Subversion 版本控制 [草稿]

针对 Subversion 1.8

(编译自 r6050)

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

Subversion 版本控制 [草稿]: 针对 Subversion 1.8: (编译自 r6050)

由 Ben Collins-Sussman, Brian W. Fitzpatrick 和 C. Michael Pilato

目录

插图清单

表格清单

范例清单

写在前面

Karl Fogel

2004 年 3 月 14 日, 芝加哥

一个差劲的常见问题列表 (Frequently Asked Questions, FAQ) 总是充斥着 作者希望听到的问题, 而不是人们真正想问的问题. 读者可能遇到过下面这种类型 的问题:

> Q: 如何使用 Glorbosoft XYZ 才能最大程度地提高团队效率?
>
> A: 很多客户想知道如何利用我们的革命性办公套件最大程度地提高 效率, 答案非常简单, 首先, 点击菜单项 `File`, 滚动到 `Increase Productivity`, 然后 ...

这种问题完全不符合 FAQ 的精神. 没人会打电话给技术支持中心询问“ 如何提高生产效率”, 相反, 人们经常问一些非常具体的问题, 例如 “怎样做才能让日历系统提前两天—而不是一天—提醒相关用户 ”, 但是编撰 FAQ 比发现真正的问题要容易得多. 构建一张真实的 FAQ 列 表需要持之以恒的辛勤工作: 在软件的整个生命周期, 需要跟踪每一个收到的 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 够真实地反映出用户的感受. 这个过程非常需要耐心, 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—最重要的是要有开放的视野和精确的记录.

我喜欢这本书的原因是它正是按照这种过程写出来的. 本书是作者和用户直接 交互的结果, 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询问同样的基本问题: 使用 Subversion 的标准流程是什么? 分支 (branch) 与标签 (tag) 的工作方式同其他 版本控制系统一样吗? 我怎么才能知道某处修改是谁做的?

日复一日看到相同问题的烦闷, 促使 Ben 在 2002 年的夏天努力了一个月, 写了一本 *The Subversion Handbook*, 这是一本 60 页 厚的手册, 涵盖了 Subversion 所有的基础使用知识. 手册没有说明定稿的最终时间, 但它随着 Subversion 的每个版本一起发布, 帮助过很多用户跨过学习之初的艰难. 当 O'Reilly 决定出版一本详尽的 Subversion 图书时, 扩充手册是最快的捷径.

新书的三位作者面临着一个不寻常的机会, 从职责上讲, 他们的任务是以目录 和草稿为基础, 自上而下地写出一部著作. 但事实上, 他们的灵感来自一些非常具体 的内容, 稳定却难以组织—Subversion 被数以千计的早期用户使用, 他们提供 了大量的反馈, 这些反馈不仅仅针对 Subversion, 还包括它的文档.

在本书的写作过程中, Ben, Mike 和 Brian 像鬼魂一样游荡在 Subversion 的邮件列表和聊天室里, 仔细研究用户遇到的实际问题, 监控反馈信息也是他 们在 CollabNet 的工作内容之一, 这给他们撰写 Subversion 文档提供了巨大的便 利. 本书建立在丰富的使用经验上, 而不是流水般脆弱的想像, 它结合了用户手册 和 FAQ 的优点, 初次阅读时, 这种二元性的优势可能并不明显. 从表面上看, 本书只是从头到尾地描述了软件的细节, 书中的内容包括一章概述, 一章不可或缺 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. 一段时间后, 当你再次翻开本书查找一些特定问题的答案时, 这种二 元性才得以显现: 书中生动的细节一定是来自不可预测的实际用例的提炼, 大多数 内容是源于用户的需求和视角.

当然, 没人可以承诺本书可以回答你的所有问题. 尽管有时候一些前人提问 的惊人一致性让你感觉是心灵感应, 你仍有可能在社区的知识库里摔跤, 空手而 归. 如果有这种情况发生, 最好的办法是把问题发送到 `<users@subversion.apache.org>`, 作者还在那里关注着社区, 不 仅仅是本书的三位作者, 还包括许多曾经做出修正与提供原始材料的人. 从社区 的角度看, 帮你解决问题也是为了改进本书和 Subversion. 他们渴望你的信息, 不仅仅可以帮助你, 也可以帮助他们. 与 Subversion 这样活跃的自由软件项目 在一起, 你不会孤军奋战.

希望本书能成为你的好伙伴.

前言

> “即使你能确认什么是完美，也不要让完美成为你的敌人，更何况你 不能确认．因为落入过去陷阱的不悦，你会在设计时因为担心自己的缺陷而无 所作为．”
>
> —Greg Hudson, Subversion 开发人员

在开源软件世界，并发版本控制系统 (Concurrent Versions System, 简称 CVS) 长久 以来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，宽松的操作，以及对网络的支持（网络使众多身处不同地方的程序员可以共享他们的工作成果），正符合了开源世界协作的精神，CVS 和它半混乱的开发模式 已经成为开源文化的基石．

但是 CVS 并非毫无缺陷，而修正这些缺陷必定会耗费大量的精力．Subversion 是以 CVS 继任者的面貌出现的新型版本控制系统，Subversion 的设计者力图通过两 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，同时尽可能避免 CVS 的显著缺陷．虽然这些努力的结果并没有引起一场版本控制系统的伟大革命，但 Subversion 确实是一个非常强大，实用，灵活的版本控制系统．

本书是为 Apache™ Subversion®[1] 1.8 系列版本撰写的．我们努力让书中的内容详尽 准确，不过 Subversion 有一个非常活跃的开发社区，已经有很多新特性和改进 计划在 Subversion 的新版本中实现，对于新版本，本书的内容可能会与实际 情况有所出入．

什么是 Subversion?

Subversion 是一个 免费/开源 的 *版本控制系统* (*version control system*, 简称 VCS)，也就是说，Subversion 可以跨越时间对文件和目录，以及它们的修改进行管理．这就允许你恢复 数据的旧版本，或检查数据的修改历史．由于这个特点，很多人把版本控制系统 看成是一种 “时间机器”．

Subversion 可以跨越网络进行操作，这就允许多个用户在不同的机器上工作．从某种程度上讲，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．由于所有的工作都被记录在 案，也就不用担心并行开发会降低软件的质量—如果出现不正确的修改，只 要撤消那一次修改即可．

某些版本控制系统同时也是 *软件配置管理* (*software configuration management*, 简称 SCM) 系统．这种系统经过精巧的设计，专门用于管理源代码树，具备许多与软件开发有关的 特性—例如理解编程语言，或者提供了程序构建工具．但 Subversion 不是 SCM，它是一个通用系统，可以管理 *任意* 类型的文件集合．对读者而言，这些文件可能是源代码文件—对别人来说，可能是一个货物清单 或数字电影．

Subversion 是正确的工具吗?

如果你是一个考虑如何使用 Subversion 的用户或系统管理员，你要问自 己的第一件事就是：“它适合这项工作吗?”，Subversion 是一个梦幻 般的锤子，但要小心不要把任何问题都当作钉子．

首先要确认的是你的工作是否需要通常意义上的版本控制．如果你需要对 旧版本的文件和目录进行归档，而且还要查看它们的修改历史，那么你就需要 使用版本控制工具．如果你要和同事（通过网络）协作编写文档，而且还要求 记录每个人的修改，此时你也可以使用版本控制工具．实际上，这正是软件 开发需要使用版本控制工具（例如 Subversion）的原因—在开

[1] 在本书的剩下部 分，我们都简单地写成 “Subversion”，你应该感谢这样做节省了 很多篇幅．

发团队 中工作本身就是一件社会活动: 源代码文件的修改经常需要被讨论, 生成, 评价, 有时候还要撤消修改. 版本控制工具对这种类型的协作工作很有帮助.

使用版本控制也是有代价的. 如果需要由你自己来管理版本控制系统, 你将 为此消耗相当多的精力, 除非把管理任务交给第三方负责. 在日常的工作中, 你不能像往常那样复制, 移动, 重命名或删除文件, 相反, 你需要按照版本控制 系统的要求来完成.

即使你可以接受使用版本控制系统带来的 代价/好处, 你也不能仅仅因为 它 可以 满足你的要求而使用它, 认真考虑一下你的 需求是否还有更好的工具可以实现. 举例来说, 由于 Subversion 把数据复制 给所有的相关人员, 所以经常有新手错误地把它当成了一种普通的分布式系统, 人们有时候使用 Subversion 管理大量的照片, 数字音乐或软件包, 但问题是 人们通常很少修改这些文件. 文件集合随着时间不断增长, 但集合中的单个 文件却没有发生变化, 在这种场景中使用 Subversion 就显得 “过犹不及”[2]. 有很多简单的工具可以用来复制 数据, 而且没有多余的修改跟踪开销, 例如 *rsync* 或 *unison*.

一旦决定好了你确实需要一个版本控制系统, 你将发现可选择的工具非常丰富. 第一次设计并发布 Subversion 时, 它的主要版本控制策略是 集中式的版本控制 (*centralized version control*)—有一个远程 的主仓库, 仓库中存放了被版本控制的数据, 用户在本地操作数据的浅拷贝副本. Subversion 发布后, 很快就显现出它在版本控制领域的领导地位, 它的使用范围 越来越广泛, 越来越多的旧版本控制系统被它取代, 即使是在今天, 它在版本控 制领域也占据着重要地位.

从那时起也发生了很多事情. 在 Subversion 开始出现的那一年, 一种新的版本控制 策略—分布式的版本控制 (*distributed version control*)—开始受到越来 越多的关注, 应用也越来越广泛. 一些版本控制工具, 例如 Git (http://git-scm.com/) 和 Mercurial (http://mercurial-scm.org/) 成为了分布式版本控制系统 (DVCS) 的宠儿. 分布式的版本控制利用不断提高的网络连接速度和低廉的存储 开销, 提出了一种和集中式模型完全不同的方法. 首先最明显的区别是它们不 需要一个远程的中央仓库, 每一个用户在本地都有一份完整的版本历史. 用户 之间仍然需要协作, 但不需要通过一台中央节点, 可以直接进行交互. 实际上, 项目版本化数据的权威 “主线” 通常只是各个协作者协商出来的 结果.

每一种版本控制系统都有它自己的长处和短处. 也许 DVCS 的两个最 大的好处是日常操作非常高效 (因为所有的数据都在本地存放) 和更强大的 分支合并 (因为合并算法是 DVCS 的核心). 缺点是分布式的版本控制更加 复杂, 这种复杂性可能会给协作带来不小的挑战. 另外, DVCS 工具工作得 很好, 部分是因为本来由中央系统持有的控制权—基于路径的访问控制, 对单个项目进行更新或回溯等—转移到了用户手止. 幸运的是, 许多明智的组织已经发现没必要对两种版本控制系统进行严谨地辩论, Subversion 和 DVCS 工具 (例如 Git) 可以和谐地一起工作, 每一种工具都 能发挥出自己的优势.

本书是关于 Subversion 的, 所以我们不会拿它和其他工具进行完整地 对比, 读者在选择版本控制工具时应该查看所有的备选, 然后从中选择一个 最适合自己和同事的工具. 如果最终选择了 Subversion, 那么你将从剩下 的几章里学到如何高效地使用 Subversion.

