded paths, perhaps including the paths that served as sources of your troublesome copy operations, too.

Finally, **svndumpfilter** takes path filtering quite literally. If you are trying to copy the history of a project rooted at `trunk/my-project` and move it into a repository of its own, you would, of course, use the **svndumpfilter include** command to keep all the changes in and under `trunk/my-project`. But the resultant dump file makes no assumptions about the repository into which you plan to load this data. Specifically, the dump data might begin with the revision that added the `trunk/my-project` directory, but it will *not* contain directives that would create the `trunk` directory itself (because `trunk` doesn't match the include filter). You'll need to make

---

[11] 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。

sure that any directories that the new dump stream expects to exist actually do exist in the target repository before trying to load the stream into that repository.

5.4.7. 版本库复制

There are several scenarios in which it is quite handy to have a Subversion repository whose version history is exactly the same as some other repository's. Perhaps the most obvious one is the maintenance of a simple backup repository, used when the primary repository has become inaccessible due to a hardware failure, network outage, or other such annoyance. Other scenarios include deploying mirror repositories to distribute heavy Subversion load across multiple servers, use as a soft-upgrade mechanism, and so on.

As of version 1.4, Subversion provides a program for managing scenarios such as these —**svnsync**. This works by essentially asking the Subversion server to "replay" revisions, one at a time. It then uses that revision information to mimic a commit of the same to another repository. Neither repository needs to be locally accessible to the machine on which **svnsync** is running —its parameters are repository URLs, and it does all its work through Subversion's Repository Access (RA) interfaces. All it requires is read access to the source repository and read/write access to the destination repository.

注意

When using **svnsync** against a remote source repository, the Subversion server for that repository must be running Subversion version 1.4 or later.

Assuming you already have a source repository that you'd like to mirror, the next thing you need is an empty target repository that will actually serve as that mirror. This target repository can use either of the available filesystem data-store backends (see 第 5.2.3 节 "选择数据存储格式"), but it must not yet have any version history in it. The protocol that **svnsync** uses to communicate revision information is highly sensitive to mismatches between the versioned histories contained in the source and target repositories. For this reason, while **svnsync** cannot *demand* that the target repository be read-only, [12] allowing the revision history in the target repository to change by any mechanism other than the mirroring process is a recipe for disaster.

警告

Do *not* modify a mirror repository in such a way as to cause its version history to deviate from that of the repository it mirrors. The only commits and revision property modifications that ever occur on that mirror repository should be those performed by the **svnsync** tool.

Another requirement of the target repository is that the **svnsync** process be allowed to modify revision properties. Because **svnsync** works within the framework of that repository's hook system, the default state of the repository (which is to disallow revision property changes; see pre-revprop-change) is insufficient. You'll need to explicitly implement the pre-revprop-change hook, and your script must allow **svnsync** to set and change revision properties. With those provisions in place, you are ready to start mirroring repository revisions.

---

[12] 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。

> **提示**
>
> 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。

Let's walk through the use of **svnsync** in a somewhat typical mirroring scenario. We'll pepper this discourse with practical recommendations, which you are free to disregard if they aren't required by or suitable for your environment.

As a service to the fine developers of our favorite version control system, we will be mirroring the public Subversion source code repository and exposing that mirror publicly on the Internet, hosted on a different machine than the one on which the original Subversion source code repository lives. This remote host has a global configuration that permits anonymous users to read the contents of repositories on the host, but requires users to authenticate to modify those repositories. (Please forgive us for glossing over the details of Subversion server configuration for the moment—those are covered thoroughly in 第 6 章 服务配置.) And for no other reason than that it makes for a more interesting example, we'll be driving the replication process from a third machine—the one that we currently find ourselves using.

First, we'll create the repository which will be our mirror. This and the next couple of steps do require shell access to the machine on which the mirror repository will live. Once the repository is all configured, though, we shouldn't need to touch it directly again.

```
$ ssh admin@svn.example.com \
      -"svnadmin create -/var/svn/svn-mirror"
admin@svn.example.com's password: ********
$
```

At this point, we have our repository, and due to our server's configuration, that repository is now "live" on the Internet. Now, because we don't want anything modifying the repository except our replication process, we need a way to distinguish that process from other would-be committers. To do so, we use a dedicated username for our process. Only commits and revision property modifications performed by the special username `syncuser` will be allowed.

We'll use the repository's hook system both to allow the replication process to do what it needs to do and to enforce that only it is doing those things. We accomplish this by implementing two of the repository event hooks—pre-revprop-change and start-commit. Our `pre-revprop-change` hook script is found in 例 5.2 "镜像版本库的 pre-revprop-change 钩子", and basically verifies that the user attempting the property changes is our `syncuser` user. If so, the change is allowed; otherwise, it is denied.

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
```

That covers revision property changes. Now we need to ensure that only the `syncuser` user is permitted to commit new revisions to the repository. We do this using a `start-commit` hook scripts such as the one in 例 5.3 "镜像版本库的 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
```

After installing our hook scripts and ensuring that they are executable by the Subversion server, we're finished with the setup of the mirror repository. Now, we get to actually do the mirroring.

The first thing we need to do with **svnsync** is to register in our target repository the fact that it will be a mirror of the source repository. We do this using the **svnsync initialize** subcommand. The URLs we provide point to the root directories of the target and source repositories, respectively. In Subversion 1.4, this is required—only full mirroring of repositories is permitted. In Subversion 1.5, though, you can use **svnsync** to mirror only some subtree of the repository, too.

```
$ svnsync help init
initialize (init): usage: svnsync initialize DEST_URL SOURCE_URL

Initialize a destination repository for synchronization from
another repository.
…
$ svnsync initialize http://svn.example.com/svn-mirror \
                      http://svn.collab.net/repos/svn \
```

```
                    ---sync-username syncuser ---sync-password
syncpass
Copied properties for revision 0.
$
```

Our target repository will now remember that it is a mirror of the public Subversion source code repository. Notice that we provided a username and password as arguments to **svnsync**—that was required by the pre-revprop-change hook on our mirror repository.

> ◆ **注意**
>
> In Subversion 1.4, the values given to **svnsync**'s `--username` and `--password` command-line options were used for authentication against both the source and destination repositories. This caused problems when a user's credentials weren't exactly the same for both repositories, especially when running in noninteractive mode (with the `--non-interactive` option).
>
> This has been fixed in Subversion 1.5 with the introduction of two new pairs of options. Use `--source-username` and `--source-password` to provide authentication credentials for the source repository; use `--sync-username` and `--sync-password` to provide credentials for the destination repository. (The old `--username` and `--password` options still exist for compatibility, but we advise against using them.)

And now comes the fun part. With a single subcommand, we can tell **svnsync** to copy all the as-yet-unmirrored revisions from the source repository to the target. [13] The **svnsync synchronize** subcommand will peek into the special revision properties previously stored on the target repository, and determine both what repository it is mirroring as well as that the most recently mirrored revision was revision 0. Then it will query the source repository and determine what the latest revision in that repository is. Finally, it asks the source repository's server to start replaying all the revisions between 0 and that latest revision. As **svnsync** get the resultant response from the source repository's server, it begins forwarding those revisions to the target repository's server as new commits.

```
$ svnsync help synchronize
synchronize (sync): usage: svnsync synchronize DEST_URL

Transfer all pending revisions to the destination from the source
with which it was initialized.
…
$ svnsync synchronize http://svn.example.com/svn-mirror
Transmitting file data -.......................................
Committed revision 1.
Copied properties for revision 1.
Transmitting file data -..
Committed revision 2.
```

---

[13] 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。

```
Copied properties for revision 2.
Transmitting file data -.....
Committed revision 3.
Copied properties for revision 3.
…
Transmitting file data -..
Committed revision 23406.
Copied properties for revision 23406.
Transmitting file data -.
Committed revision 23407.
Copied properties for revision 23407.
Transmitting file data -....
Committed revision 23408.
Copied properties for revision 23408.
$
```

Of particular interest here is that for each mirrored revision, there is first a commit of that revision to the target repository, and then property changes follow. This is because the initial commit is performed by (and attributed to) the user `syncuser`, and it is datestamped with the time as of that revision's creation. Also, Subversion's underlying repository access interfaces don't provide a mechanism for setting arbitrary revision properties as part of a commit. So **svnsync** follows up with an immediate series of property modifications that copy into the target repository all the revision properties found for that revision in the source repository. This also has the effect of fixing the author and datestamp of the revision to match that of the source repository.

Also noteworthy is that **svnsync** performs careful bookkeeping that allows it to be safely interrupted and restarted without ruining the integrity of the mirrored data. If a network glitch occurs while mirroring a repository, simply repeat the **svnsync synchronize** command, and it will happily pick up right where it left off. In fact, as new revisions appear in the source repository, this is exactly what you to do to keep your mirror up to date.

关于 svnsync

svnsync needs to be able to set and modify revision properties on the mirror repository because those properties are part of the data it is tasked with mirroring. As those properties change in the source repository, those changes need to be reflected in the mirror repository, too. But **svnsync** also uses a set of custom revision properties—stored in revision 0 of the mirror repository—for its own internal bookkeeping. These properties contain information such as the URL and UUID of the source repository, plus some additional state-tracking information.

One of those pieces of state-tracking information is a flag that essentially just means "there's a synchronization in progress right now." This is used to prevent multiple **svnsync** processes from colliding with each other while trying to mirror data to the same destination repository. Now, generally you won't need to pay any attention whatsoever to *any* of these special properties (all of which begin with the prefix `svn:sync-`). Occasionally, though, if a synchronization fails unexpectedly, Subversion never has a chance to remove this particular state flag. This causes all future synchronization attempts to fail because it appears that a synchronization is still in progress when, in fact, none is. Fortunately, recovering from this situation is as simple as removing the `svn:sync-lock` property which serves as this flag from revision 0 of the mirror repository:

```
$ svn propdel ---revprop --r0 svn:sync-lock http://
svn.example.com/svn-mirror
property -'svn:sync-lock' deleted from repository revision 0
$
```

That **svnsync** stores the source repository URL in a bookkeeping property on the mirror repository is the reason why you have to specify that URL only once, during **svnsync init**. Future synchronization operations against that mirror simply consult the special `svn:sync-from-url` property stored on the mirror itself to know where to synchronize from. This value is used literally by the synchronization process, though. So while from within CollabNet's network you can perhaps access our example source URL as `http://svn/repos/svn` (because that first `svn` magically gets `.collab.net` appended to it by DNS voodoo), if you later need to update that mirror from another machine outside CollabNet's network, the synchronization might fail (because the hostname `svn` is ambiguous). For this reason, it's best to use fully qualified source repository URLs when initializing a mirror repository rather than those that refer to only hostnames or IP addresses (which can change over time). But here again, if you need an existing mirror to start referring to a different URL for the same source repository, you can change the bookkeeping property which houses that information:

```
$ svn propset ---revprop --r0 svn:sync-from-url NEW-SOURCE-
URL \
      http://svn.example.com/svn-mirror
property -'svn:sync-from-url' set on repository revision 0
$
```

Another interesting thing about these special bookkeeping properties is that **svnsync** will not attempt to mirror any of those properties when they are found in the source repository. The reason is probably obvious, but basically boils down to **svnsync** not being able to distinguish the special properties it has merely copied from the source repository from those it needs to consult and maintain for its own bookkeeping needs. This situation could occur if, for example, you were maintaining a mirror of a mirror of a third repository. When **svnsync** sees its own special properties in revision 0 of the source repository, it simply ignores them.

There is, however, one bit of inelegance in the process. Because Subversion revision properties can be changed at any time throughout the lifetime of the repository, and because they don't leave an audit trail that indicates when they were changed, replication processes have to pay special attention to them. If you've already mirrored the first 15 revisions of a repository and someone then changes a revision property on revision 12, **svnsync** won't know to go back and patch up its copy of revision 12. You'll need to tell it to do so manually by using (or with some additional tooling around) the **svnsync copy-revprops** subcommand, which simply rereplicates all the revision properties for a particular revision or range thereof.

```
$ svnsync help copy-revprops
copy-revprops: usage: svnsync copy-revprops DEST_URL [REV[:REV2]]

Copy the revision properties in a given range of revisions to the
destination from the source with which it was initialized.
…
$ svnsync copy-revprops http://svn.example.com/svn-mirror 12
Copied properties for revision 12.
$
```

That's repository replication in a nutshell. You'll likely want some automation around such a process. For example, while our example was a pull-and-push setup, you might wish to have your primary repository push changes to one or more blessed mirrors as part of its post-commit and post-revprop-change hook implementations. This would enable the mirror to be up to date in as near to real time as is likely possible.

Also, while it isn't very commonplace to do so, **svnsync** does gracefully mirror repositories in which the user as whom it authenticates has only partial read access. It simply copies only the bits of the repository that it is permitted to see. Obviously, such a mirror is not useful as a backup solution.

In Subversion 1.5, **svnsync** grew the ability to also mirror a subset of a repository rather than the whole thing. The process of setting up and maintaining such a mirror is exactly the same as when mirroring a whole repository, except that instead of specifying the source repository's root URL when running **svnsync init**, you specify the URL of some subdirectory within that repository. Synchronization to that mirror will now copy only the bits that changed under that source repository subdirectory. There are some limitations to this support, though. First, you can't mirror multiple disjoint subdirectories of the source repository into a single mirror repository—you'd need to instead mirror some parent directory that is common to both. Second, the filtering logic is entirely path-based, so if the subdirectory you are mirroring was renamed at some point in the past, your mirror would contain only the revisions since the directory appeared at the URL you specified. And likewise, if the source subdirectory is renamed in the future, your synchronization processes will stop mirroring data at the point that the source URL you specified is no longer valid.

As far as user interaction with repositories and mirrors goes, it *is* possible to have a single working copy that interacts with both, but you'll have to jump through some hoops to make it happen. First, you need to ensure that both the primary and mirror repositories have the same repository UUID (which is not the case by default). See 第 5.4.9 节 "管理版本库的 UUID" later in this chapter for more about this.

Once the two repositories have the same UUID, you can use **svn switch** with the `--relocate` option to point your working copy to whichever of the repositories you wish to operate against, a process that is described in svn switch. There is a possible danger here, though, in that if the primary and mirror repositories aren't in close synchronization, a working copy up to date with, and pointing to, the primary repository will, if relocated to point to an out-of-date mirror, become confused about the apparent sudden loss of revisions it fully expects to be present, and it will throw errors to that effect. If this occurs, you can relocate your working copy back to the primary repository and then either wait until the mirror repository is up to date, or backdate your working copy to a revision you know is present in the sync repository, and then retry the relocation.

Finally, be aware that the revision-based replication provided by **svnsync** is only that—replication of revisions. Only information carried by the Subversion repository dump file format is available for replication. As such, **svnsync** has the same sorts of limitations that the repository dump stream has, and does not include such things as the hook implementations, repository or server configuration data, uncommitted transactions, or information about user locks on repository paths.

5.4.8. 版本库备份

Despite numerous advances in technology since the birth of the modern computer, one thing unfortunately rings true with crystalline clarity—sometimes things go very, very awry. Power outages, network connectivity dropouts, corrupt RAM, and crashed hard drives are but a taste of the evil that Fate is poised to unleash on even the most conscientious administrator. And so we arrive at a very important topic—how to make backup copies of your repository data.

There are two types of backup methods available for Subversion repository administrators—full and incremental. A full backup of the repository involves squirreling away in one sweeping action all the information required to fully reconstruct that repository in the event of a catastrophe. Usually, it means, quite literally, the duplication of the entire repository directory (which includes either a Berkeley DB or FSFS environment). Incremental backups are lesser things: backups of only the portion of the repository data that has changed since the previous backup.

As far as full backups go, the naïve approach might seem like a sane one, but unless you temporarily disable all other access to your repository, simply doing a recursive directory copy runs the risk of generating a faulty backup. In the case of Berkeley DB, the documentation describes a certain order in which database files can be copied that will guarantee a valid backup copy. A similar ordering exists for FSFS data. But you don't have to implement these algorithms yourself, because the Subversion development team has already done so. The **svnadmin hotcopy** command takes care of the minutia involved in making a hot backup of your repository. And its invocation is as trivial as the Unix **cp** or Windows **copy** operations:

```
$ svnadmin hotcopy -/var/svn/repos -/var/svn/repos-backup
```

The resultant backup is a fully functional Subversion repository, able to be dropped in as a replacement for your live repository should something go horribly wrong.

When making copies of a Berkeley DB repository, you can even instruct **svnadmin hotcopy** to purge any unused Berkeley DB logfiles (see 第 5.4.3.3 节 "删除不使用的 Berkeley DB 日志文件") from the original repository upon completion of the copy. Simply provide the `--clean-logs` option on the command line.

```
$ svnadmin hotcopy ---clean-logs -/var/svn/bdb-repos -/var/svn/bdb-repos-backup
```

Additional tooling around this command is available, too. The `tools/backup/` directory of the Subversion source distribution holds the **hot-backup.py** script. This script adds a bit of backup management atop **svnadmin hotcopy**, allowing you to keep only the most recent configured number of backups of each repository. It will automatically manage the names of the backed-up repository directories to avoid collisions with previous backups and will "rotate off" older backups, deleting them so that only the most recent ones remain. Even if you also have an incremental backup, you might want to run this program on a regular basis. For example, you might consider using **hot-backup.py** from a program scheduler (such as **cron** on Unix systems), which can cause it to run nightly (or at whatever granularity of time you deem safe).

Some administrators use a different backup mechanism built around generating and storing repository dump data. We described in 第 5.4.5 节 "版本库数据的移植" how to use **svnadmin dump** with the `--incremental` option to perform an incremental backup of a given revision or range of revisions. And of course, you can achieve a full backup variation of this by omitting the `--incremental` option to that command. There is some value in these methods, in that the format of your backed-up information is flexible—it's not tied to a particular platform, versioned filesystem type, or release of Subversion or Berkeley DB. But that flexibility comes at a cost, namely that restoring that data can take a long time—longer with each new revision committed to your repository. Also, as is the case with so many of the various backup methods, revision property changes that are made to already backed-up revisions won't get picked up by a nonoverlapping, incremental dump generation. For these reasons, we recommend against relying solely on dump-based backup approaches.

As you can see, each of the various backup types and methods has its advantages and disadvantages. The easiest is by far the full hot backup, which will always result in a perfect working replica of your repository. Should something bad happen to your live repository, you can restore from the backup with a simple recursive directory copy. Unfortunately, if you are maintaining multiple backups of your repository, these full copies will each eat up just as much disk space as your live repository. Incremental backups, by contrast, tend to be quicker to generate and smaller to store. But the restoration process can be a pain, often involving applying multiple incremental backups. And other methods have their own peculiarities. Administrators need to find the balance between the cost of making the backup and the cost of restoring it.

The **svnsync** program (see 第 5.4.7 节 "版本库复制") actually provides a rather handy middle-ground approach. If you are regularly synchronizing a read-only mirror with your main repository, in a pinch your read-only mirror is probably a good candidate for replacing that main repository if it falls over. The primary disadvantage of this method is that only the versioned repository data gets synchronized—repository configuration files, user-specified repository path locks, and other items that might live in the physical repository directory but not *inside* the repository's virtual versioned filesystem are not handled by **svnsync**.

In any backup scenario, repository administrators need to be aware of how modifications to unversioned revision properties affect their backups. Since these changes do not themselves generate new revisions, they will not trigger post-commit hooks, and may not even trigger the pre-revprop-change and post-revprop-change hooks. [14] And since you can change revision properties without respect to chronological order—you can change any revision's properties at any time—an incremental backup of the latest few revisions might not catch a property modification to a revision that was included as part of a previous backup.

Generally speaking, only the truly paranoid would need to back up their entire repository, say, every time a commit occurred. However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (such as per-commit emails or incremental dumps), a hot backup of the database might be something that a repository administrator would want to include as part of a system-wide nightly backup. It's your data—protect it as much as you'd like.

Often, the best approach to repository backups is a diversified one that leverages combinations of the methods described here. The Subversion developers, for example, back up the Subversion source code repository nightly using **hot-backup.py** and an off-site **rsync** of those full backups; keep multiple archives of all the commit and property change notification emails; and have repository mirrors maintained by various volunteers using **svnsync**. Your solution might be similar, but should be catered to your needs and that delicate balance of convenience with paranoia. And whatever you do, validate your backups from time to time—what good is a spare tire that has a hole in it? While all of this might not save your hardware from the iron fist of Fate, [15] it should certainly help you recover from those trying times.

5.4.9. 管理版本库的 UUID

Subversion repositories have a universally unique identifier (UUID) associated with them. This is used by Subversion clients to verify the identity of a repository when other forms of verification aren't good enough (such as checking the repository URL, which can change over time). Most Subversion repository administrators rarely, if ever, need to think about repository UUIDs as anything more than a trivial implementation detail of Subversion. Sometimes, however, there is cause for attention to this detail.

As a general rule, you want the UUIDs of your live repositories to be unique. That is, after all, the point of having UUIDs. But there are times when you want the repository UUIDs of two repositories to be exactly the same. For example, if you make a copy of a repository for backup purposes, you want the backup to be a perfect replica of the original so that, in the event that you have to restore that backup and replace the live repository, users don't suddenly see what looks like a different repository. When dumping and loading repository history (as described earlier in 第 5.4.5 节 "版本库数据的移植"), you get to decide whether to apply the UUID encapsulated in the data dump stream to the repository in which you are loading the data. The particular circumstance will dictate the correct behavior.

There are a couple of ways to set (or reset) a repository's UUID, should you need to. As of Subversion 1.5, this is as simple as using the **svnadmin setuuid** command. If you provide this

---

[14] **svnadmin setlog**可以被绕过钩子程序被调用。
[15] You know—the collective term for all of her "fickle fingers."

subcommand with an explicit UUID, it will validate that the UUID is well-formed and then set the repository UUID to that value. If you omit the UUID, a brand-new UUID will be generated for your repository.

```
$ svnlook uuid -/var/svn/repos
cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
$ svnadmin setuuid -/var/svn/repos   # generate a new UUID
$ svnlook uuid -/var/svn/repos
3c3c38fe-acc0-11dc-acbc-1b37ff1c8e7c
$ svnadmin setuuid -/var/svn/repos \
          cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec  # restore the
old UUID
$ svnlook uuid -/var/svn/repos
cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
$
```

For folks using versions of Subversion earlier than 1.5, these tasks are a little more complicated. You can explicitly set a repository's UUID by piping a repository dump file stub that carries the new UUID specification through **svnadmin load --force-uuid *REPOS-PATH***.

```
$ svnadmin load ---force-uuid -/var/svn/repos <<EOF
SVN-fs-dump-format-version: 2

UUID: cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
EOF
$ svnlook uuid -/var/svn/repos
cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
$
```

Having older versions of Subversion generate a brand-new UUID is not quite as simple to do, though. Your best bet here is to find some other way to generate a UUID, and then explicitly set the repository's UUID to that value.

5.5. 移动和删除版本库

Subversion repository data is wholly contained within the repository directory. As such, you can move a Subversion repository to some other location on disk, rename a repository, copy a repository, or delete a repository altogether using the tools provided by your operating system for manipulating directories—**mv**, **cp -a**, and **rm -r** on Unix platforms; **copy**, **move**, and **rmdir / s /q** on Windows; vast numbers of mouse and menu gyrations in various graphical file explorer applications, and so on.

Of course, there's often still more to be done when trying to cleanly affect changes such as this. For example, you might need to update your Subversion server configuration to point to the new location of a relocated repository or to remove configuration bits for a now-deleted repository. If you have automated processes that publish information from or about your repositories, they may need to be updated. Hook scripts might need to be reconfigured. Users

may need to be notified. The list can go on indefinitely, or at least to the extent that you've built processes and procedures around your Subversion repository.

In the case of a copied repository, you should also consider the fact that Subversion uses repository UUIDs to distinguish repositories. If you copy a Subversion repository using a typical shell recursive copy command, you'll wind up with two repositories that are identical in every way—including their UUIDs. In some circumstances, this might be desirable. But in the instances where it is not, you'll need to generate a new UUID for one of these identical repositories. See 第 5.4.9 节 "管理版本库的 UUID" for more about managing repository UUIDs.

5.6. 总结

By now you should have a basic understanding of how to create, configure, and maintain Subversion repositories. We introduced you to the various tools that will assist you with this task. Throughout the chapter, we noted common administration pitfalls and offered suggestions for avoiding them.

All that remains is for you to decide what exciting data to store in your repository, and finally, how to make it available over a network. The next chapter is all about networking.

第 6 章 服务配置

A Subversion repository can be accessed simultaneously by clients running on the same machine on which the repository resides using the `file://` method. But the typical Subversion setup involves a single server machine being accessed from clients on computers all over the office—or, perhaps, all over the world.

This chapter describes how to get your Subversion repository exposed outside its host machine for use by remote clients. We will cover Subversion's currently available server mechanisms, discussing the configuration and use of each. After reading this chapter, you should be able to decide which networking setup is right for your needs, as well as understand how to enable such a setup on your host computer.

6.1. 概述

Subversion was designed with an abstract network layer. This means that a repository can be programmatically accessed by any sort of server process, and the client "repository access" API allows programmers to write plug-ins that speak relevant network protocols. In theory, Subversion can use an infinite number of network implementations. In practice, there are only two servers at the time of this writing.

Apache is an extremely popular web server; using the **mod_dav_svn** module, Apache can access a repository and make it available to clients via the WebDAV/DeltaV protocol, which is an extension of HTTP. Because Apache is an extremely extensible server, it provides a number of features "for free," such as encrypted SSL communication, logging, integration with a number of third-party authentication systems, and limited built-in web browsing of repositories.

In the other corner is **svnserve**: a small, lightweight server program that speaks a custom protocol with clients. Because its protocol is explicitly designed for Subversion and is stateful (unlike HTTP), it provides significantly faster network operations—but at the cost of some features as well. While it can use SASL to provide a variety of authentication and encryption options, it has no logging or built-in web browsing. It is, however, extremely easy to set up and is often the best option for small teams just starting out with Subversion.

A third option is to use **svnserve** tunneled over an SSH connection. Even though this scenario still uses **svnserve**, it differs quite a bit in features from a traditional **svnserve** deployment. SSH is used to encrypt all communication. SSH is also used exclusively to authenticate, so real system accounts are required on the server host (unlike vanilla **svnserve**, which has its own private user accounts). Finally, because this setup requires that each user spawn a private, temporary **svnserve** process, it's equivalent (from a permissions point of view) to allowing a group of local users to all access the repository via `file://` URLs. Path-based access control has no meaning, since each user is accessing the repository database files directly.

表 6.1 "Subversion 服务器选项比较" 是三种典型服务器部署的总结。

表 6.1. Subversion 服务器选项比较

特性	Apache + mod_dav_svn	svnserve	穿越 SSH 隧道的 svnserve 服务器
认证选项	HTTP(S) 基本认证, X.509 证书, LDAP, NTLM 或任何 Apache httpd 已经具备的方式	缺省是 CRAM-MD5；LDAP, NTLM 或任何 SASL 支持的机制。	SSH
用户帐号选项	私有 "users" 文件，或其它 Apache httpd(LDAP，SQL 等)支持的机制	私有 "users" 文件，或其它 SASL(LDAP，SQL 等)支持的机制	系统帐号
授权选项	可以授予整个版本库的读/写权限，也可以为每个路径指定	可以授予整个版本库的读/写权限，也可以为每个路径指定	只能对整个版本库授予读/写权限
加密	通过可选的 SSL	通过可选的 SASL 特性	继承 SSH 连接
日志	Full Apache logs of each HTTP request, with optional "high-level" logging of general client operations	无日志	无日志
交互性	可以被其它 WebDAV 客户端访问	只同 svn 客户端通讯	只同 svn 客户端通讯
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
主从服务器复制	从服务器透明代理写到主服务器	只能创建只读从服务器	只能创建只读从服务器
速度	有些慢	快一点	快一点
初始设置	有些复杂	极为简单	相对简单

6.2. 选择一个服务器配置

那你应该用什么服务器？什么最好？

Obviously, there's no right answer to that question. Every team has different needs, and the different servers all represent different sets of trade-offs. The Subversion project itself doesn't endorse one server or another, or consider either server more "official" than another.

下面是你选择或者不选择某一个部署方式的原因。

6.2.1. 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：

- By default, only one authentication method is available, the network protocol is not encrypted, and the server stores clear text passwords. (All these things can be changed by configuring SASL, but it's a bit more work to do.)

- 没有任何类型的日志，甚至是错误。

- 没有内置的 WEB 浏览 (你必须自己单独安装 WEB 服务器，以及版本库浏览软件，来增加此功能)。

6.2.2. 穿越 SSH 隧道的 svnserve 服务器

为什么你会希望使用它：

- The network protocol is stateful and noticeably faster than WebDAV.

- 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：

- Only one choice of authentication method is available.

- There is no logging of any kind, not even errors.

- It requires users to be in the same system group, or use a shared SSH key.

- If used improperly, it can lead to file permission problems.

6.2.3. Apache 的 HTTP 服务器

为什么你会希望使用它：

- It allows Subversion to use any of the numerous authentication systems already integrated with Apache.

- There is no need to create system accounts on the server.

- Full Apache logging is available.

- 网络传输可以通过SSL加密。

- HTTP(S) 通常可以穿越公司防火墙。

- Built-in repository browsing is available via web browser.

- The repository can be mounted as a network drive for transparent version control (see 第 C.2 节 "自动版本化").

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

6.2.4. 推荐

In general, the authors of this book recommend a vanilla **svnserve** installation for small teams just trying to get started with a Subversion server; it's the simplest to set up and has the fewest maintenance issues. You can always switch to a more complex server deployment as your needs change.

下面是一些常见的建议和小技巧，基于多年对用户的支持：

- If you're trying to set up the simplest possible server for your group, a vanilla **svnserve** installation is the easiest, fastest route. Note, however, that your repository data will be transmitted in the clear over the network. If your deployment is entirely within your company's LAN or VPN, this isn't an issue. If the repository is exposed to the wide-open Internet, you might want to make sure that either the repository's contents aren't sensitive (e.g., it contains only open source code), or that you go the extra mile in configuring SASL to encrypt network communications.

- If you need to integrate with existing legacy identity systems (LDAP, Active Directory, NTLM, X.509, etc.), you must use either the Apache-based server or **svnserve** configured with SASL. If you absolutely need server-side logs of either server errors or client activities, an Apache-based server is your only option.

- If you've decided to use either Apache or stock **svnserve**, create a single **svn** user on your system and run the server process as that user. Be sure to make the repository directory wholly owned by the **svn** user as well. From a security point of view, this keeps the repository data nicely siloed and protected by operating system filesystem permissions, changeable by only the Subversion server process itself.

- If you have an existing infrastructure that is heavily based on SSH accounts, and if your users already have system accounts on your server machine, it makes sense to deploy an **svnserve**-over-SSH solution. Otherwise, we don't widely recommend this option to the public. It's

generally considered safer to have your users access the repository via (imaginary) accounts managed by **svnserve** or Apache, rather than by full-blown system accounts. If your deep desire for encrypted communication still draws you to this option, we recommend using Apache with SSL or **svnserve** with SASL encryption instead.

- Do *not* be seduced by the simple idea of having all of your users access a repository directly via `file://` URLs. Even if the repository is readily available to everyone via a network share, this is a bad idea. It removes any layers of protection between the users and the repository: users can accidentally (or intentionally) corrupt the repository database, it becomes hard to take the repository offline for inspection or upgrade, and it can lead to a mess of file permission problems (see 第 6.6 节 "支持多种版本库访问方法"). Note that this is also one of the reasons we warn against accessing repositories via `svn+ssh://` URLs—from a security standpoint, it's effectively the same as local users accessing via `file://`, and it can entail all the same problems if the administrator isn't careful.

6.3. svnserve - 定制的服务器

The **svnserve** program is a lightweight server, capable of speaking to clients over TCP/IP using a custom, stateful protocol. Clients contact an **svnserve** server by using URLs that begin with the `svn://` or `svn+ssh://` scheme. This section will explain the different ways of running **svnserve**, how clients authenticate themselves to the server, and how to configure appropriate access control to your repositories.

6.3.1. 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**。

6.3.1.1. svnserve 与守护进程

The easiest option is to run **svnserve** as a standalone "daemon" process. Use the `-d` option for this:

```
$ svnserve --d
$              # svnserve is now running, listening on port 3690
```

When running **svnserve** in daemon mode, you can use the `--listen-port` and `--listen-host` options to customize the exact port and hostname to "bind" to.

Once we successfully start **svnserve** as explained previously, it makes every repository on your system available to the network. A client needs to specify an *absolute* path in the repository

URL. For example, if a repository is located at `/var/svn/project1`, a client would reach it via `svn://host.example.com/var/svn/project1`. To increase security, you can pass the `-r` option to **svnserve**, which restricts it to exporting only repositories below that path. For example:

```
$ svnserve --d --r -/var/svn
…
```

Using the `-r` option effectively modifies the location that the program treats as the root of the remote filesystem space. Clients then use URLs that have that path portion removed from them, leaving much shorter (and much less revealing) URLs:

```
$ svn checkout svn://host.example.com/project1
…
```

6.3.1.2. 通过 **inetd** 调用 **svnserve**

If you want **inetd** to launch the process, you need to pass the `-i` (`--inetd`) option. In the following example, we've shown the output from running `svnserve -i` at the command line, but note that this isn't how you actually start the daemon; see the paragraphs following the example for how to configure **inetd** to start **svnserve**.

```
$ svnserve --i
( success ( 1 2 ( ANONYMOUS -) ( edit-pipeline -) -) -)
```

When invoked with the `--inetd` option, **svnserve** attempts to speak with a Subversion client via `stdin` and `stdout` using a custom protocol. This is the standard behavior for a program being run via **inetd**. The IANA has reserved port 3690 for the Subversion protocol, so on a Unix-like system you can add lines to `/etc/services` such as these (if they don't already exist):

```
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
```

Make sure "svnowner" is a user that has appropriate permissions to access your repositories. Now, when a client connection comes into your server on port 3690, **inetd** will spawn an **svnserve** process to service it. Of course, you may also want to add `-r` to the configuration line as well, to restrict which repositories are exported.

6.3.1.3. 通过隧道调用 **svnserve**

A third way to invoke **svnserve** is in tunnel mode, using the `-t` option. This mode assumes that a remote-service program such as **rsh** or **ssh** has successfully authenticated a user and is now

invoking a private **svnserve** process *as that user*. (Note that you, the user, will rarely, if ever, have reason to invoke **svnserve** with the -t at the command line; instead, the SSH daemon does so for you.) The **svnserve** program behaves normally (communicating via stdin and stdout) and assumes that the traffic is being automatically redirected over some sort of tunnel back to the client. When **svnserve** is invoked by a tunnel agent like this, be sure that the authenticated user has full read and write access to the repository database files. It's essentially the same as a local user accessing the repository via file:// URLs.

这个选项将在第 6.3.4 节 "穿越 SSH 隧道"详细讨论。

6.3.1.4. svnserve 与 Windows 服务

If your Windows system is a descendant of Windows NT (2000, 2003, XP, or Vista), you can run **svnserve** as a standard Windows service. This is typically a much nicer experience than running it as a standalone daemon with the --daemon (-d) option. Using daemon mode requires launching a console, typing a command, and then leaving the console window running indefinitely. A Windows service, however, runs in the background, can start at boot time automatically, and can be started and stopped using the same consistent administration interface as other Windows services.

You'll need to define the new service using the command-line tool **SC.EXE**. Much like the **inetd** configuration line, you must specify an exact invocation of **svnserve** for Windows to run at startup time:

```
C:\> sc create svn
       binpath= -"C:\svn\bin\svnserve.exe ---service --r C:\repos"
       displayname= -"Subversion Server"
       depend= Tcpip
       start= auto
```

This defines a new Windows service named "svn," which executes a particular **svnserve.exe** command when started (in this case, rooted at C:\repos). There are a number of caveats in the prior example, however.

First, notice that the **svnserve.exe** program must always be invoked with the --service option. Any other options to **svnserve** must then be specified on the same line, but you cannot add conflicting options such as --daemon (-d), --tunnel, or --inetd (-i). Options such as -r or --listen-port are fine, though. Second, be careful about spaces when invoking the **SC.EXE** command: the key= value patterns must have no spaces between key= and must have exactly one space before the value. Lastly, be careful about spaces in your command line to be invoked. If a directory name contains spaces (or other characters that need escaping), place the entire inner value of binpath in double quotes, by escaping them:

```
C:\> sc create svn
       binpath= -"\"C:\program files\svn\bin\svnserve.exe\" ---
service --r C:\repos"
       displayname= -"Subversion Server"
```

```
        depend= Tcpip
        start= auto
```

Also note that the word `binpath` is misleading—its value is a *command line*, not the path to an executable. That's why you need to surround it with quotes if it contains embedded spaces.

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
```

The service can also be uninstalled (i.e., undefined) by deleting its definition: **sc delete svn**. Just be sure to stop the service first! The **SC.EXE** program has many other subcommands and options; run **sc /?** to learn more about it.

6.3.2. 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。

- 取决于已定义的策略，可能发生下列事情：

  - 如果没有收到认证请求，客户端可能被允许匿名访问。

  - 客户端可以在任何认证时被要求。

  - 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(通常通过SSH)。

The **svnserve** server, by default, knows only how to send a CRAM-MD5 [1] authentication challenge. In essence, the server sends a small amount of data to the client. The client uses the MD5 hash algorithm to create a fingerprint of the data and password combined, and then sends the fingerprint as a response. The server performs the same computation with the stored password to verify that the result is identical. *At no point does the actual password travel over the network.*

If your **svnserve** server was built with SASL support, it not only knows how to send CRAM-MD5 challenges, but also likely knows a whole host of other authentication mechanisms. See 第 6.3.3 节 "让 svnserve 使用 SASL" later in this chapter to learn how to configure SASL authentication and encryption.

It's also possible, of course, for the client to be externally authenticated via a tunnel agent, such as **ssh**. In that case, the server simply examines the user it's running as, and uses this name

---

[1]见RFC 2195。

as the authenticated username. For more on this, see the later section, 第 6.3.4 节 "穿越 SSH 隧道".

As you've already guessed, a repository's `svnserve.conf` file is the central mechanism for controlling authentication and authorization policies. The file has the same format as other configuration files (see 第 7.1 节 "运行配置区"): section names are marked by square brackets (`[` and `]`), comments begin with hashes (`#`), and each section contains specific variables that can be set (`variable = value`). Let's walk through these files and learn how to use them.

6.3.2.1. 创建一个用户文件和认证域

For now, the `[general]` section of `svnserve.conf` has all the variables you need. Begin by changing the values of those variables: choose a name for a file that will contain your usernames and passwords and choose an authentication realm:

```
[general]
password-db = userfile
realm = example realm
```

The `realm` is a name that you define. It tells clients which sort of "authentication namespace" they're connecting to; the Subversion client displays it in the authentication prompt and uses it as a key (along with the server's hostname and port) for caching credentials on disk (see 第 3.11.2 节 "客户端凭证缓存"). The `password-db` variable points to a separate file that contains a list of usernames and passwords, using the same familiar format. For example:

```
[users]
harry = foopassword
sally = barpassword
```

The value of `password-db` can be an absolute or relative path to the users file. For many admins, it's easy to keep the file right in the `conf/` area of the repository, alongside `svnserve.conf`. On the other hand, it's possible you may want to have two or more repositories share the same users file; in that case, the file should probably live in a more public place. The repositories sharing the users file should also be configured to have the same realm, since the list of users essentially defines an authentication realm. Wherever the file lives, be sure to set the file's read and write permissions appropriately. If you know which user(s) **svnserve** will run as, restrict read access to the users file as necessary.

6.3.2.2. 设置访问控制

There are two more variables to set in the `svnserve.conf` file: they determine what unauthenticated (anonymous) and authenticated users are allowed to do. The variables `anon-access` and `auth-access` can be set to the value `none`, `read`, or `write`. Setting the value to `none` prohibits both reading and writing; `read` allows read-only access to the repository, and `write` allows complete read/write access to the repository. For example:

```
[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
```

The example settings are, in fact, the default values of the variables, should you forget to define them. If you want to be even more conservative, you can block anonymous access completely:

```
[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
```

The server process understands not only these "blanket" access controls to the repository, but also finer-grained access restrictions placed on specific files and directories within the repository. To make use of this feature, you need to define a file containing more detailed rules, and then set the `authz-db` variable to point to it:

```
[general]
password-db = userfile
realm = example realm

# Specific access rules for specific locations
authz-db = authzfile
```

We discuss the syntax of the `authzfile` file in detail later in this chapter, in 第 6.5 节 "基于路径的授权". Note that the `authz-db` variable isn't mutually exclusive with the `anon-access` and `auth-access` variables; if all the variables are defined at once, *all* of the rules must be satisfied before access is allowed.

6.3.3. 让 **svnserve** 使用 SASL

For many teams, the built-in CRAM-MD5 authentication is all they need from **svnserve**. However, if your server (and your Subversion clients) were built with the Cyrus Simple Authentication and Security Layer (SASL) library, you have a number of authentication and encryption options available to you.

> 什么是 SASL？
>
> The Cyrus Simple Authentication and Security Layer is open source software written by Carnegie Mellon University. It adds generic authentication and encryption capabilities to any network protocol, and as of Subversion 1.5 and later, both the **svnserve** server and **svn** client know how to make use of this library. It may or may not be available to you: if you're building Subversion yourself, you'll need to have at least version 2.1 of SASL installed on your system, and you'll need to make sure that it's detected during Subversion's build process. If you're using a prebuilt Subversion binary package, you'll have to check with the package maintainer as to whether SASL support was compiled in. SASL comes with a number of pluggable modules that represent different authentication systems: Kerberos (GSSAPI), NTLM, One-Time-Passwords (OTP), DIGEST-MD5, LDAP, Secure-Remote-Password (SRP), and others. Certain mechanisms may or may not be available to you; be sure to check which modules are provided.
>
> You can download Cyrus SASL (both code and documentation) from http://asg.web.cmu.edu/sasl/sasl-library.html.

Normally, when a subversion client connects to **svnserve**, the server sends a greeting that advertises a list of the capabilities it supports, and the client responds with a similar list of capabilities. If the server is configured to require authentication, it then sends a challenge that lists the authentication mechanisms available; the client responds by choosing one of the mechanisms, and then authentication is carried out in some number of round-trip messages. Even when SASL capabilities aren't present, the client and server inherently know how to use the CRAM-MD5 and ANONYMOUS mechanisms (see 第 6.3.2 节 "内置的认证和授权"). If server and client were linked against SASL, a number of other authentication mechanisms may also be available. However, you'll need to explicitly configure SASL on the server side to advertise them.

6.3.3.1. 使用 SASL 认证

To activate specific SASL mechanisms on the server, you'll need to do two things. First, create a `[sasl]` section in your repository's `svnserve.conf` file with an initial key-value pair:

```
[sasl]
use-sasl = true
```

Second, create a main SASL configuration file called `svn.conf` in a place where the SASL library can find it—typically in the directory where SASL plug-ins are located. You'll have to locate the plug-in directory on your particular system, such as `/usr/lib/sasl2/` or `/etc/sasl2/`. (Note that this is *not* the `svnserve.conf` file that lives within a repository!)

On a Windows server, you'll also have to edit the system registry (using a tool such as **regedit**) to tell SASL where to find things. Create a registry key named `[HKEY_LOCAL_MACHINE\SOFTWARE\Carnegie Mellon\Project Cyrus\SASL Library]`, and place two keys inside it: a key called `SearchPath` (whose value is a path to the directory containing the SASL `sasl*.dll` plug-in libraries), and a key called `ConfFile` (whose value is a path to the parent directory containing the `svn.conf` file you created).

Because SASL provides so many different kinds of authentication mechanisms, it would be foolish (and far beyond the scope of this book) to try to describe every possible server-side configuration. Instead, we recommend that you read the documentation supplied in the `doc/` subdirectory of the SASL source code. It goes into great detail about every mechanism and how to configure the server appropriately for each. For the purposes of this discussion, we'll just demonstrate a simple example of configuring the DIGEST-MD5 mechanism. For example, if your `subversion.conf` (or `svn.conf`) file contains the following:

```
pwcheck_method: auxprop
auxprop_plugin: sasldb
sasldb_path: -/etc/my_sasldb
mech_list: DIGEST-MD5
```

you've told SASL to advertise the DIGEST-MD5 mechanism to clients and to check user passwords against a private password database located at `/etc/my_sasldb`. A system administrator can then use the **saslpasswd2** program to add or modify usernames and passwords in the database:

```
$ saslpasswd2 --c --f -/etc/my_sasldb --u realm username
```

A few words of warning: first, make sure the "realm" argument to **saslpasswd2** matches the same realm you've defined in your repository's `svnserve.conf` file; if they don't match, authentication will fail. Also, due to a shortcoming in SASL, the common realm must be a string with no space characters. Finally, if you decide to go with the standard SASL password database, make sure the **svnserve** program has read access to the file (and possibly write access as well, if you're using a mechanism such as OTP).

This is just one simple way of configuring SASL. Many other authentication mechanisms are available, and passwords can be stored in other places such as in LDAP or a SQL database. Consult the full SASL documentation for details.

Remember that if you configure your server to only allow certain SASL authentication mechanisms, this forces all connecting clients to have SASL support as well. Any Subversion client built without SASL support (which includes all pre-1.5 clients) will be unable to authenticate. On the one hand, this sort of restriction may be exactly what you want ("My clients must all use Kerberos!"). However, if you still want non-SASL clients to be able to authenticate, be sure to advertise the CRAM-MD5 mechanism as an option. All clients are able to use CRAM-MD5, whether they have SASL capabilities or not.

6.3.3.2. SASL 加密

SASL is also able to perform data encryption if a particular mechanism supports it. The built-in CRAM-MD5 mechanism doesn't support encryption, but DIGEST-MD5 does, and mechanisms such as SRP actually require use of the OpenSSL library. To enable or disable different levels of encryption, you can set two values in your repository's `svnserve.conf` file:

```
[sasl]
use-sasl = true
```

```
min-encryption = 128
max-encryption = 256
```

The `min-encryption` and `max-encryption` variables control the level of encryption demanded by the server. To disable encryption completely, set both values to 0. To enable simple checksumming of data (i.e., prevent tampering and guarantee data integrity without encryption), set both values to 1. If you wish to allow—but not require—encryption, set the minimum value to 0, and the maximum value to some bit length. To require encryption unconditionally, set both values to numbers greater than 1. In our previous example, we require clients to do at least 128-bit encryption, but no more than 256-bit encryption.

6.3.4. 穿越 SSH 隧道

svnserve's built-in authentication (and SASL support) can be very handy, because it avoids the need to create real system accounts. On the other hand, some administrators already have well-established SSH authentication frameworks in place. In these situations, all of the project's users already have system accounts and the ability to "SSH into" the server machine.

It's easy to use SSH in conjunction with **svnserve**. The client simply uses the `svn+ssh://` URL scheme to connect:

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harryssh@host.example.com's password:   *****

foo
bar
baz
…
```

In this example, the Subversion client is invoking a local **ssh** process, connecting to `host.example.com`, authenticating as the user `harryssh` (according to SSH user configuration), then spawning a private **svnserve** process on the remote machine running as the user `harryssh`. The **svnserve** command is being invoked in tunnel mode (`-t`), and its network protocol is being "tunneled" over the encrypted connection by **ssh**, the tunnel agent. If the client performs a commit, the authenticated username `harryssh` will be used as the author of the new revision.

The important thing to understand here is that the Subversion client is *not* connecting to a running **svnserve** daemon. This method of access doesn't require a daemon, nor does it notice one if present. It relies wholly on the ability of **ssh** to spawn a temporary **svnserve** process, which then terminates when the network connection is closed.

When using `svn+ssh://` URLs to access a repository, remember that it's the **ssh** program prompting for authentication, and *not* the **svn** client program. That means there's no automatic password-caching going on (see 第 3.11.2 节 "客户端凭证缓存"). The Subversion client often

makes multiple connections to the repository, though users don't normally notice this due to the password caching feature. When using `svn+ssh://` URLs, however, users may be annoyed by **ssh** repeatedly asking for a password for every outbound connection. The solution is to use a separate SSH password-caching tool such as **ssh-agent** on a Unix-like system, or **pageant** on Windows.

When running over a tunnel, authorization is primarily controlled by operating system permissions to the repository's database files; it's very much the same as if Harry were accessing the repository directly via a `file://` URL. If multiple system users are going to be accessing the repository directly, you may want to place them into a common group, and you'll need to be careful about umasks (be sure to read 第 6.6 节 "支持多种版本库访问方法" later in this chapter). But even in the case of tunneling, you can still use the `svnserve.conf` file to block access, by simply setting `auth-access = read` or `auth-access = none`. [2]

You'd think that the story of SSH tunneling would end here, but it doesn't. Subversion allows you to create custom tunnel behaviors in your runtime `config` file (see 第 7.1 节 "运行配置区".) For example, suppose you want to use RSH instead of SSH. [3] In the `[tunnels]` section of your `config` file, simply define it like this:

```
[tunnels]
rsh = rsh
```

And now, you can use this new tunnel definition by using a URL scheme that matches the name of your new variable: `svn+rsh://host/path`. When using the new URL scheme, the Subversion client will actually be running the command **rsh host svnserve -t** behind the scenes. If you include a username in the URL (e.g., `svn+rsh://username@host/path`), the client will also include that in its command (**rsh username@host svnserve -t**). But you can define new tunneling schemes to be much more clever than that:

```
[tunnels]
joessh = $JOESSH -/opt/alternate/ssh --p 29934
```

This example demonstrates a couple of things. First, it shows how to make the Subversion client launch a very specific tunneling binary (the one located at `/opt/alternate/ssh`) with specific options. In this case, accessing an `svn+joessh://` URL would invoke the particular SSH binary with `-p 29934` as arguments—useful if you want the tunnel program to connect to a nonstandard port.

Second, it shows how to define a custom environment variable that can override the name of the tunneling program. Setting the `SVN_SSH` environment variable is a convenient way to override the default SSH tunnel agent. But if you need to have several different overrides for different servers, each perhaps contacting a different port or passing a different set of options to SSH, you can use the mechanism demonstrated in this example. Now if we were to set the `JOESSH` environment variable, its value would override the entire value of the tunnel variable —**$JOESSH** would be executed instead of **/opt/alternate/ssh -p 29934**.

---

[2] 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。
[3] 我们实际上不支持这个，因为RSH在安全性上显著不如SSH。

6.3.5. SSH 配置技巧

It's possible to control not only the way in which the client invokes **ssh**, but also to control the behavior of **sshd** on your server machine. In this section, we'll show how to control the exact **svnserve** command executed by **sshd**, as well as how to have multiple users share a single system account.

6.3.5.1. 初始设置

To begin, locate the home directory of the account you'll be using to launch **svnserve**. Make sure the account has an SSH public/private keypair installed, and that the user can log in via public-key authentication. Password authentication will not work, since all of the following SSH tricks revolve around using the SSH `authorized_keys` file.

If it doesn't already exist, create the `authorized_keys` file (on Unix, typically `~/.ssh/authorized_keys`). Each line in this file describes a public key that is allowed to connect. The lines are typically of the form:

```
ssh-dsa AAAABtce9euch… user@example.com
```

The first field describes the type of key, the second field is the base64-encoded key itself, and the third field is a comment. However, it's a lesser known fact that the entire line can be preceded by a `command` field:

```
command="program" ssh-dsa AAAABtce9euch… user@example.com
```

When the `command` field is set, the SSH daemon will run the named program instead of the typical tunnel-mode **svnserve** invocation that the Subversion client asks for. This opens the door to a number of server-side tricks. In the following examples, we abbreviate the lines of the file as:

```
command="program" TYPE KEY COMMENT
```

6.3.5.2. 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
```

In this example, `/path/to/svnserve` might be a custom wrapper script around **svnserve** which sets the umask (see 第 6.6 节 "支持多种版本库访问方法".) It also shows how to anchor **svnserve** in a virtual root directory, just as one often does when running **svnserve** as a daemon process. This might be done either to restrict access to parts of the system, or simply to relieve the user of having to type an absolute path in the `svn+ssh://` URL.

It's also possible to have multiple users share a single account. Instead of creating a separate system account for each user, generate a public/private key pair for each person. Then place each public key into the `authorized_users` file, one per line, and use the `--tunnel-user` option:

```
  command="svnserve --t ---tunnel-user=harry" TYPE1 KEY1
harry@example.com
  command="svnserve --t ---tunnel-user=sally" TYPE2 KEY2
sally@example.com
```

This example allows both Harry and Sally to connect to the same account via public key authentication. Each of them has a custom command that will be executed; the `--tunnel-user` option tells **svnserve** to assume that the named argument is the authenticated user. Without `--tunnel-user`, it would appear as though all commits were coming from the one shared system account.

A final word of caution: giving a user access to the server via public-key in a shared account might still allow other forms of SSH access, even if you've set the `command` value in `authorized_keys`. For example, the user may still get shell access through SSH or be able to perform X11 or general port forwarding through your server. To give the user as little permission as possible, you may want to specify a number of restrictive options immediately after the `command`:

```
  command="svnserve --t ---tunnel-user=harry",no-port-
forwarding,no-agent-forw
arding,no-X11-forwarding,no-pty TYPE1 KEY1 harry@example.com
```

Note that this all must be on one line—truly on one line—since SSH `authorized_keys` files do not even allow the conventional backslash character (\) for line continuation. The only reason we've shown it with a line break is to fit it on the physical page of a book.

6.4. httpd - Apache 的 HTTP 服务器

The Apache HTTP Server is a "heavy-duty" network server that Subversion can leverage. Via a custom module, **httpd** makes Subversion repositories available to clients via the WebDAV/DeltaV protocol, which is an extension to HTTP 1.1 (see http://www.webdav.org/ for more information). This protocol takes the ubiquitous HTTP protocol that is the core of the World Wide Web, and adds writing—specifically, versioned writing—capabilities. The result is a standardized, robust system that is conveniently packaged as part of the Apache 2.0 software, supported by numerous operating systems and third-party products, and doesn't require network administrators to open up yet another custom port. [4] While an Apache-Subversion server has more features than **svnserve**, it's also a bit more difficult to set up. With flexibility often comes more complexity.

Much of the following discussion includes references to Apache configuration directives. While some examples are given of the use of these directives, describing them in full is outside the

---

[4]他们讨厌这样做。

scope of this chapter. The Apache team maintains excellent documentation, publicly available on their web site at http://httpd.apache.org. For example, a general reference for the configuration directives is located at http://httpd.apache.org/docs-2.0/mod/directives.html.

Also, as you make changes to your Apache setup, it is likely that somewhere along the way a mistake will be made. If you are not already familiar with Apache's logging subsystem, you should become aware of it. In your `httpd.conf` file are directives that specify the on-disk locations of the access and error logs generated by Apache (the `CustomLog` and `ErrorLog` directives, respectively). Subversion's **mod_dav_svn** uses Apache's error logging interface as well. You can always browse the contents of those files for information that might reveal the source of a problem that is not clearly noticeable otherwise.

为什么是 Apache 2？

If you're a system administrator, it's very likely that you're already running the Apache web server and have some prior experience with it. At the time of this writing, Apache 1.3 is the more popular version of Apache. The world has been somewhat slow to upgrade to the Apache 2.x series for various reasons: some people fear change, especially changing something as critical as a web server. Other people depend on plug-in modules that work only against the Apache 1.3 API, and they are waiting for a 2.x port. Whatever the reason, many people begin to worry when they first discover that Subversion's Apache module is written specifically for the Apache 2 API.

The proper response to this problem is: don't worry about it. It's easy to run Apache 1.3 and Apache 2 side by side; simply install them to separate places and use Apache 2 as a dedicated Subversion server that runs on a port other than 80. Clients can access the repository by placing the port number into the URL:

```
$ svn checkout http://host.example.com:7382/repos/project
```

6.4.1. 先决条件

To network your repository over HTTP, you basically need four components, available in two packages. You'll need Apache **httpd** 2.0, the **mod_dav** DAV module that comes with it, Subversion, and the **mod_dav_svn** filesystem provider module distributed with Subversion. Once you have all of those components, the process of networking your repository is as simple as:

- 配置好httpd 2.0，并且使用**mod_dav**启动

- Installing the **mod_dav_svn** backend to **mod_dav**, which uses Subversion's libraries to access the repository

- 配置你的`httpd.conf`来输出（或者说暴露）版本库

You can accomplish the first two items either by compiling **httpd** and Subversion from source code or by installing prebuilt binary packages of them on your system. For the most up-

to-date information on how to compile Subversion for use with the Apache HTTP Server, as well as how to compile and configure Apache itself for this purpose, see the INSTALL file in the top level of the Subversion source code tree.

6.4.2. 基本的 Apache 配置

Once you have all the necessary components installed on your system, all that remains is the configuration of Apache via its httpd.conf file. Instruct Apache to load the **mod_dav_svn** module using the LoadModule directive. This directive must precede any other Subversion-related configuration items. If your Apache was installed using the default layout, your **mod_dav_svn** module should have been installed in the modules subdirectory of the Apache install location (often /usr/local/apache2). The LoadModule directive has a simple syntax, mapping a named module to the location of a shared library on disk:

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Note that if **mod_dav** was compiled as a shared object (instead of statically linked directly to the **httpd** binary), you'll need a similar LoadModule statement for it, too. Be sure that it comes before the **mod_dav_svn** line:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

At a later location in your configuration file, you now need to tell Apache where you keep your Subversion repository (or repositories). The Location directive has an XML-like notation, starting with an opening tag and ending with a closing tag, with various other configuration directives in the middle. The purpose of the Location directive is to instruct Apache to do something special when handling requests that are directed at a given URL or one of its children. In the case of Subversion, you want Apache to simply hand off support for URLs that point at versioned resources to the DAV layer. You can instruct Apache to delegate the handling of all URLs whose path portions (the part of the URL that follows the server's name and the optional port number) begin with /repos/ to a DAV provider whose repository is located at /var/svn/repository using the following httpd.conf syntax:

```
<Location -/repos>
  DAV svn
  SVNPath -/var/svn/repository
</Location>
```

If you plan to support multiple Subversion repositories that will reside in the same parent directory on your local disk, you can use an alternative directive—SVNParentPath—to indicate that common parent directory. For example, if you know you will be creating multiple Subversion repositories in a directory /var/svn that would be accessed via URLs such as http://my.server.com/svn/repos1, http://my.server.com/svn/repos2, and so on, you could use the httpd.conf configuration syntax in the following example:

```
<Location -/svn>
  DAV svn

  # any -"/svn/foo" URL will map to a repository -/var/svn/foo
  SVNParentPath -/var/svn
</Location>
```

Using the previous syntax, Apache will delegate the handling of all URLs whose path portions begin with `/svn/` to the Subversion DAV provider, which will then assume that any items in the directory specified by the `SVNParentPath` directive are actually Subversion repositories. This is a particularly convenient syntax in that, unlike the use of the `SVNPath` directive, you don't have to restart Apache to create and network new repositories.

Be sure that when you define your new `Location`, it doesn't overlap with other exported locations. For example, if your main `DocumentRoot` is exported to `/www`, do not export a Subversion repository in `<Location  /www/repos>`. If a request comes in for the URI `/www/repos/foo.c`, Apache won't know whether to look for a file `repos/foo.c` in the `DocumentRoot`, or whether to delegate **mod_dav_svn** to return `foo.c` from the Subversion repository. The result is often an error from the server of the form `301 Moved Permanently`.

服务器名称和复制请求

Subversion makes use of the `COPY` request type to perform server-side copies of files and directories. As part of the sanity checking done by the Apache modules, the source of the copy is expected to be located on the same machine as the destination of the copy. To satisfy this requirement, you might need to tell **mod_dav** the name you use as the hostname of your server. Generally, you can use the `ServerName` directive in `httpd.conf` to accomplish this.

```
ServerName svn.example.com
```

If you are using Apache's virtual hosting support via the `NameVirtualHost` directive, you may need to use the `ServerAlias` directive to specify additional names by which your server is known. Again, refer to the Apache documentation for full details.

At this stage, you should strongly consider the question of permissions. If you've been running Apache for some time now as your regular web server, you probably already have a collection of content—web pages, scripts, and such. These items have already been configured with a set of permissions that allows them to work with Apache, or more appropriately, that allows Apache to work with those files. Apache, when used as a Subversion server, will also need the correct permissions to read and write to your Subversion repository.

You will need to determine a permission system setup that satisfies Subversion's requirements without messing up any previously existing web page or script installations. This might mean changing the permissions on your Subversion repository to match those in use by other things that Apache serves for you, or it could mean using the `User` and `Group` directives in

`httpd.conf` to specify that Apache should run as the user and group that owns your Subversion repository. There is no single correct way to set up your permissions, and each administrator will have different reasons for doing things a certain way. Just be aware that permission-related problems are perhaps the most common oversight when configuring a Subversion repository for use with Apache.

6.4.3. 认证选项

At this point, if you configured `httpd.conf` to contain something such as the following:

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/svn
</Location>
```

your repository is "anonymously" accessible to the world. Until you configure some authentication and authorization policies, the Subversion repositories that you make available via the `Location` directive will be generally accessible to everyone. In other words:

- 任何人可以使用Subversion客户端来从版本库URL取出一个工作副本(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。

- 任何人可以提交到版本库。

Of course, you might have already set up a `pre-commit` hook script to prevent commits (see 第 5.3.2 节 "实现版本库钩子"). But as you read on, you'll see that it's also possible to use Apache's built-in methods to restrict access in specific ways.

6.4.3.1. 配置 HTTP 认证

The easiest way to authenticate a client is via the HTTP Basic authentication mechanism, which simply uses a username and password to verify that a user is who she says she is. Apache provides an **htpasswd** utility for managing the list of acceptable usernames and passwords. Let's grant commit access to Sally and Harry. First, we need to add them to the password file:

```
$ ### 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
$
```

Next, you need to add some more `httpd.conf` directives inside your `Location` block to tell Apache what to do with your new password file. The `AuthType` directive specifies the type of authentication system to use. In this case, we want to specify the `Basic` authentication system. `AuthName` is an arbitrary name that you give for the authentication domain. Most browsers will display this name in the pop-up dialog box when the browser is querying the user for her name and password. Finally, use the `AuthUserFile` directive to specify the location of the password file you created using **htpasswd**.

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 -/var/svn
  AuthType Basic
  AuthName -"Subversion repository"
  AuthUserFile -/etc/svn-auth-file
</Location>
```

This `<Location>` block is not yet complete, and it will not do anything useful. It's merely telling Apache that whenever authorization is required, Apache should harvest a username and password from the Subversion client. What's missing here, however, are directives that tell Apache *which* sorts of client requests require authorization. Wherever authorization is required, Apache will demand authentication as well. The simplest thing to do is protect all requests. Adding `Require valid-user` tells Apache that all requests require an authenticated user:

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/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(第 6.4.4 节 "授权选项")来得到`Require`的细节，和授权政策的其他设置方法。

One word of warning: HTTP Basic Auth passwords pass in very nearly plain text over the network, and thus are extremely insecure.

Another option is to not use Basic authentication, but to use Digest authentication instead. Digest authentication allows the server to verify the client's identity *without* passing the plain-text password over the network. Assuming that the client and server both know the user's password, they can verify that the password is the same by using it to apply a hashing function to a one-time bit of information. The server sends a small random-ish string to the client; the client uses

the user's password to hash the string; the server then looks to see whether the hashed value is what it expected.

Configuring Apache for Digest authentication is also fairly easy, and only a small variation on our prior example. Be sure to consult Apache's documentation for full details.

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/svn
  AuthType Digest
  AuthName -"Subversion repository"
  AuthDigestDomain -/svn/
  AuthUserFile -/etc/svn-auth-file
  Require valid-user
</Location>
```

If you're looking for maximum security, public key cryptography is the best solution. It may be best to use some sort of SSL encryption, so that clients authenticate via `https://` instead of `http://`; at a bare minimum, you can configure Apache to use a self-signed server certificate. [5] Consult Apache's documentation (and OpenSSL documentation) about how to do that.

6.4.3.2. SSL 证书管理

Businesses that need to expose their repositories for access outside the company firewall should be conscious of the possibility that unauthorized parties could be "sniffing" their network traffic. SSL makes that kind of unwanted attention less likely to result in sensitive data leaks.

If a Subversion client is compiled to use OpenSSL, it gains the ability to speak to an Apache server via `https://` URLs. The Neon library used by the Subversion client is not only able to verify server certificates, but can also supply client certificates when challenged. When the client and server have exchanged SSL certificates and successfully authenticated one another, all further communication is encrypted via a session key.

It's beyond the scope of this book to describe how to generate client and server certificates and how to configure Apache to use them. Many other books, including Apache's own documentation, describe this task. But what we *can* cover here is how to manage server and client certificates from an ordinary Subversion client.

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

If the client receives a server certificate, it needs to verify that it trusts the certificate: is the server really who it claims to be? The OpenSSL library does this by examining the signer of the

---

[5] 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。

server certificate, or *certificate authority* (CA). If OpenSSL is unable to automatically trust the CA, or if some other problem occurs (such as an expired certificate or hostname mismatch), the Subversion command-line client will ask you whether you want to trust the server certificate anyway:

```
$ 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?
```

This dialogue should look familiar; it's essentially the same question you've probably seen coming from your web browser (which is just another HTTP client like Subversion). If you choose the `(p)`ermanent option, the server certificate will be cached in your private runtime `auth/` area in just the same way your username and password are cached (see 第 3.11.2 节 "客户端凭证缓存"). If cached, Subversion will automatically trust this certificate in future negotiations.

Your runtime `servers` file also gives you the ability to make your Subversion client automatically trust specific CAs, either globally or on a per-host basis. Simply set the `ssl-authority-files` variable to a semicolon-separated list of PEM-encoded CA certificates:

```
[global]
ssl-authority-files = -/path/to/CAcert1.pem;/path/to/CAcert2.pem
```

Many OpenSSL installations also have a predefined set of "default" CAs that are nearly universally trusted. To make the Subversion client automatically trust these standard authorities, set the `ssl-trust-default-ca` variable to `true`.

When talking to Apache, a Subversion client might also receive a challenge for a client certificate. Apache is asking the client to identify itself: is the client really who it says it is? If all goes correctly, the Subversion client sends back a private certificate signed by a CA that Apache trusts. A client certificate is usually stored on disk in encrypted format, protected by a local password. When Subversion receives this challenge, it will ask you for a path to the certificate and the password that protects it:

```
$ 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':  ********
…
```

Notice that the client certificate is a "p12" file. To use a client certificate with Subversion, it must be in PKCS#12 format, which is a portable standard. Most web browsers are already able to import and export certificates in that format. Another option is to use the OpenSSL command-line tools to convert existing certificates into PKCS#12.

Again, the runtime `servers` file allows you to automate this challenge on a per-host basis. Either or both pieces of information can be described in runtime variables:

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = -/path/to/my/cert.p12
ssl-client-cert-password = somepassword
```

Once you've set the `ssl-client-cert-file` and `ssl-client-cert-password` variables, the Subversion client can automatically respond to a client certificate challenge without prompting you. [6]

6.4.4. 授权选项

At this point, you've configured authentication, but not authorization. Apache is able to challenge clients and confirm identities, but it has not been told how to allow or restrict access to the clients bearing those identities. This section describes two strategies for controlling access to your repositories.

6.4.4.1. 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。

You can restrict access on all repository operations by adding the `Require  valid-user` directive to your `<Location>` block. Using our previous example, this would mean that only clients that claimed to be either `harry` or `sally` and that provided the correct password for their respective username would be allowed to do anything with the Subversion repository:

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/svn
```

---

[6]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。

```
  # 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>
```

Sometimes you don't need to run such a tight ship. For example, Subversion's own source code repository at http://svn.collab.net/repos/svn allows anyone in the world to perform read-only repository tasks (such as checking out working copies and browsing the repository with a web browser), but restricts all write operations to authenticated users. To do this type of selective restriction, you can use the `Limit` and `LimitExcept` configuration directives. Like the `Location` directive, these blocks have starting and ending tags, and you would nest them inside your `<Location>` block.

The parameters present on the `Limit` and `LimitExcept` directives are HTTP request types that are affected by that block. For example, if you wanted to disallow all access to your repository except the currently supported read-only operations, you would use the `LimitExcept` directive, passing the `GET`, `PROPFIND`, `OPTIONS`, and `REPORT` request type parameters. Then the previously mentioned `Require  valid-user` directive would be placed inside the `<LimitExcept>` block instead of just inside the `<Location>` block.