Subversion 的历史

2000 年, CollabNet 公司 (http://www.collab.net) 开始寻找 CVS 替代产品的开发人员, CollabNet 提供了一个名为 CollabNet Enterprise Edition (CEE) 的协作软件套装[3], 这套软件的一个组成部分就是 版本控制系统. 虽然 CEE 最初使用 CVS 作为版本控制系统, 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 没有可以自由使用的替代品. 于是 CollabNet 决定重新开发一个新的版本控制 系统, 该系统要保留 CVS 的基本思想, 但是要修正 CVS 的错误和不合理的特性.

[2] 还有另一种说法, “用别克 打苍蝇”
[3] CEE 现在已经被 CollabNet TeamForge 代替

2000 年 2 月，CollabNet 联系到 *Open Source Development with CVS* (Coriolis, 1999) 的作者 Karl Fogel，询问他是否愿意为这个新项目工作．巧合的是，当时 Karl 正在和朋友 Jim Blandy 讨论设计一个新的版本控制系统．1995 年 他们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，但还是经常使用 CVS 进行日常工作．使用 CVS 的挫折促使 Jim 认真地思考如何管理版本化的数据．他当时不仅使用了名字 Subversion，而且已经完成了版本库的最初设计．所以当 CollabNet 提出邀请时，Karl 马上同意为这个项目工作，同时 Jim 也得到了他的雇主—Red Hat 软件 公司—的同意，允许他为这个项目工作，并且没有限定期限．CollabNet 雇佣了 Karl 和 Ben Collins-Sussman，详细设计从 2000 年 5 月开始，在 CollabNet 的 Brian Behlendorf 和 Jason Robbins，以及 Greg Stein（独立开发者，参与了 WebDAV/DeltaV 规范的制订）恰到好处的激励 下，Subversion 吸引到了众多开发者的注意，结果是许多对 CVS 有过失望 经历的人都很乐意为这个项目做些事情．

最初的设计小组设计了几个简单的开发目标．他们不想在版本控制方法 学中开垦处女地，他们只想修正 CVS．他们决定 Subversion 应该符合 CVS 的特征，保留相同的开发模式，但不能包含 CVS 的显著缺陷．尽管 Subversion 不必成为 CVS 的完全替代品，但应该与 CVS 保持足够的相似性，使得 CVS 用户可以轻松地迁移到 Subversion．

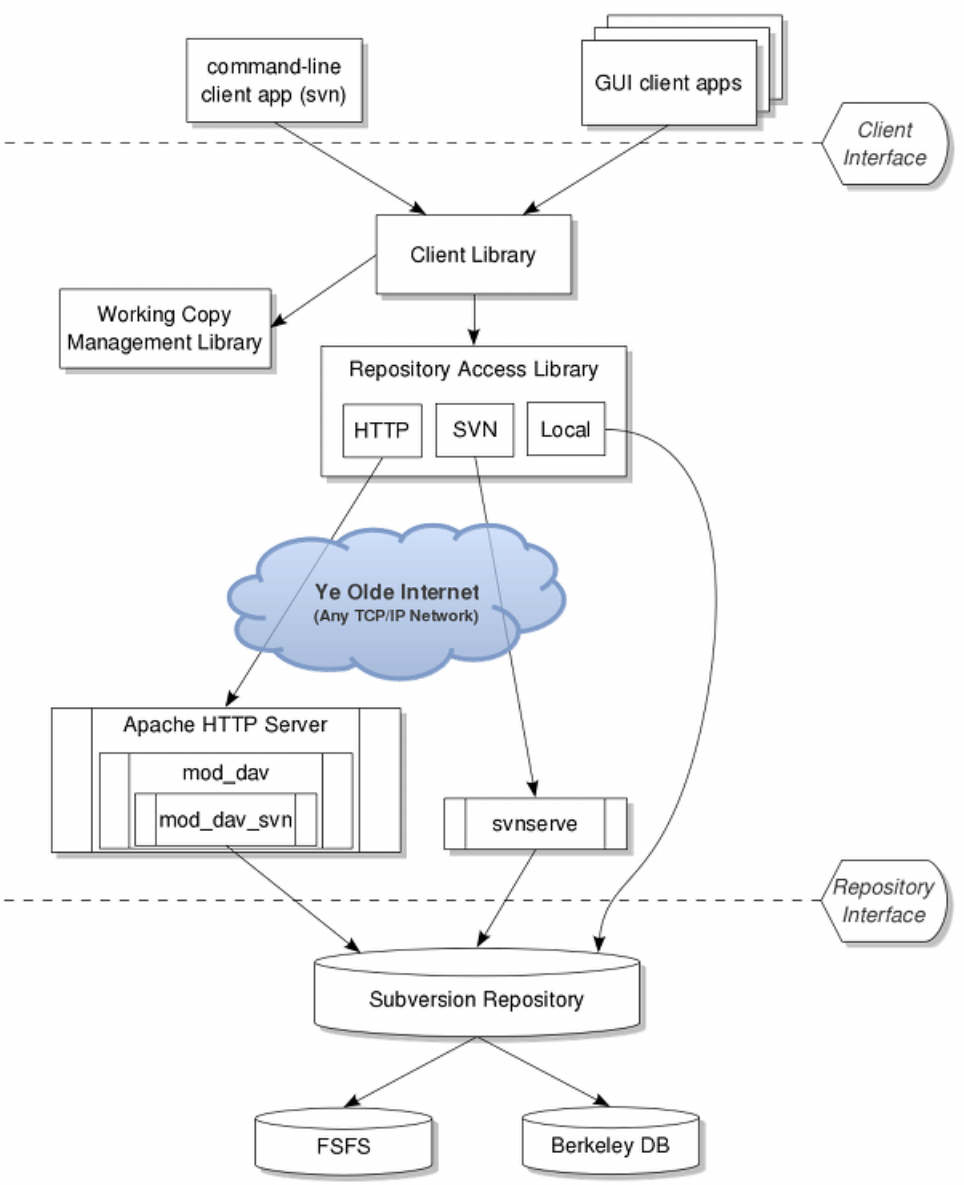
经过 14 个月的编码，在 2001 年 8 月 31 日，Subversion 实现了 “自我寄生”，也就是说开发人员不再使用 CVS，而是使用 Subversion 管理 Subversion 的源代码．

虽然 CollabNet 发起了这个项目，而且仍然提供资金支持（为一些全职的 Subversion 开发者提供薪水），但 Subversion 像其他许多开源项目一样，由一套松散透明，鼓励能者多劳的规则管理．2009 年，CollabNet 和 Subversion 开发人员合作，把 Subversion 项目纳入 Apache Software Foundation（简称 ASF)—世界著名的开源项目集合．Subversion 的技术根基，社区和开发工 作同 ASF 非常契合，ASF 的很多成员同时也是 Subversion 的活跃贡献者．2010 年，Subversion 成为了 ASF 的顶级项目之一，其项目主页迁移至 http://subversion.apache.org，Subversion 被重新命名为 “Apache Subversion”．

Subversion 的架构

图 1 “Subversion 的架构” 展视了 Subversion 设计结构的 “俯视” 图．

图 1. Subversion 的架构



图中的一端是存放所有版本化数据的 Subversion 仓库，另一端是 Subversion 客户端程序，客户端程序管理着部分版本化数据在本地的映射．两端之间是穿过仓库访问 (Repository Access) 层的多条访问路径，其中 一些路径跨越计算机网络，通过网络服务器对仓库进行访问，其他一些路径 则不经过网络，直接访问仓库.

Subversion 的组件

安装好的 Subversion 由好几个组件构成，下面将简单地介绍 一下这些组件，这些描述或许过于简略，不太容易理解，但是不要担心，本书后面的章节会用更多的内容详细介绍这些组件.

svn

客户端命令行工具

svnversion

用于报告工作副本状态（就项目的版本号而言）的工具

svnlook

可以直接检查仓库的工具

svnadmin

用于创建，调整或修复 Subversion 仓库的工具

mod_dav_svn

可插拔的 Apache HTTP 服务器模块，该模块允许用户通过网络访问 仓库

svnserve

一个定制的，可独立运行的服务器程序，可以以守护进程方式运行，也可以被 SSH 调用，这是另一种允许用户通过网络访问仓库的方法

svndumpfilter

过滤 Subversion 仓库转储数据流的程序

svnsync

可以跨越网络对仓库进行增量镜像备份的程序

svnrdump

可以跨越网络对仓库历史进行转储和加载的程序

svnmucc

该工具支持在没有工作副本的情况下，在一个单独的提交中对多个 仓库执行基于 URL 的操作

Subversion 的演化

本书的第一版由 O'Reilly Media 于 2004 年出版，也就是 Subversion 1.0 发布后不久，此后 Subversion 项目仍然在不断地发行新版本．下面是 Subversion 1.0 后发布的每个新版本的主要变化，注意下面的修改日志 并不完整，详细信息可以访问 Subversion 项目主页 http://subversion.apache.org.

Subversion 1.1 (2004 年 9 月)

1.1 版引入了 FSFS, 一种扁平文件的仓库存放选项，虽然 Berkeley DB 后端仍然被广泛使用，但是 FSFS 的门槛较低，管理 需求也更容易满足，FSFS 成为了新建仓库的默认选项．1.1 版还支持 对符号链接进行版本控制，URL 自动封装，以及本地化的用户接口.

Subversion 1.2 (2005 年 5 月)

1.2 版支持在服务器端对文件加锁，实现对特定资源的串行化提交. Subversion 是一种并行的版本控制系统，但特定类型的二进制文件（例如音频文件）不支持合并，使用锁机制实现了对这种类型的文件进行 版本控制和保护. 1.2 版还实现了一个完整的 WebDAV 自动化版本控制，允许 Subversion 仓库被当成网络文件夹进行挂载. 最后，Subversion 1.2 开始使用一种新的，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 版为 *svnserve* 服务器引入了基于路径 的授权控制，该特性以前只有 Apache 服务器才支持. Apache 服务器 增加了新的日志特性，Subversion 为其他语言提供的 API 也取得了 巨大的进步.

Subversion 1.4 (2006 年 9 月)

1.4 版引入了一个新工具—*svnsync* —通过网络完成仓库的单向复制. 工作副本的一个重大修改是 其元数据不再使用 XML（客户端的运行速度变得更快）. 以 Berkeley DB 作为后端的仓库获得了在服务器崩溃后自动恢复的能力.

Subversion 1.5 (2008 年 6 月)

1.5 版的开发时间比之前版本长了很多，但取得的效果是巨大的：分支与合并的半自动跟踪. 这为用户带来了非常大的便利，也让 Subversion 远远超越 CVS，进入了商业竞争者的行列，同 Perforce 和 ClearCase 并肩. Subversion 1.5 还增加了很多用户比较关注的 特性，例如交互式的冲突解决，稀疏检出，变更列表的客户端管理，外部定义的新语法，为 *svnserve* 新增 SASL 授权支持.

Subversion 1.6 (2009 年 3 月)

通过引入结构性冲突，1.6 版的分支与合并变得更加健壮，已有的几个 特性也得到了增强：更多的交互式冲突解决选项，稀疏检出支持不可伸缩 和完全的排他性，基于文件的外部定义，*svnserve* 支持与 *mod_dav_svn* 类似的操作日志. 另外，客户端命令行工具支持新的，用于引用仓库 URL 的缩写语法.