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/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>
```

These are only a few simple examples. For more in-depth information about Apache access control and the `Require` directive, take a look at the `Security` section of the Apache documentation's tutorials collection at http://httpd.apache.org/docs-2.0/misc/tutorials.html.

6.4.4.2. 每目录访问控制

It's possible to set up finer-grained permissions using a second Apache httpd module, **mod_authz_svn**. This module grabs the various opaque URLs passing from client to server, asks

mod_dav_svn to decode them, and then possibly vetoes requests based on access policies defined in a configuration file.

If you've built Subversion from source code, **mod_authz_svn** is automatically built and installed alongside **mod_dav_svn**. Many binary distributions install it automatically as well. To verify that it's installed correctly, make sure it comes right after **mod_dav_svn**'s `LoadModule` directive in `httpd.conf`:

```
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
```

To activate this module, you need to configure your `Location` block to use the `AuthzSVNAccessFile` directive, which specifies a file containing the permissions policy for paths within your repositories. (In a moment, we'll discuss the format of that file.)

Apache is flexible, so you have the option to configure your block in one of three general patterns. To begin, choose one of these basic configuration patterns. (The following examples are very simple; look at Apache's own documentation for much more detail on Apache authentication and authorization options.)

The simplest block is to allow open access to everyone. In this scenario, Apache never sends authentication challenges, so all users are treated as "anonymous." (See 例 6.1 "匿名访问的配置样例".)

例 6.1. 匿名访问的配置样例

```
<Location -/repos>
  DAV svn
  SVNParentPath -/var/svn

  # our access control policy
  AuthzSVNAccessFile -/path/to/access/file
</Location>
```

On the opposite end of the paranoia scale, you can configure your block to demand authentication from everyone. All clients must supply credentials to identify themselves. Your block unconditionally requires authentication via the `Require valid-user` directive, and it defines a means to authenticate. (See 例 6.2 "认证访问的配置样例".)

例 6.2. 认证访问的配置样例

```
<Location -/repos>
  DAV svn
  SVNParentPath -/var/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>
```

A third very popular pattern is to allow a combination of authenticated and anonymous access. For example, many administrators want to allow anonymous users to read certain repository directories, but want only authenticated users to read (or write) more sensitive areas. In this setup, all users start out accessing the repository anonymously. If your access control policy demands a real username at any point, Apache will demand authentication from the client. To do this, use both the `Satisfy Any` and `Require valid-user` directives together. (See 例 6.3 "混合认证/匿名访问的配置样例".)

例 6.3. 混合认证/匿名访问的配置样例

```
<Location -/repos>
  DAV svn
  SVNParentPath -/var/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>
```

Once you've settled on one of these three basic `httpd.conf` templates, you need to create your file containing access rules for particular paths within the repository. We describe this later in this chapter, in 第 6.5 节 "基于路径的授权".

6.4.4.3. 禁用基于路径的检查

The **mod_dav_svn** module goes through a lot of work to make sure that data you've marked "unreadable" doesn't get accidentally leaked. This means it needs to closely monitor all of the paths and file-contents returned by commands such as **svn checkout** and **svn update**. If these commands encounter a path that isn't readable according to some authorization policy, the path is typically omitted altogether. In the case of history or rename tracing—for example, running a command such as `svn cat -r OLD foo.c` on a file that was renamed long ago—the rename tracking will simply halt if one of the object's former names is determined to be read-restricted.

All of this path checking can sometimes be quite expensive, especially in the case of **svn log**. When retrieving a list of revisions, the server looks at every changed path in each revision and checks it for readability. If an unreadable path is discovered, it's omitted from the list of the revision's changed paths (normally seen with the `--verbose` option), and the whole log message is suppressed. Needless to say, this can be time-consuming on revisions that affect a large number of files. This is the cost of security: even if you haven't configured a module such as **mod_authz_svn** at all, the **mod_dav_svn** module is still asking Apache **httpd** to run authorization checks on every path. The **mod_dav_svn** module has no idea what authorization modules have been installed, so all it can do is ask Apache to invoke whatever might be present.

On the other hand, there's also an escape hatch of sorts, which allows you to trade security features for speed. If you're not enforcing any sort of per-directory authorization (i.e., not using **mod_authz_svn** or similar module), you can disable all of this path checking. In your `httpd.conf` file, use the `SVNPathAuthz` directive as shown in 例 6.4 "禁用所有的路径检查".

例 6.4. 禁用所有的路径检查

```
<Location -/repos>
  DAV svn
  SVNParentPath -/var/svn

  SVNPathAuthz off
</Location>
```

The `SVNPathAuthz` directive is "on" by default. When set to "off," all path-based authorization checking is disabled; **mod_dav_svn** stops invoking authorization checks on every path it discovers.

6.4.5. 额外的糖果

We've covered most of the authentication and authorization options for Apache and **mod_dav_svn**. But there are a few other nice features that Apache provides.

6.4.5.1. 版本库浏览

One of the most useful benefits of an Apache/WebDAV configuration for your Subversion repository is that the youngest revisions of your versioned files and directories are immediately available for viewing via a regular web browser. Since Subversion uses URLs to identify versioned resources, those URLs used for HTTP-based repository access can be typed directly into a web browser. Your browser will issue an HTTP GET request for that URL; based on whether that URL represents a versioned directory or file, **mod_dav_svn** will respond with a directory listing or with file contents.

Since the URLs do not contain any information about which version of the resource you wish to see, **mod_dav_svn** will always answer with the youngest version. This functionality has the wonderful side effect that you can pass around Subversion URLs to your peers as references to documents, and those URLs will always point at the latest manifestation of that document. Of course, you can even use the URLs as hyperlinks from other web sites, too.

> 我可以看到老的修订版本吗？
>
> With an ordinary web browser? In one word: nope. At least, not with **mod_dav_svn** as your only tool.
>
> Your web browser speaks ordinary HTTP only. That means it knows only how to GET public URLs, which represent the latest versions of files and directories. According to the WebDAV/DeltaV specification, each server defines a private URL syntax for older versions of resources, and that syntax is opaque to clients. To find an older version of a file, a client must follow a specific procedure to "discover" the proper URL; the procedure involves issuing a series of WebDAV PROPFIND requests and understanding DeltaV concepts. This is something your web browser simply can't do.
>
> So, to answer the question, one obvious way to see older revisions of files and directories is by passing the `--revision` (`-r`) argument to the **svn list** and **svn cat** commands. To browse old revisions with your web browser, however, you can use third-party software. A good example of this is ViewVC (http://viewvc.tigris.org/). ViewVC was originally written to display CVS repositories through the Web, [7] and the latest releases are able to understand Subversion repositories as well.

6.4.5.1.1. 正确的 MIME 类型

When browsing a Subversion repository, the web browser gets a clue about how to render a file's contents by looking at the Content-Type: header returned in Apache's response to the HTTP GET request. The value of this header is some sort of MIME type. By default, Apache will tell the web browsers that all repository files are of the "default" MIME type, typically text/plain. This can be frustrating, however, if a user wishes repository files to render as something more meaningful—for example, it might be nice to have a foo.html file in the repository actually render as HTML when browsing.

To make this happen, you need only to make sure that your files have the proper svn:mime-type set. We discuss this in more detail in 第 3.3.1 节 "文件内容类型", and you can

---

[7]Back then, it was called ViewCVS.

even configure your client to automatically attach proper `svn:mime-type` properties to files entering the repository for the first time; see 第 3.2.4 节 "自动设置属性".

So in our example, if one were to set the `svn:mime-type` property to `text/html` on file `foo.html`, Apache would properly tell your web browser to render the file as HTML. One could also attach proper `image/*` MIME-type properties to image files and ultimately get an entire web site to be viewable directly from a repository! There's generally no problem with this, as long as the web site doesn't contain any dynamically generated content.

6.4.5.1.2. 定制外观

You generally will get more use out of URLs to versioned files—after all, that's where the interesting content tends to lie. But you might have occasion to browse a Subversion directory listing, where you'll quickly note that the generated HTML used to display that listing is very basic, and certainly not intended to be aesthetically pleasing (or even interesting). To enable customization of these directory displays, Subversion provides an XML index feature. A single `SVNIndexXSLT` directive in your repository's `Location` block of `httpd.conf` will instruct **mod_dav_svn** to generate XML output when displaying a directory listing, and to reference the XSLT stylesheet of your choice:

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/svn
  SVNIndexXSLT -"/svnindex.xsl"
  …
</Location>
```

Using the `SVNIndexXSLT` directive and a creative XSLT stylesheet, you can make your directory listings match the color schemes and imagery used in other parts of your web site. Or, if you'd prefer, you can use the sample stylesheets provided in the Subversion source distribution's `tools/xslt/` directory. Keep in mind that the path provided to the `SVNIndexXSLT` directory is actually a URL path—browsers need to be able to read your stylesheets to make use of them!

6.4.5.1.3. 列出版本库

If you're serving a collection of repositories from a single URL via the `SVNParentPath` directive, then it's also possible to have Apache display all available repositories to a web browser. Just activate the `SVNListParentPath` directive:

```
<Location -/svn>
  DAV svn
  SVNParentPath -/var/svn
  SVNListParentPath on
  …
</Location>
```

If a user now points her web browser to the URL `http://host.example.com/svn/`, she'll see a list of all Subversion repositories sitting in `/var/svn`. Obviously, this can be a security problem, so this feature is turned off by default.

6.4.5.2. Apache 日志

Because Apache is an HTTP server at heart, it contains fantastically flexible logging features. It's beyond the scope of this book to discuss all of the ways logging can be configured, but we should point out that even the most generic `httpd.conf` file will cause Apache to produce two logs: `error_log` and `access_log`. These logs may appear in different places, but are typically created in the logging area of your Apache installation. (On Unix, they often live in `/usr/local/apache2/logs/`.)

The `error_log` describes any internal errors that Apache runs into as it works. The `access_log` file records every incoming HTTP request received by Apache. This makes it easy to see, for example, which IP addresses Subversion clients are coming from, how often particular clients use the server, which users are authenticating properly, and which requests succeed or fail.

Unfortunately, because HTTP is a stateless protocol, even the simplest Subversion client operation generates multiple network requests. It's very difficult to look at the `access_log` and deduce what the client was doing—most operations look like a series of cryptic `PROPPATCH`, `GET`, `PUT`, and `REPORT` requests. To make things worse, many client operations send nearly identical series of requests, so it's even harder to tell them apart.

mod_dav_svn, however, can come to your aid. By activating an "operational logging" feature, you can ask **mod_dav_svn** to create a separate log file describing what sort of high-level operations your clients are performing.

To do this, you need to make use of Apache's `CustomLog` directive (which is explained in more detail in Apache's own documentation). Be sure to invoke this directive *outside* your Subversion `Location` block:

```
<Location -/svn>
  DAV svn
  …
</Location>

CustomLog logs/svn_logfile -"%t %u %{SVN-ACTION}e" env=SVN-ACTION
```

In this example, we're asking Apache to create a special logfile, `svn_logfile`, in the standard Apache `logs` directory. The `%t` and `%u` variables are replaced by the time and username of the request, respectively. The really important parts are the two instances of `SVN-ACTION`. When Apache sees that variable, it substitutes the value of the `SVN-ACTION` environment variable, which is automatically set by **mod_dav_svn** whenever it detects a high-level client action.

So, instead of having to interpret a traditional `access_log` like this:

```
[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
…
```

you can peruse a much more intelligible `svn_logfile` like this:

```
[26/Jan/2007:22:24:20 --0600] -- get-dir -/tags r1729 props
[26/Jan/2007:22:24:27 --0600] -- update -/trunk r1729
depth=infinity send-copyfrom-args
[26/Jan/2007:22:25:29 --0600] -- status -/trunk/foo r1729
depth=infinity
[26/Jan/2007:22:25:31 --0600] sally commit r1730
```

完全记录的日志动作见 "高级日志" 一节。

6.4.5.3. 通过代理写

One of the nice advantages of using Apache as a Subversion server is that it can be set up for simple replication. For example, suppose that your team is distributed across four offices around the globe. The Subversion repository can exist only in one of those offices, which means the other three offices will not enjoy accessing it—they're likely to experience significantly slower traffic and response times when updating and committing code. A powerful solution is to set up a system consisting of one *master* Apache server and several *slave* Apache servers. If you place a slave server in each office, users can check out a working copy from whichever slave is closest to them. All read requests go to their local slave. Write requests get automatically routed to the single master server. When the commit completes, the master then automatically "pushes" the new revision to each slave server using the **svnsync** replication tool.

This configuration creates a huge perceptual speed increase for your users, because Subversion client traffic is typically 80–90% read requests. And if those requests are coming from a *local* server, it's a huge win.

In this section, we'll walk you through a standard setup of this single-master/multiple-slave system. However, keep in mind that your servers must be running at least Apache 2.2.0 (with **mod_proxy** loaded) and Subversion 1.5 (**mod_dav_svn**).

6.4.5.3.1. 配置服务器

First, configure your master server's `httpd.conf` file in the usual way. Make the repository available at a certain URI location, and configure authentication and authorization however you'd like. After that's done, configure each of your "slave" servers in the exact same way, but add the special `SVNMasterURI` directive to the block:

```
<Location -/svn>
  DAV svn
  SVNPath -/var/svn/repos
  SVNMasterURI http://master.example.com/svn
  …
</Location>
```

This new directive tells a slave server to redirect all write requests to the master. (This is done automatically via Apache's **mod_proxy** module.) Ordinary read requests, however, are still serviced by the slaves. Be sure that your master and slave servers all have matching authentication and authorization configurations; if they fall out of sync, it can lead to big headaches.

Next, we need to deal with the problem of infinite recursion. With the current configuration, imagine what will happen when a Subversion client performs a commit to the master server. After the commit completes, the server uses **svnsync** to replicate the new revision to each slave. But because **svnsync** appears to be just another Subversion client performing a commit, the slave will immediately attempt to proxy the incoming write request back to the master! Hilarity ensues.

The solution to this problem is to have the master push revisions to a different `<Location>` on the slaves. This location is configured to *not* proxy write requests at all, but to accept normal commits from (and only from) the master's IP address:

```
<Location -/svn-proxy-sync>
  DAV svn
  SVNPath -/var/svn/repos
  Order deny,allow
  Deny from all
  # Only let the server's IP address access this Location:
  Allow from 10.20.30.40
  …
</Location>
```

6.4.5.3.2. 设置复制

Now that you've configured your `Location` blocks on master and slaves, you need to configure the master to replicate to the slaves. This is done the usual way— using **svnsync**. If you're not familiar with this tool, see 第 5.4.7 节 "版本库复制" for details.

First, make sure that each slave repository has a `pre-revprop-change` hook script which allows remote revision property changes. (This is standard procedure for being on the receiving end of **svnsync**.) Then log into the master server and configure each of the slave repository URIs to receive data from the master repository on the local disk:

```
$ svnsync init http://slave1.example.com/svn-proxy-sync file://var/
svn/repos
Copied properties for revision 0.
```

```
$ svnsync init http://slave2.example.com/svn-proxy-sync file://var/
svn/repos
Copied properties for revision 0.
$ svnsync init http://slave3.example.com/svn-proxy-sync file://var/
svn/repos
Copied properties for revision 0.

# Perform the initial replication

$ svnsync sync http://slave1.example.com/svn-proxy-sync
Transmitting file data -....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data -.......
Committed revision 2.
Copied properties for revision 2.
…

$ svnsync sync http://slave2.example.com/svn-proxy-sync
Transmitting file data -....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data -.......
Committed revision 2.
Copied properties for revision 2.
…

$ svnsync sync http://slave3.example.com/svn-proxy-sync
Transmitting file data -....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data -.......
Committed revision 2.
Copied properties for revision 2.
…
```

After this is done, we configure the master server's `post-commit` hook script to invoke **svnsync** on each slave server:

```
#!/bin/sh
# Post-commit script to replicate newly committed revision to
slaves

svnsync sync http://slave1.example.com/svn-proxy-sync > -/dev/
null 2>&1
svnsync sync http://slave2.example.com/svn-proxy-sync > -/dev/
null 2>&1
```

```
svnsync sync http://slave3.example.com/svn-proxy-sync > -/dev/
null 2>&1
```

The extra bits on the end of each line aren't necessary, but they're a sneaky way to allow the sync commands to run in the background so that the Subversion client isn't left waiting forever for the commit to finish. In addition to this `post-commit` hook, you'll need a `post-revprop-change` hook as well so that when a user, say, modifies a log message, the slave servers get that change also:

```
#!/bin/sh
# Post-revprop-change script to replicate revprop-changes to slaves

REV=${2}
svnsync copy-revprops http://slave1.example.com/svn-proxy-sync
${REV} > -/dev/null 2>&1
svnsync copy-revprops http://slave2.example.com/svn-proxy-sync
${REV} > -/dev/null 2>&1
svnsync copy-revprops http://slave3.example.com/svn-proxy-sync
${REV} > -/dev/null 2>&1
```

The only thing we've left out here is what to do about locks. Because locks are strictly enforced by the master server (the only place where commits happen), we don't technically need to do anything. Many teams don't use Subversion's locking features at all, so it may be a nonissue for you. However, if lock changes aren't replicated from master to slaves, it means that clients won't be able to query the status of locks (e.g., **svn status -u** will show no information about repository locks). If this bothers you, you can write `post-lock` and `post-unlock` hook scripts that run **svn lock** and **svn unlock** on each slave machine, presumably through a remote shell method such as SSH. That's left as an exercise for the reader!

6.4.5.3.3. 告诫

Your master/slave replication system should now be ready to use. A couple of words of warning are in order, however. Remember that this replication isn't entirely robust in the face of computer or network crashes. For example, if one of the automated **svnsync** commands fails to complete for some reason, the slaves will begin to fall behind. For example, your remote users will see that they've committed revision 100, but then when they run **svn update**, their local server will tell them that revision 100 doesn't yet exist! Of course, the problem will be automatically fixed the next time another commit happens and the subsequent **svnsync** is successful—the sync will replicate all waiting revisions. But still, you may want to set up some sort of out-of-band monitoring to notice synchronization failures and force **svnsync** to run when things go wrong.

我们可以为 **svnserve** 设置复制吗？

If you're using **svnserve** instead of Apache as your server, you can certainly configure your repository's hook scripts to invoke **svnsync** as we've shown here, thereby causing automatic replication from master to slaves. Unfortunately, at the time of this writing there is no way to make slave **svnserve** servers automatically proxy write requests back to the master server. This means your users would only be able to check out read-only working copies from the slave servers. You'd have to configure your slave servers to disallow write access completely. This might be useful for creating read-only "mirrors" of popular open source projects, but it's not a transparent proxying system.

6.4.5.4. 其它的 **Apache** 特性

Several of the features already provided by Apache in its role as a robust web server can be leveraged for increased functionality or security in Subversion as well. The Subversion client is able to use SSL (the Secure Sockets Layer, discussed earlier). If your Subversion client is built to support SSL, it can access your Apache server using `https://` and enjoy a high-quality encrypted network session.

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。

Finally, because **mod_dav_svn** is speaking a subset of the WebDAV/DeltaV protocol, it's possible to access the repository via third-party DAV clients. Most modern operating systems (Win32, OS X, and Linux) have the built-in ability to mount a DAV server as a standard network "shared folder." This is a complicated topic, but also wondrous when implemented. For details, read 附 录 C, *WebDAV 和自动版本*.

Note that there are a number of other small tweaks one can make to **mod_dav_svn** that are too obscure to mention in this chapter. For a complete list of all `httpd.conf` directives that **mod_dav_svn** responds to, see "指令"一节.

6.5. 基于路径的授权

Both Apache and **svnserve** are capable of granting (or denying) permissions to users. Typically this is done over the entire repository: a user can read the repository (or not), and she can write to the repository (or not). It's also possible, however, to define finer-grained access rules. One set of users may have permission to write to a certain directory in the repository, but not others; another directory might not even be readable by all but a few special people.

Both servers use a common file format to describe these path-based access rules. In the case of Apache, one needs to load the **mod_authz_svn** module and then add the `AuthzSVNAccessFile` directive (within the `httpd.conf` file) pointing to your own rules file. (For a full explanation, see 第 6.4.4.2 节 "每目录访问控制".) If you're using **svnserve**, you need to make the `authz-db` variable (within `svnserve.conf`) point to your rules file.

你真的需要基于路径的访问控制吗？

A lot of administrators setting up Subversion for the first time tend to jump into path-based access control without giving it a lot of thought. The administrator usually knows which teams of people are working on which projects, so it's easy to jump in and grant certain teams access to certain directories and not others. It seems like a natural thing, and it appeases the administrator's desire to maintain tight control of the repository.

Note, though, that there are often invisible (and visible!) costs associated with this feature. In the visible category, the server needs to do a lot more work to ensure that the user has the right to read or write each specific path; in certain situations, there's very noticeable performance loss. In the invisible category, consider the culture you're creating. Most of the time, while certain users *shouldn't* be committing changes to certain parts of the repository, that social contract doesn't need to be technologically enforced. Teams can sometimes spontaneously collaborate with each other; someone may want to help someone else out by committing to an area she doesn't normally work on. By preventing this sort of thing at the server level, you're setting up barriers to unexpected collaboration. You're also creating a bunch of rules that need to be maintained as projects develop, new users are added, and so on. It's a bunch of extra work to maintain.

Remember that this is a version control system! Even if somebody accidentally commits a change to something she shouldn't, it's easy to undo the change. And if a user commits to the wrong place with deliberate malice, it's a social problem anyway, and that the problem needs to be dealt with outside Subversion.

So, before you begin restricting users' access rights, ask yourself whether there's a real, honest need for this, or whether it's just something that "sounds good" to an administrator. Decide whether it's worth sacrificing some server speed, and remember that there's very little risk involved; it's bad to become dependent on technology as a crutch for social problems. [8]

As an example to ponder, consider that the Subversion project itself has always had a notion of who is allowed to commit where, but it's always been enforced socially. This is a good model of community trust, especially for open source projects. Of course, sometimes there *are* truly legitimate needs for path-based access control; within corporations, for example, certain types of data really can be sensitive, and access needs to be genuinely restricted to small groups of people.

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。

The syntax of the file is the same familiar one used by `svnserve.conf` and the runtime configuration files. Lines that start with a hash (#) are ignored. In its simplest form, each section names a repository and path within it, as well as the authenticated usernames are the option names within each section. The value of each option describes the user's level of access to the repository path: either `r` (read-only) or `rw` (read/write). If the user is not mentioned at all, no access is allowed.

---

[8] 本书的共同主题！

To be more specific: the value of the section names is either of the form `[repos-name:path]` or of the form `[path]`. If you're using the `SVNParentPath` directive, it's important to specify the repository names in your sections. If you omit them, a section such as `[/some/dir]` will match the path `/some/dir` in *every* repository. If you're using the `SVNPath` directive, however, it's fine to only define paths in your sections—after all, there's only one repository.

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

In this first example, the user `harry` has full read and write access on the `/branches/calc/bug-142` directory in the `calc` repository, but the user `sally` has read-only access. Any other users are blocked from accessing this directory.

Of course, permissions are inherited from parent to child directory. That means we can specify a subdirectory with a different access policy for Sally:

```
[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
```

Now Sally can write to the `testing` subdirectory of the branch, but can still only read other parts. Harry, meanwhile, continues to have complete read/write access to the whole branch.

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 =
```

In this example, Harry has read/write access to the entire `bug-142` tree, but has absolutely no access at all to the `secret` subdirectory within it.

提示

The thing to remember is that the most specific path always matches first. The server tries to match the path itself, and then the parent of the path, then the parent of that, and so on. The net effect is that mentioning a specific path in the access file will always override any permissions inherited from parent directories.

By default, nobody has any access to the repository at all. That means that if you're starting with an empty file, you'll probably want to give at least read permission to all users at the root of the repository. You can do this by using the asterisk variable (*), which means "all users":

```
[/]
* = r
```

This is a common setup; notice that no repository name is mentioned in the section name. This makes all repositories world-readable to all users. Once all users have read access to the repositories, you can give explicit `rw` permission to certain users on specific subdirectories within specific repositories.

The asterisk variable (*) is also worth special mention because it's the *only* pattern that matches an anonymous user. If you've configured your server block to allow a mixture of anonymous and authenticated access, all users start out accessing anonymously. The server looks for a * value defined for the path being accessed; if it can't find one, it demands real authentication from the client.

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
```

Groups can be granted access control just like users. Distinguish them with an "at" (@) prefix:

```
[calc:/projects/calc]
@calc-developers = rw

[paint:/projects/paint]
jane = r
@paint-developers = rw
```

Another important fact is that the *first* matching rule is the one which gets applied to a user. In the prior example, even though Jane is a member of the `paint-developers` group (which has read/write access), the `jane = r` rule will be discovered and matched before the group rule, thus denying Jane write access.

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
```

Subversion 1.5 brings another useful feature to the access file syntax: username aliases. Some authentication systems expect and carry relatively short usernames of the sorts we've been describing here—`harry`, `sally`, `joe`, and so on. But other authentication systems—such as those which use LDAP stores or SSL client certificates—may carry much more complex usernames. For example, Harry's username in an LDAP-protected system might be `CN=Harold Hacker,OU=Engineers,DC=red-bean,DC=com`. With usernames like that, the access file can become quite bloated with long or obscure usernames that are easy to mistype. Fortunately, username aliases allow you to have to type the correct complex username only once, in a statement which assigns to it a more easily digestable alias.

```
[aliases]
harry = CN=Harold Hacker,OU=Engineers,DC=red-bean,DC=com
sally = CN=Sally Swatterbug,OU=Engineers,DC=red-bean,DC=com
joe = CN=Gerald I. Joseph,OU=Engineers,DC=red-bean,DC=com
…
```

Once you've defined a set of aliases, you can refer to the users elsewhere in the access file via their aliases in all the same places you could have instead used their actual usernames. Simply prepend an ampersand to the alias to distinguish it from a regular username:

```
[groups]
calc-developers = &harry, &sally, &joe
paint-developers = &frank, &sally, &jane
everyone = @calc-developers, @paint-developers
```

You might also choose to use aliases if your users' usernames change frequently. Doing so allows you to need to update only the aliases table when these username changes occur, instead of doing global-search-and-replace operations on the whole access file.

部分可读性和检出

If you're using Apache as your Subversion server and have made certain subdirectories of your repository unreadable to certain users, you need to be aware of a possible nonoptimal behavior with **svn checkout**.

When the client requests a checkout or update over HTTP, it makes a single server request and receives a single (often large) server response. When the server receives the request, that is the *only* opportunity Apache has to demand user authentication. This has some odd side effects. For example, if a certain subdirectory of the repository is readable only by user Sally, and user Harry checks out a parent directory, his client will respond to the initial authentication challenge as Harry. As the server generates the large response, there's no way it can resend an authentication challenge when it reaches the special subdirectory; thus the subdirectory is skipped altogether, rather than asking the user to reauthenticate as Sally at the right moment. In a similar way, if the root of the repository is anonymously world-readable, the entire checkout will be done without authentication—again, skipping the unreadable directory, rather than asking for authentication partway through.

6.6. 支持多种版本库访问方法

You've seen how a repository can be accessed in many different ways. But is it possible—or safe—for your repository to be accessed by multiple methods simultaneously? The answer is yes, provided you use a bit of foresight.

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)；

- An **svnserve** process—either a daemon or one launched by **inetd**—running as a particular fixed user

- 一个Apache **httpd**进程，以一个固定用户运行

The most common problem administrators run into is repository ownership and permissions. Does every process (or user) in the preceding list have the rights to read and write the repository's underlying data files? Assuming you have a Unix-like operating system, a straightforward approach might be to place every potential repository user into a new `svn` group, and make the repository wholly owned by that group. But even that's not enough, because a process may write to the database files using an unfriendly umask—one that prevents access by other users.

So the next step beyond setting up a common group for repository users is to force every repository-accessing process to use a sane umask. For users accessing the repository directly, you can make the **svn** program into a wrapper script that first runs **umask 002** and then runs the real **svn** client program. You can write a similar wrapper script for the **svnserve** program, and add a **umask 002** command to Apache's own startup script, `apachectl`. For example:

```
$ cat -/usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real -"$@"
```

Another common problem is often encountered on Unix-like systems. If your repository is backed by Berkeley DB, for example, it occasionally creates new log files to journal its actions. Even if the Berkeley DB repository is wholly owned by the **svn** group, these newly created log files won't necessarily be owned by that same group, which then creates more permissions problems for your users. A good workaround is to set the group SUID bit on the repository's `db` directory. This causes all newly created log files to have the same group owner as the parent directory.

Once you've jumped through these hoops, your repository should be accessible by all the necessary processes. It may seem a bit messy and complicated, but the problems of having multiple users sharing write access to common files are classic ones that are not often elegantly solved.

Fortunately, most repository administrators will never *need* to have such a complex configuration. Users who wish to access repositories that live on the same machine are not limited to using `file://` access URLs—they can typically contact the Apache HTTP server or **svnserve** using `localhost` for the server name in their `http://` or `svn://` URL. And maintaining multiple server processes for your Subversion repositories is likely to be more of a headache than necessary. We recommend that you choose a single server that best meets your needs and stick with it!

---

svn+ssh 服务器检查列表

It can be quite tricky to get a bunch of users with existing SSH accounts to share a repository without permissions problems. If you're confused about all the things that you (as an administrator) need to do on a Unix-like system, here's a quick checklist that resummarizes some of the topics discussed in this section:

- 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。

- Your users need to use a sane umask when accessing the repository, so make sure **svnserve** (`/usr/bin/svnserve`, or wherever it lives in `$PATH`) is actually a wrapper script that runs **umask 002** and executes the real **svnserve** binary.

- Take similar measures when using **svnlook** and **svnadmin**. Either run them with a sane umask or wrap them as just described.

第 7 章 定制你的 Subversion 体验

Version control can be a complex subject, as much art as science, that offers myriad ways of getting stuff done. Throughout this book, you've read of the various Subversion command-line client subcommands and the options that modify their behavior. In this chapter, we'll look into still more ways to customize the way Subversion works for you—setting up the Subversion runtime configuration, using external helper applications, Subversion's interaction with the operating system's configured locale, and so on.

7.1. 运行配置区

Subversion provides many optional behaviors that the user can control. Many of these options are of the kind that a user would wish to apply to all Subversion operations. So, rather than forcing users to remember command-line arguments for specifying these options and to use them for every operation they perform, Subversion uses configuration files, segregated into a Subversion configuration area.

The Subversion *configuration area* is a two-tiered hierarchy of option names and their values. Usually, this boils down to a special directory that contains *configuration files* (the first tier), which are just text files in standard INI format (with "sections" providing the second tier). You can easily edit these files using your favorite text editor (such as Emacs or vi), and they contain directives read by the client to determine which of several optional behaviors the user prefers.

7.1.1. 配置区布局

The first time the **svn** command-line client is executed, it creates a per-user configuration area. On Unix-like systems, this area appears as a directory named `.subversion` in the user's home directory. On Win32 systems, Subversion creates a folder named `Subversion`, typically inside the `Application Data` area of the user's profile directory (which, by the way, is usually a hidden directory). However, on this platform, the exact location differs from system to system and is dictated by the Windows Registry. [1] We will refer to the per-user configuration area using its Unix name, `.subversion`.

In addition to the per-user configuration area, Subversion also recognizes the existence of a system-wide configuration area. This gives system administrators the ability to establish defaults for all users on a given machine. Note that the system-wide configuration area alone does not dictate mandatory policy—the settings in the per-user configuration area override those in the system-wide one, and command-line arguments supplied to the **svn** program have the final word on behavior. On Unix-like platforms, the system-wide configuration area is expected to be the `/etc/subversion` directory; on Windows machines, it looks for a `Subversion` directory inside the common `Application Data` location (again, as specified by the Windows Registry). Unlike the per-user case, the **svn** program does not attempt to create the system-wide configuration area.

---

[1] 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。

The per-user configuration area currently contains three files—two configuration files (`config` and `servers`), and a `README.txt` file, which describes the INI format. At the time of their creation, the files contain default values for each of the supported Subversion options, mostly commented out and grouped with textual descriptions about how the values for the key affect Subversion's behavior. To change a certain behavior, you need only to load the appropriate configuration file into a text editor, and to modify the desired option's value. If at any time you wish to have the default configuration settings restored, you can simply remove (or rename) your configuration directory and then run some innocuous **svn** command, such as **svn --version**. A new configuration directory with the default contents will be created.

The per-user configuration area also contains a cache of authentication data. The `auth` directory holds a set of subdirectories that contain pieces of cached information used by Subversion's various supported authentication methods. This directory is created in such a way that only the user herself has permission to read its contents.

7.1.2. 配置和 Windows 注册表

In addition to the usual INI-based configuration area, Subversion clients running on Windows platforms may also use the Windows Registry to hold the configuration data. The option names and their values are the same as in the INI files. The "file/section" hierarchy is preserved as well, though addressed in a slightly different fashion—in this schema, files and sections are just levels in the Registry key tree.

Subversion looks for system-wide configuration values under the `HKEY_LOCAL_MACHINE \Software\Tigris.org\Subversion` key. For example, the `global-ignores` option, which is in the `miscellany` section of the `config` file, would be found at `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config \Miscellany\global-ignores`. Per-user configuration values should be stored under `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

Registry-based configuration options are parsed *before* their file-based counterparts, so they are overridden by values found in the configuration files. In other words, Subversion looks for configuration information in the following locations on a Windows system; lower-numbered locations take precedence over higher-numbered locations:

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

Also, the Windows Registry doesn't really support the notion of something being "commented out." However, Subversion will ignore any option key whose name begins with a hash (#) character. This allows you to effectively comment out a Subversion option without

deleting the entire key from the Registry, obviously simplifying the process of restoring that option.

The **svn** command-line client never attempts to write to the Windows Registry and will not attempt to create a default configuration area there. You can create the keys you need using the **REGEDIT** program. Alternatively, you can create a .reg file (such as the one in 例 7.1 "注册表条目(.reg)文件样例"), and then double-click on that file's icon in the Explorer shell, which will cause the data to be merged into your Registry.

例 7.1. 注册表条目(.reg)文件样例

```
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\Config\auth]
"#store-passwords"="yes"
"#store-auth-creds"="yes"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#editor-cmd"="notepad"
"#diff-cmd"=""
"#diff3-cmd"=""
"#diff3-has-program-arg"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config
\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]
```

The previous example shows the contents of a .reg file, which contains some of the most commonly used configuration options and their default values. Note the presence of both system-wide (for network proxy-related options) and per-user settings (editor programs and password

storage, among others). Also note that all the options are effectively commented out. You need only to remove the hash (#) character from the beginning of the option names and set the values as you desire.

7.1.3. 配置选项

In this section, we will discuss the specific runtime configuration options that Subversion currently supports.

7.1.3.1. 服务器

The `servers` file contains Subversion configuration options related to the network layers. There are two special section names in this file—`groups` and `global`. The `groups` section is essentially a cross-reference table. The keys in this section are the names of other sections in the file; their values are *globs*—textual tokens that possibly contain wildcard characters—that are compared against the hostnames of the machine to which Subversion requests are sent.

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
…

[collabnet]
…
```

When Subversion is used over a network, it attempts to match the name of the server it is trying to reach with a group name under the `groups` section. If a match is made, Subversion then looks for a section in the `servers` file whose name is the matched group's name. From that section, it reads the actual network configuration settings.