Subversion 1.7 (2011 年 10 月)

1.7 版主要是对 Subversion 的已有组件进行了两次大检修， 其中最大的变化是重新实现了工作副本的管理函数库 *libsvn_wc*，也就是所谓的 “WC-NG”. 第二个变化是为客户端与服务器间的交互引入了一种更光滑的 HTTP 协 议. Subversion 1.7 还增加了很多新特性，修复了很多问题，在性能 上也有明显的提升.

Subversion 1.8 (2013 年 6 月)

Subversion 1.8 的客户端在跟踪文件和目录的重命名上更加详尽，它的 *svn merge* 更加智能，选项 `--reintegrate` 对它来说根本就是多余的了. 增加 了新的版本控制属性，其中一些属性可以从父目录继承，这就允许仓库 为属性的自动设置和忽略文件模式传递默认值，用户间的一致性得到增 强，而这种一致性以前需要通过用户间的交流才能完成. 交互式的冲突 解决引入了一种新的命令行文件合并工具. 同样，Subversion 1.8 还 增加了许多新特性，修复了很多问题，在性能上也有提升.

目标听众

本书是为了那些在计算机领域有着丰富知识，并且希望使用 Subversion 管理数据的读者准备的. 尽管 Subversion 可以在多种不同的操作系统上运行，但是它的主要用户界面是基于命令行的，所以本书主要讨论 Subversion 的 命令行工具 (*svn*) 和其他一些辅助工具.

为方便讨论，本书的例子假设读者使用的是类 Unix 操作系统，并且熟悉 Unix 和命令行界面，当然，*svn* 也可以在非 Unix 平台上 运行，例如 Microsoft Windows. 除了一些微小的差别（例如在路径中使用 反斜杠 (\)—而不是斜杠 (/)—作为分隔符），在 Windows 上运行 *svn* 的输入 和输出与在 Unix 平台上运行完全一致.

本书的大多数读者应该是那些需要管理代码变化的程序员或系统管理员，这也是 Subversion 最普遍的用途，因此本书的例子主要关注源代码文件，但是 Subversion 可以对任意类型的文件进行版本控制—图片，音频，数据库，文档等，对 Subversion 而言，任意类型的数据也只是数据而已.

本书假定读者没有使用过版本控制工具，我们同时也尽了最大的努力，让 CVS（或其他版本控制系统）用户可以轻松地过渡到 Subversion. 边栏可能会时不时 地介绍一些和 CVS 相关的内容，附录 B, *针对 CVS 用户的 Subversion 介绍* 总结了 Subversion 和 CVS 的区别.

另外需要注意的是书中的源代码示例仅仅是例子而已，虽然它们可以被 编译器编译通过，但列举它们只是为了说明问题，并非为了展示优秀的编程 风格.

如何阅读本书

技术书籍总是面临这样一种两难境地：是迎合 “自上而下” 的读者，还是 “自下而上” 的读者. 自上而下的读者喜欢快速 浏览文档，先对系统的工作方式有一个整体上的了解，然后才开始使用软件；自下而上的读者是一个 “在实践中学习” 的人—他喜欢 先开始使用软件，通过实际操作来体会软件的运行方式，只在必要时才会参考 文档. 大多数图书会针对某一类读者，本书更倾向于自上而下的读者（如果你 阅读了本节，说明你属于自上而下的类型）. 但是自下而上的读者也不要感到失望，本书布局是为了对 Subversion 的各个主题进行广泛的探讨，但是每一章节都包含 了大量可供实践的例子. 想要马上动手试一下的读者可以跳到 附录 A, *Subversion 快速入门*.

无论读者属于哪一种风格，本书的目标是对不同背景的人都能有所帮助 —无论是没有接触过版本控制的新手，还是经验丰富的系统管理员. 根据 背景的不同，某些章节可能对你更有价值，而某些章节可能只需要浏览一下即可. 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”：

有经验的系统管理员

> 假设读者已经使用过版本控制系统，需要快速搭建 Subversion 服务器并运行起来，第 5 章 *仓库管理* 和 第 6 章 *服务器配置* 介绍了如何创建你的第一个仓库，并将其发布到网络上. 第 2 章 *基本用法* 和 附录 B, *针对 CVS 用户的 Subversion 介绍* 是 Subversion 客户端工具的快速学习 指南.

新用户

> 如果系统管理员已经搭建好了 Subversion 服务器，你只需要学习如何 使用客户端工具. 如果你从未使用过版本控制系统，那么 第 1 章 *基本概念* 介绍了版本控制系统的重要思想. 第 2 章 *基本用法* 介绍了 Subversion 客户端工具的基本用法.

高级用户

> 无论是普通用户还是系统管理员，你的项目最终总会越来越大，这时 你就要学习更加高级的 Subversion 功能，例如 Subversion 的属性支持（第 3 章 *高级主题*），如何使用分支与合并（第 4 章 *分支与合并*），如何配置运行时参数（第 7 章 *定制自己的 Subversion 体验*）等. 这些章节一开始不会显得 特别重要，但是熟悉了基本操作之后还是非常有必要了解一下.

开发人员

假设你已经对 Subversion 非常熟练了，现在想要扩展 Subversion，或以它为基础，开发新的软件，第 8 章 嵌入 *Subversion* 介绍 了相关的内容.

本书的结尾部分是 Subversion 的参考手册— 第 II 部分 “Subversion 命令行参考手册”，其中介绍了 Subversion 的所有命令，附录中还 包含了很多很有用的主题，阅读完本书后，这些章节最有可能是需要你回顾的内容.

本书组织

每一章要介绍的内容如下：

第 1 章 基本概念

介绍版本控制基础和不同的版本控制模型，以及 Subversion 的仓库，工作副本和版本号.

第 2 章 基本用法

介绍 Subversion 用户在一天的工作中将会如何使用 Subversion 客户端工具去获取，修改和提交数据.

第 3 章 高级主题

介绍普通用户最终将会用到的更复杂的功能，例如版本控制的元数据，文件锁和限定版本号.

第 4 章 分支与合并

介绍分支，合并和标签，包括分支与合并的最佳使用方法，常见用法，如何撤消修改，如何方便地从一个分支切换到另一个分支.

第 5 章 仓库管理

介绍 Subversion 仓库基础，包括如何创建，配置和维护一个仓库，以及其中要用到的工具.

第 6 章 服务器配置

介绍如何配置 Subversion 服务器，以及访问仓库的不同的方法：`http`，`svn` 和本地磁盘访问. 还介绍了关于认证，授权和匿名访问的细节.

第 7 章 定制自己的 *Subversion* 体验

介绍 Subversion 客户端配置文件，如何处理国际化文本，以及如何 在 Subversion 中使用外部工具.

第 8 章 嵌入 *Subversion*

从程序员的视角介绍 Subversion 的内部构造，Subversion 文件系统，工作副本的管理区. 还介绍了如何使用公共 API 对 Subversion 进行二次 开发.

第 II 部分 “Subversion 命令行参考手册”

详细地介绍 *svn*, *svnadmin*, *svnlook* 的每个子命令，并带有丰富的使用示例.

附录 A, *Subversion 快速入门*

专门为没有耐心的读者而写，快速地介绍了如何安装 Subversion 并开始使用它.

附录 B, *针对 CVS 用户的 Subversion 介绍*

介绍了 Subversion 与 CVS 的异同点，对于如何纠正长期使用 CVS 所养成的坏习惯也提出了大量的建议. 包含的内容有 Subversion 的版本号，对目录进行版本控制，离线操作，命令 *update* 与 *status*, 分支，标签，元数据，冲突解决和授权.

附录 C, *WebDAV 与自动版本控制*

介绍了 WebDAV 和 DeltaV 的细节，以及如何通过配置 Subversion, 使得仓库可以被挂载成可读写的 DAV 共享目录.

附录 E, *版权*

本书所使用的创作共享署名授权协议的一份副本.

本书是免费的

本书起源于 Subversion 开发人员编写的文档，这些文档后来被合并到一个 单独的项目中，并加以重写，也正因为如此，本书使用的授权是免费的（见 附录 E, *版权*). 实际上，本书的编写过程一直是公开 进行的，最初是 Subversion 项目的一部分，这就意味着以下两点：

- 你总能在本书的 Subversion 仓库中找到本书的最新版本.
- 你可以按照自己的要求修改本书并重新发布—它使用的授权是免费 的，唯一的约束是你必须保留原始作者的贡献. 当然，我们更希望你把反馈 和补丁发送给 Subversion 开发社区，而不是发布自己的私有版本.

本书主页和不同语言的翻译版存放在 http://svnbook.red-bean.com, 在那里你可以找到本书不同 格式的最新版和打过标签的版本，以及访问本书 Subversion 仓库的方法 (Subversion 仓库存放了本书的 DocBook XML 源代码). 非常欢迎（甚至鼓励）读者向我们反馈. 针对本书源代码的注释，建议和补丁请发送至 `<svnbook-dev@red-bean.com>`.

致谢

如果没有 Subversion, 本书也就失去了存在的意义，因此笔者感谢 Brian Behlendorf 和 CollabNet 资助了如此优秀和雄心勃勃的开源项目；感谢 Jim Blandy 给出了 Subversion 的名字和原始设计—我们爱你，Jim; 感谢 Karl Fogel, 他既是一个好友，也是一个优秀的社区领导者[4]

感谢 O'Reilly 及其专业的编辑团队，为本书的润色工作做出的巨大贡献，他们是 Chuck Toporek, Linda Mui, Tatiana Apandi, Mary Brady 和 Mary Treseler.

最后，我们还要感谢评审过本书，向本书提过建议和补丁的无数志愿者，完整地列出这些人的名单有点不太实际，但是他们的名字将永远地留在本书的 提交历史里!

[4] 噢，还要感谢 Karl 工作过于繁忙，我们才能有撰写本书的机会.

部分 I. 初识 Subversion

目录

第 1 章 基本概念

本章将对 Subversion 及其版本控制方法进行简短的介绍，我们首先介绍 一些通用的版本控制概念，然后再介绍 Subversion 特有的概念，最后再展示 一些使用 Subversion 的简单例子.

虽然本章都是以程序的源代码作为版本控制的对象，但是 Subversion 可以 管理任意类型的文件—它并非是程序员的专用工具.

版本控制基础

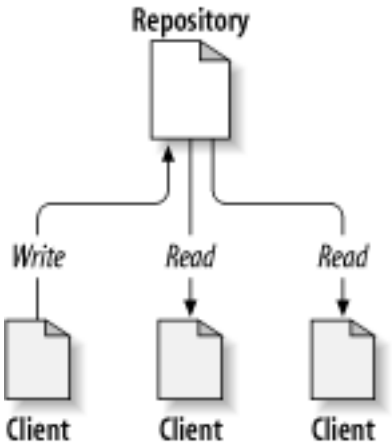
版本控制系统用于跟踪文件和目录在时间上的增量版本. 当然，仅仅跟踪一 个用户或用户组不同版本的文件和目录并不会让版本控制系统显得多么高级，它的真正用处是允许用户查看每个版本所发生的变化，对任意一个修改进行 撤消.

本节将介绍一些版本控制系统比较高层的组成部分和概念，内容只限于 现代的版本控制系统—在如今这个时代，如果一个版本控制系统无法在 广域网下工作，估计没多少人愿意使用它.