The `global` section contains the settings that are meant for all of the servers not matched by one of the globs under the `groups` section. The options available in this section are exactly the same as those that are valid for the other server sections in the file (except, of course, the special `groups` section), and are as follows:

http-proxy-exceptions
  This specifies a comma-separated list of patterns for repository hostnames that should be accessed directly, without using the proxy machine. The pattern syntax is the same as is used in the Unix shell for filenames. A repository hostname matching any of these patterns will not be proxied.

http-proxy-host
  This specifies the hostname of the proxy computer through which your HTTP-based Subversion requests must pass. It defaults to an empty value, which means that Subversion will not attempt to route HTTP requests through a proxy computer, and will instead attempt to contact the destination machine directly.

`http-proxy-port`

This specifies the port number on the proxy host to use. It defaults to an empty value.

`http-proxy-username`

This specifies the username to supply to the proxy machine. It defaults to an empty value.

`http-proxy-password`

This specifies the password to supply to the proxy machine. It defaults to an empty value.

`http-timeout`

This specifies the amount of time, in seconds, to wait for a server response. If you experience problems with a slow network connection causing Subversion operations to time out, you should increase the value of this option. The default value is `0`, which instructs the underlying HTTP library, Neon, to use its default timeout setting.

`http-compression`

This specifies whether Subversion should attempt to compress network requests made to DAV-ready servers. The default value is `yes` (though compression will occur only if that capability is compiled into the network layer). Set this to `no` to disable compression, such as when debugging network transmissions.

`http-library`

Subversion provides a pair of repository access modules that understand its WebDAV network protocol. The original one, which shipped with Subversion 1.0, is `libsvn_ra_neon` (though back then it was called `libsvn_ra_dav`). Newer Subversion versions also provide `libsvn_ra_serf`, which uses a different underlying implementation and aims to support some of the newer HTTP concepts.

At this point, `libsvn_ra_serf` is still considered experimental, though it appears to work in the common cases quite well. To encourage experimentation, Subversion provides the `http-library` runtime configuration option to allow users to specify (generally, or in a per-server-group fashion) which WebDAV access module they'd prefer to use—`neon` or `serf`.

`http-auth-types`

This option is a semicolon-delimited list of authentication types supported by the Neon-based WebDAV repository access modules. Valid members of this list are `basic`, `digest`, and `negotiate`.

`neon-debug-mask`

This is an integer mask that the underlying HTTP library, Neon, uses for choosing what type of debugging output to yield. The default value is `0`, which will silence all debugging output. For more information about how Subversion makes use of Neon, see 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`
> If a host (or set of hosts) requires an SSL client certificate, you'll normally be prompted for a path to your certificate. By setting this variable to that same path, Subversion will be able to find your client certificate automatically without prompting you. There's no standard place to store your certificate on disk; Subversion will grab it from any path you specify.

`ssl-client-cert-password`
> If your SSL client certificate file is encrypted by a passphrase, Subversion will prompt you for the passphrase whenever the certificate is used. If you find this annoying (and don't mind storing the password in the `servers` file), you can set this variable to the certificate's passphrase. You won't be prompted anymore.

`store-plaintext-passwords`
> This variable is only important on UNIX-like systems. It controls what the Subversion client does in case the password for the current authentication realm can only be cached on disk in unencrypted form, in the `~/.subversion/auth/` caching area. You can set it to `yes` or `no` to enable or disable caching of passwords in unencrypted form, respectively. The default setting is `ask`, which causes the Subversion client to ask you each time a *new* password is about to be added to the `~/.subversion/auth/` caching area.

7.1.3.2. 配置

The `config` file contains the rest of the currently available Subversion runtime options—those not related to networking. There are only a few options in use as of this writing, but they are again grouped into sections in expectation of future additions.

The `auth` section contains settings related to Subversion's authentication and authorization against the repository. It contains the following:

`store-passwords`
> This instructs Subversion to cache, or not to cache, passwords that are supplied by the user in response to server authentication challenges. The default value is `yes`. Set this to `no` to disable this on-disk password caching. You can override this option for a single instance of the **svn** command using the `--no-auth-cache` command-line parameter (for those subcommands that support it). For more information, see 第 3.11.2 节 "客户端凭证缓存".

`store-auth-creds`
> This setting is the same as `store-passwords`, except that it enables or disables on-disk caching of *all* authentication information: usernames, passwords, server certificates, and any other types of cacheable credentials.

The `helpers` section controls which external applications Subversion uses to accomplish its tasks. Valid options in this section are:

`editor-cmd`
> This specifies the program Subversion will use to query the user for certain types of textual metadata or when interactively resolving conflicts. See 第 7.3 节 "使用外置编辑器" for more details on using external text editors with Subversion.

diff-cmd

> This specifies the absolute path of a differencing program, used when Subversion generates "diff" output (such as when using the **svn diff** command). By default, Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See 第 7.4 节 "使用外置比较与合并工具" for more details on using such programs.

diff3-cmd

> This specifies the absolute path of a three-way differencing program. Subversion uses this program to merge changes made by the user with those received from the repository. By default, Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See 第 7.4 节 "使用外置比较与合并工具" for more details on using such programs.

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。

merge-tool-cmd

> This specifies the program that Subversion will use to perform three-way merge operations on your versioned files. See 第 7.4 节 "使用外置比较与合并工具" for more details on using such programs.

The tunnels section allows you to define new tunnel schemes for use with **svnserve** and svn:// client connections. For more details, see 第 6.3.4 节 "穿越 SSH 隧道".

The miscellany section is where everything that doesn't belong elsewhere winds up. [2] In this section, you can find:

global-ignores

> When running the **svn status** command, Subversion lists unversioned files and directories along with the versioned ones, annotating them with a ? character (see 第 2.4.3.1 节 "查看你的修改概况"). Sometimes it can be annoying to see uninteresting, unversioned items—for example, object files that result from a program's compilation—in this display. The global-ignores option is a list of whitespace-delimited globs that describe the names of files and directories that Subversion should not display unless they are versioned. The default value is *.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store.
>
> As well as **svn status**, the **svn add** and **svn import** commands also ignore files that match the list when they are scanning a directory. You can override this behavior for a single instance of any of these commands by explicitly specifying the filename, or by using the --no-ignore command-line flag.
>
> For information on finer-grained control of ignored items, see 第 3.4 节 "忽略未版本控制的条目".

---

[2] 就是一个大杂烩?

`enable-auto-props`
> This instructs Subversion to automatically set properties on newly added or imported files. The default value is `no`, so set this to `yes` to enable this feature. The `auto-props` section of this file specifies which properties are to be set on which files.

`log-encoding`
> This variable sets the default character set encoding for commit log messages. It's a permanent form of the `--encoding` option (see 第 9.1.1 节 "svn 选项"). The Subversion repository stores log messages in UTF-8 and assumes that your log message is written using your operating system's native locale. You should specify a different encoding if your commit messages are written in any other encoding.

`use-commit-times`
> Normally your working copy files have timestamps that reflect the last time they were touched by any process, whether your own editor or some **svn** subcommand. This is generally convenient for people developing software, because build systems often look at timestamps as a way of deciding which files need to be recompiled.
>
> In other situations, however, it's sometimes nice for the working copy files to have timestamps that reflect the last time they were changed in the repository. The **svn export** command always places these "last-commit timestamps" on trees that it produces. By setting this config variable to `yes`, the **svn checkout**, **svn update**, **svn switch**, and **svn revert** commands will also set last-commit timestamps on files that they touch.

`mime-types-file`
> This option, new to Subversion 1.5, specifies the path of a MIME types mapping file, such as the `mime.types` file provided by the Apache HTTP Server. Subversion uses this file to assign MIME types to newly added or imported files. See 第 3.2.4 节 "自动设置属性" and 第 3.3.1 节 "文件内容类型" for more about Subversion's detection and use of file content types.

`preserved-conflict-file-exts`
> The value of this option is a space-delimited list of file extensions that Subversion should preserve when generating conflict filenames. By default, the list is empty. This option is new to Subversion 1.5.
>
> When Subversion detects conflicting file content changes, it defers resolution of those conflicts to the user. To assist in the resolution, Subversion keeps pristine copies of the various competing versions of the file in the working copy. By default, those conflict files have names constructed by appending to the original filename a custom extension such as `.mine` or `.REV` (where `REV` is a revision number). A mild annoyance with this naming scheme is that on operating systems where a file's extension determines the default application used to open and edit that file, appending a custom extension prevents the file from being easily opened by its native application. For example, if the file `ReleaseNotes.pdf` was conflicted, the conflict files might be named `ReleaseNotes.pdf.mine` or `ReleaseNotes.pdf.r4231`. While your system might be configured to use Adobe's Acrobat Reader to open files whose extensions are `.pdf`, there probably isn't an application configured on your system to open all files whose extensions are `.r4231`.

You can fix this annoyance by using this configuration option, though. For files with one of the specified extensions, Subversion will append to the conflict file names the custom extension just as before, but then also reappend the file's original extension. Using the previous example, and assuming that `pdf` is one of the extensions configured in this list thereof, the conflict files generated for `ReleaseNotes.pdf` would instead be named `ReleaseNotes.pdf.mine.pdf` and `ReleaseNotes.pdf.r4231.pdf`. Because each file ends in `.pdf`, the correct default application will be used to view them.

`interactive-conflicts`
> This is a Boolean option that specifies whether Subversion should try to resolve conflicts interactively. If its value is `yes` (which is the default value), Subversion will prompt the user for how to handle conflicts in the manner demonstrated in 第 2.4.5 节 "解决冲突(合并别人的修改)". Otherwise, it will simply flag the conflict and continue its operation, postponing resolution to a later time.

`no-unlock`
> This Boolean option corresponds to **svn commit**'s `--no-unlock` option, which tells Subversion not to release locks on files you've just committed. If this runtime option is set to `yes`, Subversion will never release locks automatically, leaving you to run **svn unlock** explicitly. It defaults to `no`.

The `auto-props` section controls the Subversion client's ability to automatically set properties on files when they are added or imported. It contains any number of key-value pairs in the format `PATTERN = PROPNAME=VALUE[;PROPNAME=VALUE ...]`, where `PATTERN` is a file pattern that matches one or more filenames and the rest of the line is a semicolon-delimited set of property assignments. Multiple matches on a file will result in multiple propsets for that file; however, there is no guarantee that auto-props will be applied in the order in which they are listed in the config file, so you can't have one rule "override" another. You can find several examples of auto-props usage in the `config` file. Lastly, don't forget to set `enable-auto-props` to `yes` in the `miscellany` section if you want to enable auto-props.

7.2. 本地化

Localization is the act of making programs behave in a region-specific way. When a program formats numbers or dates in a way specific to your part of the world or prints messages (or accepts input) in your native language, the program is said to be *localized*. This section describes steps Subversion has made toward localization.

7.2.1. 理解区域设置

Most modern operating systems have a notion of the "current locale"—that is, the region or country whose localization conventions are honored. These conventions—typically chosen by some runtime configuration mechanism on the computer—affect the way in which programs present data to the user, as well as the way in which they accept user input.

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"
$
```

The output is a list of locale-related environment variables and their current values. In this example, the variables are all set to the default `C` locale, but users can set these variables to specific country/language code combinations. For example, if one were to set the `LC_TIME` variable to `fr_CA`, programs would know to present time and date information formatted according to a French-speaking Canadian's expectations. And if one were to set the `LC_MESSAGES` variable to `zh_TW`, programs would know to present human-readable messages in Traditional Chinese. Setting the `LC_ALL` variable has the effect of changing every locale variable to the same value. The value of `LANG` is used as a default value for any locale variable that is unset. To see the list of available locales on a Unix system, run the command **`locale -a`**.

On Windows, locale configuration is done via the "Regional and Language Options" control panel item. There you can view and select the values of individual settings from the available locales, and even customize (at a sickening level of detail) several of the display formatting conventions.

7.2.2. Subversion 对区域设置的使用

The Subversion client, **svn**, honors the current locale configuration in two ways. First, it notices the value of the `LC_MESSAGES` variable and attempts to print all messages in the specified language. For example:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

This behavior works identically on both Unix and Windows systems. Note, though, that while your operating system might have support for a certain locale, the Subversion client still may not be able to speak the particular language. In order to produce localized messages, human volunteers must provide translations for each language. The translations are written using the GNU gettext package, which results in translation modules that end with the `.mo` filename extension. For example, the German translation file is named `de.mo`. These translation files are installed somewhere on your system. On Unix, they typically live in `/usr/share/locale/`, while on Windows they're often found in the `share\locale\` folder in Subversion's installation area. Once installed, a module is named after the program for which it provides translations. For example, the `de.mo` file may ultimately end up installed as `/usr/share/`

locale/de/LC_MESSAGES/subversion.mo. By browsing the installed .mo files, you can see which languages the Subversion client is able to speak.

The second way in which the locale is honored involves how **svn** interprets your input. The repository stores all paths, filenames, and log messages in Unicode, encoded as UTF-8. In that sense, the repository is *internationalized*—that is, the repository is ready to accept input in any human language. This means, however, that the Subversion client is responsible for sending only UTF-8 filenames and log messages into the repository. To do this, it must convert the data from the native locale into UTF-8.

For example, suppose you create a file named caffè.txt, and then when committing the file, you write the log message as "Adesso il caffè è più forte." Both the filename and the log message contain non-ASCII characters, but because your locale is set to it_IT, the Subversion client knows to interpret them as Italian. It uses an Italian character set to convert the data to UTF-8 before sending it off to the repository.

Note that while the repository demands UTF-8 filenames and log messages, it *does not* pay attention to file contents. Subversion treats file contents as opaque strings of bytes, and neither client nor server makes an attempt to understand the character set or encoding of the contents.

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:
…
```

Errors such as this typically occur when the Subversion client has received a UTF-8 string from the repository, but not all of the characters in that string can be represented using the encoding of the current locale. For example, if your locale is en_US but a collaborator has committed a Japanese filename, you're likely to see this error when you receive the file during an **svn update**.

The solution is either to set your locale to something that *can* represent the incoming UTF-8 data, or to change the filename or log message in the repository. (And don't forget to slap your collaborator's hand—projects should decide on common languages ahead of time so that all participants are using the same locale.)

7.3. 使用外置编辑器

The most obvious way to get data into Subversion is through the addition of files to version control, committing changes to those files, and so on. But other pieces of information besides merely versioned file data live in your Subversion repository. Some of these bits of information —commit log messages, lock comments, and some property values—tend to be textual in nature and are provided explicitly by users. Most of this information can be provided to the Subversion command-line client using the --message (-m) and --file (-F) options with the appropriate subcommands.

Each of these options has its pros and cons. For example, when performing a commit, `--file` (`-F`) works well if you've already prepared a text file that holds your commit log message. If you didn't, though, you can use `--message` (`-m`) to provide a log message on the command line. Unfortunately, it can be tricky to compose anything more than a simple one-line message on the command line. Users want more flexibility—multiline, free-form log message editing on demand.

Subversion supports this by allowing you to specify an external text editor that it will launch as necessary to give you a more powerful input mechanism for this textual metadata. There are several ways to tell Subversion which editor you'd like use. Subversion checks the following things, in the order specified, when it wants to launch such an editor:

1. 命令行选项`--editor-cmd`

2. `SVN_EDITOR` environment variable

3. `editor-cmd` runtime configuration option

4. `VISUAL` environment variable

5. `EDITOR` environment variable

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）

The value of any of these options or variables is the beginning of a command line to be executed by the shell. Subversion appends to that command line a space and the pathname of a temporary file to be edited. So, to be used with Subversion, the configured or specified editor needs to support an invocation in which its last command-line parameter is a file to be edited, and it should be able to save the file in place and return a zero exit code to indicate success.

As noted, external editors can be used to provide commit log messages to any of the committing subcommands (such as **svn commit** or **import**, **svn mkdir** or **delete** when provided a URL target, etc.), and Subversion will try to launch the editor automatically if you don't specify either of the `--message` (`-m`) or `--file` (`-F`) options. The **svn propedit** command is built almost entirely around the use of an external editor. And beginning in version 1.5, Subversion will also use the configured external text editor when the user asks it to launch an editor during interactive conflict resolution. Oddly, there doesn't appear to be a way to use external editors to interactively provide lock comments.

7.4. 使用外置比较与合并工具

The interface between Subversion and external two- and three-way differencing tools harkens back to a time when Subversion's only contextual differencing capabilities were built around invocations of the GNU diffutils toolchain, specifically the **diff** and **diff3** utilities. To get the kind of behavior Subversion needed, it called these utilities with more than a handful of options and parameters, most of which were quite specific to the utilities. Some time later, Subversion grew its own internal differencing library, and as a failover mechanism, the `--diff-cmd` and `--diff3-cmd` options were added to the Subversion command-line client so that users could more easily indicate that they preferred to use the GNU diff and diff3 utilities instead of the newfangled internal diff library. If those options were used, Subversion would simply ignore the internal diff library, and fall back to running those external programs, lengthy argument lists and all. And that's where things remain today.

It didn't take long for folks to realize that having such easy configuration mechanisms for specifying that Subversion should use the external GNU diff and diff3 utilities located at a particular place on the system could be applied toward the use of other differencing tools, too. After all, Subversion didn't actually verify that the things it was being told to run were members of the GNU diffutils toolchain. But the only configurable aspect of using those external tools is their location on the system—not the option set, parameter order, and so on. Subversion continues to throw all those GNU utility options at your external diff tool regardless of whether that program can understand those options. And that's where things get unintuitive for most users.

The key to using external two- and three-way differencing tools (other than GNU diff and diff3, of course) with Subversion is to use wrapper scripts, which convert the input from Subversion into something that your differencing tool can understand, and then to convert the output of your tool back into a format that Subversion expects—the format that the GNU tools would have used. The following sections cover the specifics of those expectations.

注意

The decision on when to fire off a contextual two- or three-way diff as part of a larger Subversion operation is made entirely by Subversion and is affected by, among other things, whether the files being operated on are human-readable as determined by their `svn:mime-type` property. This means, for example, that even if you had the niftiest Microsoft Word-aware differencing or merging tool in the universe, it would never be invoked by Subversion as long as your versioned Word documents had a configured MIME type that denoted that they were not human-readable (such as `application/msword`). For more about MIME type settings, see 第 3.3.1 节 "文件内容类型"

Subversion 1.5 introduces interactive resolution of conflicts (described in 第 2.4.5 节 "解决冲突（合并别人的修改）"), and one of the options provided to users is the ability to launch a third-party merge tool. If this action is taken, Subversion will consult the `merge-tool-cmd` runtime configuration option to find the name of an external merge tool and, upon finding one, will launch that tool with the appropriate input files. This differs from the configurable three-way differencing tool in a couple of ways. First, the differencing tool is always used to handle three-way differences, whereas the merge tool is employed only when three-way difference application has detected a conflict. Second, the interface is much cleaner—your configured merge tool need only accept as command-line parameters four path specifications: the base file, the "theirs" file (which contains upstream changes), the "mine" file (which contains local modifications), and the path of the file where the final resolved contents should be stored.

7.4.1. 外置 diff

Subversion calls external diff programs with parameters suitable for the GNU diff utility, and expects only that the external program will return with a successful error code. For most alternative diff programs, only the sixth and seventh arguments—the paths of the files that represent the left and right sides of the diff, respectively—are of interest. Note that Subversion runs the diff program once per modified file covered by the Subversion operation, so if your program runs in an asynchronous fashion (or is "backgrounded"), you might have several instances of it all running simultaneously. Finally, Subversion expects that your program return

an error code of 1 if your program detected differences, or 0 if it did not—any other error code is considered a fatal error. [3]

例 7.2 "diffwrap.py" and 例 7.3 "diffwrap.bat" are templates for external diff tool wrappers in the Python and Windows batch scripting languages, respectively.

例 7.2. diffwrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite diff program here.
DIFF = -"/usr/local/bin/my-diff-tool"

# Subversion provides the paths we need as the last two parameters.
LEFT  = sys.argv[-2]
RIGHT = sys.argv[-1]

# Call the diff command (change the following line to make sense for
# your diff program).
cmd = [DIFF, -'--left', LEFT, -'--right', RIGHT]
os.execv(cmd[0], cmd)

# Return an errorcode of 0 if no differences were detected, 1 if some were.
# Any other errorcode will be treated as fatal.
```

---

[3]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。"

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 last two
parameters.
REM These are parameters 6 and 7 (unless you use svn diff --x, in
REM which case, all bets are off).
SET LEFT=%6
SET RIGHT=%7

REM Call the diff command (change the following line to make
sense for
REM your diff 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.
```

7.4.2. 外置 diff3

Subversion calls external merge programs with parameters suitable for the GNU diff3 utility, expecting that the external program will return with a successful error code and that the full file contents that result from the completed merge operation are printed on the standard output stream (so that Subversion can redirect them into the appropriate version-controlled file). For most alternative merge programs, only the ninth, tenth, and eleventh arguments, the paths of the files which represent the "mine," "older," and "yours" inputs, respectively, are of interest. Note that because Subversion depends on the output of your merge program, your wrapper script must not exit before that output has been delivered to Subversion. When it finally does exit, it should return an error code of 0 if the merge was successful, or 1 if unresolved conflicts remain in the output—any other error code is considered a fatal error.

例 7.4 "diff3wrap.py" and 例 7.5 "diff3wrap.bat" are templates for external merge tool wrappers in the Python and Windows batch scripting languages, respectively.

例 7.4. diff3wrap.py

```python
#!/usr/bin/env python
import sys
import os

# Configure your favorite diff program here.
DIFF3 = -"/usr/local/bin/my-merge-tool"

# Subversion provides the paths we need as the last three
parameters.
MINE  = sys.argv[-3]
OLDER = sys.argv[-2]
YOURS = sys.argv[-1]

# Call the merge command (change the following line to make sense
for
# your merge program).
cmd = [DIFF3, -'--older', OLDER, -'--mine', MINE, -'--yours',
YOURS]
os.execv(cmd[0], cmd)

# 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 last three
parameters.
REM These are parameters 9, 10, and 11.  But we have access to only
REM nine parameters at a time, so we shift our nine-parameter
window
REM 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.
```

7.5. 总结

Sometimes there's a single right way to do things; sometimes there are many. Subversion's developers understand that while the majority of its exact behaviors are acceptable to most of its users, there are some corners of its functionality where such a universally pleasing approach doesn't exist. In those places, Subversion offers users the opportunity to tell it how *they* want it to behave.

In this chapter, we explored Subversion's runtime configuration system and other mechanisms by which users can control those configurable behaviors. If you are a developer, though, the next chapter will take you one step further. It describes how you can further customize your Subversion experience by writing your own software against Subversion's libraries.

第 8 章 嵌入 Subversion

Subversion has a modular design: it's implemented as a collection of libraries written in C. Each library has a well-defined purpose and application programming interface (API), and that interface is available not only for Subversion itself to use, but for any software that wishes to embed or otherwise programmatically control Subversion. Additionally, Subversion's API is available not only to other C programs, but also to programs written in higher-level languages such as Python, Perl, Java, and Ruby.

This chapter is for those who wish to interact with Subversion through its public API or its various language bindings. If you wish to write robust wrapper scripts around Subversion functionality to simplify your own life, are trying to develop more complex integrations between Subversion and other pieces of software, or just have an interest in Subversion's various library modules and what they offer, this chapter is for you. If, however, you don't foresee yourself participating with Subversion at such a level, feel free to skip this chapter with the confidence that your experience as a Subversion user will not be affected.

8.1. 分层的库设计

Each of Subversion's core libraries can be said to exist in one of three main layers—the Repository layer, the Repository Access (RA) layer, or the Client layer (see 图 1 "Subversion 的架构" in the Preface). We will examine these layers shortly, but first, let's briefly summarize Subversion's various libraries. For the sake of consistency, we will refer to the libraries by their extensionless Unix library names (`libsvn_fs`, `libsvn_wc`, `mod_dav_svn`, etc.).

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_local
 本地版本库访问模块

libsvn_ra_neon
> WebDAV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
> 工作副本管理库

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

The fact that the word "miscellaneous" appears only once in the previous list is a good sign. The Subversion development team is serious about making sure that functionality lives in the right layer and libraries. Perhaps the greatest advantage of the modular design is its lack of complexity from a developer's point of view. As a developer, you can quickly formulate that kind of "big picture" that allows you to pinpoint the location of certain pieces of functionality with relative ease.

Another benefit of modularity is the ability to replace a given module with a whole new library that implements the same API without affecting the rest of the code base. In some sense, this happens within Subversion already. The libsvn_ra_local, libsvn_ra_neon, libsvn_ra_serf, and libsvn_ra_svn libraries each implement the same interface, all working as plug-ins to libsvn_ra. And all four communicate with the Repository layer —libsvn_ra_local connects to the repository directly; the other three do so over a network. The libsvn_fs_base and libsvn_fs_fs libraries are another pair of libraries that implement the same functionality in different ways—both are plug-ins to the common libsvn_fs library.

The client itself also highlights the benefits of modularity in the Subversion design. Subversion's libsvn_client library is a one-stop shop for most of the functionality necessary for designing a working Subversion client (see 第 8.1.3 节 "客户端层"). So while the Subversion distribution provides only the **svn** command-line client program, several third-party programs provide various forms of graphical client UIs. These GUIs use the same APIs that the stock command-line client does. This type of modularity has played a large role in the proliferation of available Subversion clients and IDE integrations and, by extension, to the tremendous adoption rate of Subversion itself.

8.1.1. 版本库层

When referring to Subversion's Repository layer, we're generally talking about two basic concepts—the versioned filesystem implementation (accessed via `libsvn_fs`, and supported by its `libsvn_fs_base` and `libsvn_fs_fs` plug-ins), and the repository logic that wraps it (as implemented in `libsvn_repos`). These libraries provide the storage and reporting mechanisms for the various revisions of your version-controlled data. This layer is connected to the Client layer via the Repository Access layer, and is, from the perspective of the Subversion user, the stuff at the "other end of the line."

The Subversion filesystem is not a kernel-level filesystem that one would install in an operating system (such as the Linux ext2 or NTFS), but instead is a virtual filesystem. Rather than storing "files" and "directories" as real files and directories (the kind you can navigate through using your favorite shell program), it uses one of two available abstract storage backends—either a Berkeley DB database environment or a flat-file representation. (To learn more about the two repository backends, see 第 5.2.3 节 "选择数据存储格式".) There has even been considerable interest by the development community in giving future releases of Subversion the ability to use other backend database systems, perhaps through a mechanism such as Open Database Connectivity (ODBC). In fact, Google did something similar to this before launching the Google Code Project Hosting service: they announced in mid-2006 that members of its open source team had written a new proprietary Subversion filesystem plug-in that used Google's ultra-scalable Bigtable database for its storage.

The filesystem API exported by `libsvn_fs` contains the kinds of functionality you would expect from any other filesystem API—you can create and remove files and directories, copy and move them around, modify file contents, and so on. It also has features that are not quite as common, such as the ability to add, modify, and remove metadata ("properties") on each file or directory. Furthermore, the Subversion filesystem is a versioning filesystem, which means that as you make changes to your directory tree, Subversion remembers what your tree looked like before those changes. And before the previous changes. And the previous ones. And so on, all the way back through versioning time to (and just beyond) the moment you first started adding things to the filesystem.

All the modifications you make to your tree are done within the context of a Subversion commit transaction. The following is a simplified general routine for modifying your filesystem:

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。

Once you have committed your transaction, your filesystem modifications are permanently stored as historical artifacts. Each of these cycles generates a single new revision of your tree, and each revision is forever accessible as an immutable snapshot of "the way things were."

事务的其它信息

The notion of a Subversion transaction can become easily confused with the transaction support provided by the underlying database itself, especially given the former's close proximity to the Berkeley DB database code in `libsvn_fs_base`. Both types of transaction exist to provide atomicity and isolation. In other words, transactions give you the ability to perform a set of actions in an all-or-nothing fashion—either all the actions in the set complete with success, or they all get treated as though *none* of them ever happened —and in a way that does not interfere with other processes acting on the data.

Database transactions generally encompass small operations related specifically to the modification of data in the database itself (such as changing the contents of a table row). Subversion transactions are larger in scope, encompassing higher-level operations such as making modifications to a set of files and directories that are intended to be stored as the next revision of the filesystem tree. If that isn't confusing enough, consider the fact that Subversion uses a database transaction during the creation of a Subversion transaction (so that if the creation of a Subversion transaction fails, the database will look as though we had never attempted that creation in the first place)!

Fortunately for users of the filesystem API, the transaction support provided by the database system itself is hidden almost entirely from view (as should be expected from a properly modularized library scheme). It is only when you start digging into the implementation of the filesystem itself that such things become visible (or interesting).

Most of the functionality the filesystem interface provides deals with actions that occur on individual filesystem paths. That is, from outside the filesystem, the primary mechanism for describing and accessing the individual revisions of files and directories comes through the use of path strings such as `/foo/bar`, just as though you were addressing files and directories through your favorite shell program. You add new files and directories by passing their paths-to-be to the right API functions. You query for information about them by the same mechanism.

Unlike most filesystems, though, a path alone is not enough information to identify a file or directory in Subversion. Think of a directory tree as a two-dimensional system, where a node's siblings represent a sort of left-and-right motion, and navigating into the node's subdirectories represents a downward motion. 图 8.1 "二维的文件和目录" shows a typical representation of a tree as exactly that.

图 **8.1.** 二维的文件和目录
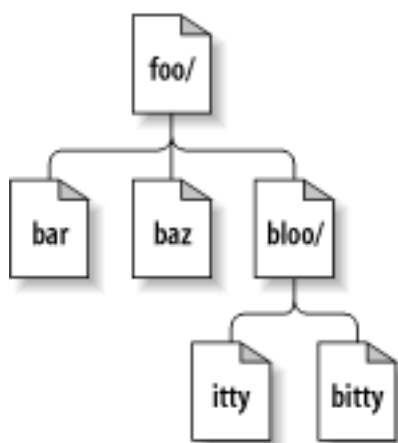
The difference here is that the Subversion filesystem has a nifty third dimension that most filesystems do not have—Time! [1] In the filesystem interface, nearly every function that has a *path* argument also expects a *root* argument. This `svn_fs_root_t` argument describes either a revision or a Subversion transaction (which is simply a revision in the making) and provides that third dimension of context needed to understand the difference between `/foo/bar` in revision 32, and the same path as it exists in revision 98. 图 8.2 "版本时间 - 第三维！" shows revision history as an added dimension to the Subversion filesystem universe.

图 **8.2.** 版本时间 - 第三维！
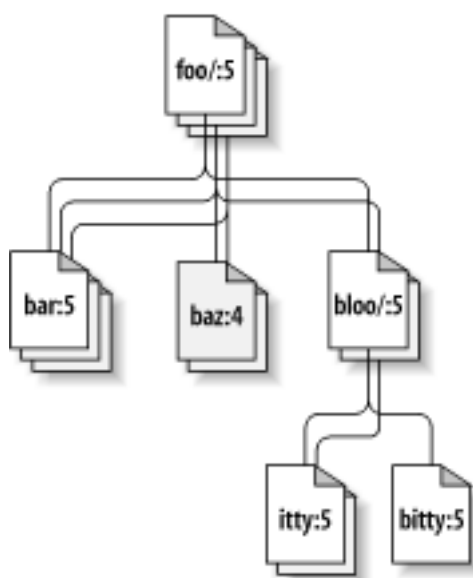
As we mentioned earlier, the `libsvn_fs` API looks and feels like any other filesystem, except that it has this wonderful versioning capability. It was designed to be usable by any program interested in a versioning filesystem. Not coincidentally, Subversion itself is interested in that functionality. But while the filesystem API should be sufficient for basic file and directory versioning support, Subversion wants more—and that is where `libsvn_repos` comes in.

---

[1] 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。

The Subversion repository library (`libsvn_repos`) sits (logically speaking) atop the `libsvn_fs` API, providing additional functionality beyond that of the underlying versioned filesystem logic. It does not completely wrap each and every filesystem function—only certain major steps in the general cycle of filesystem activity are wrapped by the repository interface. Some of these include the creation and commit of Subversion transactions and the modification of revision properties. These particular events are wrapped by the repository layer because they have hooks associated with them. A repository hook system is not strictly related to implementing a versioning filesystem, so it lives in the repository wrapper library.

The hooks mechanism is but one of the reasons for the abstraction of a separate repository library from the rest of the filesystem code. The `libsvn_repos` API provides several other important utilities to Subversion. These include the abilities to:

- 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

- Generate a human-readable "dump" of the filesystem—a complete representation of the revisions in the filesystem.

- 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。

8.1.2. 版本库访问层

If the Subversion Repository layer is at "the other end of the line," the Repository Access (RA) layer is the line itself. Charged with marshaling data between the client libraries and the repository, this layer includes the `libsvn_ra` module loader library, the RA modules themselves (which currently includes `libsvn_ra_neon`, `libsvn_ra_local`, `libsvn_ra_serf`, and `libsvn_ra_svn`), and any additional libraries needed by one or more of those RA modules (such as the `mod_dav_svn` Apache module or `libsvn_ra_svn`'s server, **svnserve**).

Since Subversion uses URLs to identify its repository resources, the protocol portion of the URL scheme (usually `file://`, `http://`, `https://`, `svn://`, or `svn+ssh://`) is used to determine which RA module will handle the communications. Each module registers a list of the protocols it knows how to "speak" so that the RA loader can, at runtime, determine which module to use for the task at hand. You can determine which RA modules are available to the Subversion command-line client, and what protocols they claim to support, by running **svn --version**:

```
$ svn ---version
svn, version 1.5.0 (r31699)
   compiled Jun 18 2008, 09:57:36
```

```
Copyright (C) 2000-2008 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_neon -: Module for accessing a repository via WebDAV
protocol using Neon.
  -- 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_serf -: Module for accessing a repository via WebDAV
protocol using serf.
  -- handles -'http' scheme
  -- handles -'https' scheme


$
```

The public API exported by the RA layer contains functionality necessary for sending and receiving versioned data to and from the repository. And each of the available RA plug-ins is able to perform that task using a specific protocol—`libsvn_ra_dav` speaks HTTP/WebDAV (optionally using SSL encryption) with an Apache HTTP Server that is running the `mod_dav_svn` Subversion server module; `libsvn_ra_svn` speaks a custom network protocol with the **svnserve** program; and so on.

For those who wish to access a Subversion repository using still another protocol, that is precisely why the Repository Access layer is modularized! Developers can simply write a new library that implements the RA interface on one side and communicates with the repository on the other. Your new library can use existing network protocols or you can invent your own. You could use interprocess communication (IPC) calls, or—let's get crazy, shall we?—you could even implement an email-based protocol. Subversion supplies the APIs; you supply the creativity.

8.1.3. 客户端层

On the client side, the Subversion working copy is where all the action takes place. The bulk of functionality implemented by the client-side libraries exists for the sole purpose of managing working copies—directories full of files and other subdirectories that serve as a sort of local, editable "reflection" of one or more repository locations—and propagating changes to and from the Repository Access layer.