仓库

版本控制系统的核心是仓库，它是存放系统数据的中央位置. 仓库通常以 文件系统树 (*filesystem tree*) 的形式存放信息，文件系统树是文件和目录的分层结构. 有任意数量的 客户端 (*client*) 连接到仓库，对其中的文件进行读写 访问. 通过向仓库写数据，客户端将信息暴露给其他人；通过读取数据，客户端获得了其他人的信息，如 图 1.1 “典型的客户端/服务器系统” 所示：

图 1.1. 典型的客户端/服务器系统



为什么这很重要? 目前来看，仓库像是一个典型的文件服务器. 确实是，但是它和你平时用的文件服务器并不完全相同，仓库的独特之处是随着文件 不断地发生变化，它会记住文件的每一个版本.

当客户端从仓库读取数据时，它通常只会读取到文件系统树的最新版本，但是版本控制系统客户端的一个重要功能是它也可以读到早先版本的文件 系统树. 版本控制系统客户端可以询问历史问题，例如 “在上周三时，这个目录中存放的是什么内容?” 和 “修改这个文件的最后 一个人是谁? 他改了什么?” 这些是任意一个版本控制系统都 要有能力解决的问题.

工作副本

版本控制系统的核心价值在于它可以跟踪文件和目录的版本，但是其他 软件不会操作 “文件和目录的版本”，大多数软件只能理解 如何操作某种特定类型文件的 单一 版本，那么用户 如何才能以一种实实在在的方式，和一个包含了不同文件多个版本的，抽象的，远程的仓库进行交互？用户的字处理软件，演示软件，源代码编辑器，网页设计软件—以及其他一些只能处理单一版本文件的程序— 如何才能访问仓库中的文件？答案是 工作副本 (*working copy*).

顾名思义，工作副本是仓库的特定版本数据在本地的副本，用户可以 自由地对它进行操作. 对其他软件来说，工作副本[1] 只是一个普通的本地目录，所以即使它们不具备版本控制功能也可以对 工作副本进行读写. 版本控制系统的客户端工具负责管理工作副本，以及 与仓库通信.

版本控制模型

如果说版本控制系统的首要工作是跟踪文件和目录在时间上的不同版本， 那么它的次要工作就是支持协作编辑和数据共享. 不同的版本控制系统在实现 后者时可能会采用不同的策略，理解这些策略的差别非常重要，这主要基于以 下两点考虑：首先，这可以帮助你比较不同的版本控制系统，特别是在你遇到 了一个和 Subversion 相似的系统；然后，这可以帮助你更高效地使用 Subversion，因为 Subversion 支持多种不同的工作模式.

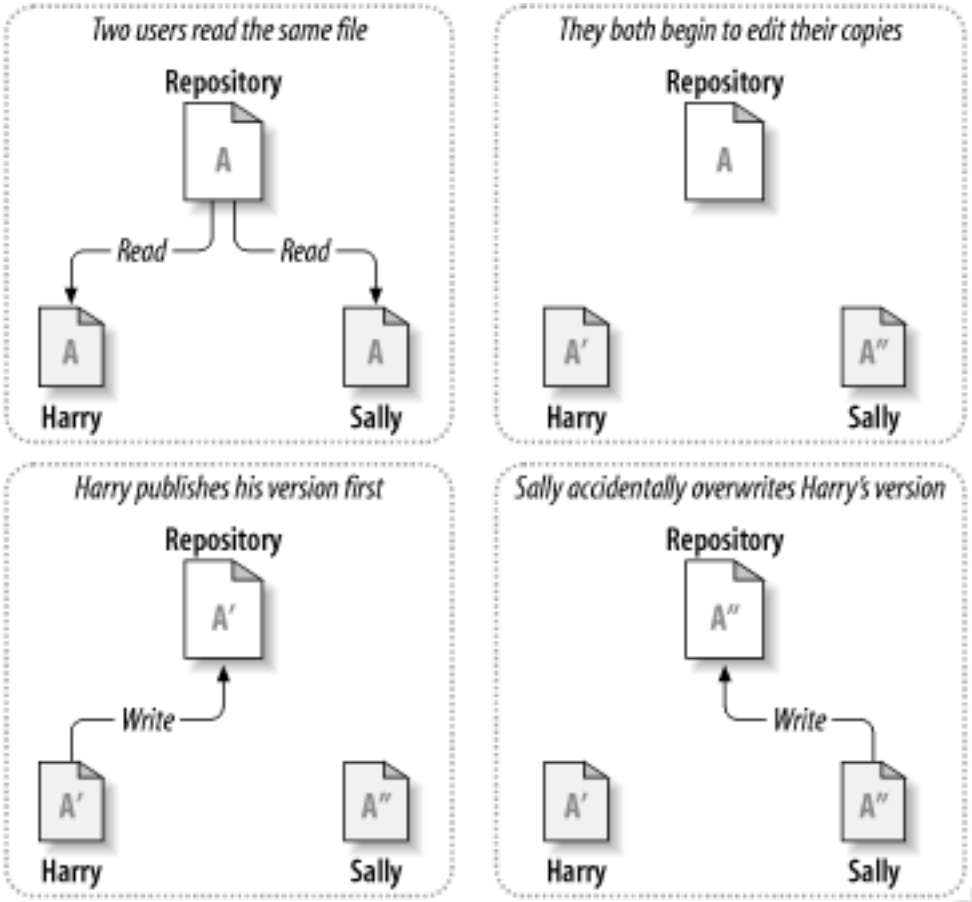
文件共享的问题

所有的版本控制系统都要解决一个基本问题：如何允许用户共享信息，同时避免他们无意之间互相干扰？一个用户无意间覆盖了其他用户的修改 —这种情况经常发生.

考虑 图 1.2 “需要避免的情况” 所示的情境. 假设现在有两个同事，Harry 和 Sally，他们在同一时间修改 了仓库中的同一文件. 如果 Harry 先把修改保存到仓库中，后面 Sally 就有可能用他的新版本文件覆盖掉 Harry 的版本. 虽然 Harry 的修改不会就此丢失 (因为版本控制系统会记住每一次修改)，但是 Harry 的修改 不会出现在 Sally 的新版本文件中，因为他从未看到过 Harry 的修改. 从效果上来看，Harry 的修改丢失了—至少是对文件的最新版本来说 —而且很有可能是无意间导致的，我们决不能让这种情况发生.

[1] 术语 “工作副本” 可以应用到任意一个文件版本的 本地实例，但是大多数人谈到工作副本时，都指的是一整个目录，其中 包含了被版本控制系统管理的文件与子目录

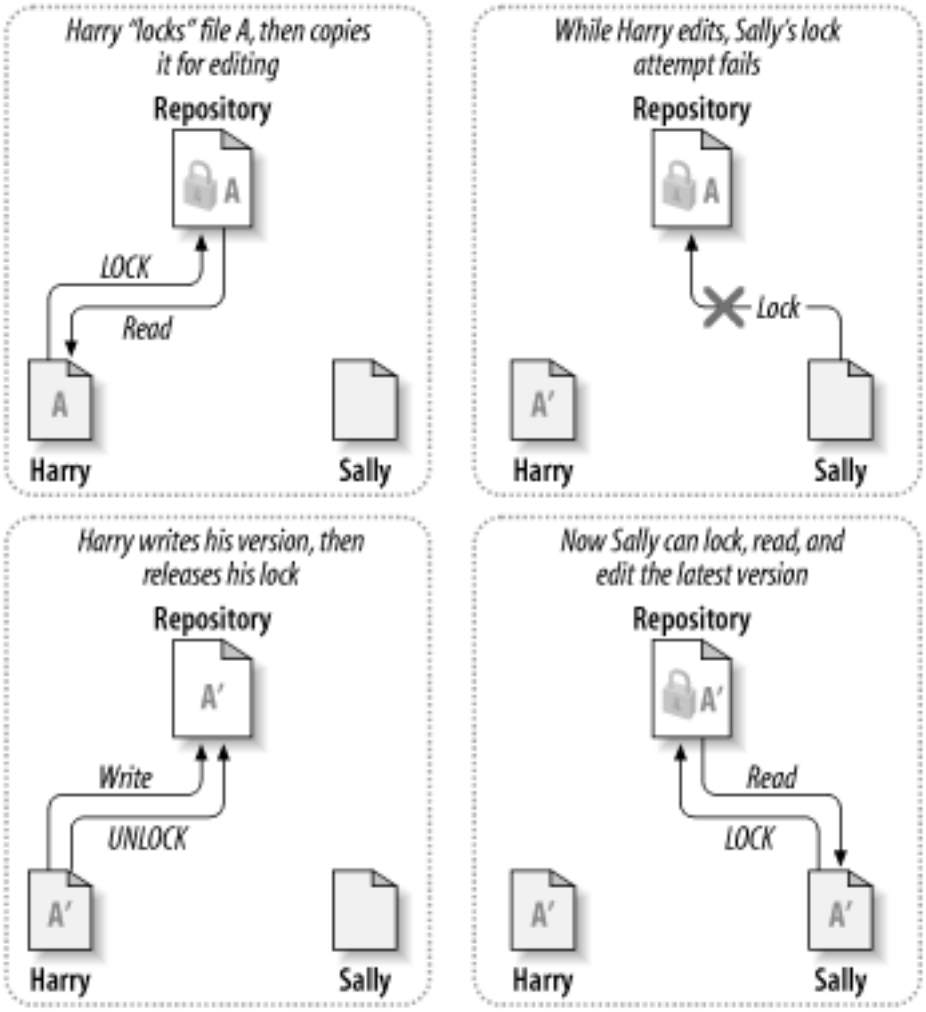
图 1.2. 需要避免的情况



加锁-修改-解锁 解决方案

为了解决用户间互相干扰工作的问题，许多版本控制系统都采用了 *加锁–修改–解锁* (*lock-modify-unlock*) 模型. 在这个模型中，仓库一次只允许一个用户修改同一文件，这种独占策略通过锁进行管理. Harry 在修改之前要先对文件进行 “加锁” (lock), 如果 Harry 已经锁住了文件，Sally 就不能再对同一文件进行加锁，也就不能 修改文件. 他所能做的就是等待 Harry 完成修改，保存文件，然后释放 锁. Harry 解锁后才能轮到 Sally 加锁，然后他可能会得到一个新版本的 文件并开始编辑. 图 1.3 “加锁–修改–解锁 解决方案” 展示了 工作流程.

图 1.3. 加锁-修改-解锁 解决方案



加锁-修改-解锁 模型的问题是限制比较多，经常会成为用户的麻烦：

- 加锁可能会导致管理上的问题. 有时候 Harry 在锁住一个文件后可能会忘了给它解锁，同时 Sally 还在焦急地等着. 如果 Harry 去度假了，那么 Sally 必须找到 仓库管理员，让他释放 Harry 的锁. 这种情况会浪费大量的时间.

- 加锁可能会导致不必要的串行化. 如果 Harry 想要修改文件的开头部分，而 Sally 只想修改同一文件 的结尾部分? 此时他们的修改就不会重叠. 如果他们的修改可以恰当 地合并在一起，那他们就可以同时编辑文件，完全不会产生任何问题. 此时对文件进行加锁就完全没有必要.

- 加锁可能会造成安全上的错觉. 假设 Harry 加锁并修改了文件 A，同时 Sally 加锁并修改了文件 B，文件 A 和文件 B 在内容是互相依赖的，如果 Harry 和 Sally 的 修改在语义上是不兼容的，那将会如何? 文件 A 和文件 B 可能无法 再正常工作. 加锁-修改-解锁 模型对这种情况无能为力—但是用 户会错误地认为只要在加锁后修改就是安全的. 通过加锁，Harry 和 Sally 都错误地认为自己的修改是安全的，也就不会事先和对方沟通. 锁机制常常会代替真正的交流.