Subversion's working copy library, `libsvn_wc`, is directly responsible for managing the data in the working copies. To accomplish this, the library stores administrative information

about each working copy directory within a special subdirectory. This subdirectory, named `.svn`, is present in each working copy directory and contains various other files and directories that record state and provide a private workspace for administrative action. For those familiar with CVS, this `.svn` subdirectory is similar in purpose to the `CVS` administrative directories found in CVS working copies. For more information about the `.svn` administrative area, see 第 8.2 节 "进入工作副本的管理区" later in this chapter.

The Subversion client library, `libsvn_client`, has the broadest responsibility; its job is to mingle the functionality of the working copy library with that of the Repository Access layer, and then to provide the highest-level API to any application that wishes to perform general revision control actions. For example, the function `svn_client_checkout()` takes a URL as an argument. It passes this URL to the RA layer and opens an authenticated session with a particular repository. It then asks the repository for a certain tree, and sends this tree into the working copy library, which then writes a full working copy to disk (`.svn` directories and all).

The client library is designed to be used by any application. While the Subversion source code includes a standard command-line client, it should be very easy to write any number of GUI clients on top of the client library. New GUIs (or any new client, really) for Subversion need not be clunky wrappers around the included command-line client—they have full access via the `libsvn_client` API to the same functionality, data, and callback mechanisms that the command-line client uses. In fact, the Subversion source code tree contains a small C program (which you can find at `tools/examples/minimal_client.c`) that exemplifies how to wield the Subversion API to create a simple client program.

直接绑定 - 关于正确性

Why should your GUI program bind directly with a `libsvn_client` instead of acting as a wrapper around a command-line program? Besides simply being more efficient, it can be more correct as well. A command-line program (such as the one supplied with Subversion) that binds to the client library needs to effectively translate feedback and requested data bits from C types to some form of human-readable output. This type of translation can be lossy. That is, the program may not display all of the information harvested from the API or may combine bits of information for compact representation.

If you wrap such a command-line program with yet another program, the second program has access only to already interpreted (and as we mentioned, likely incomplete) information, which it must *again* translate into *its* representation format. With each layer of wrapping, the integrity of the original data is potentially tainted more and more, much like the result of making a copy of a copy (of a copy...) of a favorite audio or video cassette.

But the most compelling argument for binding directly to the APIs instead of wrapping other programs is that the Subversion project makes compatibility promises regarding its APIs. Across minor versions of those APIs (such as between 1.3 and 1.4), no function's prototype will change. In other words, you aren't forced to update your program's source code simply because you've upgraded to a new version of Subversion. Certain functions might be deprecated, but they still work, and this gives you a buffer of time to eventually embrace the newer APIs. These kinds of compatibility promises do not exist for Subversion command-line program output, which is subject to change from release to release.

8.2. 进入工作副本的管理区

像我们前面提到的，每个 Subversion 工作副本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副本目录的管理数据。Subversion 使用 `.svn` 中的信息来追踪如下的数据：

- 工作副本中展示的目录和文件在版本库中的位置

- 工作副本中当前展示的文件和目录的版本、

- 所有可能附加在文件和目录上的用户定义的属性。

- 工作副本文件的原始(未编辑)的版本。

The Subversion working copy administration area's layout and contents are considered implementation details not really intended for human consumption. Developers are encouraged to use Subversion's public APIs, or the tools that Subversion provides, to access and manipulate the working copy data, instead of directly reading or modifying those files. The file formats employed by the working copy library for its administrative data do change from time to time—a fact that the public APIs do a great job of hiding from the average user. In this section, we expose some of these implementation details sheerly to appease your overwhelming curiosity.

8.2.1. 条目文件

Perhaps the single most important file in the `.svn` directory is the `entries` file. It contains the bulk of the administrative information about the versioned items in a working copy directory. This one file tracks the repository URLs, pristine revision, file checksums, pristine text and property timestamps, scheduling and conflict state information, last-known commit information (author, revision, timestamp), local copy history—practically everything that a Subversion client is interested in knowing about a versioned (or to-be-versioned) resource!

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。

The format of the `.svn/entries` file has changed over time. Originally an XML file, it now uses a custom—though still human-readable—file format. While XML was a great choice for early developers of Subversion who were frequently debugging the file's contents (and Subversion's behavior in light of them), the need for easy developer debugging has diminished as Subversion has matured and has been replaced by the user's need for snappier performance. Be aware that Subversion's working copy library automatically upgrades working copies from one format to another—it reads the old formats and writes the new—which saves you the hassle of checking out a new working copy, but can also complicate situations where different versions of Subversion might be trying to use the same working copy.

8.2.2. 原始副本和属性文件

As mentioned before, the `.svn` directory also holds the pristine "text-base" versions of files. You can find those in `.svn/text-base`. The benefits of these pristine copies are multiple—

network-free checks for local modifications and difference reporting, network-free reversion of modified or missing files, more efficient transmission of changes to the server—but they come at the cost of having each versioned file stored at least twice on disk. These days, this seems to be a negligible penalty for most files. However, the situation gets uglier as the size of your versioned files grows. Some attention is being given to making the presence of the "text-base" an option. Ironically, though, it is as your versioned files' sizes get larger that the existence of the "text-base" becomes more crucial—who wants to transmit a huge file across a network just because she wants to commit a tiny change to it?

Similar in purpose to the "text-base" files are the property files and their pristine "prop-base" copies, located in `.svn/props` and `.svn/prop-base`, respectively. Since directories can have properties too, there are also `.svn/dir-props` and `.svn/dir-prop-base` files.

8.3. 使用 API

Developing applications against the Subversion library APIs is fairly straightforward. Subversion is primarily a set of C libraries, with header (`.h`) files that live in the `subversion/include` directory of the source tree. These headers are copied into your system locations (e.g., `/usr/local/include`) when you build and install Subversion itself from source. These headers represent the entirety of the functions and types meant to be accessible by users of the Subversion libraries. The Subversion developer community is meticulous about ensuring that the public API is well documented—refer directly to the header files for that documentation.

When examining the public header files, the first thing you might notice is that Subversion's datatypes and functions are namespace-protected. That is, every public Subversion symbol name begins with `svn_`, followed by a short code for the library in which the symbol is defined (such as `wc`, `client`, `fs`, etc.), followed by a single underscore (`_`), and then the rest of the symbol name. Semipublic functions (used among source files of a given library but not by code outside that library, and found inside the library directories themselves) differ from this naming scheme in that instead of a single underscore after the library code, they use a double underscore (`__`). Functions that are private to a given source file have no special prefixing and are declared `static`. Of course, a compiler isn't interested in these naming conventions, but they help to clarify the scope of a given function or datatype.

Another good source of information about programming against the Subversion APIs is the project's own hacking guidelines, which you can find at http://subversion.tigris.org/hacking.html. This document contains useful information, which, while aimed at developers and would-be developers of Subversion itself, is equally applicable to folks developing against Subversion as a set of third-party libraries. [2]

8.3.1. Apache 可移植运行库

Along with Subversion's own datatypes, you will see many references to datatypes that begin with `apr_`—symbols from the Apache Portable Runtime (APR) library. APR is Apache's portability library, originally carved out of its server code as an attempt to separate the OS-

---

[2] 当然，Subversion使用Subversion的API。

specific bits from the OS-independent portions of the code. The result was a library that provides a generic API for performing operations that differ mildly—or wildly—from OS to OS. While the Apache HTTP Server was obviously the first user of the APR library, the Subversion developers immediately recognized the value of using APR as well. This means that there is practically no OS-specific code in Subversion itself. Also, it means that the Subversion client compiles and runs anywhere that the Apache HTTP Server does. Currently, this list includes all flavors of Unix, Win32, BeOS, OS/2, and Mac OS X.

In addition to providing consistent implementations of system calls that differ across operating systems, [3] APR gives Subversion immediate access to many custom datatypes, such as dynamic arrays and hash tables. Subversion uses these types extensively. But perhaps the most pervasive APR datatype, found in nearly every Subversion API prototype, is the `apr_pool_t`— the APR memory pool. Subversion uses pools internally for all its memory allocation needs (unless an external library requires a different memory management mechanism for data passed through its API), [4] and while a person coding against the Subversion APIs is not required to do the same, she *is* required to provide pools to the API functions that need them. This means that users of the Subversion API must also link against APR, must call `apr_initialize()` to initialize the APR subsystem, and then must create and manage pools for use with Subversion API calls, typically by using `svn_pool_create()`, `svn_pool_clear()`, and `svn_pool_destroy()`.

使用内存池编程

Almost every developer who has used the C programming language has at some point sighed at the daunting task of managing memory usage. Allocating enough memory to use, keeping track of those allocations, freeing the memory when you no longer need it—these tasks can be quite complex. And of course, failure to do those things properly can result in a program that crashes itself, or worse, crashes the computer.

Higher-level languages, on the other hand, either take the job of memory management away from you completely or make it something you toy with only when doing extremely tight program optimization. Languages such as Java and Python use *garbage collection*, allocating memory for objects when needed, and automatically freeing that memory when the object is no longer in use.

APR provides a middle-ground approach called *pool-based memory management*. It allows the developer to control memory usage at a lower resolution—per chunk (or "pool") of memory, instead of per allocated object. Rather than using `malloc()` and friends to allocate enough memory for a given object, you ask APR to allocate the memory from a memory pool. When you're finished using the objects you've created in the pool, you destroy the entire pool, effectively de-allocating the memory consumed by *all* the objects you allocated from it. Thus, rather than keeping track of individual objects that need to be de-allocated, your program simply considers the general lifetimes of those objects and allocates the objects in a pool whose lifetime (the time between the pool's creation and its deletion) matches the object's needs.

---

[3]Subversion使用尽可能多ANSI系统调用和数据类型。
[4]Neon和Berkeley DB就是这种库的例子。

8.3.2. URL 和路径需求

With remote version control operation as the whole point of Subversion's existence, it makes sense that some attention has been paid to internationalization (i18n) support. After all, while "remote" might mean "across the office," it could just as well mean "across the globe." To facilitate this, all of Subversion's public interfaces that accept path arguments expect those paths to be canonicalized—which is most easily accomplished by passing them through the `svn_path_canonicalize()` function—and encoded in UTF-8. This means, for example, that any new client binary that drives the `libsvn_client` interface needs to first convert paths from the locale-specific encoding to UTF-8 before passing those paths to the Subversion libraries, and then reconvert any resultant output paths from Subversion back into the locale's encoding before using those paths for non-Subversion purposes. Fortunately, Subversion provides a suite of functions (see `subversion/include/svn_utf.h`) that any program can use to do these conversions.

Also, Subversion APIs require all URL parameters to be properly URI-encoded. So, instead of passing `file:///home/username/My  File.txt` as the URL of a file named `My    File.txt`, you need to pass `file:///home/username/My %20File.txt`. Again, Subversion supplies helper functions that your application can use—`svn_path_uri_encode()` and `svn_path_uri_decode()`, for URI encoding and decoding, respectively.

8.3.3. 使用 C 和 C++ 以外的语言

If you are interested in using the Subversion libraries in conjunction with something other than a C program—say, a Python or Perl script—Subversion has some support for this via the Simplified Wrapper and Interface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig`. They are still maturing, but they are usable. These bindings allow you to call Subversion API functions indirectly, using wrappers that translate the datatypes native to your scripting language into the datatypes needed by Subversion's C libraries.

Significant efforts have been made toward creating functional SWIG-generated bindings for Python, Perl, and Ruby. To some extent, the work done preparing the SWIG interface files for these languages is reusable in efforts to generate bindings for other languages supported by SWIG (which include versions of C#, Guile, Java, MzScheme, OCaml, PHP, and Tcl, among others). However, some extra programming is required to compensate for complex APIs that SWIG needs some help translating between languages. For more information on SWIG itself, see the project's web site at http://www.swig.org/.

Subversion also has language bindings for Java. The javahl bindings (located in `subversion/bindings/java` in the Subversion source tree) aren't SWIG-based, but are instead a mixture of Java and hand-coded JNI. Javahl covers most Subversion client-side APIs and is specifically targeted at implementors of Java-based Subversion clients and IDE integrations.

Subversion's language bindings tend to lack the level of developer attention given to the core Subversion modules, but can generally be trusted as production-ready. A number of scripts and applications, alternative Subversion GUI clients, and other third-party tools are successfully using Subversion's language bindings today to accomplish their Subversion integrations.

It's worth noting here that there are other options for interfacing with Subversion using other languages: alternative bindings for Subversion that aren't provided by the Subversion development community at all. You can find links to these alternative bindings on the Subversion project's links page (at http://subversion.tigris.org/links.html), but there are a couple of popular ones we feel are especially noteworthy. First, Barry Scott's PySVN bindings (http://pysvn.tigris.org/) are a popular option for binding with Python. PySVN boasts of a more Pythonic interface than the more C-like APIs provided by Subversion's own Python bindings. And if you're looking for a pure Java implementation of Subversion, check out SVNKit (http://svnkit.com/), which is Subversion rewritten from the ground up in Java.

SVNKit 与 javahl

In 2005, a small company called TMate announced the 1.0.0 release of JavaSVN—a pure Java implementation of Subversion. Since then, the project has been renamed to SVNKit (available at http://svnkit.com/) and has seen great success as a provider of Subversion functionality to various Subversion clients, IDE integrations, and other third-party tools.

The SVNKit library is interesting in that, unlike the javahl library, it is not merely a wrapper around the official Subversion core libraries. In fact, it shares no code with Subversion at all. But while it is easy to confuse SVNKit with javahl, and easier still to not even realize which of these libraries you are using, folks should be aware that SVNKit differs from javahl in some significant ways. First, SVNKit is not developed as open source software and seems to have at any given time only a few developers working on it. Also, SVNKit's license is more restrictive than that of Subversion. Finally, by aiming to be a pure Java Subversion library, SVNKit is limited in which portions of Subversion can be reasonably cloned while still keeping up with Subversion's releases. This has already happened once—SVNKit cannot access BDB-backed Subversion repositories via the `file://` protocol because there's no pure Java implementation of Berkeley DB that is file-format-compatible with the native implementation of that library.

That said, SVNKit has a well-established track record of reliability. And a pure Java solution is much more robust in the face of programming errors—a bug in SVNKit might raise a catchable Java Exception, but a bug in the Subversion core libraries as accessed via javahl can bring down your entire Java Runtime Environment. So, weigh the costs when choosing a Java-based Subversion implementation.

8.3.4. 代码样例

例 8.1 "使用版本库层" contains a code segment (written in C) that illustrates some of the concepts we've been discussing. It uses both the repository and filesystem interfaces (as can be determined by the prefixes `svn_repos_` and `svn_fs_` of the function names, respectively) to create a new revision in which a directory is added. You can see the use of an APR pool, which is passed around for memory allocation purposes. Also, the code reveals a somewhat obscure fact about Subversion error handling—all Subversion errors must be explicitly handled to avoid memory leakage (and in some cases, application failure).

```
pool));

    -/* Now that we have started a new Subversion transaction, get
a root
```

```
     * object that represents that transaction.
     */
```

例 **8.1.** 使用版本库层

```
    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);
}
```

Note that in 例 8.1 "使用版本库层", the code could just as easily have committed the transaction using `svn_fs_commit_txn()`. But the filesystem API knows nothing about the repository library's hook mechanism. If you want your Subversion repository to automatically perform some set of non-Subversion tasks every time you commit a transaction (e.g., sending an email that describes all the changes made in that transaction to your developer mailing list), you need to use the `libsvn_repos`-wrapped version of that function, which adds the hook triggering functionality—in this case, `svn_repos_fs_commit_txn()`. (For more information regarding Subversion's repository hooks, see 第 5.3.2 节 "实现版本库钩子".)

Now let's switch languages. 例 8.2 "使用 Python 处理版本库层" is a sample program that uses Subversion's SWIG Python bindings to recursively crawl the youngest repository revision, and to print the various paths reached during the crawl.

```
        # Print the name of this path.
        print directory + -"/"

        # Get the directory entries for DIRECTORY.
        entries = svn.fs.svn_fs_dir_entries(root, directory)
```

例 8.2. 使用 Python 处理版本库层

```
        # 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)
```

This same program in C would need to deal with APR's memory pool system. But Python handles memory usage automatically, and Subversion's Python bindings adhere to that convention. In C, you'd be working with custom datatypes (such as those provided by the APR library) for representing the hash of entries and the list of paths, but Python has hashes (called "dictionaries") and lists as built-in datatypes, and it provides a rich collection of functions for operating on those types. So SWIG (with the help of some customizations in Subversion's language bindings layer) takes care of mapping those custom datatypes into the native datatypes of the target language. This provides a more intuitive interface for users of that language.

The Subversion Python bindings can be used for working copy operations, too. In the previous section of this chapter, we mentioned the `libsvn_client` interface and how it exists for the sole purpose of simplifying the process of writing a Subversion client. 例 8.3 "一个 Python 状态爬虫" is a brief example of how that library can be accessed via the SWIG Python bindings to re-create a scaled-down version of the **svn status** command.

```
          # Print the path, minus the bit that overlaps with the
root of
          # the status crawl 嵌入 Subversion
```

例 8.3. 一个 Python 状态爬虫

```
          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)

    # Do the status crawl, using _status_callback() as our
callback function.
    revision = svn.core.svn_opt_revision_t()
    revision.type = svn.core.svn_opt_revision_head
    svn.client.svn_client_status2(wc_path, revision,
_status_callback,
                                  svn.core.svn_depth_infinity,
verbose,
                                  0, 0, 1, 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.apr_err,
e.message))
        sys.exit(1)
```

As was the case in 例 8.2 "使用 Python 处理版本库层", this program is pool-free and uses, for the most part, normal Python datatypes. The call to `svn_client_ctx_t()` is deceiving because the public Subversion API has no such function—this just happens to be a case where SWIG's automatic language generation bleeds through a little bit (the function is a sort of factory function for Python's version of the corresponding complex C structure). Also note that the path passed to this program (like the last one) gets run through `svn_path_canonicalize()`, because to *not* do so runs the risk of triggering the underlying Subversion C library's assertions about such things, which translates into rather immediate and unceremonious program abortion.

8.4. 总结

One of Subversion's greatest features isn't something you get from running its command-line client or other tools. It's the fact that Subversion was designed modularly and provides a stable, public API so that others—like yourself, perhaps—can write custom software that drives Subversion's core logic.

In this chapter, we took a closer look at Subversion's architecture, examining its logical layers and describing that public API, the very same API that Subversion's own layers use to communicate with each other. Many developers have found interesting uses for the Subversion API, from simple repository hook scripts, to integrations between Subversion and some other application, to completely different version control systems. What unique itch will *you* scratch with it?

第 9 章 Subversion 完全参考

This chapter is intended to be a complete reference to using Subversion. This includes the command-line client (**svn**) and all its subcommands, as well as the repository administration programs (**svnadmin** and **svnlook**) and their respective subcommands.

9.1. Subversion 命令行客户端：svn

To use the command-line client, type **svn**, the subcommand you wish to use, [1] and any options or targets that you wish to operate on—the subcommand and the options need not appear in a specific order. For example, all of the following are valid ways to use **svn status**:

```
$ 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，以及第 3.2 节 "属性" 中的管理属性的命令。

9.1.1. svn 选项

While Subversion has different options for its subcommands, all options exist in a single namespace—that is, each option is guaranteed to mean the same thing regardless of the subcommand you use it with. For example, `--verbose` (`-v`) always means "verbose output," regardless of the subcommand you use it with.

The **svn** command-line client usually exits quickly with an error if you pass it an option which does not apply to the specified subcommand. But as of Subversion 1.5, several of the options which apply to all—or nearly all—of the subcommands have been deemed acceptable by all subcommands, even if they have no effect on some of them. They appear grouped together in the command-line client's usage messages as global options. This was done to assist folks who write scripts which wrap the command-line client. These global options are as follows:

`--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`
Prevents caching of authentication information (e.g., username and password) in the Subversion runtime configuration directories.

`--non-interactive`
Disables all interactive prompting. Some examples of interactive prompting include requests for authentication credentials and conflict resolution decisions. This is useful if you're

---
[1]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。

running Subversion inside an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--password` *PASSWD*
> Specifies the password to use when authenticating against a Subversion server. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--username` *NAME*
> Specifies the username to use when authenticating against a Subversion server. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

The rest of the options apply and are accepted by only a subset of the subcommand. They are as follows:

`--accept` *ACTION*
> Specifies an action for automatic conflict resolution. Possible actions are `postpone`, `base`, `mine-full`, `theirs-full`, `edit`, and `launch`.

`--auto-props`
> 开启 auto-props，覆盖 config 文件中的 enable-auto-props 指示。

`--change` (`-c`) *ARG*
> Used as a means to refer to a specific "change" (a.k.a. a revision). This option is syntactic sugar for "-r ARG-1:ARG".

`--changelist` *ARG*
> Instructs Subversion to operate only on members of the changelist named *ARG*. You can use this option multiple times to specify sets of changelists.

`--cl` *ARG*
> 选项 `--changelist` 的别名。

`--depth` *ARG*
> Instructs Subversion to limit the scope of an operation to a particular tree depth. *ARG* is one of `empty`, `files`, `immediates`, or `infinity`.

`--diff-cmd` *CMD*
> Specifies an external program to use to show differences between files. When **svn diff** is invoked without this option, it uses Subversion's internal diff engine, which provides unified diffs by default. If you want to use an external diff program, use `--diff-cmd`. You can pass options to the diff program with the `--extensions` option (more on that later in this section).

`--diff3-cmd` *CMD*
> 指定一个外置程序用来合并文件。

`--dry-run`
> Goes through all the motions of running a command, but makes no actual changes—either on disk or in the repository.

`--editor-cmd` *CMD*

Specifies an external program to use to edit a log message or a property value. See the `editor-cmd` section in 第 7.1.3.2 节 "配置" for ways to specify a default editor.

`--encoding` *ENC*

Tells Subversion that your commit message is encoded in the charset provided. The default is your operating system's native locale, and you should specify the encoding if your commit message is in any other encoding.

`--extensions` (`-x`) *ARGS*

Specifies an argument or arguments that Subversion should pass to an external diff command. This option is valid only when used with the **svn diff** or **svn merge** commands, with the `--diff-cmd` option. If you wish to pass multiple arguments, you must enclose all of them in quotes (e.g., **svn diff --diff-cmd /usr/bin/diff -x "-b -E"**).

`--file` (`-F`) *FILENAME*

Uses the contents of the named file for the specified subcommand, though different subcommands do different things with this content. For example, **svn commit** uses the content as a commit log, whereas **svn propset** uses it as a property value.

`--force`

Forces a particular command or operation to run. Subversion will prevent you from performing some operations in normal usage, but you can pass the force option to tell Subversion "I know what I'm doing as well as the possible repercussions of doing it, so let me at 'em." This option is the programmatic equivalent of doing your own electrical work with the power on—if you don't know what you're doing, you're likely to get a nasty shock.

`--force-log`

Forces a suspicious parameter passed to the `--message` (`-m`) or `--file` (`-F`) option to be accepted as valid. By default, Subversion will produce an error if parameters to these options look like they might instead be targets of the subcommand. For example, if you pass a versioned file's path to the `--file` (`-F`) option, Subversion will assume you've made a mistake, that the path was instead intended as the target of the operation, and that you simply failed to provide some other—unversioned—file as the source of your log message. To assert your intent and override these types of errors, pass the `--force-log` option to subcommands that accept log messages.

`--help` (`-h`或`-?`)

If used with one or more subcommands, shows the built-in help text for each. If used alone, it displays the general client help text.

`--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`

Tells Subversion to ignore externals definitions and the external working copies managed by them.

`--incremental`

打印适合串联的输出格式。

`--keep-changelists`
告诉 Subversion 在提交后不要删除修改集。

`--keep-local`
保留文件或目录的版本版本 (用于命令 **svn delete**)。

`--limit (-l)` *NUM*
Shows only the first *NUM* log messages.

`--message (-m)` *MESSAGE*
Indicates that you will specify either a log message or a lock comment on the command line, following this option. For example:

```
$ 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-auto-props`
关闭**auto-**props，覆盖config文件中的enable-auto-props指示。

`--no-diff-deleted`
Prevents Subversion from printing differences for deleted files. The default behavior when you remove a file is for **svn diff** to print the same differences that you would see if you had left the file but removed all the content.

`--no-ignore`
Shows files in the status listing that would normally be omitted since they match a pattern in the global-ignores configuration option or the svn:ignore property. See 第 7.1.3.2 节 "配置" and 第 3.4 节 "忽略未版本控制的条目" for more information.

`--no-unlock`
Tells Subversion not to automatically unlock files (the default commit behavior is to unlock all files listed as part of the commit). See 第 3.7 节 "锁定" for more information.

`--non-recursive (-N)`
Deprecated. Stops a subcommand from recursing into subdirectories. Most subcommands recurse by default, but some subcommands—usually those that have the potential to remove or undo your local modifications—do not.

`--notice-ancestry`
在计算差异时关注祖先。

`--old` *ARG*
使用*ARG*作为旧的目标(结合**svn diff**使用)。

`--parents`
Creates and adds nonexistent or nonversioned parent subdirectories to the working copy or repository as part of an operation. This is useful for automatically creating multiple subdirectories where none currently exist. If performed on a URL, all the directories will be created in a single commit.

`--quiet` (`-q`)
　　请求客户端在执行操作时只显示重要信息。

`--record-only`
　　标记版本已合并(使用 `--revision`)。

`--recursive` (`-R`)
　　让子命令递归子目录，大多数子命令缺省是递归的。

`--reintegrate`
　　Used with the **svn merge** subcommand, merges all of the source URL's changes into the working copy. See 第 4.3.2 节 "保持分支同步" for details.

`--relocate` *FROM TO [PATH...]*
　　使用子命令 **svn switch**，修改你的工作副本所引用的版本库位置。当版本库的位置修改了，而你有一个工作副本，希望继续使用时非常有用。见**svn switch**的例子。

`--remove` *ARG*
　　从变更列表 *ARG* 清除

`--revision` (`-r`) *REV*
　　Indicates that you're going to supply a revision (or range of revisions) for a particular operation. You can provide revision numbers, keywords, or dates (in curly braces) as arguments to the revision option. If you wish to offer a range of revisions, you can provide two revisions separated by a colon. For example:

```
$ 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}
```

　　见第 3.1.1 节 "修订版本关键字" 查看更多信息。

`--revprop`
　　Operates on a revision property instead of a property specific to a file or directory. This option requires that you also pass a revision with the `--revision` (`-r`) option.

`--set-depth` *ARG*
　　Sets the sticky depth on a directory in a working copy to one of `empty`, `files`, `immediates`, or `infinity`.

`--show-revs` *ARG*
　　Used to make **svn mergeinfo** display either `merged` or `eligible` revisions.

`--show-updates` (`-u`)
　　Causes the client to display information about which files in your working copy are out of date. This doesn't actually update any of your files—it just shows you which files will be updated if you then use **svn update**.

`--stop-on-copy`
> Causes a Subversion subcommand that traverses the history of a versioned resource to stop harvesting that historical information when a copy—that is, a location in history where that resource was copied from another location in the repository—is encountered.

`--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。

`--use-merge-history` (`-g`)
> 从合并历史使用或显示额外的信息。

`--verbose` (`-v`)
> Requests that the client print out as much information as it can while running any subcommand. This may result in Subversion printing out additional fields, detailed information about every file, or additional information regarding its actions.

`--version`
> Prints the client version info. This information includes not only the version number of the client, but also a listing of all repository access modules that the client can use to access a Subversion repository. With `--quiet` (`-q`) it prints only the version number in a compact form.

`--with-all-revprops`
> Used with the `--xml` option to **svn log**, will retrieve and display all revision properties in the log output.

`--with-revprop` *ARG*
> When used with any command that writes to the repository, sets the revision property, using the *NAME=VALUE* format, *NAME* to *VALUE*. When used with **svn log** in `--xml` mode, this displays the value of *ARG* in the log output.

`--xml`
> 使用XML格式打印输出。

9.1.2. svn 子命令

Here are the various subcommands for the **svn** program. For the sake of brevity, we omit the global options (described in 第 9.1.1 节 "svn 选项") from the subcommand descriptions which follow.

名称

svn add — 添加文件, 目录或符号链。

概要

```
svn add PATH...
```

描述

Schedule files, directories, or symbolic links in your working copy for addition to the repository. They will be uploaded and added to the repository on your next commit. If you add something and change your mind before committing, you can unschedule the addition using **svn revert**.

别名

无

改变

工作副本2

访问版本库

否

选项

```
--auto-props
--depth ARG
--force
--no-auto-props
--no-ignore
--parents
--quiet (-q)
--targets FILENAME
```

例子

添加一个文件到工作副本:

```
$ 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 ---depth=empty otherdir
A        otherdir
```

Normally, the command **svn add *** will skip over any directories that are already under version control. Sometimes, however, you may want to add every unversioned object in your working copy, including those hiding deeper. Passing the --force option makes **svn add** recurse into versioned directories:

```
$ svn add * ---force
A        foo.c
A        somedir/bar.c
A  (bin)  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

Show author and revision information inline for the specified files or URLs. Each line of text is annotated at the beginning with the author (username) and the revision number for the last change to that line.

别名

praise, **annotate**, **ann**

改变

无2

访问版本库

是

选项

```
--extensions (-x) ARG
--force
--incremental
--revision (-r) ARG
--use-merge-history (-g)
--verbose (-v)
--xml
```

例子

If you want to see blame-annotated source for `readme.txt` in your test repository:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry You should read this.
```

Even if **svn blame** says that Harry last modified `readme.txt` in revision 5, you'll have to examine exactly what the revision changed to be sure that Harry changed the *context* of the line —he may have adjusted just the whitespace.

If you use the `--xml` option, you can get XML output describing the blame annotations, but not the contents of the lines themselves:

```
$ svn blame ---xml http://svn.red-bean.com/repos/test/readme.txt
<?xml version="1.0"?>
<blame>
<target
   path="sandwich.txt">
<entry
   line-number="1">
<commit
   revision="3">
<author>sally</author>
<date>2008-05-25T19:12:31.428953Z</date>
</commit>
</entry>
<entry
   line-number="2">
<commit
   revision="5">
<author>harry</author>
<date>2008-05-29T03:26:12.293121Z</date>
</commit>
</entry>
</target>
</blame>
```

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

访问版本库

是

选项

```
--revision (-r) REV
```

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.
```

提示

If your working copy is out of date (or you have local modifications) and you want to see the HEAD revision of a file in your working copy, **svn cat -r HEAD *FILENAME*** will automatically fetch the HEAD revision of the specified path:

```
$ cat foo.c
This file is in my local working copy
and has changes that I've made.
```

```
$ svn cat --r HEAD foo.c
Latest revision fresh from the repository!
```

```
$ svn cat --r HEAD foo.c
Latest revision fresh from the repository!
```

名称

svn changelist — 关联(解耦)本地路径到修改列表。

概要

```
changelist CLNAME TARGET...

changelist ---remove TARGET...
```

描述

Used for dividing files in a working copy into a changelist (logical named grouping) in order to allow users to easily work on multiple file collections within a single working copy.

别名

cl

改变

工作副本2

访问版本库

否

选项

```
--changelist ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--remove
--targets ARG
```

例子

Edit three files, add them to a changelist, then commit only files in that changelist:

```
$ svn cl issue1729 foo.c bar.c baz.c
Path -'foo.c' is now a member of changelist -'issue1729'.
Path -'bar.c' is now a member of changelist -'issue1729'.
Path -'baz.c' is now a member of changelist -'issue1729'.

$ svn status
A      someotherfile.c
```

```
A       test/sometest.c

--- Changelist -'issue1729':
A       foo.c
A       bar.c
A       baz.c

$ svn commit ---changelist issue1729 --m -"Fixing Issue 1729."
Adding         bar.c
Adding         baz.c
Adding         foo.c
Transmitting file data -...
Committed revision 2.

$ svn status
A       someotherfile.c
A       test/sometest.c
```

注意，只有在变更列表*issue1729*中的文件被提交了。

名称

svn checkout — 从版本库取出一个工作副本。

概要

svn checkout URL[@REV]... [PATH]

描述

Check out a working copy from a repository. If *PATH* is omitted, the basename of the URL will be used as the destination. If multiple URLs are given, each will be checked out into a subdirectory of *PATH*, with the name of the subdirectory being the basename of the URL.

别名

co

改变

创建一个工作副本

访问版本库

是

选项

```
--depth ARG
--force
--ignore-externals
--quiet (-q)
--revision (-r) REV
```

例子

取出一个工作副本到mine目录：

```
$ svn checkout file:///var/svn/repos/test mine
A  mine/a
A  mine/b
A  mine/c
A  mine/d
Checked out revision 20.
$ ls
mine
```

检出两个目录到两个单独的工作副本：

```
$ svn checkout file:///var/svn/repos/test  file:///var/svn/repos/
quiz
A  test/a
A  test/b
A  test/c
A  test/d
Checked out revision 20.
A  quiz/l
A  quiz/m
Checked out revision 13.
$ ls
quiz  test
```

检出两个目录到两个单独的工作副本，但是将两个目录都放到working-copies：

```
$ svn checkout file:///var/svn/repos/test  file:///var/svn/repos/
quiz working-copies
A  working-copies/test/a
A  working-copies/test/b
A  working-copies/test/c
A  working-copies/test/d
Checked out revision 20.
A  working-copies/quiz/l
A  working-copies/quiz/m
Checked out revision 13.
$ 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副本：

```
$ svn checkout file:///var/svn/repos/test mine
A  mine/a
A  mine/b
^C
svn: The operation was interrupted
svn: caught SIGINT

$ svn checkout file:///var/svn/repos/test mine
A  mine/c
^C
svn: The operation was interrupted
svn: caught SIGINT

$ svn update mine
A  mine/d
```

```
Updated to revision 20.
```

If you wish to check out some revision other than the most recent one, you can do so by providing the `--revision` (`-r`) option to the **svn checkout** command:

```
$ svn checkout --r 2 file:///var/svn/repos/test mine
A  mine/a
Checked out revision 2.
```

```
$ svn checkout --r 2 file:///var/svn/repos/test mine
```

名称

svn cleanup — 递归清理工作副本

概要

```
svn cleanup [PATH...]
```

描述

Recursively clean up the working copy, removing working copy locks and resuming unfinished operations. If you ever get a `working copy locked` error, run this command to remove stale locks and get your working copy into a usable state again.

If, for some reason, an **svn update** fails due to a problem running an external diff program (e.g., user input or network failure), pass the `--diff3-cmd` to allow cleanup to complete any merging with your external diff program. You can also specify any configuration directory with the `--config-dir` option, but you should need these options extremely infrequently.

别名

无

改变

工作副本2

访问版本库

否

选项

```
--diff3-cmd CMD
```

例子

Well, there's not much to the examples here, as **svn cleanup** generates no output. If you pass no *PATH*, then "`.`" is used:

```
$ svn cleanup

$ svn cleanup -/var/svn/working-copy
```

名称

svn commit — 将修改从工作副本发送到版本库。

概要

svn commit [PATH...]

描述

Send changes from your working copy to the repository. If you do not supply a log message with your commit by using either the --file or --message option, **svn** will launch your editor for you to compose a commit message. See the editor-cmd list entry in 第 7.1.3.2 节 "配置".

svn commit will send any lock tokens that it finds and will release locks on all *PATH*s committed (recursively) unless --no-unlock is passed.

> **提示**
>
> If you begin a commit and Subversion launches your editor to compose the commit message, you can still abort without committing your changes. If you want to cancel your commit, just quit your editor without saving your commit message and Subversion will prompt you to either abort the commit, continue with no message, or edit the message again.

别名

ci (short for "check in"; not **co**, which is an alias for the **checkout** subcommand)

改变

工作副本；版本库

访问版本库

是

选项

```
--changelist ARG
--depth ARG
--editor-cmd ARG
--encoding ENC
--file (-F) FILE
--force-log
--keep-changelists
--message (-m) TEXT
--no-unlock
```

```
--quiet (-q)
--targets FILENAME
--with-revprop ARG
```

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.
```

Commit a modification to the file foo.c (explicitly specified on the command line) with the commit message in a file named msg:

```
$ 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 — 拷贝工作副本的一个文件或目录到版本库。

概要

svn copy SRC[@REV]... DST

描述

Copy one or more files in a working copy or in the repository. When copying multiple sources, they will be added as children of *DST*, which must be a directory. *SRC* and *DST* can each be either a working copy (WC) path or 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
    Complete server-side copy. This is usually used to branch and tag.

When copying multiple sources, they will be added as children of *DST*, which must be a directory.

If no peg revision (i.e., *@REV*) is supplied, by default the BASE revision will be used for files copied from the working copy, while the HEAD revision will be used for files copied from a URL.

> ### 注意
>
> You can only copy files within a single repository. Subversion does not support cross-repository copying.

别名

cp

改变

如果目标是 URL，则是版本库；如果目标是 WC 路径，则是工作副本。

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

```
--editor-cmd EDITOR
--encoding ENC
--file (-F) FILE
--force-log
--message (-m) TEXT
--parents
--quiet (-q)
--revision (-r) REV
--with-revprop ARG
```

例子

Copy an item within your working copy (this schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +   bar.txt
```

拷贝工作副本的一个文件或目录到版本库：

```
$ svn cp bat.c baz.c qux.c src
A         src/bat.c
A         src/baz.c
A         src/qux.c
```

Copy revision 8 of bat.c into your working copy under a different name:

```
$ svn cp --r 8 bat.c ya-old-bat.c
A         ya-old-bat.c
```

Copy an item in your working copy to a URL in the repository (this is an immediate commit, so you must supply a commit message):

```
$ svn copy near.txt file:///var/svn/repos/test/far-away.txt --
m -"Remote copy."

Committed revision 8.
```

Copy an item from the repository to your working copy (this just schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy file:///var/svn/repos/test/far-away --r 6 near-here
A         near-here
```

提示

这是恢复死掉文件的推荐方式！

最后，是在URL之间拷贝：

```
$ svn copy file:///var/svn/repos/test/far-away \
        file:///var/svn/repos/test/over-there --m -"remote
copy."

Committed revision 9.


$ svn copy file:///var/svn/repos/test/trunk \
        file:///var/svn/repos/test/tags/0.6.32-prerelease --
m -"tag tree"

Committed revision 12.
```


提示

This is the easiest way to "tag" a revision in your repository—just **svn copy** that revision (usually HEAD) into your tags directory.

And don't worry if you forgot to tag—you can always specify an older revision and tag anytime:

```
$ svn copy --r 11 file:///var/svn/repos/test/trunk \
        file:///var/svn/repos/test/tags/0.6.32-prerelease \
        --m -"Forgot to tag at rev 11"

Committed revision 13.
```

名称

svn delete — 从工作副本或版本库删除一个项目。

概要

```
svn delete PATH...

svn delete URL...
```

描述

Items specified by *PATH* are scheduled for deletion upon the next commit. Files (and directories that have not been committed) are immediately removed from the working copy unless the `--keep-local` option is given. The command will not remove any unversioned or modified items; use the `--force` option to override this behavior.

Items specified by URL are deleted from the repository via an immediate commit. Multiple URLs are committed atomically.

别名

del, **remove**, **rm**

改变

如果操作对象是文件则是工作副本变化；如果对象是URL则会影响版本库。

访问版本库

对URL操作时访问

选项

```
--editor-cmd EDITOR
--encoding ENC
--file (-F) FILE
--force
--force-log
--keep-local
--message (-m) TEXT
--quiet (-q)
--targets FILENAME
--with-revprop ARG
```

例子

Using **svn** to delete a file from your working copy deletes your local copy of the file, but it merely schedules the file to be deleted from the repository. When you commit, the file is deleted in the repository.

```
$ 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:///var/svn/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 — 显示两个版本或路径的差异。

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

- Display the differences between two paths. You can use **svn diff** in the following ways:

- Use just **svn diff** to display local modifications in a working copy.

- Display the changes made to *TARGET*s as they are seen in *REV* between two revisions. *TARGET*s may be all working copy paths or all *URL*s. If *TARGET*s are working copy paths, *N* defaults to BASE and *M* to the working copy; if *TARGET*s are *URL*s, *N* must be specified and *M* defaults to HEAD. The -c M option is equivalent to -r N:M where N = M-1. Using -c -M does the reverse: -r M:N where N = M-1.

- Display the differences between *OLD-TGT* as it was seen in *OLDREV* and *NEW-TGT* as it was seen in *NEWREV*. *PATH*s, if given, are relative to *OLD-TGT* and *NEW-TGT* and restrict the output to differences for those paths. *OLD-TGT* and *NEW-TGT* may be working copy paths or *URL[@REV]*. *NEW-TGT* defaults to *OLD-TGT* if not specified. -r N makes *OLDREV* default to N; -r N:M makes *OLDREV* default to N and *NEWREV* default to M.

svn diff OLD-URL[@OLDREV] NEW-URL[@NEWREV] is shorthand for **svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV]**.

svn diff -r N:M URL is shorthand for **svn diff -r N:M --old=URL --new=URL**.

svn diff [-r N[:M]] URL1[@N] URL2[@M] is shorthand for **svn diff [-r N[:M]] --old=URL1 --new=URL2**.

If *TARGET* is a URL, then revs N and M can be given either via the --revision option or by using the "@" notation as described earlier.

If *TARGET* is a working copy path, the default behavior (when no --revision option is provided) is to display the differences between the base and working copies of *TARGET*. If a --revision option is specified in this scenario, though, it means:

```
--revision N:M
```
    服务器比较 *TARGET@N* 和 *TARGET@M*。

```
--revision N
```
    The client compares *TARGET@N* against the working copy.

If the alternate syntax is used, the server compares *URL1* and *URL2* at revisions *N* and *M*, respectively. If either *N* or *M* is omitted, a value of HEAD is assumed.

By default, **svn diff** ignores the ancestry of files and merely compares the contents of the two files being compared. If you use --notice-ancestry, the ancestry of the paths in question will be taken into consideration when comparing revisions (i.e., if you run **svn diff** on two files with identical contents but different ancestry, you will see the entire contents of the file as having been removed and added again).

别名

di

改变

无2

访问版本库

获得工作副本非BASE修订版本的区别时会

选项

```
--change (-c) ARG
--changelist ARG
--depth ARG
--diff-cmd CMD
--extensions (-x) -"ARGS"
--force
--new ARG
--no-diff-deleted
--notice-ancestry
--old ARG
--revision (-r) ARG
--summarize
--xml
```

例子

比较BASE和你的工作副本(**svn diff**最经常的用法)：

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS  (revision 4404)
+++ COMMITTERS  (working copy)
```

See what changed in the file COMMITTERS revision 9115:

```
$ svn diff --c 9115 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

察看你的工作副本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3500trunk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3500trunk中的三个文件：

```
$ svn diff --r 3000:3500 ---old http://svn.collab.net/repos/svn/
trunk \
          COMMITTERS README HACKING
```

如果你有工作副本，你不必输入这么长的URL：

```
$ svn diff --r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Use `--diff-cmd` *CMD* `-x` to pass arguments directly to the external diff program:

```
$ svn diff ---diff-cmd -/usr/bin/diff --x -"-i --b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
```

Lastly, you can use the `--xml` option along with the `--summarize` option to view XML describing the changes that occurred between revisions, but not the contents of the diff itself:

```
$ svn diff ---summarize ---xml http://svn.red-bean.com/repos/
test@r2 \
           http://svn.red-bean.com/repos/test
<?xml version="1.0"?>
<diff>
<paths>
<path
   props="none"
   kind="file"
   item="modified">http://svn.red-bean.com/repos/test/
sandwich.txt</path>
<path
   props="none"
   kind="file"
   item="deleted">http://svn.red-bean.com/repos/test/burrito.txt</
path>
<path
   props="none"
   kind="dir"
   item="added">http://svn.red-bean.com/repos/test/snacks</path>
</paths>
</diff>
```

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

The first form exports a clean directory tree from the repository specified by *URL*—at revision *REV* if it is given; otherwise, at HEAD, into *PATH*. If *PATH* is omitted, the last component of the *URL* is used for the local directory name.

The second form exports a clean directory tree from the working copy specified by *PATH1* into *PATH2*. All local changes will be preserved, but files not under version control will not be copied.

别名

无

改变

本地磁盘

访问版本库

只有当从URL导出时会访问

选项

```
--depth ARG
--force
--ignore-externals
--native-eol EOL
--quiet (-q)
--revision (-r) REV
```

例子

从你的工作副本导出(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:///var/svn/repos my-export
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

When rolling operating-system-specific release packages, it can be useful to export a tree that uses a specific EOL character for line endings. The `--native-eol` option will do this, but it affects only files that have `svn:eol-style = native` properties attached to them. For example, to export a tree with all CRLF line endings (possibly for a Windows `.zip` file distribution):

```
$ svn export file:///var/svn/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

访问版本库

否

选项

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

Recursively commit a copy of *PATH* to *URL*. If *PATH* is omitted, "." is assumed. Parent directories are created in the repository as necessary. Unversionable items such as device files and pipes are ignored even if `--force` is specified.

别名

无

改变

版本库

访问版本库

是

选项

```
--auto-props
--depth ARG
--editor-cmd EDITOR
--encoding ENC
--file (-F) FILE
--force
--force-log
--message (-m) TEXT
--no-auto-props
--no-ignore
--quiet (-q)
--with-revprop ARG
```

例子

This imports the local directory `myproj` into `trunk/misc` in your repository. The directory `trunk/misc` need not exist before you import into it—**svn import** will recursively create directories for you.

```
$ svn import --m -"New import" myproj \
           http://svn.red-bean.com/repos/trunk/misc
Adding         myproj/sample.txt
…
Transmitting file data -........
Committed revision 16.
```

Be aware that this will *not* create a directory named `myproj` in the repository. If that's what you want, simply add `myproj` to the end of the URL:

```
$ 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.
```

After importing data, note that the original tree is *not* under version control. To start working, you still need to **svn checkout** a fresh working copy of the tree.

名称

svn info — 显示本地或远程条目的信息。

概要

svn info [TARGET[@REV]...]

描述

打印你的工作副本路径和 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(date)

- Lock失效时间(date)

Additional kinds of information available only for working copy paths are:

- Schedule

- 复制自 URL

- 复制自 rev

- 正文最后更新

- 属性最后更新

- Checksum

- Conflict previous base file

- Conflict previous working file

- Conflict current base file

- Conflict properties file

别名

无

改变

无2

访问版本库

对URL操作时访问

选项

```
--changelist ARG
--depth ARG
--incremental
--recursive (-R)
--revision (-r) REV
--targets FILENAME
--xml
```

例子

svn info 会展示工作副本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: d6aeb60b0662ccceb6bce4bac344cb66
```

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)
Properties Last Updated: 2003-01-16 23:39:02 --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)
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 --0600 (Wed, 15 Jan 2003)
Lock Comment (1 line):
My test lock comment
```

Lastly, **svn info** output is available in XML format by passing the`--xml` option:

```
$ svn info ---xml http://svn.red-bean.com/repos/test
<?xml version="1.0"?>
```

```
<info>
<entry
   kind="dir"
   path="."
   revision="1">
<url>http://svn.red-bean.com/repos/test</url>
<repository>
<root>http://svn.red-bean.com/repos/test</root>
<uuid>5e7d134a-54fb-0310-bd04-b611643e5c25</uuid>
</repository>
<wc-info>
<schedule>normal</schedule>
<depth>infinity</depth>
</wc-info>
<commit
   revision="1">
<author>sally</author>
<date>2003-01-15T23:35:12.847647Z</date>
</commit>
</entry>
</info>
```

名称

svn list — 列出版本库目录的条目。

概要

```
svn list [TARGET[@REV]...]
```

描述

列出版本库中每一个 *TARGET* 文件和 *TARGET* 目录的内容。如果 *TARGET* 是工作副本路径，会使用对应的版本库 URL。

缺省的 *TARGET* 是 "．"，意味着当前工作副本的版本库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

- If locked, the letter "O" (see the preceding section on svn info for details).

- 大小(单位字节)

- 最后提交的日期时间

With `--xml`, output is in XML format (with a header and an enclosing document element unless `--incremental` is also specified). All of the information is present; the `--verbose` option is not accepted.

别名

ls

改变

无2

访问版本库

是

选项

```
--depth ARG
--incremental
--recursive (-R)
--revision (-r) REV
```

```
--verbose (-v)
--xml
```

例子

如果你希望在没有下载工作副本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:///var/svn/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/
```

You can also get **svn list** output in XML format with the --xml option:

```
$ svn list ---xml http://svn.red-bean.com/repos/test
<?xml version="1.0"?>
<lists>
<list
   path="http://svn.red-bean.com/repos/test">
<entry
   kind="dir">
<name>examples</name>
<size>0</size>
<commit
   revision="24">
<author>harry</author>
<date>2008-01-18T06:35:53.048870Z</date>
</commit>
</entry>
...
</list>
</lists>
```

更多细节见第 2.5.3.2 节 "svn list"。

名称

svn lock — Lock working copy paths or URLs in the repository so that no other user can commit changes to them.

概要

```
svn lock TARGET...
```

描述

Lock each *TARGET*. If any *TARGET* is already locked by another user, print a warning and continue locking the rest of the *TARGET*s. Use `--force` to steal a lock from another user or working copy.

别名

无

改变

工作副本，版本库

访问版本库

是

选项

```
--encoding ENC
--file (-F) FILE
--force
--force-log
--message (-m) TEXT
--targets FILENAME
```

例子

在工作副本锁定两个文件：

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user -'harry'.
'house.jpg' locked by user -'harry'.
```

锁定工作副本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 -'/var/svn/repos/db'

$ svn lock ---force tree.jpg
'tree.jpg' locked by user -'harry'.
```

没有工作副本的情况下锁定文件：

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user -'harry'.
```

更多细节见第 3.7 节 "锁定"。

名称

svn log — 显示提交日志信息。

概要

```
svn log [PATH]

svn log URL[@REV] [PATH...]
```

描述

Shows log messages from the repository. If no arguments are supplied, **svn log** shows the log messages for all files and directories inside (and including) the current working directory of your working copy. You can refine the results by specifying a path, one or more revisions, or any combination of the two. The default revision range for a local path is `BASE:1`.

If you specify a URL alone, it prints log messages for everything the URL contains. If you add paths past the URL, only messages for those paths under that URL will be printed. The default revision range for a URL is `HEAD:1`.

With `--verbose`, **svn log** will also print all affected paths with each log message. With `--quiet`, **svn log** will not print the log message body itself (this is compatible with `--verbose`).

Each log message is printed just once, even if more than one of the affected paths for that revision were explicitly requested. Logs follow copy history by default. Use `--stop-on-copy` to disable this behavior, which can be useful for determining branch points.

别名

无

改变

无2

访问版本库

是

选项

```
--change (-c) ARG
--incremental
--limit (-l) NUM
--quiet (-q)
--revision (-r) REV
--stop-on-copy
--targets FILENAME
```

```
--use-merge-history (-g)
--verbose (-v)
--with-all-revprops
--with-revprop ARG
--xml
```

例子

You can see the log messages for all the paths that changed in your working copy by running **svn log** from the top:

```
$ 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副本，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
…
```

The --verbose option causes **svn log** to include information about the paths that were changed in each displayed revision. These paths appear, one path per line of output, with action codes that indicate what type of change was made to the path.

```
$ svn log --v 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
Changed paths:
   M -/foo.c

Added defines.
------------------------------------------------------------------------
r31 -| harry -| 2003-01-10 12:25:08 --0600 (Fri, 10 Jan 2003) -|
1 line
Changed paths:
   A -/bar.c

Added new file bar.c
------------------------------------------------------------------------
r28 -| sally -| 2003-01-07 21:48:33 --0600 (Tue, 07 Jan 2003) -|
3 lines
…
```

svn log uses just a handful of action codes, and they are similar to the ones the **svn update** command uses:

A
　　项目已增加。

D

　　项目已删除。

M

　　条目属性改变了，注意开头的空格。

R

　　在同样的位置，项目被代替。

In addition to the action codes which precede the changed paths, **svn log** with the `--verbose` option will note whether a path was added or replaced as the result of a copy operation. It does so by printing (`from COPY-FROM-PATH:COPY-FROM-REV`) after such paths.

When you're concatenating the results of multiple calls to the log command, you may want to use the `--incremental` option. **svn log** normally prints out a dashed line at the beginning of a log message, after each subsequent log message, and following the final log message. If you ran **svn log** on a range of two revisions, you would get this:

```
$ 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。

```
$ svn log ---xml ---incremental --r 1 sandwich.txt
<logentry
   revision="1">
<author>harry</author>
<date>2008-06-03T06:35:53.048870Z</date>
<msg>Initial Import.</msg>
</logentry>
```

提示

Sometimes when you run **svn log** on a specific path and a specific revision, you see no log information output at all, as in the following:

```
$ svn log --r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

That just means the path wasn't modified in that revision. To get log information for that revision, either run the log operation against the repository's root URL, or specify a path that you happen to know was changed in that revision:

```
$ 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副本路径。

概要

svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]

svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]

svn merge [[-c M]... -| [-r N:M]...] [SOURCE[@REV] [WCPATH]]

描述

In the first form, the source URLs are specified at revisions *N* and *M*. These are the two sources to be compared. The revisions default to HEAD if omitted.

In the second form, the URLs corresponding to the source working copy paths define the sources to be compared. The revisions must be specified.

In the third form, *SOURCE* can be either a URL or a working copy path (in which case its corresponding URL is used). If not specified, *SOURCE* will be the same as *WCPATH*. *SOURCE* in revision *REV* is compared as it existed between revisions *N* and *M* for each revision range provided. If *REV* is not specified, HEAD is assumed.

-c M is equivalent to -r <M-1>:M, and -c -M does the reverse: -r M:<M-1>. If no revision ranges are specified, the default range of 1:HEAD is used. Multiple -c and/or -r instances may be specified, and mixing of forward and reverse ranges is allowed—the ranges are internally compacted to their minimum representation before merging begins (which may result in no-op).

WCPATH is the working copy path that will receive the changes. If *WCPATH* is omitted, a default value of ".". is assumed, unless the sources have identical basenames that match a file within ".". In this case, the differences will be applied to that file.

Subversion will internally track metadata about the merge operation only if the two sources are ancestrally related—if the first source is an ancestor of the second or vice versa. This is guaranteed to be the case when using the third form. Unlike **svn diff**, the merge command takes the ancestry of a file into consideration when performing a merge operation. This is very important when you're merging changes from one branch into another and you've renamed a file on one branch but not the other.

别名

无

改变

工作副本2

访问版本库

只有在对URL操作时会

选项

```
--accept ARG
--change (-c) REV
--depth ARG
--diff3-cmd CMD
--dry-run
--extensions (-x) ARG
--force
--ignore-ancestry
--quiet (-q)
--record-only
--reintegrate
--revision (-r) REV
```

例子

Merge a branch back into the trunk (assuming that you have an up-to-date working copy of the trunk):

```
$ svn merge ---reintegrate \
          http://svn.example.com/repos/calc/branches/my-calc-
branch
--- Merging differences between repository URLs into -'.':
U    button.c
U    integer.c
U    Makefile
 U   -.

$ # build, test, verify, -...

$ svn commit --m -"Merge my-calc-branch back into trunk!"
Sending        -.
Sending        button.c
Sending        integer.c
Sending        Makefile
Transmitting file data -..
Committed revision 391.
```

合并一个单独文件的修改：

```
$ cd myproj
```

```
$ svn merge --r 30:31 thhgttg.txt
U  thhgttg.txt
```

```
$ svn merge --r 30:31 thhgttg.txt
U  thhgttg.txt
```

名称

svn mergeinfo — Query merge-related information. See 第 4.3.3 节 "合并信息和预览" for details.

概要

svn mergeinfo SOURCE_URL[@REV] [TARGET[@REV]...]

描述

Query information related to merges (or potential merges) between *SOURCE-URL* and *TARGET*. If the `--show-revs` option is not provided, display revisions which have been merged from *SOURCE-URL* to *TARGET*. Otherwise, display either `merged` or `eligible` revisions as specified by the `--show-revs` option.

别名

无

改变

无2

访问版本库

是

选项

--revision (-r) REV

例子

Find out which changesets your trunk directory has already received as well as what changesets it's still eligible to receive:

```
$ svn mergeinfo branches/test
Path: branches/test
  Source path: -/trunk
    Merged ranges: r2:13
    Eligible ranges: r13:15
```

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

Create a directory with a name given by the final component of the *PATH* or *URL*. A directory specified by a working copy *PATH* is scheduled for addition in the working copy. A directory specified by a URL is created in the repository via an immediate commit. Multiple directory URLs are committed atomically. In both cases, all the intermediate directories must already exist unless the `--parents` option is used.

别名

无

改变

工作副本；如果是对URL操作则会影响版本库

访问版本库

只有在对URl操作时会

选项

```
--editor-cmd EDITOR
--encoding ENC
--file (-F) FILE
--force-log
--message (-m) TEXT
--parents
--quiet (-q)
--with-revprop ARG
```

例子

在工作副本创建一个目录：

```
$ 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.
```

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副本或者是版本库。

> **提示**
>
> 这个命令同**svn copy**加一个**svn delete**等同。

When moving multiple sources, they will be added as children of *DST*, which must be a directory.

> **注意**
>
> Subversion does not support moving between working copies and URLs. In addition, you can only move files within a single repository—Subversion does not support cross-repository moving. Subversion supports the following types of moves within a single repository:

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

工作副本；如果是对URL操作则会影响版本库

访问版本库

只有在对URl操作时会

选项

```
--editor-cmd EDITOR
--encoding ENC
--file (-F) FILE
--force
```

```
--force-log
--message (-m) TEXT
--parents
--quiet (-q)
--revision (-r) REV
--with-revprop ARG
```

例子

移动工作拷bede一个文件：

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

移动工作副本的一些文件到子目录：

```
$ svn move baz.c bat.c qux.c src
A         src/baz.c
D         baz.c
A         src/bat.c
D         bat.c
A         src/qux.c
D         qux.c
```

移动版本库中的文件(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

This removes properties from files, directories, or revisions. The first form removes versioned properties in your working copy, and the second removes unversioned remote properties on a repository revision (*TARGET* determines only which repository to access).

别名

pdel, **pd**

改变

工作副本；对URL操作时是版本库

访问版本库

只有在对URl操作时会

选项

```
--changelist ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
```

例子

删除你的工作副本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 — Edit the property of one or more items under version control. See svn propset later in this chapter.

概要

```
svn propedit PROPNAME TARGET...

svn propedit PROPNAME ---revprop --r REV [TARGET]
```

描述

Edit one or more properties using your favorite editor. The first form edits versioned properties in your working copy, and the second edits unversioned remote properties on a repository revision (*TARGET* determines only which repository to access).

别名

pedit, **pe**

改变

工作副本；对URL操作时是版本库

访问版本库

只有在对URl操作时会

选项

```
--editor-cmd EDITOR
--encoding ENC
--file (-F) ARG
--force
--force-log
--message (-m) ARG
--revision (-r) REV
--revprop
--with-revprop ARG
```

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

Print the value of a property on files, directories, or revisions. The first form prints the versioned property of an item or items in your working copy, and the second prints unversioned remote properties on a repository revision. See 第 3.2 节 "属性" for more information on properties.

别名

pget, **pg**

改变

工作副本；对URL操作时是版本库

访问版本库

只有在对URl操作时会

选项

```
--changelist ARG
--depth ARG
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--xml
```

例子

检查工作副本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.
```

Lastly, you can get **svn propget** output in XML format with the `--xml` option:

```
$ svn propget ---xml svn:ignore -.
<?xml version="1.0"?>
<properties>
<target
   path="">
<property
   name="svn:ignore">*.o
</property>
</target>
</properties>
```

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

List all properties on files, directories, or revisions. The first form lists versioned properties in your working copy, and the second lists unversioned remote properties on a repository revision (*TARGET* determines only which repository to access).

别名

plist, **pl**

改变

工作副本；对URL操作时是版本库

访问版本库

只有在对URl操作时会

选项

```
--changelist ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--verbose (-v)
--xml
```

例子

你可以使用 **proplist** 察看你的工作副本的一个项目属性：

```
$ svn proplist foo.c
Properties on -'foo.c':
  svn:mime-type
  svn:keywords
```

```
  owner
```

使用 --verbose 选项，**svn proplist** 可以非常便利的显示属性值：

```
$ svn proplist ---verbose foo.c
Properties on -'foo.c':
  svn:mime-type -: text/plain
  svn:keywords -: Author Date Rev
  owner -: sally
```

Lastly, you can get **svn proplist** output in xml format with the --xml option:

```
$ svn proplist ---xml
<?xml version="1.0"?>
<properties>
<target
   path=".">
<property
   name="svn:ignore"/>
</target>
</properties>
```

名称

svn propset — Set *PROPNAME* to *PROPVAL* on files, directories, or revisions.

概要

```
svn propset PROPNAME [PROPVAL -| --F VALFILE] PATH...
```

```
svn propset PROPNAME ---revprop --r REV [PROPVAL -| --F VALFILE]
[TARGET]
```

描述

Set *PROPNAME* to *PROPVAL* on files, directories, or revisions. The first example creates a versioned, local property change in the working copy, and the second creates an unversioned, remote property change on a repository revision (*TARGET* determines only which repository to access).

> **提示**
>
> Subversion has a number of "special" properties that affect its behavior. See 第 9.10 节 "Subversion 属性" later in this chapter for more on these properties.

别名

pset, *ps*

改变

工作副本；对URL操作时是版本库

访问版本库

只有在对URl操作时会

选项

```
--changelist ARG
--depth ARG
--encoding ENC
--file (-F) FILE
--force
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--targets FILENAME
```

例子

设置文件的 MIME 类型：

```
$ 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，你没有工作副本，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'
```

Lastly, you can tell **propset** to take its input from a file. You could even use this to set the contents of a property to something binary:

```
$ svn propset owner-pic --F sally.jpg moo.c
property -'owner-pic' set on -'moo.c'
```

注意

By default, you cannot modify revision properties in a Subversion repository. Your repository administrator must explicitly enable revision property modifications by creating a hook named `pre-revprop-change`. See for more information on hook scripts.

名称

svn resolve — 解决工作副本文件或目录的冲突。

概要

svn resolve PATH...

描述

Resolve "conflicted" state on working copy files or directories. This routine does not semantically resolve conflict markers; however, it replaces *PATH* with the version specified by the `--accept` argument and then removes conflict-related artifact files. This allows *PATH* to be committed again—that is, it tells Subversion that the conflicts have been "resolved.". You can pass the following arguments to the `--accept` command depending on your desired resolution:

base
    Choose the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.

working
    Assuming that you've manually handled the conflict resolution, choose the version of the file as it currently stands in your working copy.

mine-full
    Resolve all conflicted files with copies of the files as they stood immediately before you ran **svn update**.

theirs-full
    Resolve all conflicted files with copies of the files that were fetched from the server when you ran **svn update**.

关于解决冲突的深入介绍可以看第 2.4.5 节 "解决冲突（合并别人的修改）"。

别名

无

改变

工作副本2

访问版本库

否

选项

--accept ARG

```
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

例子

Here's an example where, after a postponed conflict resolution during update, **svn resolve** replaces the all conflicts in file `foo.c` with your edits:

```
$ svn up
Conflict discovered in -'foo.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: p
C    foo.c
Updated to revision 5.

$ svn resolve ---accept mine-full foo.c
Resolved conflicted state of -'foo.c'
```

名称

svn resolved — *Deprecated*. Remove "conflicted" state on working copy files or directories.

概要

```
svn resolved PATH...
```

描述

This command has been deprecated in favor of running **svn resolve --accept working** **PATH**. See svn resolve in the preceding section for details.

Remove "conflicted" state on working copy files or directories. This routine does not semantically resolve conflict markers; it merely removes conflict-related artifact files and allows *PATH* to be committed again; that is, it tells Subversion that the conflicts have been "resolved." See 第 2.4.5 节 "解决冲突（合并别人的修改）" for an in-depth look at resolving conflicts.

别名

无

改变

工作副本2

访问版本库

否

选项

```
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

例子

如果你在更新时得到冲突，你的工作副本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** 让你的工作副本知道你已经完成了所有事情。

> ⛔ **警告**
>
> 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副本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

Reverts any local changes to a file or directory and resolves any conflicted states. **svn revert** will revert not only the contents of an item in your working copy, but also any property changes. Finally, you can use it to undo any scheduling operations that you may have performed (e.g., files scheduled for addition or deletion can be "unscheduled").

别名

无

改变

工作副本2

访问版本库

否

选项

```
--changelist ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

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--depth=infinity选项：

```
$ svn revert ---depth=infinity -.
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
```

警告

svn revert is inherently dangerous, since its entire purpose is to throw away data —namely, your uncommitted changes. Once you've reverted, Subversion provides *no way* to get back those uncommitted changes.

If you provide no targets to **svn revert**, it will do nothing—to protect you from accidentally losing changes in your working copy, **svn revert** requires you to provide at least one target.

名称

svn status — 打印工作副本文件和目录的状态。

概要

```
svn status [PATH...]
```

描述

Print the status of working copy files and directories. With no arguments, it prints only locally modified items (no repository access). With `--show-updates`, it adds working revision and server out-of-date information. With `--verbose`, it prints full revision information on every item. With `--quiet`, it prints only summary information about locally modified items.

输出的前六列都是一个字符宽，每一列给出了工作副本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'`
Item has been replaced in your working copy. This means the file was scheduled for deletion, and then a new file with the same name was scheduled for addition in its place.

`'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。

`'!'`
Item is missing (e.g., you moved or deleted it without using **svn**). This also indicates that a directory is incomplete (a checkout or update was interrupted).

`'~'`

Item is versioned as one kind of object (file, directory, link), but has been replaced by a different kind of object.

The second column tells the status of a file's or directory's properties:

`' '`

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。

The third column is populated only if the working copy directory is locked (see 第 2.6 节 "有时你只需要清理"):

`' '`

项目没有锁定。

`'L'`

项目已经锁定。

The fourth column is populated only if the item is scheduled for addition-with-history:

`' '`

没有历史预定要提交。

`'+'`

历史预定要伴随提交。

The fifth column is populated only if the item is switched relative to its parent (see 第 4.5 节 "使用分支"):

`' '`

项目是它的父目录的孩子。

`'S'`

项目已经转换。

The sixth column is populated with lock information:

`' '`

When `--show-updates` is used, the file is not locked. If `--show-updates` is *not* used, this merely means that the file is not locked in this working copy.

K

文件锁定在工作副本。

O

File is locked either by another user or in another working copy. This appears only when `--show-updates` is used.

T

File was locked in this working copy, but the lock has been "stolen" and is invalid. The file is currently locked in the repository. This appears only when `--show-updates` is used.

B

File was locked in this working copy, but the lock has been "broken" and is invalid. The file is no longer locked. This appears only when `--show-updates` is used.

The out-of-date information appears in the seventh column (only if you pass the `--show-updates` option):

' '

这个项目在工作副本是最新的。

'*'

在服务器这个项目有了新的修订版本。

The remaining fields are variable width and delimited by spaces. The working revision is the next field if the `--show-updates` or `--verbose` option is passed.

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。

工作副本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

访问版本库

只有使用`--show-updates`时会访问

选项

```
--changelist ARG
--depth ARG
--ignore-externals
--incremental
--no-ignore
--quiet (-q)
--show-updates (-u)
--verbose (-v)
--xml
```

例子

这是查看你在工作副本所做的修改的最简单的方法。

```
$ svn status wc
 M     wc/bar.c
A  +   wc/qax.c
```

If you want to find out what files in your working copy are out of date, pass the `--show-updates` option (this will *not* make any changes to your working copy). Here you can see that `wc/foo.c` has changed in the repository since we last updated our working copy:

```
$ svn status ---show-updates wc
 M            965     wc/bar.c
        *     965     wc/foo.c
A  +          965     wc/qax.c
Status against revision:    981
```

> **注意**
>
> `--show-updates` *only* places an asterisk next to items that are out of date (i.e., items that will be updated from the repository if you later use **svn update**). `--show-updates` does *not* cause the status listing to reflect the repository's version of the item (although you can see the revision number in the repository by passing the `--verbose` option).

The most information you can get out of the status subcommand is as follows:

```
$ 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
```

Lastly, you can get **svn status** output in XML format with the `--xml` option:

```
$ svn status ---xml wc
<?xml version="1.0"?>
<status>
<target
   path="wc">
<entry
   path="qax.c">
<wc-status
   props="none"
   item="added"
   revision="0">
</wc-status>
</entry>
```

```
<entry
   path="bar.c">
<wc-status
   props="normal"
   item="modified"
   revision="965">
<commit
   revision="965">
<author>sally</author>
<date>2008-05-28T06:35:53.048870Z</date>
</commit>
</wc-status>
</entry>
</target>
</status>
```

关于**svn status**的更多例子可以见第 2.4.3.1 节 "查看你的修改概况"。

名称

svn switch — 把工作副本更新到别的URL。

概要

```
svn switch URL[@PEGREV] [PATH]

switch ---relocate FROM TO [PATH...]
```

描述

The first variant of this subcommand (without the `--relocate` option) updates your working copy to point to a new URL—usually a URL that shares a common ancestor with your working copy, although not necessarily. This is the Subversion way to move a working copy to a new branch. If specified, *PEGREV* determines in which revision the target is first looked up. See 第 4.5 节 "使用分支" for an in-depth look at switching.