复制-修改-合并 解决方案

Subversion, CVS 和许多其他的版本控制系统使用 复制–修改–合并 (*copy-modify-merge*) 模型作为锁机制的替代品. 在这个模型中, 每一个用户的 客户端都与仓库通信, 在本地创建一份私有的工作副本, 然后用户可以同时 地, 互不干扰地修改自己的私有副本, 最后, 私有副本被合并到一个新的 最终版本. 为了支持 复制–修改–合并 模型, 版本控制系统通常会提供合并 操作, 但是归根到底, 必须由用户自己来确保合并的结果是正确的.

我们通过例子来说明. 假设 Harry 和 Sally 各自创建了同一项目 的工作副本, 并在各自的工作副本中修改了同一文件 A. Sally 先把 修改保存到仓库中, 后面 Harry 试图保存自己的修改时, 仓库告诉他 文件 A 已经 过时 (*out of date*) 了, 换句话说, 自从他上一次复制了文件 A 之后, 仓库中 的文件 A 被更新了. 于是, Harry 告诉客户端把仓库中文件 A 的更新合 并到他的工作副本中 (这里不妨假设 Sally 的修改没有和他的修改重叠), 修改合并后, Harry 再一次向仓库保存了他自己的修改. 图 1.4 “复制–修改–合并 解决方案” 和 图 1.5 “复制–修改–合并 解决方案 (续)” 展示了 工作流程.

图 1.4. 复制–修改–合并 解决方案

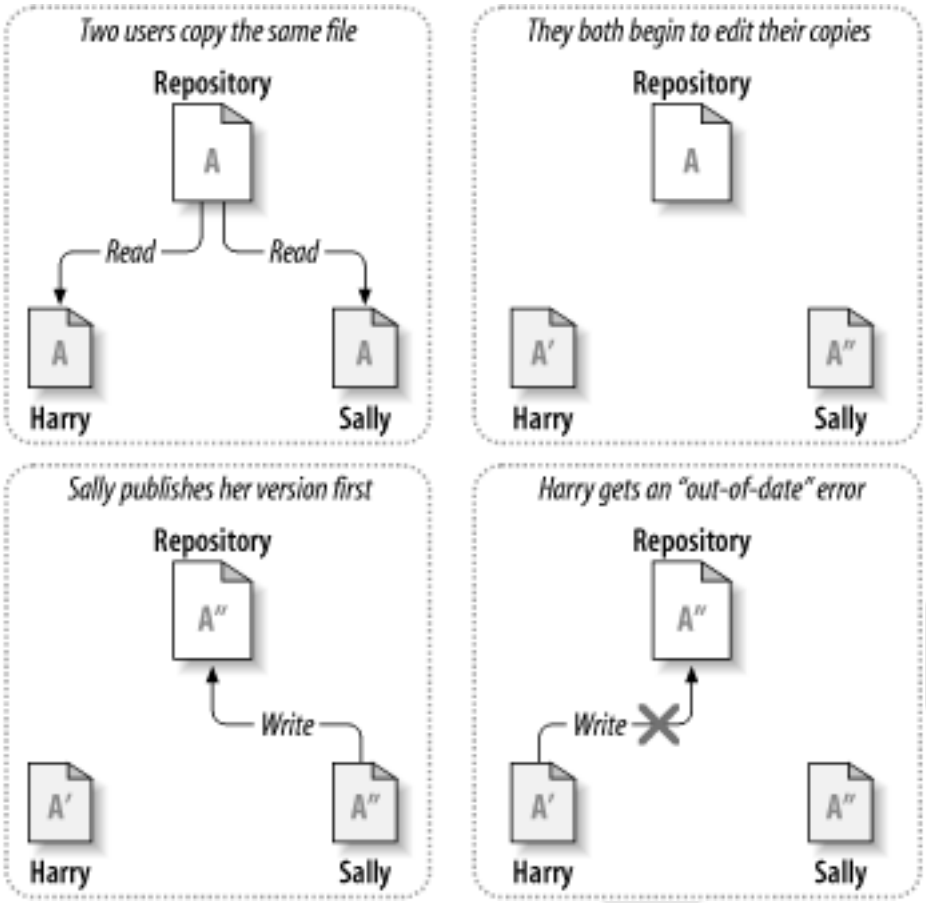
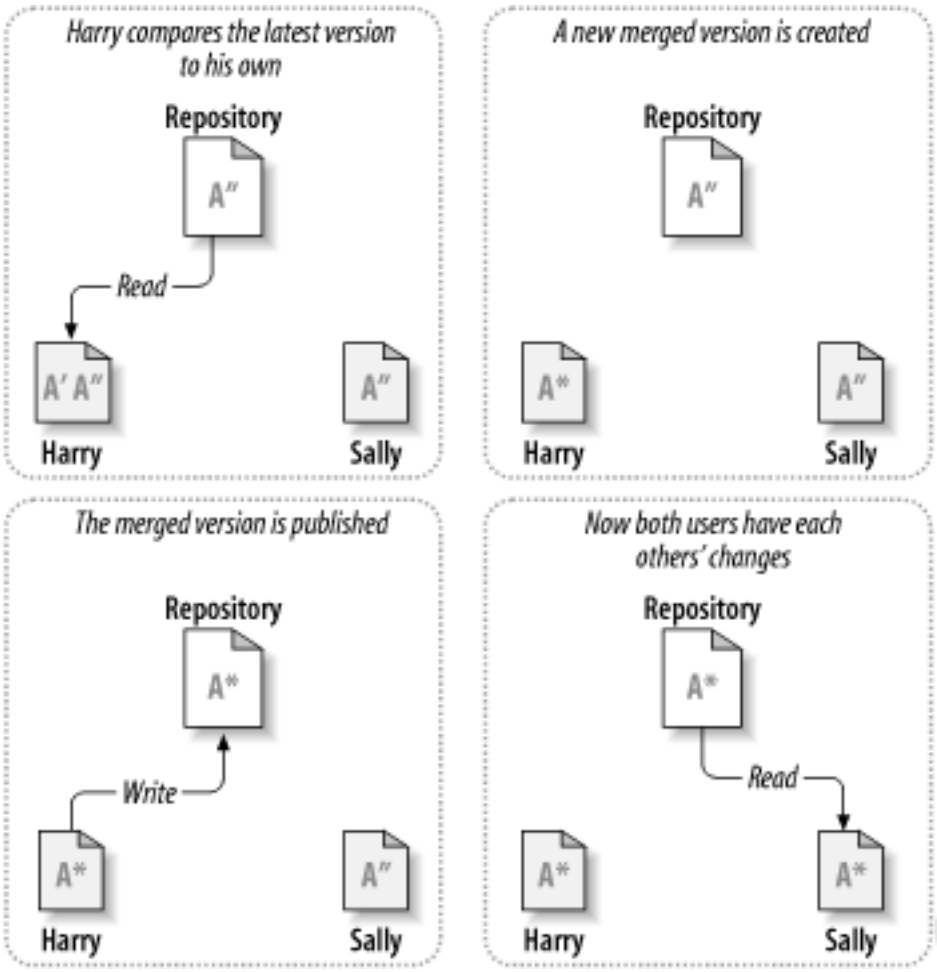


图 1.5. 复制–修改–合并 解决方案 (续)



但是如果 Sally 的修改和 Harry 重叠了，那又会产生什么结果? 这种情况称为 冲突 (*conflict*), 通常不是什么大问题. 当 Harry 告诉客户端把仓库的最新修改合并到他的工作副本时，副本中的文件 A 被标记为冲突状态: Harry 可以同时看到互相冲突的两套修改，并对它们 进行手工选择. 软件不会自动地解决冲突，只有人类才能理解冲突并做出 正确地选择. Harry 把重叠的修改解决后—可能是在和 Sally 沟通 之后—就可以把合并后的文件安全地保存到仓库中.

复制–修改–合并 模型看起来好像有点混乱，但是在实际使用中它 运行地很流畅. 用户可以并发地工作，不用等待其他人，当用户操作 同一文件时，经验表明他们的大多数修改不会重叠，冲突情况其实很少发 生. 解决冲突花费的时间通常要比使用锁机制浪费的时间要少得多.

上面所说的问题都涉及到一个关键因素: 用户间的沟通. 如果用户间 缺乏沟通，发生语法冲突和语义冲突的概率都会增加. 没有一个版本控制 系统可以强制用户沟通或检测语义冲突，所以不要认为使用锁机制可以完 全避免产生冲突，在实际使用中，锁机制似乎会影响工作效率.

什么时候使用锁是必须的

虽然人们通常认为 加锁-修改-解锁 模型对协作开发是有害的，但某些情况下却是最合适的.

复制-修改-合并 模型要求文件是支持合并的—也就是说文件 是基于行的文本文件（例如程序源代码文件），但是对二进制文件（例如 图片和音频文件）来说，合并有冲突的修改几乎是不可能完成的. 在这 种情况下，串行地修改文件就显得非常有必要. 如果没有串行访问，用户花费大量时间作出的修改很可能会被丢弃.

Subversion 以 复制-修改-合并 模型为主， 但是对某些类型的文件 仍然需要使用 加锁-修改-解锁 模型， 我们将在 “锁”一节 介绍如何在 Subversion 中使用 加锁-修改-解锁 模型.

Subversion 的版本控制方法

我们已经提到 Subversion 是一个现代的，支持网络的版本控制系统. 在 “版本控制基础”一节 说过（从较高的层次 看待版本控制）， 仓库是存放 Subversion 的版本控制数据的中央位置， 用户及 其软件通过工作副本与仓库交互. 本节介绍 Subversion 的版本控制实现方法.

Subversion 的仓库

Subversion 实现仓库的方式与其他版本控制系统非常类似. 与工作副本 不同，一个 Subversion 仓库是一个抽象的实体，可以被 Subversion 的库函数 和工具进行独占性地操作. 因为大多数用户是在工作副本中通过客户端工具 与 Subversion 交互，所以本书主要讨论工作副本以及如何操作它，关于 仓库的细节请参考 第 5 章 仓库管理.

在 Subversion 中，每一个版本控制系统用户都有的客户端对象 —一个目录，目录中除了存放被版本控制的文件外，还有用于跟踪 文件和与服务器通信的元数据—叫做 工作副本 (working copy). 虽然有些版本控制系统使用 “仓库” 表示 存放在客户端的对象，但这种说法并不正确，而且在 Subversion 中会让 用户感到困惑.

“Subversion 的工作副本”一节 介绍工作副本.