If `--force` is used, unversioned obstructing paths in the working copy do not automatically cause a failure if the switch attempts to add the same path. If the obstructing path is the same type (file or directory) as the corresponding path in the repository, it becomes versioned but its contents are left untouched in the working copy. This means that an obstructing directory's unversioned children may also obstruct and become versioned. For files, any content differences between the obstruction and the repository are treated like a local modification to the working copy. All properties from the repository are applied to the obstructing path.

As with most subcommands, you can limit the scope of the switch operation to a particular tree depth using the `--depth` option. Alternatively, you can use the `--set-depth` option to set a new "sticky" working copy depth on the switch target. Currently, the depth of a working copy directory can only be increased (telescoped more deeply); you cannot make a directory more shallow.

`--relocate`选项导致**svn switch**做不同的事情：它更新你的工作副本指向到**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

工作副本2

访问版本库

是

选项

```
--accept ARG
--depth ARG
--diff3-cmd CMD
--force
--ignore-externals
--quiet (-q)
--relocate
--revision (-r) REV
--set-depth ARG
```

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.
```

To switch back, just provide the URL to the location in the repository from which you originally checked out your working copy:

```
$ 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.
```

提示

You can switch just part of your working copy to a branch if you don't want to switch your entire working copy.

Sometimes an administrator might change the location (or apparent location) of your repository —in other words, the content of the repository doesn't change, but the repository's root URL does. For example, the hostname may change, the URL scheme may change, or any part of the URL that leads to the repository itself may change. Rather than check out a new working copy, you can have the **svn switch** command "rewrite" your working copy's administrative metadata to refer to the new repository location. If you use the `--relocate` option to **svn switch**, Subversion will contact the repository to validate the relocation request (looking for the repository at the new URL, of course), and then do this metadata rewriting. No file contents will be changed as the result of this type of switch operation—this is a metadata-only modification to the working copy.

```
$ svn checkout file:///var/svn/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:///var/svn/repos'

$ svn switch ---relocate file:///var/svn/repos file:///tmp/
newlocation -.
$ svn update
At revision 3.
```

警告

Be careful when using the `--relocate` option. If you mistype the argument, you might end up creating nonsensical URLs within your working copy that render the whole workspace unusable and tricky to fix. It's also important to understand exactly when one should or shouldn't use `--relocate`. Here's the rule of thumb:

- If the working copy needs to reflect a new directory *within* the repository, use just **svn switch**.

- If the working copy still reflects the same repository directory, but the location of the repository itself has changed, use **svn switch** with the `--relocate` option.

名称

svn unlock — 解锁工作副本路径或URL。

概要

```
svn unlock TARGET...
```

描述

Unlock each *TARGET*. If any *TARGET* is locked by another user or no valid lock token exists in the working copy, print a warning and continue unlocking the rest of the *TARGET*s. Use `--force` to break a lock belonging to another user or working copy.

别名

无

改变

工作副本，版本库

访问版本库

是

选项

```
--force
--targets FILENAME
```

例子

解锁工作副本中的两个文件：

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

解锁工作副本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.
```

没有工作副本时解锁一个文件：

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

更多细节见第 3.7 节 "锁定"。

名称

svn update — 更新你的工作副本。

概要

```
svn update [PATH...]
```

描述

svn update brings changes from the repository into your working copy. If no revision is given, it brings your working copy up to date with the HEAD revision. Otherwise, it synchronizes the working copy to the revision given by the --revision option. As part of the synchronization, **svn update** also removes any stale locks (see 第 2.6 节 "有时你只需要清理") found in the working copy.

For each updated item, it prints a line that starts with a character reporting the action taken. These characters have the following meaning:

A
    添加

B
    损坏的锁（仅第三列）

D
    删除

U
    更新

C
    冲突

G
    合并

E
    已经存在

A character in the first column signifies an update to the actual file, whereas updates to the file's properties are shown in the second column. Lock information is printed in the third column.

As with most subcommands, you can limit the scope of the update operation to a particular tree depth using the --depth option. Alternatively, you can use the --set-depth option to set a new "sticky" working copy depth on the update target. Currently, the depth of a working copy directory can only be increased (telescoped more deeply); you cannot make a directory more shallow.

别名

up

改变

工作副本2

访问版本库

是

选项

```
--accept ARG
--changelist
--depth ARG
--diff3-cmd CMD
--editor-cmd ARG
--force
--ignore-externals
--quiet (-q)
--revision (-r) REV
--set-depth ARG
```

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.
```

你也可以将工作副本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.
```

> **提示**
>
> If you want to examine an older revision of a single file, you may want to use **svn cat** instead—it won't change your working copy.

9.2. svnadmin

svnadmin is the administrative tool for monitoring and repairing your Subversion repository. For detailed information on repository administration, see the maintenance section for 第 5.4.1.1 节 "svnadmin".

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。

9.2.1. svnadmin 选项

Options in **svmadin** are global, just as they are in **svn**:

`--bdb-log-keep`
   (Berkeley DB-specific.) Disable automatic log removal of database logfiles. Having these logfiles around can be convenient if you need to restore from a catastrophic repository failure.

`--bdb-txn-nosync`
   (Berkeley DB-specific.) Disables fsync when committing database transactions. Used with the **svnadmin create** command to create a Berkeley DB-backed repository with `DB_TXN_NOSYNC` enabled (which improves speed but has some risks associated with it).

`--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`
   By default, when loading data into repository that already contains revisions, **svnadmin** will ignore the `UUID` from the dump stream. This option will cause the repository's `UUID` to be set to the `UUID` from the stream.

`--ignore-uuid`
   By default, when loading an empty repository, **svnadmin** will ignore the `UUID` from the dump stream. This option will force that UUID to be ignored (useful for overriding your configuration file if it has `--force-uuid` set).

`--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/。

`--pre-1.4-compatible`
 When creating a new repository, use a format that is compatible with versions of Subversion earlier than Subversion 1.4.

`--pre-1.5-compatible`
 When creating a new repository, use a format that is compatible with versions of Subversion earlier than Subversion 1.5.

`--revision` (`-r`) *ARG*
 指定一个操作的修订版本。

`--quiet`
 Do not show normal progress—show only errors.

`--use-post-commit-hook`
 When loading a dump file, runs the repository's `post-commit` hook after finalizing each newly loaded revision.

`--use-post-revprop-change-hook`
 When changing a revision property, runs the repository's `post-revprop-change` hook after changing the revision property.

`--use-pre-commit-hook`
 When loading a dump file, runs the repository's `pre-commit` hook before finalizing each newly loaded revision. If the hook fails, aborts the commit and terminates the load process.

`--use-pre-revprop-change-hook`
 When changing a revision property, runs the repository's `pre-revprop-change` hook before changing the revision property. If the hook fails, aborts the modification and terminates.

9.2.2. svnadmin 子命令

Here are the various subcommands for the **svnadmin** program.

名称

svnadmin crashtest — 方针进程崩溃。

概要

svnadmin crashtest REPOS_PATH

描述

Open the repository at *REPOS_PATH*, then abort, thus simulating a process that crashes while holding an open repository handle. This is used for testing automatic repository recovery (a new feature in Berkeley DB 4.4). It's unlikely that you'll need to run this command.

选项

无

例子

```
$ svnadmin crashtest -/var/svn/repos
Aborted
```

很激动，不是吗？

名称

svnadmin create — 创建一个新的空的版本库。

概要

svnadmin create REPOS_PATH

描述

Create a new, empty repository at the path provided. If the provided directory does not exist, it will be created for you. [1] As of Subversion 1.2, **svnadmin** creates new repositories with the FSFS filesystem backend by default.

While **svnadmin create** will create the base directory for a new repository, it will not create intermediate directories. For example, if you have an empty directory named /var/svn, creating /var/svn/repos will work, while attempting to create /var/svn/subdirectory/repos will fail with an error.

选项

--bdb-log-keep
--bdb-txn-nosync
--config-dir DIR
--fs-type TYPE
--pre-1.4-compatible
--pre-1.5-compatible

例子

Creating a new repository is this easy:

```
$ svnadmin create -/var/svn/repos
```

In Subversion 1.0, a Berkeley DB repository is always created. In Subversion 1.1, a Berkeley DB repository is the default repository type, but an FSFS repository can be created using the --fs-type option:

```
$ svnadmin create -/var/svn/repos ---fs-type fsfs
```

---

[1] 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 exists in current versions of Subversion only for historical reasons. This command is deprecated and no longer needed.

It dates from a time when Subversion offered administrators greater control over compression strategies in the repository. This turned out to be a lot of complexity for *very* little gain, and this "feature" was deprecated.

选项

--quiet (-q)
--revision (-r) REV

名称

svnadmin dump — Dump the contents of the filesystem to `stdout`.

概要

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental] [--
deltas]
```

描述

Dump the contents of the filesystem to `stdout` in a "dump file" portable format, sending feedback to `stderr`. Dump revisions *LOWER* rev through *UPPER* rev. If no revisions are given, dump all revision trees. If only *LOWER* is given, dump that one revision tree. See 第 5.4.5 节 "版本库数据的移植" for a practical use.

By default, the Subversion dump stream contains a single revision (the first revision in the requested revision range) in which every file and directory in the repository in that revision is presented as though that whole tree was added at once, followed by other revisions (the remainder of the revisions in the requested range), which contain only the files and directories that were modified in those revisions. For a modified file, the complete full-text representation of its contents, as well as all of its properties, are presented in the dump file; for a directory, all of its properties are presented.

Two useful options modify the dump file generator's behavior. The first is the `--incremental` option, which simply causes that first revision in the dump stream to contain only the files and directories modified in that revision, instead of being presented as the addition of a new tree, and in exactly the same way that every other revision in the dump file is presented. This is useful for generating a relatively small dump file to be loaded into another repository that already has the files and directories that exist in the original repository.

The second useful option is `--deltas`. This option causes **svnadmin dump** to, instead of emitting full-text representations of file contents and property lists, emit only deltas of those items against their previous versions. This reduces (in some cases, drastically) the size of the dump file that **svnadmin dump** creates. There are, however, disadvantages to using this option—deltified dump files are more CPU-intensive to create, cannot be operated on by **svndumpfilter**, and tend not to compress as well as their nondeltified counterparts when using third-party tools such as **gzip** and **bzip2**.

选项

```
--deltas
--incremental
--quiet (-q)
--revision (-r) REV
```

例子

转储整个版本库：

```
$ svnadmin dump -/var/svn/repos > full.dump
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
```

从版本库增量转储一个单独的事务：

```
$ svnadmin dump -/var/svn/repos --r 21 ---incremental > incr.dump
* Dumped revision 21.
```

名称

svnadmin help — 求助！

概要

```
svnadmin help [SUBCOMMAND...]
```

描述

This subcommand is useful when you're trapped on a desert island with neither a Net connection nor a copy of this book.

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

This subcommand makes a full "hot" backup of your repository, including all hooks, configuration files, and, of course, database files. If you pass the `--clean-logs` option, **svnadmin** will perform a hot copy of your repository, and then remove unused Berkeley DB logs from the original repository. You can run this command at any time and make a safe copy of the repository, regardless of whether other processes are using the repository.

选项

`--clean-logs`

警告

就像第 5.2.3.1 节 "Berkeley DB"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 — Ask Berkeley DB which logfiles exist for a given Subversion repository (applies only to repositories using the `bdb` backend).

概要

```
svnadmin list-dblogs REPOS_PATH
```

描述

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the logfiles accumulate, although most are no longer used and can be deleted to reclaim disk space. See 第 5.4.3 节 "管理磁盘空间" for more information.

名称

svnadmin list-unused-dblogs — Ask Berkeley DB which logfiles can be safely deleted (applies only to repositories using the `bdb` backend).

概要

svnadmin list-unused-dblogs REPOS_PATH

描述

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the logfiles accumulate, although most are no longer used and can be deleted to reclaim disk space. See 第 5.4.3 节 "管理磁盘空间" for more information.

例子

Remove all unused logfiles from the repository:

```
$ svnadmin list-unused-dblogs -/var/svn/repos
/var/svn/repos/log.0000000031
/var/svn/repos/log.0000000032
/var/svn/repos/log.0000000033

$ svnadmin list-unused-dblogs -/var/svn/repos -| xargs rm
## disk space reclaimed!
```

名称

svnadmin load — Read a repository dump stream from stdin.

概要

svnadmin load REPOS_PATH

描述

Read a repository dump stream from stdin, committing new revisions into the repository's filesystem. Send progress feedback to stdout.

选项

--force-uuid
--ignore-uuid
--parent-dir
--quiet (-q)
--use-post-commit-hook
--use-pre-commit-hook

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 -/var/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 \
               -/var/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 [PATH-IN-REPOS]

描述

Print descriptions of all locks in repository *REPOS_PATH* underneath the path *PATH-IN-REPOS*. If *PATH-IN-REPOS* is not provided, it defaults to the root directory of the repository.

选项

无

例子

显示了版本库/var/svn/repos中一个锁定的文件：

```
$ svnadmin lslocks -/var/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

Print the names of all uncommitted transactions. See 第 5.4.3.2 节 "删除终止的事务" for information on how uncommitted transactions are created and what you should do with them.

例子

List all outstanding transactions in a repository:

```
$ svnadmin lstxns -/var/svn/repos/
1w
1x
```

名称

svnadmin recover — Bring a repository database back into a consistent state (applies only to repositories using the `bdb` backend). In addition, if `repos/conf/passwd` does not exist, it will create a default passwordfile .

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 -/var/svn/repos/
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Recovering the database requires an exclusive lock on the repository. (This is a "database lock"; see the sidebar 锁定的三种含义.) If another process is accessing the repository, then **svnadmin recover** will error:

```
$ svnadmin recover -/var/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 -/var/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

Remove one or more locks from each *LOCKED_PATH*.

选项

无

例子

这删除了版本库/var/svn/repos里tree.jpg和house.jpg文件上的锁定：

```
$ svnadmin rmlocks -/var/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

Delete outstanding transactions from a repository. This is covered in detail in 第 5.4.3.2 节 "删除终止的事务".

选项

--quiet (-q)

例子

删除命名的事物:

$ svnadmin rmtxns -/var/svn/repos/ 1w 1x

很幸运,**lstxns**的输出作为**rmtxns**输入工作良好:

$ svnadmin rmtxns -/var/svn/repos/ `svnadmin lstxns -/var/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。

This is similar to using **svn propset** with the --revprop option to set the svn:log property on a revision, except that you can also use the option --bypass-hooks to avoid running any pre- or post-commit hooks, which is useful if the modification of revision properties has not been enabled in the pre-revprop-change hook.

> ⛔ **警告**
>
> 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

--bypass-hooks
--revision (-r) REV

例子

设置修订版本19的日志信息为文件msg的内容：

$ svnadmin setlog -/var/svn/repos/ --r 19 msg

名称

svnadmin setrevprop — Set a property on a revision.

概要

svnadmin setrevprop REPOS_PATH --r REVISION NAME FILE

描述

Set the property *NAME* on revision *REVISION* to the contents of *FILE*. Use `--use-pre-revprop-change-hook` or `--use-post-revprop-change-hook` to trigger the revision property-related hooks (e.g., if you want an email notification sent from your `post-revprop-change-hook`).

选项

```
--revision (-r) ARG
--use-post-revprop-change-hook
--use-pre-revprop-change-hook
```

例子

The following sets the revision property `repository-photo` to the contents of the file `sandwich.png`:

```
$svnadmin setrevprop -/var/svn/repos --r 0 repository-photo
sandwich.png
```

As you can see, **svnadmin setrevprop** has no output upon success.

名称

svnadmin setuuid — 重置版本库的UUID。

概要

svnadmin setuuid REPOS_PATH [NEW_UUID]

描述

Reset the repository UUID for the repository located at *REPOS_PATH*. If *NEW_UUID* is provided, use that as the new repository UUID; otherwise, generate a brand-new UUID for the repository.

选项

无

例子

If you've **svnsync**ed /var/svn/repos to /var/svn/repos-new and intend to use repos-new as your canonical repository, you may want to change the UUID for repos-new to the UUID of repos so that your users don't have to check out a new working copy to accommodate the change:

```
$ svnadmin setuuid -/var/svn/repos-new 2109a8dd-854f-0410-ad31-
d604008985ab
```

As you can see, **svnadmin setuuid** has no output upon success.

名称

svnadmin upgrade — 升级版本库到支持的最新方案版本。

概要

svnadmin upgrade REPOS_PATH

描述

升级位于 *REPOS_PATH* 的版本库到支持的最新方案版本。

This functionality is provided as a convenience for repository administrators who wish to make use of new Subversion functionality without having to undertake a potentially costly full repository dump and load operation. As such, the upgrade performs only the minimum amount of work needed to accomplish this while still maintaining the integrity of the repository. While a dump and subsequent load guarantee the most optimized repository state, **svnadmin upgrade** does not.

警告

在升级之前，你**永远**都应该备份版本库。

选项

无

例子

Upgrade the repository at path /var/repos/svn:

```
$ svnadmin upgrade -/var/repos/svn
Repository lock acquired.
Please wait; upgrading the repository may take some time...

Upgrade completed.
```

名称

svnadmin verify — 验证版本库保存的数据。

概要

```
svnadmin verify REPOS_PATH
```

描述

Run this command if you wish to verify the integrity of your repository. This basically iterates through all revisions in the repository by internally dumping all revisions and discarding the output—it's a good idea to run this on a regular basis to guard against latent hard disk failures and "bitrot." If this command fails—which it will do at the first sign of a problem—that means your repository has at least one corrupted revision, and you should restore the corrupted revision from a backup (you did make a backup, didn't you?).

选项

```
--quiet (-q)
--revision (-r) ARG
```

例子

检验挂起的版本库：

```
$ svnadmin verify -/var/svn/repos/
* Verified revision 1729.
```

9.3. svnlook

svnlook is a command-line utility for examining different aspects of a Subversion repository. It does not make any changes to the repository—it's just used for "peeking." **svnlook** is typically used by the repository hooks, but a repository administrator might find it useful for diagnostic purposes.

Since **svnlook** works via direct repository access (and thus can be used only on the machine that holds the repository), it refers to the repository with a path, not a 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。

9.3.1. svnlook 选项

Options in **svnlook** are global, just as they are in **svn** and **svnadmin**; however, most options apply to only one subcommand since the functionality of **svnlook** is (intentionally) limited in scope:

`--copy-info`
导致 **svnlook changed** 复制源显示详细信息。

`--no-diff-deleted`
Prevents **svnlook diff** from printing differences for deleted files. The default behavior when a file is deleted in a transaction/revision is to print the same differences that you would see if you had left the file but removed all the content.

`--no-diff-added`
Prevents **svnlook diff** from printing differences for added files. The default behavior when you add a file is to print the same differences that you would see if you had added the entire contents of an existing (empty) file.

`--revision (-r)`
Specifies a particular revision number that you wish to examine.

`--revprop`
Operates on a revision property instead of a property specific to a file or directory. This option requires that you also pass a revision with the `--revision (-r)` option.

`--transaction (-t)`
Specifies a particular transaction ID that you wish to examine.

`--show-ids`
Shows the filesystem node revision IDs for each path in the filesystem tree.

9.3.2. svnlook 子命令

Here are the various subcommands for the **svnlook** program.

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) TXN
```

例子

svnlook author垂手可得，但是并不令人激动：

```
$ svnlook author --r 40 -/var/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) TXN
```

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 -/var/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'`
　　Properties of item changed; note the leading underscore

`'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

```
--copy-info
--revision (-r) REV
--transaction (-t) TXN
```

例子

This shows a list of all the changed files and directories in revision 39 of a test repository. Note that the first changed item is a directory, as evidenced by the trailing /:

```
$ svnlook changed --r 39 -/var/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
```

如下是显示文件重命名修订版本的例子：

```
$ svnlook changed --r 64 -/var/svn/repos
A    trunk/vendors/baker/toast.txt
D    trunk/vendors/baker/bread.txt
```

Unfortunately, nothing in the preceding output reveals the connection between the deleted and added files. Use the --copy-info option to make this relationship more apparent:

```
$ svnlook changed --r 64 ---copy-info -/var/svn/repos
A + trunk/vendors/baker/toast.txt
    (from trunk/vendors/baker/bread.txt:r63)
D    trunk/vendors/baker/bread.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) TXN
```

例子

显示测试版本库修订版本40的日期：

```
$ svnlook date --r 40 -/var/svn/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

```
--diff-copy-from
--no-diff-added
--no-diff-deleted
--revision (-r) REV
--transaction (-t) TXN
```

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 -/var/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
================================================================
```

If a file has a nontextual `svn:mime-type` property, the differences are not explicitly shown.

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) TXN
```

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 -/var/svn/repos
trunk/vendors/deli/
```

名称

svnlook help — 求助！

概要

```
Also svnlook --h and svnlook --?.
```

描述

Displays the help message for **svnlook**. This command, like its brother, **svn help**, is also your friend, even though you never call it anymore and forgot to invite it to your last party.

选项

无

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

```
--limit (-l) ARG
--revision (-r) REV
--show-ids
```

例子

This shows the history output for the path /branches/bookstore as of revision 13 in our sample repository:

```
$ svnlook history --r 13 -/var/svn/repos -/branches/bookstore ---
show-ids
REVISION   PATH <ID>
--------   ----------
      13   -/branches/bookstore <1.1.r13/390>
      12   -/branches/bookstore <1.1.r12/413>
      11   -/branches/bookstore <1.1.r11/0>
       9   -/trunk <1.0.r9/551>
       8   -/trunk <1.0.r8/131357096>
       7   -/trunk <1.0.r7/294>
       6   -/trunk <1.0.r6/353>
       5   -/trunk <1.0.r5/349>
       4   -/trunk <1.0.r4/332>
       3   -/trunk <1.0.r3/335>
       2   -/trunk <1.0.r2/295>
       1   -/trunk <1.0.r1/532>
```

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(字节)和日志信息，然后是一个换行符。

选项

```
--revision (-r) REV
--transaction (-t) TXN
```

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 -/var/svn/repos
sally
2003-02-22 17:44:49 --0600 (Sat, 22 Feb 2003)
16
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

Print all information available for the lock at *PATH_IN_REPOS*. If *PATH_IN_REPOS* is not locked, print nothing.

选项

无

例子

这描述了文件tree.jpg的锁定。

```
$ svnlook lock -/var/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 — 日志信息本身，后接换行。

概要

```
svnlook log REPOS_PATH
```

描述

打印日志信息。

选项

```
--revision (-r) REV
--transaction (-t) TXN
```

例子

这显示了实例版本库在修订版本40的日志输出：

```
$ svnlook log -/var/svn/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
--revprop
--transaction (-t) TXN

例子

这显示了HEAD修订版本中文件/trunk/sandwich的"seasonings"属性的值：

$ svnlook pg -/var/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

List the properties of a path in the repository. With --verbose, show the property values too.

别名

pl, **plist**

选项

```
--revision (-r) REV
--revprop
--transaction (-t) TXN
--verbose (-v)
```

例子

这显示了HEAD修订版本中/trunk/README的属性名称：

```
$ svnlook proplist -/var/svn/repos -/trunk/README
  original-author
  svn:mime-type
```

This is the same command as in the preceding example, but this time showing the property values as well:

```
$ svnlook ---verbose proplist -/var/svn/repos -/trunk/README
  original-author -: harry
  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

```
--full-paths
--non-recursive (-N)
--revision (-r) REV
--show-ids
--transaction (-t) TXN
```

例子

This shows the tree output (with nodeIDs) for revision 13 in our sample repository:

```
$ svnlook tree --r 13 -/var/svn/repos ---show-ids
/ <0.0.r13/811>
 trunk/ <1.0.r9/551>
  button.c <2.0.r9/238>
  Makefile <3.0.r7/41>
  integer.c <4.0.r6/98>
 branches/ <5.0.r13/593>
  bookstore/ <1.1.r13/390>
   button.c <2.1.r12/85>
   Makefile <3.0.r7/41>
   integer.c <4.1.r13/109>
```

名称

svnlook uuid — 打印版本库的 UUID。

概要

svnlook uuid REPOS_PATH

描述

Print the UUID for the repository. The UUID is the repository's *u*niversal *u*nique *id*entifier. The Subversion client uses this identifier to differentiate between one repository and another.

选项

无

例子

```
$ svnlook uuid -/var/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。

选项

无

例子

这显示了在实例版本库显示最年轻的修订版本：

```
$ svnlook youngest -/var/svn/repos/
42
```

9.4. svnsync

svnsync is the Subversion remote repository mirroring tool. Put simply, it allows you to replay the revisions of one repository into another one.

In any mirroring scenario, there are two repositories: the source repository, and the mirror (or "sink") repository. The source repository is the repository from which **svnsync** pulls revisions. The mirror repository is the destination for the revisions pulled from the source repository. Each of the repositories may be local or remote—they are only ever addressed by their URLs.

The **svnsync** process requires only read access to the source repository; it never attempts to modify it. But obviously, **svnsync** requires both read and write access to the mirror repository.

警告

svnsync is very sensitive to changes made in the mirror repository that weren't made as part of a mirroring operation. To prevent this from happening, it's best if the **svnsync** process is the only process permitted to modify the mirror repository.

9.4.1. svnsync 选项

Options in **svnsync** are global, just as they are in **svn** and **svnadmin**:

```
--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
```
Prevents caching of authentication information (e.g., username and password) in the Subversion runtime configuration directories.

```
--non-interactive
```
In the case of an authentication failure or insufficient credentials, prevents prompting for credentials (e.g., username or password). This is useful if you're running Subversion inside an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

```
--quiet (-q)
```
请求客户端在执行操作时只显示重要信息。

```
--source-password PASSWD
```
指定你要同步的源 Subversion 服务器的密码。如果没有提供，或者不正确，在需要时，Subversion 会提示你输入。

```
--source-username NAME
```
指定你要同步的源 Subversion 服务器的用户。如果没有提供，或者不正确，在需要时，Subversion 会提示你输入。

```
--sync-password PASSWD
```
指定你要同步的目标 Subversion 服务器的密码。如果没有提供，或者不正确，在需要时，Subversion 会提示你输入。

```
--sync-username NAME
```
指定你要同步的目标 Subversion 服务器的用户。如果没有提供，或者不正确，在需要时，Subversion 会提示你输入。

9.4.2. svnsync 子命令

Here are the various subcommands for the **svnsync** program.

名称

svnsync copy-revprops — Copy all revision properties for a particular revision (or range of revisions) from the source repository to the mirror repository.

概要

```
svnsync copy-revprops DEST_URL [REV[:REV2]
```

描述

Because Subversion revision properties can be changed at any time, it's possible that the properties for some revision might be changed after that revision has already been synchronized to another repository. Because the **svnsync synchronize** command operates only on the range of revisions that have not yet been synchronized, it won't notice a revision property change outside that range. Left as is, this causes a deviation in the values of that revision's properties between the source and mirror repositories. **svnsync copy-revprops** is the answer to this problem. Use it to resynchronize the revision properties for a particular revision or range of revisions.

别名

无

选项

```
--config-dir DIR
--no-auth-cache
--non-interactive
--quiet (-q)
--source-password ARG
--source-username ARG
--sync-password ARG
--sync-username ARG
```

例子

为单个修订版本重新同步修订版本属性：

```
$ svnsync copy-revprops file:///var/svn/repos-mirror 6
Copied properties for revision 6.
$
```

名称

svnsync help — 求助！

概要

```
svnsync help
```

描述

This subcommand is useful when you're trapped in a foreign prison with neither a Net connection nor a copy of this book, but you do have a local Wi-Fi network running and you'd like to sync a copy of your repository over to the backup server that Ira The Knife is running over in cell block D.

别名

无

选项

无

名称

svnsync initialize — Initialize a mirror repository for synchronization from the source repository.

概要

```
svnsync initialize MIRROR_URL SOURCE_URL
```

描述

svnsync initialize verifies that a repository meets the requirements of a new mirror repository —that it has no previous existing version history and that it allows revision property modifications—and records the initial administrative information that associates the mirror repository with the source repository. This is the first **svnsync** operation you run on a would-be mirror repository.

别名

init

选项

```
--config-dir DIR
--no-auth-cache
--non-interactive
--quiet (-q)
--source-password ARG
--source-username ARG
--sync-password ARG
--sync-username ARG
```

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:///var/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:///var/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

The **svnsync synchronize** command does all the heavy lifting of a repository mirroring operation. After consulting with the mirror repository to see which revisions have already been copied into it, it then begins to copy any not-yet-mirrored revisions from the source repository.

svnsync synchronize can be gracefully canceled and restarted.

As of Subversion 1.5, you can limit **svnsync** to a subdirectory of the source repository by specifying the subdirectory as part of the *SOURCE_URL*.

别名

sync

选项

```
--config-dir DIR
--no-auth-cache
--non-interactive
--quiet (-q)
--source-password ARG
--source-username ARG
--sync-password ARG
--sync-username ARG
```

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:///var/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.
$
```

9.5. 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。

Regardless of the access method, once the client has selected a repository by transmitting its URL, **svnserve** reads a file named conf/svnserve.conf in the repository directory to determine repository-specific settings such as what authentication database to use and what authorization policies to apply. See 第 6.3 节 "svnserve - 定制的服务器 " for details of the svnserve.conf file.

9.5.1. svnserve 选项

Unlike the previous commands we've described, **svnserve** has no subcommands—it is controlled exclusively by options.

--daemon (-d)
   Causes **svnserve** to run in daemon mode. **svnserve** backgrounds itself and accepts and serves TCP/IP connections on the svn port (3690, by default).

--foreground
   When used together with -d, causes **svnserve** to stay in the foreground. This is mainly useful for debugging.

--inetd (-i)
   导致 **svnserve** 使用 stdin 和 stdout 文件描述符，适用于 **inetd** 守护进程。

--help (-h)
   显示有用的摘要和选项。

--listen-host=*HOST*
   svnserve监听的*HOST*，可能是一个主机名或是一个IP地址。

--listen-once (-X)
   Causes **svnserve** to accept one connection on the svn port, serve it, and exit. This option is mainly useful for debugging.

`--listen-port=`*`PORT`*
> Causes **svnserve** to listen on *`PORT`* when run in daemon mode. (FreeBSD daemons listen only on tcp6 by default—this option tells them to also listen on tcp4.)

`--pid-file `*`FILENAME`*
> Causes **svnserve** to write its process ID to *`FILENAME`*, which must be writable by the user under which **svnserve** is running.

`--root=`*`ROOT`* (`-r=`*`ROOT`*)
> Sets the virtual root for repositories served by **svnserve**. The pathname in URLs provided by the client will be interpreted relative to this root and will not be allowed to escape this root.

`--threads` (`-T`)
> When running in daemon mode, causes **svnserve** to spawn a thread instead of a process for each connection (e.g., for when running on Windows). The **svnserve** process still backgrounds itself at startup time.

`--tunnel` (`-t`)
> Causes **svnserve** to run in tunnel mode, which is just like the **inetd** mode of operation (both modes serve one connection over `stdin`/`stdout`, and then exit), except that the connection is considered to be preauthenticated with the username of the current UID. This flag is automatically passed for you by the client when running over a tunnel agent such as **ssh**. That means there's rarely any need for *you* to pass this option to **svnserve**. So, if you find yourself typing `svnserve --tunnel` on the command line and wondering what to do next, see 第 6.3.4 节 "穿越 SSH 隧道".

`--tunnel-user NAME`
> Used in conjunction with the `--tunnel` option, tells **svnserve** to assume that *`NAME`* is the authenticated user, rather than the UID of the **svnserve** process. This is useful for users wishing to share a single system account over SSH, but to maintain separate commit identities.

`--version`
> 显示版本信息，版本库后端存在和可用的模块列表，然后退出。

9.6. svndumpfilter

svndumpfilter is a command-line utility for removing history from a Subversion dump file by either excluding or including paths beginning with one or more named prefixes. For details, see 第 5.4.1.3 节 "svndumpfilter".

9.6.1. svndumpfilter 选项

Options in **svndumpfilter** are global, just as they are in **svn** and **svnadmin**:

`--drop-empty-revs`
> If filtering causes any revision to be empty (i.e., causes no change to the repository), removes these revisions from the final dump file.

`--renumber-revs`
> 在过滤后，重新分配版本号。

`--skip-missing-merge-sources`
Skips merge sources that have been removed as part of the filtering. Without this option, **svndumpfilter** will exit with an error if the merge source for a retained path is removed by filtering.

`--preserve-revprops`
If all nodes in a revision are removed by filtering and `--drop-empty-revs` is not passed, the default behavior of **svndumpfilter** is to remove all revision properties except for the date and the log message (which will merely indicate that the revision is empty). Passing this option will preserve existing revision properties (which may or may not make sense since the related content is no longer present in the dump file).

`--quiet`
不显示过滤统计。

9.6.2. svndumpfilter 子命令

svndumpfilter 有许多子命令。

名称

svndumpfilter exclude — Filter out nodes with given prefixes from the dump stream.

概要

svndumpfilter exclude PATH_PREFIX...

描述

This can be used to exclude nodes that begin with one or more *PATH_PREFIX*es from a filtered dump file.

选项

```
--drop-empty-revs
--preserve-revprops
--quiet
--renumber-revs
--skip-missing-merge-sources
```

例子

If we have a dump file from a repository with a number of different picnic-related directories in it, but we want to keep everything *except* the sandwiches part of the repository, we'll exclude only that path:

```
$ svndumpfilter exclude sandwiches < dumpfile > filtered-dumpfile
Excluding prefixes:
   -'/sandwiches'

Revision 0 committed as 0.
Revision 1 committed as 1.
Revision 2 committed as 2.
Revision 3 committed as 3.
Revision 4 committed as 4.

Dropped 1 node(s):
   -'/sandwiches'
```

名称

svndumpfilter include — Filter out nodes without given prefixes from dump stream.

概要

svndumpfilter include PATH_PREFIX...

描述

Can be used to include nodes that begin with one or more *PATH_PREFIX*es in a filtered dump file (thus excluding all other paths).

选项

```
--drop-empty-revs
--preserve-revprops
--quiet
--renumber-revs
--skip-missing-merge-sources
```

例子

If we have a dump file from a repository with a number of different picnic-related directories in it, but want to keep only the sandwiches part of the repository, we'll include only that path:

```
$ svndumpfilter include sandwiches < dumpfile > filtered-dumpfile
Including prefixes:
   -'/sandwiches'

Revision 0 committed as 0.
Revision 1 committed as 1.
Revision 2 committed as 2.
Revision 3 committed as 3.
Revision 4 committed as 4.

Dropped 3 node(s):
   -'/drinks'
   -'/snacks'
   -'/supplies'
```

名称

svndumpfilter help — 求助！

概要

```
svndumpfilter help [SUBCOMMAND...]
```

描述

Displays the help message for **svndumpfilter**. Unlike other help commands documented in this chapter, there is no witty commentary for this help command. The authors of this book deeply regret the omission.

选项

无

9.7. svnversion

名称

svnversion — 总结工作副本的本地修订版本。

概要

svnversion [OPTIONS] [WC_PATH [TRAIL_URL]]

描述

svnversion is a program for summarizing the revision mixture of a working copy. The resultant revision number, or revision range, is written to standard output.