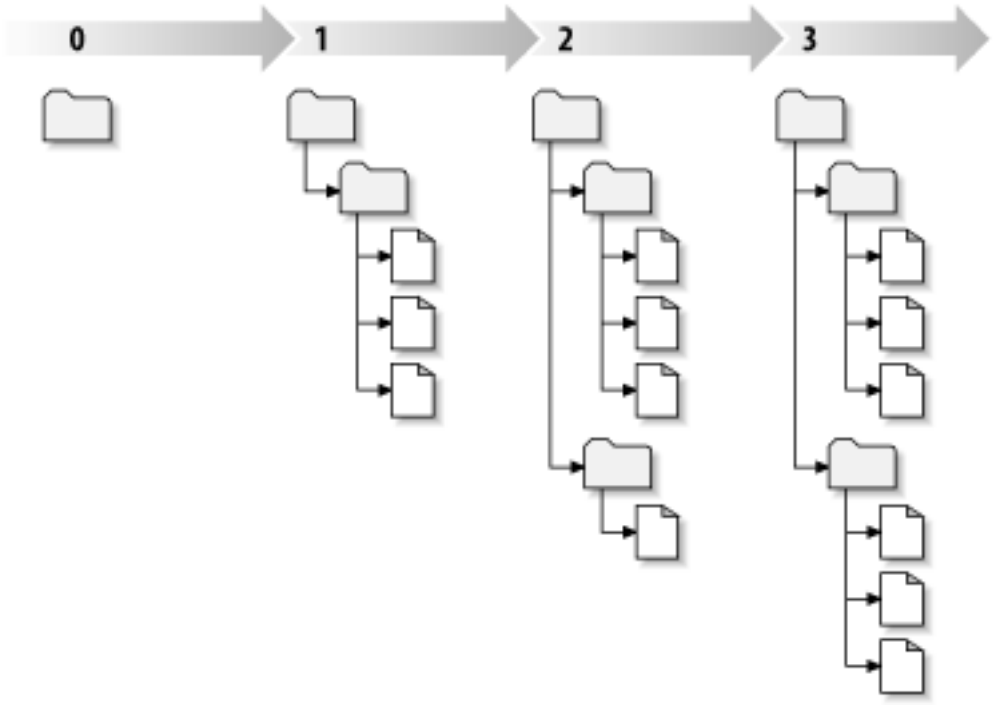
版本号

Subversion 客户端将任意多的文件和目录的修改作为一个原子事务提交 给仓库. 原子事务的意思是要么所有的修改都被仓库接受，要么一个也没有. Subversion 尽量保证即使是在程序崩溃，操作系统崩溃，网络断开和有其他 用户干扰的情况下，也能维持住原子性.

仓库每接受一次提交都会为文件系统树创建一个新状态，叫作一个 版本号 (*revision*). 每一个版本号都与一个独一无二的自然数相关联，后一个版本号都比前一个 大一. 新创建的仓库的初始版本号是 0，除了一个空的根目录外，什么也没 有.

图 1.6 “文件系统树在时间上的变化” 以可视化的方式 展示了仓库的版本号在时间上的变化. 想像有一个由版本号号码组成的队列，从 0 开始，从左向右伸展. 每一个版本号下面都挂着一个文件系统树，每一 个文件系统树都是本次提交后仓库的 “快照” (snapshot).

图 1.6. 文件系统树在时间上的变化



全局的版本号号码

不像其他大多数的版本控制系统，Subversion 的版本号被应用到 整个仓库，而不是单个文件. 每一个版本号都对应 着仓库的一个提交后的状态，例如版本号 N 表示仓库在第 N 次提交后的状 态. 如果有一个 Subversion 用户在讨论 “*foo.c* 的版本号 5”，实际上他们说的是 “在版本号 5 中的 *foo.c*”. 需要注意的是一个文件 在版本号 N 和 M 中可能是一样的！很多版本控制系统使用的版本号是针 对每个文件的，所以全局的版本号号码一开始可能会让用户觉得不太寻常（以前的 CVS 用户可以从 附录 B, *针对 CVS 用户的 Subversion 介绍* 得到更多的 细节）.

仓库寻址

Subversion 客户端工具使用 URL 识别仓库中的文件与目录. 在大部分情况下，这些 URL 使用标准的语法，允许在 URL 中包含服务器的域名和端口号.

- http://svn.example.com/svn/project
- http://svn.example.com:9834/repos

Subversion 仓库的 URL 不仅限于 `http://`，因为 Subversion 向客户端提供了几种不同的通信方式，所以根据具体的仓库 访问方式，用于寻址仓库的 URL 参数也会有微妙的差别. 表 1.1 “访问仓库的 URL 参数” 展示了不同的 URL 模式 如何映射到仓库的访问方式. 关于 Subversion 服务器选项的更多内容，见 第 6 章 服务器配置.

表 1.1. 访问仓库的 URL 参数

模式	访问方式
`file:///`	直接仓库访问（仓库在本地磁盘上）

模式	访问方式
`http://`	通过 WebDAV 协议访问可识别 Subversion 的 Apache 服务器
`https://`	和 `http://` 相同，但是增加了 SSL 封装（加密和授权）
`svn://`	通过传统的协议访问 `svnserve` 服务器
`svn+ssh://`	和 `svn://` 相同，但是增加了 SSH 隧道

Subversion 处理 URL 的方式有一些细微的差别，例如包含 `file://` 的 URL 要么以 `localhost` 作为服务器名，要么不含有服务器名：

- file:///var/svn/repos
- file://localhost/var/svn/repos

如果工作副本和仓库不在同一个驱动器上，那么 Windows 用户在使用 `file://` 模式时需要用到一种非官方的 “标准” 语法．下面是两个例子，其中 `X` 表示仓库所在的驱动器盘符：

- file:///X:/var/svn/repos
- file:///X|/var/svn/repos

注意，虽然 Windows 的路径使用反斜杠，但是 URL 仍然需要使用 正斜杠．另外还要注意的是在命令行上输入 `file:///X|/` 形式的字符串时，你需要用双引号把它包裹起来，这样的话竖线符就不会 被翻译成管道．

你不能在网页浏览器上输入 Subversion 的 `file://` 形式的 URL 来访问仓库，如果真这样做了，网页浏览器会以访问普通文件系 统的方式显示目录中文件的内容．因为，Subversion 的资源存放在一个虚拟 的文件系统中（见 “仓库层” 一节），而网页浏览器不知道如何与这种文件系统进行交互．

Subversion 客户端会自动对 URL 进行编码，就像网页浏览器那样．例如，URL `http://host/path with space/project/españa` —其中包含了空格和非 ASCII 字符—被 Subversion 自动解释成 `http://host/path%20with%20space/project/espa%C3%B1a`．如果 URL 包含空格，就要用双引号把它包裹起来，这样的话 Shell 就不会 错误地把它切分成多个参数．

Subversion 在处理 URL 参数（包括本地路径）时，有一个例外情况需 要特别注意．如果 URL 或本地路径的最后一个分量含有符号 `@`，为了让 Subversion 能够正确地对资源进行寻址，你需要使用一种特殊的语法—具体内容将在 “限定版本号与实施版本号” 一节 介绍．

Subversion 1.6 引入了一个新记号—脱字符 (`^`) —用来表示仓库根目录的 URL．比如说用户可以用 `^/tags/bigsandwich/` 表示项目根目录中的 */tags/bigsandwich* 的 URL，这种 URL 称为 *仓库的相对 URL* (*repository-relative URL*)．这种语法只能在 工作副本中使用—客户端需要从工作副本的元数据中获取仓库根目录的 URL．另外，使用仓库的相对 URL 访问仓库的根目录时需要写成 `^/`（末尾要有一个斜杠），而不是 `^`．Windows 用户不要忘了在他们的操作系统中，脱字符是一个转义字符，因此为了表示一个脱字符，需要写成 `^^`．

Subversion 的工作副本

一个 Subversion 工作副本是用户本地系统中的一个普通目录，用户可以按照 自己的要求对存放在目录中的文件进行编辑，如果是源代码文件，用户也可以 按照通常的方式对它们进行编译．工作副本是用户的私有工作空间：除非用户 明确地要求

Subversion, 否则它不会让工作副本合并其他人的修改，也不会把 用户的修改暴露给其他人. 用户可以为同一个项目创建多个工作副本.

如果用户修改了工作副本中的文件，并且确认了修改是正确的，此时 可以使用 Subversion 提供的命令来“发布” 修改（通过 把修改保存到仓库中），于是项目中的其他人就可以看到你的修改. 如果其他人也发布了他们的修改，Subversion 也提供了命令把他们的修改 合并到你的工作副本中（通过读取仓库）. 可以看到，仓库是每个用户发布的 修改的中间人—修改并非从一个工作副本直接传递到另一个工作副本.

工作副本还会包含一些额外的文件，这些文件由 Subversion 创建并维护，用于命令的正常运行. 每一个工作副本中都有一个名为 *.svn* 的子目录，它是工作副本的 管理目录 (*administrative directory*). 管理目录中的文件可以帮助 Subversion 识别哪些文件含有 未发布的修改，哪些文件是过时的.

在 1.7 版以前，Subversion 在工作副本的每一个子目录内都维护了 一个 *.svn* 目录. Subversion 1.7 在存放和 维护工作副本元数据上提出了一种全新的方法，从外面看最显著的变化 是每个工作副本只创建了一个 *.svn* 目录，存 放在工作副本的根目录下.

工作副本的工作原理

Subversion 为工作副本中的每一个文件记录两项信息：

- 文件的版本号（这被称为文件的 工作版本号 (*working revision*))
- 一个时间戳，记录了本地文件最近一次被仓库更新是在什么时候

有了这些信息后，通过与仓库通信，Subversion 就可以判断出 工作副本中的每一个文件处于以下 4 种状态中的哪一种：

当前未修改的

文件在工作副本中未被修改，并且在工作版本号之后还没有 人提交过该文件的修改. 对文件执行 *svn commit* 和 *svn update* 都不会产生任何效果.

当前已修改的

文件在工作副本中已被修改，并且在一次更新以来还没有人 向仓库提交过该文件的修改. 本地有未提交的修改，于是执行 *svn commit* 将会成功地把修改提交到仓库中，而 *svn update* 不会产生任何效果.

过时未修改的

文件在工作副本中未被修改，但是在上一次更新之后有人往 仓库提交了该文件的修改. 为了让文件和最新版本保持同步，应 该执行更新操作. 对文件执行 *svn commit* 不会产生任何效果，执行 *svn update* 将 把仓库中的最新修改合并到文件中.

过时且已修改的

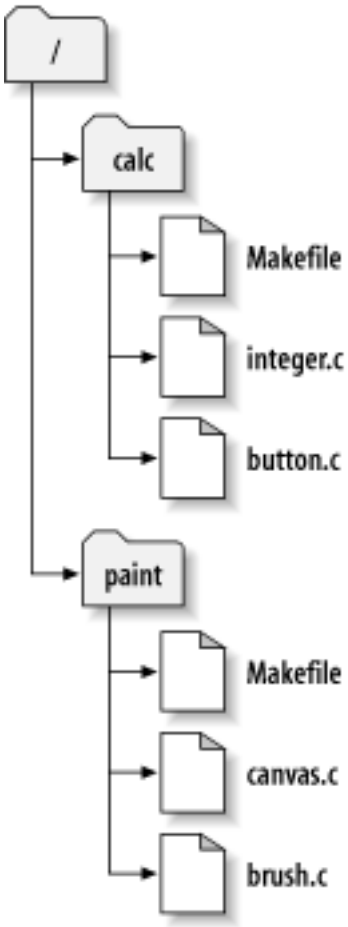
文件在本地工作副本和仓库都被修改了. 对文件执行 *svn commit* 会由于文件已过时而失败. 首先应该更新文件，命令 *svn update* 尝试 把仓库的修改合并到本地. 如果 Subversion 不能自动地以一种 合理的方式完成合并，就会把冲突交由用户来解决.

工作副本的基本操作

一个典型的 Subversion 仓库经常存放着若干个项目的文件，一般来说，每一个项目都是仓库文件系统树的一个子目录．在这种目录 布局下，用户的一个工作副本就对应着仓库中一个特定的子目录.

举例来说，假设你有一个包含了两个软件项目的仓库，这两个项目是 `paint` 和 `calc`，每一个项目 都有一个属于自己的目录，如 图 1.7 “仓库的文件系统” 所示.

图 1.7. 仓库的文件系统



为了得到一个工作副本，你必须 检出 (*checkout*) 仓库的某些子树（术语 检出 听起来好像会涉及到加锁和资源的预留，但实际 上并没有，它仅仅是为用户创建一份仓库的工作副本）．举例来说，如果检出 */calc*，用户将会得到这样一份工作副本：

```
$ 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/
$
```

靠近左边界的几个字母 A 指出 Subversion 正 在往工作副本中添加项目 (item). 现在你就有了仓库 `/calc` 的一份私有副本, 外加一项额外的目录— *.svn*—目录里存放了 Subversion 需要的 额外信息, 我们在前面已经介绍过了.

假设用户修改了文件 *button.c*, 因为 *.svn* 记录了文件原来的修改日期和内容, 所以 Subversion 可以检测到用户修改了文件, 但是 Subversion 不会自动地 发布修改, 除非用户显式地告诉它要这么做. “发布修改” 这个操作更常见的说法是向仓库 提交 (*committing*) 或 检入 (*checking in*) 修改.

用户为了发布修改, 需要使用 Subversion 的命令 *svn commit*:

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
$
```

用户对文件 *button.c* 的修改现在就已经正式 提交到了仓库中, 提交日志还附带了一条描述修改的注解 (在上面的例子中是修改了一个拼写错误). 如果有另一个用户检出了 */calc* 的工作副本, 他就会在最新版本 *button.c* 中看到用户新提交的修改.

假设你有一个同事 Sally, 在你修改 *button.c* 的同时, 他也检出了一个 */calc* 的工作副本. 当 你把 *button.c* 的修改提交到仓库后, Sally 的 工作副本并不会自动地把修改同步到本地—只有在用户的显式要求下, Subversion 才会更新工作副本.

为了把工作副本更新到最新的状态, Sally 可以要求 Subversion 更新 (*update*) 他的工作 副本, 用到的命令是 *svn update*. 如果在 Sally 检 出工作副本之后, 有人向仓库提交了修改, 命令就会把这些修改都合并到他 的工作副本中.

```
$ pwd
/home/sally/calc
$ ls -A
Makefile button.c integer.c .svn/
$ svn update
Updating '.':
U    button.c
Updated to revision 57.
$
```

上面 *svn update* 的输出指出了 Subversion 更新了 *button.c* 的内容. 注意, Sally 不需要 指定应该更新哪些文件, 根据 *.svn* 和仓库中的 信息, Subversion 可以自动判断出哪些文件需要更新.

版本号混合的工作副本

尽量保持灵活性是 Subversion 的总体原则, 其中一项灵活性是 Subversion 支持同一个工作副本中的文件和目录可以拥有不同的工作版本号. Subversion 的工作副本不必总是对应仓库中的一个单一的版本号, 其中的文件可以来自 不同的版本号. 例如, 假设用户从仓库检出了一个工作副本, 而该仓库最新 的版本号是 4:

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

此时的工作副本对应仓库的版本号 4. 如果用户修改了文件 *button.c*, 并提交了修改, 假设在此之前没有其他 用户向提交提交过修改, 那么刚才的提交将会为仓库创建版本号 5, 工作副本变成了:

calc/
  Makefile:4
  integer.c:4
  button.c:5

假设这时候 Sally 提交了 *integer.c* 的 修改，创建了版本号 6. 如果你执行了命令 *svn update*, Subversion 就会把工作副本更新到最新版，变成：

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

Sally 对 *integer.c* 的修改出现在了你的工作 副本中，你的修改依然保留在 *button.c* 中. 在这 个例子里，*Makefile* 在版本号 4, 5 和 6 中都 保持不变. 于是，在工作副本的根目录执行了 *svn update* 后，工作副本才精确地对应到了仓库的 同一个版本号.

更新和提交是分开的

Subversion 的一条基本规则是一次 “推送” (push) 操作不会产生一次 “抓取” (pull) 操作，反之依然成立. 理由是用户准备好向仓库提交修改并不表示他已经准备好接收其他人提交 的修改，另外，如果用户的修改还未完全完成，命令 *svn update* 应该把仓库的修改合并到本地，但不 应该强迫用户提交本地未完成的修改.

这条规则主要的副作用是工作副本必须记录额外的信息来跟踪混合 的版本号，同时还要能够处理版本号混合的情况. 目录也是版本库的一部 分，这使得情况变得更加复杂.

举例来说，假设你有一个版本号是 10 的工作副本，检出后有人往 仓库提交了修改，仓库最新的版本号是 14. 你修改了文件 *foo.html*, 然后向仓库提交了修改，创建了新版本 号 15, 提交完成后，许多 Subversion 新手可能会认为工作副本的版本号 会自动更新到 16, 但事实并非如此. 在版本号 10 和 15 之间，仓库可 能发生了任意次数的修改，但是客户端对此一无所知，因为你并没有执行 *svn update*, 而 *svn commit* 并不会自动从仓库抓取更新. 如果让 *svn commit* 自动下载更新，它就会把工作副本整体的版本号更新到 15—但是这 样做就违背了基本规则 “推送和抓取是分开”. 于是，客户 端唯一能做的安全操作是只把文件 *foo.html* 的 版本号更新到 15, 工作副本中的其他文件与目录的版本号依然停留在 10. 只有在执行 *svn update* 后，仓库的最新修改才会 被下载到本地，并把工作副本的版本号更新到 15.

版本号混合是正常情况

事实上，*每次* 执行 *svn commit* 都会产生新的版本号混合的情况，刚被 提交的文件或目录的版本号是工作副本中的最大值. 再经历过几次提交 后（提交之间没有执行更新操作），工作副本的版本号就已经处于一种非常 混乱的情况，即使在此期间只有一个人在往仓库提交修改，这种情况也会 发生. 为了查看版本号的混乱情况，带上选项 `--verbose` (`-v`) 执行 *svn status* (参考 “查看修改的整体概述”一节).

新用户常常没有意识到他们的工作副本包含了混合的版本号，有时候 可能会让他们感到很困惑，因为很多客户端命令对文件或目录的版本号 很敏感. 例如，命令 *svn log* 会列出文件或目录 的修改历史（见 “生成历史修改列表”一节), 当用户对某个文件执行 *svn log* 时，他很可能想看到文件的 全部修改历史，但是如果文件在工作副本中的版本号太老了（常常是因 为太久没有执行过 *svn update*), 那么较新的修改 历史就不会显示出来.

版本号混合是有益的

如果项目足够复杂，你就会发现只把工作副本的某一部分 回退 (*backdate*) 到一个较旧的版本会很方便—我们将会在 第 2 章 基本用法 介绍如何完成这种操作. 也许是你想要测试存放在某个子目录中的子模块 早期版本，又或许是你想要查出某个文件的问题是在什么时候第一次出现. 这是版本控制系统的 “时间机器” 特性，该特性允许用户把 工作副本的任意一部分在时间上向前或向后移动.

版本号混合的限制

不过，在使用工作副本的版本号混合特性时会有一些限制条件.

首先，如果你删除了过时的文件或目录，则不能提交删除. 因为如果 仓库中有更新的版本，该限制就可以避免用户在没有看到新版本的情况下 做出错误的决定.

然后，除非目录是最新的，否则不能提交该目录的元数据修改 (我们 将在 第 3 章 高级主题 介绍如何为项目 (item) 添加 属性). 目录的工作版本号定义了一个条目和属性的特定集合，提交过时 的目录的属性修改可能会销毁用户还没有看到的属性.

最后，从 Subversion 1.7 开始，含有版本号混合情况的工作副本 不能作为合并操作的目标 (引入这个限制的原因和前面两条类似).

小结

本章介绍了 Subversion 的很多基本概念:

- 介绍了中央版本库，客户端工作副本，以及仓库版本号树组成的队列.
- 通过几个简单的例子，介绍了如何利用 “复制-修改-合并” 模型，和同事协作使用 Subversion 发布和接收修改.
- 关于 Subversion 如何跟踪和管理工作副本的信息，本章也进行了一 些介绍.

到这里为止，对于 Subversion 的工作方式读者应该有了一个很直观的感受，有了本章的基础知识作为后盾，读者可以接着阅读下一章，下一章会更加详细地 介绍 Subversion 的命令和特性.

第 2 章 基本用法

理论是很用的，但是实际使用它们的乐趣却很简单．现在我们开始介绍使用 Subversion 的细节，到这一章结束时，读者在日常工作中使用 Subversion 将 不会遇到太大的问题．本章首先介绍如何把文件纳入 Subversion，然后对代码 进行首次检出，接下来将对代码进行一些修改，并检查修改前后的具体差异．读 者还将会看到如何把其他人的修改应用到自己的工作副本中，检查修改并解决可 能的冲突．

本章不会介绍 Subversion 的所有命令 — 而是以对话的方式介绍在使 用 Subversion 的过程中最经常遇到的问题．本章假设读者已经读过并理解了 第 1 章 基本概念 的内容，而且熟悉 Subversion 的一般模型．关 于全部命令的完整参考手册，请阅读 svn 参考手册—Subversion 命令行客户端．

另外，本章还假设读者已经拥有了一个已存在的 Subversion 仓库．没有仓库 就没有工作副本，没有工作副本就无法练习本章的内容．在因特网上可以找到许多 提供免费或廉价的 Subversion 仓库托管服务的网站，如果读者想要自己创建仓库，请阅读 第 5 章 仓库管理．为了练习本章的例子，读者必须对 Subversion 仓库拥有访问权限．

最后，如果某个 Subversion 命令需要通过网络连接仓库，这可能需要用户 认证．为简单起见，本章的所有例子都会避开和认证相关的内容．需要注意的是，如果读者想把本章介绍的内容应用到某个真实世界中的 Subversion 实例，你很可 能需要向服务器提供用户名与密码．关于认证和客户端证书的更多内容，请阅读 “客户端证书”一节．

帮助!

本书的目标是成为 Subversion 新老用户的助手与信息来源，不过，Subversion 的命令行工具本身就带有丰富的帮助文档，如此一来，读者就 用不着每次都从书架上拿这本书．命令 *svn help* 是 打开内置文档的入口：

```
$ svn help
usage: svn <subcommand> [options] [args]
Subversion command-line client, version 1.8.13.
Type 'svn help <subcommand>' for help on a specific subcommand.
Type 'svn --version' to see the program version and RA modules
  or 'svn --version --quiet' to see just the version number.

Most subcommands take file and/or directory arguments, recursing
on the directories.  If no arguments are supplied to such a
command, it recurses on the current directory (inclusive) by default.

Available subcommands:
   add
   blame (praise, annotate, ann)
   cat
…
```

上面的输出内容已经说了，读者可以用 `svn help` *`SUBCOMMAND`* 查看特定子命令的帮助信息，Subversion 会输出这个子命令完整的使用方法，包括它的语法，选项和功能：

```
$ svn help help
help (?, h): Describe the usage of this program or its subcommands.
```

```
usage: help [SUBCOMMAND...]