通常在构建过程中利用其输出定义程序的版本号码。

TRAIL_URL, if present, is the trailing portion of the URL used to determine whether *WC_PATH* itself is switched (detection of switches within *WC_PATH* does not rely on *TRAIL_URL*).

When *WC_PATH* is not defined, the current directory will be used as the working copy path. *TRAIL_URL* cannot be defined if *WC_PATH* is not explicitly given.

选项

Like **svnserve**, **svnversion** has no subcommands—only options:

--no-newline (-n)
   忽略输出的尾端新行。

--committed (-c)
   使用最后修改的版本而不是当前的(例如，本地存在的最高修订版本)版本。

--help (-h)
   打印帮助摘要。

--version
   打印 **svnversion** 的版本，无错退出。

例子

如果工作副本都是一样的修订版本(例如，在更新后那一刻)，打印的修订版本是：

```
$ svnversion
4168
```

You can add *TRAIL_URL* to make sure the working copy is not switched from what you expect. Note that the *WC_PATH* is required in this command:

```
$ svnversion -. -/var/svn/trunk
4168
```

对于混合修订版本的工作副本，修订版本的范围会被打印：

```
$ svnversion
4123:4168
```

如果工作副本包含修改，后面会紧跟一个"M"：

```
$ svnversion
4168M
```

如果工作副本已经跳转，后面会有一个"S"：

```
$ svnversion
4168S
```

因此，这里是一个混合修订版本，跳转的工作副本包含了一些本地修改：

```
$ svnversion
4212:4168MS
```

如果从一个目录而不是工作副本调用，**svnversion**假定它是一个导出的工作副本并且打印"exported"：

```
$ svnversion
exported
```

9.8. mod_dav_svn

名称

mod_dav_svn 配置指令 — 通过 Apache HTTP 服务器提供 Subversion 版本库服务的配置说明。

描述

This section briefly describes each Subversion Apache configuration directive. For an in-depth description of configuring Apache with Subversion, see 第 6.4 节 "httpd - Apache 的 HTTP 服务器".)

指令

These are the `httpd.conf` directives that apply to **mod_dav_svn**:

DAV svn
> Must be included in any `Directory` or `Location` block for a Subversion repository. It tells **httpd** to use the Subversion backend for `mod_dav` to handle all requests.

SVNAllowBulkUpdates On|Off
> Toggles support for all-inclusive responses to update-style `REPORT` requests. Subversion clients use `REPORT` requests to get information about directory tree checkouts and updates from **mod_dav_svn**. They can ask the server to send that information in one of two ways: with the entirety of the tree's information in one massive response, or with a *skelta* (a skeletal representation of a tree delta) which contains just enough information for the client to know what *additional* data to request from the server. When this directive is included with a value of `Off`, **mod_dav_svn** will only ever respond to these `REPORT` requests with skelta responses, regardless of the type of responses requested by the client.
>
> Most folks won't need to use this directive at all. It primarily exists for administrators who wish—for security or auditing reasons—to force Subversion clients to fetch individually all the files and directories needed for updates and checkouts, thus leaving an audit trail of `GET` and `PROPFIND` requests in Apache's logs. The default value of this directive is `On`.

SVNAutoversioning On|Off
> When its value is `On`, allows write requests from WebDAV clients to result in automatic commits. A generic log message is auto-generated and attached to each revision. If you enable autoversioning, you'll likely want to set `ModMimeUsePathInfo On` so that `mod_mime` can set `svn:mime-type` to the correct MIME type automatically (as best as `mod_mime` is able to, of course). For more information, see 附录 C, *WebDAV 和自动版本*. The default value of this directive is `Off`.

SVNPath *directory-path*
> Specifies the location in the filesystem for a Subversion repository's files. In a configuration block for a Subversion repository, either this directive or `SVNParentPath` must be present, but not both.

SVNSpecialURI *component*
> Specifies the URI component (namespace) for special Subversion resources. The default is `!svn`, and most administrators will never use this directive. Set this only if there is a pressing

need to have a file named `!svn` in your repository. If you change this on a server already in use, it will break all of the outstanding working copies, and your users will hunt you down with pitchforks and flaming torches.

`SVNReposName` *name*

Specifies the name of a Subversion repository for use in `HTTP GET` responses. This value will be prepended to the title of all directory listings (which are served when you navigate to a Subversion repository with a web browser). This directive is optional.

`SVNIndexXSLT` *directory-path*

Specifies the URI of an XSL transformation for directory indexes. This directive is optional.

`SVNParentPath` *directory-path*

Specifies the location in the filesystem of a parent directory whose child directories are Subversion repositories. In a configuration block for a Subversion repository, either this directive or `SVNPath` must be present, but not both.

`SVNPathAuthz` `On|Off|short_circuit`

Controls path-based authorization by enabling subrequests (`On`), disabling subrequests (`Off`; see 第 6.4.4.3 节 "禁用基于路径的检查"), or querying **mod_authz_svn** directly (`short_circuit`). The default value of this directive is `On`.

`SVNListParentPath` `On|Off`

When set to `On`, allows a `GET` of `SVNParentPath`, which results in a listing of all repositories under that path. The default setting is `Off`.

`SVNMasterURI` *url*

Specifies a URI to the master Subversion repository (used for a write-through proxy).

`SVNActivitiesDB` *directory-path*

Specifies the location in the filesystem where the activities database should be stored. By default, **mod_dav_svn** creates and uses a directory in the repository called `dav/activities.d`. The path specified with this option must be an absolute path.

If specified for an `SVNParentPath` area, **mod_dav_svn** appends the basename of the repository to the path specified here. For example:

```
<Location -/svn>
  DAV svn

  # any -"/svn/foo" URL will map to a repository in
  # -/net/svn.nfs/repositories/foo
  SVNParentPath       -"/net/svn.nfs/repositories"

  # any -"/svn/foo" URL will map to an activities db in
  #  -/var/db/svn/activities/foo
  SVNActivitiesDB     -"/var/db/svn/activities"
</Location>
```

高级日志

This is a list of Subversion action log messages produced by Apache's high-level logging mechanism, followed by an example of the log message. See 第 6.4.5.2 节 "Apache 日志" for details on logging.

检出或导出
```
checkout-or-export /path r62 depth=infinity
```

Commit
```
commit harry r100
```

Diffs
```
diff /path r15:20 depth=infinity ignore-ancestry

diff /path1@15 /path2@20 depth=infinity ignore-ancestry
```

获取目录
```
get-dir /trunk r17 text
```

获取文件
```
get-file /path r20 props
```

获取文件版本
```
get-file-revs /path r12:15 include-merged-revisions
```

获取合并信息
```
get-mergeinfo (/path1 /path2)
```

Lock
```
lock /path steal
```

Log
```
log     (/path1,/path2,/path3)    r20:90    discover-changed-paths
revprops=()
```

版本重演 (svnsync)
```
replay /path r19
```

修订版本属性修改
```
change-rev-prop r50 propertyname
```

修订版本属性列表
```
rev-proplist r34
```

状态
```
status /path r62 depth=infinity
```

Switch
```
switch /pathA /pathB@50 depth=infinity
```

Unlock
```
unlock /path break
```

更新
```
update /path r17 send-copyfrom-args
```

9.9. mod_authz_svn

名称

mod_authz_svn 配置指令 — 通过 Apache HTTP 服务器提供基于路径授权的 Subversion 版本库服务的配置指令。

描述

This section briefly describes each Apache configuration directive offered by **mod_authz_svn**. For an in-depth description of using path-based authorization in Subversion, see 第 6.5 节 "基于路径的授权".)

指令

These are the `httpd.conf` directives that apply to **mod_authz_svn**:

AuthzSVNAccessFile *file-path*
> Consult *file-path* for access rules describing the permissions for paths in Subversion repository.

AuthzSVNAnonymous On|Off
> Set to `Off` to disable two special-case behaviours of this module: interaction with the `Satisfy Any` directive and enforcement of the authorization policy even when no `Require` directives are present. The default value of this directive is `On`.

AuthzSVNAuthoritative On|Off
> Set to `Off` to allow access control to be passed along to lower modules. The default value of this directive is `On`.

AuthzSVNNoAuthWhenAnonymousAllowed On|Off
> Set to `On` to suppress authentication and authorization for requests which anonymous users are allowed to perform. The default value of this directive is `On`.

9.10. Subversion 属性

Subversion allows users to invent arbitrarily named versioned properties on files and directories, as well as unversioned properties on revisions. The only restriction is on properties whose names begin with `svn:` (those are reserved for Subversion's own use). While these properties may be set by users to control Subversion's behavior, users may not invent new `svn:` properties.

9.10.1. 版本控制的属性

These are the versioned properties that Subversion reserves for its own use:

svn:executable
> If present on a file, the client will make the file executable in Unix-hosted working copies. See 第 3.3.2 节 "文件的可执行性".

`svn:mime-type`

If present on a file, the value indicates the file's MIME type. This allows the client to decide whether line-based contextual merging is safe to perform during updates, and can also affect how the file behaves when fetched via a web browser. See 第 3.3.1 节 "文件内容类型".

`svn:ignore`

If present on a directory, the value is a list of *unversioned* file patterns to be ignored by **svn status** and other subcommands. See 第 3.4 节 "忽略未版本控制的条目".

`svn:keywords`

If present on a file, the value tells the client how to expand particular keywords within the file. See 第 3.5 节 "关键字替换".

`svn:eol-style`

If present on a file, the value tells the client how to manipulate the file's line-endings in the working copy and in exported trees. See 第 3.3.3 节 "行结束字符序列" and svn export earlier in this chapter.

`svn:externals`

If present on a directory, the value is a multiline list of other paths and URLs the client should check out. See 第 3.8 节 "外部定义".

`svn:special`

If present on a file, indicates that the file is not an ordinary file, but a symbolic link or other special object. [1]

`svn:needs-lock`

If present on a file, tells the client to make the file read-only in the working copy, as a reminder that the file should be locked before editing begins. See 第 3.7.4 节 "锁定交流".

`svn:mergeinfo`

Used by Subversion to track merge data. See 第 4.3.3 节 "合并信息和预览" for details, but you should never edit this property unless you *really* know what you're doing.

9.10.2. 未版本控制的属性

These are the unversioned properties that Subversion reserves for its own use:

`svn:author`

If present, contains the authenticated username of the person who created the revision. (If not present, the revision was committed anonymously.)

`svn:date`

Contains the UTC time the revision was created, in ISO 8601 format. The value comes from the *server* machine's clock, not the client's.

`svn:log`

保存了描述修订版本的日志信息。

---

[1]As of this writing, symbolic links are indeed the only "special" objects. But there might be more in future releases of Subversion.

```
svn:autoversioned
```
If present, the revision was created via the autoversioning feature. See 第 C.2 节 "自动版本化".

9.11. 版本库钩子

下面是 Subversion 提供的版本库钩子：

名称

start-commit — 开始提交的通知

描述

The start-commit hook is run before the commit transaction is even created. It is typically used to decide whether the user has commit privileges at all.

If the start-commit hook program returns a nonzero exit value, the commit is stopped before the commit transaction is even created, and anything printed to stderr is marshalled back to the client.

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

3. Colon-separated list of capabilities that a client passes to the server, including depth, mergeinfo, and log-revprops (new in Subversion 1.5).

普通用户

访问控制（例如，因为某些原因，临时禁止提交）。

只允许支持某些特性的客户端访问。

名称

pre-commit — 在提交结束之前提醒。

描述

The `pre-commit` hook is run just before a commit transaction is promoted to a new revision. Typically, this hook is used to protect against commits that are disallowed due to content or location (e.g., your site might require that all commits to a certain branch include a ticket number from the bug tracker, or that the incoming log message is nonempty).

If the `pre-commit` hook program returns a nonzero exit value, the commit is aborted, the commit transaction is removed, and anything printed to `stderr` is marshalled back to the client.

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

The `post-commit` hook is run after the transaction is committed and a new revision is created. Most people use this hook to send out descriptive emails about the commit or to notify some other tool (such as an issue tracker) that a commit has happened. Some configurations also use this hook to trigger backup processes.

If the `post-commit` hook returns a nonzero exit status, the commit *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

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. 提交创建的修订版本号

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

The `pre-revprop-change` hook is run immediately prior to the modification of a revision property when performed outside the scope of a normal commit. Unlike the other hooks, the default state of this one is to deny the proposed action. The hook must actually exist and return a zero exit value before a revision property modification can happen.

If the `pre-revprop-change` hook doesn't exist, isn't executable, or returns a nonzero exit value, no change to the property will be made, and anything printed to `stderr` is marshalled back to the client.

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的新值传递给钩子程序。

普通用户

访问控制；变更确认和控制

名称

post-revprop-change — 修订版本属性修改成功的通知

描述

The `post-revprop-change` hook is run immediately after the modification of a revision property when performed outside the scope of a normal commit. As you can derive from the description of its counterpart, the `pre-revprop-change` hook, this hook will not run at all unless the `pre-revprop-change` hook is implemented. It is typically used to send email notification of the property change.

If the `post-revprop-change` hook returns a nonzero exit status, the change *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

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

属性改变通知

名称

pre-lock — 路径锁定尝试的通知。

描述

The `pre-lock` hook runs whenever someone attempts to lock a path. It can be used to prevent locks altogether or to create a more complex policy specifying exactly which users are allowed to lock particular paths. If the hook notices a preexisting lock, it can also decide whether a user is allowed to "steal" the existing lock.

如果 `pre-lock` 钩子返回非零值，锁定动作会终止，任何输出到 `stderr` 的内容都会返回到客户端。

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. 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

The `post-lock` hook runs after one or more paths have been locked. It is typically used to send email notification of the lock event.

If the `post-lock` hook returns a nonzero exit status, the lock *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

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

锁定通知

名称

pre-unlock — 路径解锁尝试的通知。

描述

The `pre-unlock` hook runs whenever someone attempts to remove a lock on a file. It can be used to create policies that specify which users are allowed to unlock particular paths. It's particularly important for determining policies about lock breakage. If user A locks a file, is user B allowed to break the lock? What if the lock is more than a week old? These sorts of things can be decided and enforced by the hook.

如果 `pre-unlock` 返回非零值，解锁操作会终止，任何输出到 `stderr` 的内容都会返回到客户端。

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. 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

The `post-unlock` hook runs after one or more paths have been unlocked. It is typically used to send email notification of the unlock event.

If the `post-unlock` hook returns a nonzero exit status, the unlock *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

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

附录 A. Subversion 快速入门指南

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，然后以工作副本检出，继续我们会给出本书的相关章节的链接。

🚫 **警告**

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。

A.1. 安装 Subversion

Subversion 是基于 APR—Apache 可移植运行库构建的。APR 提供了 Subversion 需要的全部与操作系统相关的操作接口，如磁盘访问，网络访问，内存管理等。这使得 Subversion 能够使用 Apache 作为其网络服务器程序之一，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。APR是一个独立的程序库，任何应用程序都可以使用它。相反，它意味着 Subversion 能够在所有可运行 Apache 服务器的操作系统上运行，如 Windows, Linux, 各种 BSD, Mac OS X, Netware 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(RPM, DEB, ports 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 Subversion 1.4 开始，所有依赖的库(如 apr，apr-**util** 和 neon 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源代码的工作副本：[1]

```
$ 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
…
```

---

[1]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。

上面的命令会检出最新(尚未发布)的 Subversion 源代码版本到你的当前工作目录的名为 `subversion` 的子目录中。很明显，你可以调整最后的参数改为你需要的。不管你怎么称呼它，在操作完成后，你已经有了 Subversion 的源代码。当然，你还是需要得到一些支持库(apr，apr-util 等等)——参见工作副本根目录的 `INSTALL` 以了解详情。

A.2. 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…。"

这是一个快速教程，能够帮助你熟悉 Subversion 的基本配置和操作。在结束这个教程时，你应该对 Subversion 的典型用法有了一个基本认识。

注意

在类 Unix 操作系统上运行附录中的例子(假定你做了显而易见的调整，本教程也能在 Windows 命令行执行)，首先需要正确安装 Subversion 客户端程序 **svn**，以及管理工具 **svnadmin**。我们还假定你使用 Subversion 1.2 或更新版本(可以运行 **svn --version** 来检查)。

Subversion 的所有版本化数据都储存在中心版本库中。因此我们需要先创建一个版本库：

```
$ svnadmin create -/var/svn/repos
$ ls -/var/svn/repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

这个命令创建了一个新目录 `/var/svn/repos`，并在其中创建了一个 Subversion 版本库。这个目录里主要保存了数据库文件(还有其它一些文件)。如果你察看它，不会找到你的版本化文件。参见第 5 章 版本库管理，以了解更多版本库创建和维护方面的信息。

在 Subversion 中没有"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，关于这方面内容的论述，参见第 5.2.1 节 "规划你的版本库结构"。不论是哪种方式，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`。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(参见第 2.2 节 "导入数据到你的版本库")将其导入到版本库中：

```
$ svn import -/tmp/myproject file:///var/svn/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"工作副本"，这是一个私有工作区。从 Subversion 版本库里"检出" myproject/trunk 工作副本的操作如下：

```
$ svn checkout file:///var/svn/repos/myproject/trunk myproject
A  myproject/foo.c
A  myproject/bar.c
A  myproject/Makefile
…
Checked out revision 1.
```

　　现在，myproject 目录下有一个版本库数据的个人副本。我们可以在这个工作副本中编辑文件，并将修改提交到版本库中。

• 进入工作副本目录，编辑某个文件的内容。

• 运行 **svn diff**，以标准差异格式查看你的修改。

• 运行 **>svn commit**，将你的修改提交到版本库中。

• 运行 **svn update**，让你的工作副本更新到版本库的"最新"版本。

完整的工作副本操作指南，请参见第 2 章 **基本使用**。

现在，你可以选择将 Subversion 版本库发布到网络上。参见第 6 章 **服务配置**，以了解不同服务器软件的使用以及配置方法。

完整的工作副本操作指南，请参见第 2 章 **基本使用**。

附录 B. CVS 用户的 Subversion 指南

这个附录可以作为 CVS 用户开始使用 Subversion 的指南，实质上就是在"10000 英尺"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 基础用户，但是仍然需要一些新的特性和设计来修正一些 CVS "不好的"行为，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。

B.1. 版本号现在不同了

在 CVS 中，版本号是针对每个文件的，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 中，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树；本质上，版本库是一系列树，每棵树都有一个版本号。当有人谈论"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 的版本 5"是不正确的，相反，一个人会说"foo.c 在版本 5 出现"。同样，我们在假定文件的进展时也要小心，在 CVS 中，文件 foo.c 的版本 5 和 6 一定是不同的，在 Subversion 中，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。"

关于此专题的更多细节，请参见第 1.3.3 节 "修订版本"。

B.2. 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"调度"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版本号。(更准确的说，谈论"版本 5 的目录foo/"是正确的。)

让我们再讨论最后一点。目录版本化是一个困难的问题；因为我们希望允许混合版本的工作副本，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 的版本 5 就是一个谎言。然

而，如果说我们在提交之后增加了 `foo` 的版本号码，这也是一个谎言；还有一些 `foo` 的修改我们没有得到，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**，所有的变更都会与版本库同步，并且目录的新版本号会正确设置。**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，你的工作副本会保存"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。但是再次说，这还是一个谎言，因为这个目录还没有发生添加和删除，因为还没有发生更新。**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第 1.3.5 节 "混合修订版本的工作副本"。

B.3. 更多离线操作

近些年来，磁盘空间变得异常便宜和富余，但是网络带宽还没有，因此 Subversion 的工作副本为紧缺资源进行了优化。

管理目录 `.svn` 与 CVS 有同样的目的，除了它还保存了只读的文件"原始"副本，这允许你做许多离线操作：

svn status
显示你所做的本地修改(见第 2.4.3.1 节 "查看你的修改概况")

svn diff
显示修改的详细信息(见see 第 2.4.3.2 节 "检查你的本地修改的详情")

svn revert
删除你的本地修改(见第 2.4.4 节 "取消本地修改")

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— **svn revert** —是新的；它不仅仅撤销本地修改，也会取消如增加和删除的调度操作，这是推荐的恢复文件方式；删除文件，然后运行**svn update**也可以工作，但是这样侮辱了更新操作的作用，而且，我们在这个主题…

B.4. 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。

命令 **cvs status** 有两个目的：第一，显示用户在工作副本的所有本地修改，第二，显示给用户哪些文件是最新的。很不幸，因为 CVS 的输出难以阅读，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副本。

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。

B.4.1. 状态

svn status 打印所有本地修改的文件。缺省情况下，不会连接版本库。然而这个命令接受一些选项，下面是是一些最常用的:

-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)。

svn status 命令有两种输出格式。默认是"短"格式，本地修改的输出就像这样:

```
$ 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，增加了两列。第二列的星号表示了文件或目录是否过期。第三列显示了工作副本的版本号。在上面的例子里，星号表示如果进行更新，faces.html 会被修改，而bloo.png 则是版本库新加的文件(bloo.png前面的版本号为空，表示这个文件在工作副本不存在)。

此刻，你应该赶快看一下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 第 3.8 节 -"#部定义"
?   Resource is not under version control
```

```
!     Resource is missing or incomplete (removed by a tool other
than
        Subversion)
```

关于 **svn status** 的详细讨论，参见第 2.4.3.1 节"查看你的修改概况"。

B.4.2. 更新

svn update会更新你的工作副本，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，参见第 2.4.1 节"更新你的工作副本"。

B.5. 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。

> **警告**
>
> 因为 Subversion 把分支和标签作为普通目录看待，你项目不同的开发线可能位于主项目目录的不同子目录里。所以记住要检出你所要的保存特定开发线的 URL，而不是项目的根 URL。如果你错误的检出了项目的根，你会惊讶的发现你的项目副本包含了所有的分支和标签。[1]

B.6. 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(或者称为"属性")。属性是在工作副本中文件或目录关联的任意名称/值对。

使用 **svn propset** 和 **svn propget**子命令，可以设置或得到属性名称。列出对象所有的属性，使用 **svn proplist**。

更多信息见第 3.2 节"属性"。

B.7. 解决冲突

CVS 使用嵌入"冲突标志"来标记冲突，并且在更新或合并操作时打印 C。历史上，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 solves this problem in a pair of ways. First, when a conflict occurs in a file, Subversion records the fact that the file is in a state of conflict, and won't allow you to commit changes to that file until you explicitly resolve the conflict. Second, Subversion 1.5 provides

---

[1]如果在检出完成之前没有消耗完磁盘空间的话。

interactive conflict resolution, which allows you to resolve conflicts as they happen instead of having to go back and do so after the update or merge operation completes. See 第 2.4.5 节 "解决冲突(合并别人的修改)" for more about conflict resolution in Subversion.

B.8. 二进制文件和行结束标记转换

在大多数情况下，Subversion 比 CVS 能更好的处理二进制文件。因为 CVS 使用 RCS，它只能存储已经修改的二进制文件的完整副本。但是 Subversion 使用二进制差异算法来表示文件的区别，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增量的(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(因为关键字扩展和行结束符号的转换)。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(详情见第 3.5 节 "关键字替换"和第 3.3.3 节 "行结束字符序列")这样做，它不会做任何关键字扩展或行结束符号的转换操作。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换的状态。

第二，Subversion 维护了一个内部的观念来区别一个文件是"文本"文件，还是"二进制"文件，但这个观念**只**在工作副本非常重要。在执行 **svn update** 期间，Subversion 会对本地修改的文本文件执行基于上下文的合并，但是对二进制文件不会。

为了检测一个基于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。否则 Subversion 认为它是二进制文件。Subversion 也会在执行 **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)。

B.9. 版本化的模块

不像 CVS，Subversion 工作副本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)，然后执行 **svn update** 会适当的更新工作副本，增加或删除组件。

Subversion定义了模块作为一个目录属性的目录列表：见第 3.8 节 "外部定义"。

B.10. 认证

通过 CVS 的 pserver，你需要在读写操作之前"登陆"到服务器(使用 **cvs login** 命令)—即使是匿名操作。当 Subversion 版本库使用 Apache 的 **httpd** 或 **svnserve** 作为服务器时，你不需要在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)。所以如果你的工作副本是全球可读的，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，当使用基于 SSH 隧道的 **svnserve** 服务器时，使用 `svn+ssh://` 的 URL 方案。在这种情况下，**ssh** 会在通道刚开始时无条件的要求认证。

B.11. 迁移 CVS 版本库到 Subversion

或许让 CVS 用户熟悉 Subversion 的最好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策略等。但是还是有许多工具声称，至少部分具备了转换 CVS 版本库为 Subversion 版本库的能力。

最流行的(好像是最成熟的)转换工具是 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。这个工具只运行一次：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 Subversion 版本库或可移植的 Subversion 转储文件。参见其网站，以了解详细的指令和附加说明。

附录 C. WebDAV 和自动版本

WebDAV 是 HTTP 的一个扩展，作为一个文件共享的标准不断流行。当今的操作系统变得极端的 web 化，许多内置了对装载 WebDAV 服务器导出的"共享"的支持。

如果你使用 Apache 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 如何与支持 WebDAV 的软件交互工作。

C.1. 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的扩展方法来把 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 加载(类似于 NFS 或 SMB) 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 遗漏了版本概念，几年之后，另一个委员会负责撰写 RFC 3253。新的 RFC 为 WebDAV 增加了版本概念，将"V"加入"DAV"—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 包含了基本的 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—仅举几例，如 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。也有几个商业的 WebDAV 服务器，例如 Microsoft 自己的 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 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要。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。它已经有了支持 WebDAV 服务的模块。DeltaV 是一个较新的规范。希望 Subversion 服务器模块(**mod_dav_svn**)最终能够成为一个开源的 DeltaV 参考实现。但非常不幸，DeltaV 的版本模型过于具体，并且与 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** 不是一个完全实现的 DeltaV 服务器，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，这种情况是否值得补救。改变 Subversion 的设计来匹配 DeltaV 看起来相当不现实，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 特性，但缺乏这样做的动力—几乎没有能与之交户的 DeltaV 客户端。

C.2. 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，服务器会接受基本 WebDAV 客户端的写请求。服务器的行为就像客户端**已经**发出了一些列正确的版本请求，执行了提交。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，所以这个特性的用例适用于管理员与非技术用户一起工作。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。他们操作这个目录就像普通目录一样：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东西。任何管理员(或博学的用户)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 和后续的版本来说，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 -/var/svn/repository
  SVNAutoversioning on
</Location>
```

当激活了 Subversion 自动版本化，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通用的日志信息。

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，导致产生数量非常巨大的自动提交版本。例如，当保存数据时，许多客户端会使用一个 PUT 请求来创建一个 0 字节的文件，然后紧跟一个 PUT 请求写真实的文件数据。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，这会产生更多的提交。

如果你有发送邮件的 post-commit 钩子程序，你可能想完全禁止发送邮件，或者只针对部分目录发送邮件；它依赖于你认为大量的邮件通知是否还有价值。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 共享的目录在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)的人检出一份工作副本，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 -/var/svn/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 类型。这个模块查看文件的扩展名，有可能的话还包括检查内容；如果文件符合某个通用模式，就会自动设置文件的属性 svn:mime-type。

C.3. 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。这些分类清楚的定义了 WebDAV 用户可用的功能。表 C.1 "常用的 WebDAV 客户端" 将支持 WebDAV 的常见软件进行了分类，并提供了的简短描述。可以在后面的章节找到这些软件的详细信息。

表 C.1. 常用的 WebDAV 客户端

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，允许对 WebDAV 的 URL 直接读写
cadaver	独立的 WebDAV 应用程序		X	X	命令行 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
Adobe Dreamweaver	独立的 WebDAV 应用程序	X			Web 创作软件，可以直接读写 WebDAV 的 URL

软件	类型	Windows	Mac	Linux	描述
Microsoft Office	独立的 WebDAV 应用程序	X			Office 产品套件，有几个组件可以直接读写 WebDAV 的 URL
Microsoft Web 文件夹	文件浏览器 WebDAV 扩展	X			GUI 文件浏览器程序，可以对 WebDAV 共享执行目录树操作
GNOME Nautilus	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
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 共享的操作系统。
Novell NetDrive	WebDAV 文件系统实现	X			为远程 WebDAV 共享分配 Windows 驱动器盘符的驱动映射程序
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			文件传输软件，也就是可以为加载的远程 WebDAV 共享赋予 Windows 驱动器盘符。
davfs2	WebDAV 文件系统实现			X	允许加载 WebDAV 共享的 Linux 文件系统驱动

C.3.1. 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。

C.3.1.1. Microsoft `Office`，`Dreamweaver`，`Photoshop`

在 Windows 中，有许多已知的应用程序支持 WebDAV 客户端功能，例如微软的 `Office`，[1]Adobe 的 Photoshop 和 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 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` 对话框完全不允许人们输入路径或 URL。这好像看起来 Macintosh 版本的应用程序不会支持 WebDAV 特性了，但实际上是因为 OS X 已经实现了底层的文件系统级的 WebDAV 支持。

C.3.1.2. `Cadaver`，DAV 浏览器

Cadaver 是一个简单的 Unix 命令行的 WebDAV 共享浏览和修改程序。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(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 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，并复制, 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。采用类 Apache 的许可证，网站是 http://www.ics.uci.edu/~webdav/。DAV Explorer 与 cadaver 的功能差不多，优点可移植，并有一个用户友好的 GUI 程序。它也是最早支持 WebDAV 访问控制协议(RFC 3744)的客户端之一。

当然，在这种情况下 DAV Explorer 的 ACL 支持没有任何用处，因为 **mod_dav_svn** 不支持 ACL。事实上，Cadaver 和 DAV Explorer 支持的一些有限的 DeltaV 命令也并不是很有用，因为它们不允许 `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。

---

[1]由于某些原因，`Microsoft` Access 不再支持 `WebDAV`，但是其它 Office 软件仍旧支持 `WebDAV`。

C.3.2. 文件浏览器的 **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，在共享的项目上执行基本的树编辑操作。例如 Windows 浏览器可以以"网络位置"方式浏览 WebDAV 服务器。用户可以拖入和拖出文件，或者是改名，复制或删除其中的文件。但是因为它只是文件浏览器的一个特性，所以 DAV 对普通应用不可见。所有的 DAV 交互必须通过浏览器界面。

C.3.2.1. Microsoft Web 文件夹

Microsoft 是 WebDAV 规范最早的支持者，从 Windows 98 开始提供客户端，被称作"网络文件夹"。这个客户端在 Windows NT 4.0 和 Windows 2000 上也存在。

最早的网络文件夹客户端是浏览器的扩展，它是浏览文件系统的 GUI 程序。它工作良好。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 的发行，`Microsoft` 开始了网络文件夹的新实现，叫做"`WebDAV Mini-Redirector.`"。这个新的实现是文件系统级的客户端，允许 WebDAV 装载到驱动器盘符上。不幸的是，这个实现充满难以相信的问题。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 HTTP 基本认证，按照 `HOST\username` 格式发送用户名。这类互操作性问题在网络上大量传播，使大量用户受挫。即使是 Apache WebDAV 的作者 Greg Stein 也建议不要对 Apache 服务器使用 XP 的网络文件夹。

Windows Vista's initial implementation of Web Folders seems to be almost the same as XP's, so it has the same sort of problems. With luck, Microsoft will remedy these issues in a Vista Service Pack.

However, there seem to be workarounds for both XP and Vista that allow Web Folders to work against Apache. Users have mostly reported success with these techniques, so we'll relay them here.

On Windows XP, you have two options. First, search Microsoft's web site for update KB90730, "Software Update for Web Folders." This may fix all your problems. If it doesn't, it seems that the original pre-XP Web Folders implementation is still buried within the system. You can unearth it by going to Network Places and adding a new network place. When prompted, enter the URL of the repository, but *include a port number* in the URL. For example, you should enter **http://host/repos** as **http://host:80/repos** instead. Respond to any authentication prompts with your Subversion credentials.

On Windows Vista, the same KB90730 update may clear everything up. But there may still be other issues. Some users have reported that Vista considers all `http://` connections insecure, and thus will always fail any authentication challenges from Apache unless the connection happens over `https://`. If you're unable to connect to the Subversion repository via SSL, you can tweak the system registry to turn off this behavior. Just change the value of the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\WebClient\Parameters\BasicAuthLevel` key from **1** to **2**. A final warning: be sure to set up the Web

Folder to point to the repository's root directory (/), rather than some subdirectory such as /trunk. Vista Web Folders seems to work only against repository roots.

In general, while these workarounds may function for you, you might get a better overall experience using a third-party WebDAV client such as WebDrive or NetDrive.

C.3.2.2. `Nautilus, Konqueror`

Nautilus 是 GNOME 桌面(http://www.gnome.org)的官方文件管理/浏览器,KDE 桌面(http://www.kde.org)的则是 Konqueror。两个应用程序都是浏览器级别的 WebDAV 客户端,对自动版本化的版本库工作良好。

在 GNOME 的 Nautilus 中,选择 File→Open `location`,在显示的对话框中输入 URL。版本库就会象其它文件系统一样显示。

在 KDE 的 Konqueror 中,你需要在地址栏使用 `webdav://` 方案来输入 URL。如果你输入 `http://` 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**,而不是 **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。

C.3.3. WebDAV 的文件系统实现

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 的透明访问,应用程序甚至意识不到发生了 WebDAV 请求。

C.3.3.1. `WebDrive, NetDrive`

WebDrive 和 NetDrive 都是优秀的商业产品,允许将 WebDAV 绑定到 Windows 的盘符。从而你可以同对真实硬盘一样操作这些 WebDAV 后端支持的共享。WebDrive 可以从 South River Technologies(http://www.southrivertech.com) 购买。Novell 的 NetDrive 可以在网络上免费下载,但用户还是需要有一个 NetWare 许可证。

C.3.3.2. 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→Connect to Server 菜单项。输入 WebDAV 的 URL,会在桌面显示一个磁盘,就像其它装载的卷。你也可以从 Darwin 终端通过 **mount** 命令装载类型为 webdav 的文件系统:

```
$ 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 不能装载为可读写,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。

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 中的 BrowserMatch 指令:

```
BrowserMatch -"^WebDAVFS/1.[012]" redirect-carefully
```

C.3.3.3. Linux davfs2

Linux davfs2 是一个 Linux 内核的文件系统模块，开发主页是 http://dav.sourceforge.net/。一旦安装 davfs2，你就可以使用通常的 Linux mount 命令装载 WebDAV 网络共享：

```
$ mount.davfs http://host/repos -/mnt/dav
```

附录 **D.** 版权

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
Work is licensed under the CCPL, neither party will use the
trademark
"Creative Commons" or any related trademark or logo of Creative
Commons without the prior written consent of Creative Commons. Any
permitted use will be in compliance with Creative Commons'

then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

==

索引