Global options:
  --username ARG           : specify a username ARG
  --password ARG           : specify a password ARG
…
```

> **选项 (Options), 开关 (Switches) 和标志 (Flags), 这都是什 么鬼?**
>
> Subversion 的客户端命令行工具拥有大量的命令修饰符，有些人把它们 叫作 “开关” 或 “标志” — 本书把它们叫 作 “选项”. 读者将会看到特定的 *svn* 子命令支持的选项，再加上所有子命令都支持的全局选项，全局选项显示在 子命令帮助信息靠近底部的位置.
>
> Subversion 的选项有两种形式: 短选项由一个连字符和一个英文字母 组成，长选项以两个连续的连字符开始，后跟几个英文字母和连字符 (例如 短选项 `-s` 和长选项 `--this-is-a-long-option`), 每一个选项都至少有一个长格式. 有些选项—例如 `--changelist`—会有一个缩写的长格式别名 (`--cl`), 只有特定的几个选项 — 通常是最经常用到的选项 — 才会拥有一个额外的短格式. 为了使书中的内容更加清晰，在例子中我们通常 使用选项的长格式，但是在描述选项时，如果该选项存在短格式，我们会 同时列出长格式（为了清晰）和短格式（为了方便记忆）. 读者在执行 Subversion 命令时，可以自由选择选项的格式.

许多基于 Unix 的 Subversion 发行版都包含了手册页，可以通过命令 *man* 打开，但是手册页往往只是指出了帮助信息的真正 位置，例如项目网址. 另外，有些公司会通过论坛和有偿咨询提供 Subversion 的帮助与支持. 当然，还有资源最丰富的因特网. 获取 Subversion 的帮助从 来就不是一件难事.

往仓库中添加数据

往 Subversion 仓库中添加新文件有 2 种办法: *svn import* 和 *svn add*. 这里先介绍 *svn import*, *svn add* 在本章的后面 再作介绍.

导入文件和目录

命令 *svn import* 可以快速地向仓库中添加新文件 或目录. *svn import* 不要求工作副本，新增的文件会 马上提交到仓库中. 使用该命令的典型情况是用户想要把一个已存在的目录 添加到 Subversion 仓库中，例如:

```
$ svn import /path/to/mytree \
             http://svn.example.com/svn/repo/some/project \
             -m "Initial import"
Adding         mytree/foo.c
Adding         mytree/bar.c
Adding         mytree/subdir
Adding         mytree/subdir/quux.h

Committed revision 1.
$
```

上面的例子把本地目录 *mytree* 中的内容添加到 仓库的 *some/project* 目录中. 注意，用户在导入 前无需创建新目录 — *svn import* 会自动完成这 些工作. 提交后，用户就可以在仓库中看到新增的文件和目录:

```
$ svn list http://svn.example.com/svn/repo/some/project
bar.c
foo.c
subdir/
$
```

注意，导入完成后，本地的原始目录并没有被转换成工作副本，为了能够以 版本控制的方式对文件进行管理，用户仍然需要创建一份最新的工作副本.

推荐的仓库布局

为了方便用户管理数据，Subversion 提供了很大的灵活性. Subversion 只是简单地对目录和文件进行版本控制，不会给它们附上特殊的意义，用户 完全可以按照自己的喜好来决定数据的布局. 不过，这种灵活性有时也会带 来一些麻烦，如果用户同时在 2 个或多个布局完全不同的仓库中浏览，而这 些仓库的布局又没有规律，用户往往会感到迷失，不知身在何处.

为了避免这种问题，我们推荐读者遵循传统的仓库布局（这种布局 出现在很久以前，在 Subversion 项目早期阶段就已经开始使用），传统布局的 特点是，仓库中的目录名可以向用户传达出与它们所存放的数据相关的信息. 大多数项目都有一条公认的开发 “主线”，或者叫作 主干 (*trunk*)；还有一 些 分支 (*branches*)，分 支是某一条开发线的分叉；还有一些 标签 (*tags*)，标签是某一条开发线的稳定版快照. 我们首先建议 每一个项目在仓库中都一个公认的 项目根目录 (*project root*)，目录中只存放和该项目相关的 数据. 然后，我们建议每一个项目根目录下都有一个表示开发主线的 *trunk* 子目录，存放所有分支的 *branches* 子目录，存放所有标签的 *tags* 子目录. 如果仓库只存放单个项目，那么仓库的根目录也可以作为项目根目录.

这里有一些例子：

```
$ svn list file:///var/svn/single-project-repo
trunk/
branches/
tags/
$ svn list file:///var/svn/multi-project-repo
project-A/
project-B/
$ svn list file:///var/svn/multi-project-repo/project-A
trunk/
branches/
tags/
$
```

关于标签和分支的更多内容，我们将在 第 4 章 分支与合并 介绍. 如果用户要为多个项目创建仓库，其中的细节与建议请参考 “仓库布局”一节. 关于项目根目录的更多 内容，我们将在 “规划仓库的组织方式”一节 介绍.

名字中有什么

Subversion 尽量不去限制用户想要管理的数据类型，文件的内容及其 属性被看成二进制数据进行存放和传输， “文件内容类型”一节 将会介绍如何向 Subversion 提示某个文件不需要 “文本” 操作. 然而，在有些 地方，Subversion 会限制其中所存放的数据类型.

某些数据由 Subversion 内部进行处理 — 例如属性名，文件路径，日志消息 — 这些数据使用 UTF-8 编码，不过这并不意味着和 Subversion 交互时一定要使用 UTF-8 编码，一般情况下，Subversion 客户端会自动对 编码进行转换，不需要用户参与.

在 WebDAV 交换数据过程和版本比较老的 Subversion 管理文件中，文件路径是 XML 的属性值，属性名是 XML 的标签名，此时文件路径只能包含合法的 XML (1.0) 字符，属性也只能使用 ASCII 字符. Subversion 禁止在文件路径 中出现 TAB, CR 和 LF 这些字符，这是为了避免在差异比较时，或者在命令 *svn log* 和 *svn status* 和输出中 文件路径被错误地断开.

听起来有许多规则需要记住，但是在实际使用时这些限制极少会对用户产 生影响. 只要用户的本地化设置和 UTF-8 兼容，而且不在文件路径中使用 控制字符，在和 Subversion 通信时就不会产生任何问题. 客户端命令行工具 也能提供帮助 — 为了方便内部使用，用户在输入 URL 时，命令行工具 自动地对非法路径字符进行转义，从而创建出方便内部使用的合法路径.

在选择有效的路径名时，Subversion 不是唯一的限制因素，使用多个 操作系统的开发团队还需要考虑不同操作系统对路径名的限制. 例如，Windows 不允许文件名包含冒号，但是使用 Linux 系统的用户可以轻易地 在仓库中添加这样的文件，这些文件将不能被 Windows 用户检出. 有些 操作系统区分文件名的大小写，而有些不区分. 所以和 Subversion 相比，用户更应该注意不同的操作系统和文件系统所产生的限制.

创建工作副本

大多数时候，用户开始使用仓库是通过执行 检出 (*checkout*) 命令. 检出仓库中的目录将会在用户的 本地主机上创建一个该目录的工作副本. 除非特意指定，否则这个副本将包 含仓库最新版本的数据：

```
$ svn checkout http://svn.example.com/svn/repo/trunk
A    trunk/README
A    trunk/INSTALL
A    trunk/src/main.c
A    trunk/src/header.h
…
Checked out revision 8810.
$
```

上面的例子检出的是主干目录，但用户也可以轻易地检出更深层的子目录，只需要在检出命令的参数中写上子目录对应的 URL 即可：

```
$ svn checkout http://svn.example.com/svn/repo/trunk/src
A    src/main.c
A    src/header.h
A    src/lib/helpers.c
…
Checked out revision 8810.
$
```

因为 Subversion 用的是 复制–修改–合并 模型，而非 加锁–修改–解锁（见 “版本控制模型”一节），所以用户马上就可以修改 工作副本里的文件与目录. 工作副本就像一个普通目录，用户可以编辑或 重命名里面的文件，甚至可以删除整个工作副本.

虽然工作副本 “像一个普通目录”，用户可以按照自己 的意愿编辑里面的文件， 但是其他事情必须告诉给 Subversion. 例如，如果 用户想要复制或移动工作副本中的某个文件或目录，必须使用 *svn copy* 或 *svn move*, 而不是 操作系统提供的复制与移动命令. 关于它们的更多内容会在后面讲到.

除非用户准备提交修改，否则不需要通知 Subversion 服务器你做了哪些 修改.

目录 .svn 里有什么东西?

工作副本的根目录 — 1.7 版以前是每个目录及其子目录 — 都有一个用于管理的子目录 *.svn*. 通常情况下，操作 系统的目录列表指令不会显示该目录，但它是一个非常重要的目录，无论用户 做什么操作，都不能删除或修改其中的内容，Subversion 管理工作副本的信息 都存放在这个目录里.

在上面的两个例子中, Subversion 在本地创建的目录名是检出命令中 URL 参数的最后一个分量. 如果用户只向 *svn checkout* 提供了 URL 参数, 那么根据最后一个分量来创建目录对用户来说就比较方便. 不过 Subversion 客户端命令行工具也允许用户自己指定一个目录名, 例如:

```
$ svn checkout http://svn.example.com/svn/repo/trunk my-working-copy
A    my-working-copy/README
A    my-working-copy/INSTALL
A    my-working-copy/src/main.c
A    my-working-copy/src/header.h
…
Checked out revision 8810.
$
```

如果用户指定的本地目录不存在, *svn checkout* 会自动创建该目录.

基本工作周期

Subversion 支持的特性与选项非常丰富， 但是能够在日常工作中用到的却很 少. 本节将介绍日常工作中最常用到的 Subversion 操作.

典型的工作周期就像:

1. 更新工作副本. 这会用到命令 *svn update*.

2. 修改. 最常见的修改就是编辑已有文件的内容，但有时还要添加，删除，复制和移动文件或目录 — 命令 *svn add*, *svn delete*, *svn copy* 和 *svn move* 负责 处理工作副本的结构性调整.

3. 审查修改. 用命令 *svn status* 和 *svn diff* 查看工作副本发生了哪些变化.

4. 修正错误. 人无完人，在审查修改时用户可 能会发现某些修改是不正确的. 有时候修正错误最简单的方式是撤消所有的修改，重新开始. 命令 *svn revert* 可以把文件或目 录恢复到修改前的样子.

5. 解决冲突 (合并其他人的修改). 当一个用户 正在修改文件时，其他人可能已经把自己的修改提交到了服务器上. 为了防止 在提交修改时，由于工作副本过旧导致提交失败，用户需要把其他人的修改 更新到本地，用到的命令是 *svn update*. 如果命令 的执行结果有冲突产生，用户需要用命令 *svn resolve* 解决冲突.

6. 发布 (提交) 修改. 命令 *svn commit* 把工作副本的修改提交到仓库中，如果修改 被接受，其他用户就可以看到这些修改.

更新工作副本

如果某个项目正在被多个工作副本修改，用户就需要更新自己本地的 工作副本，以获取其他人提交的修改．这些修改可能来自团队中的其他开发 人员，也可能是自己在其他地方提交的修改．Subversion 不允许用户向过时 的文件或目录提交修改，所以在开始修改前，最好保证本地工作副本的内容 是最新的．

命令 *svn update* 把仓库上的最新数据同步到本地 的工作副本：

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

从上面的例子可以看到，在你最后一次更新了工作副本后，有人修改了 *foo.c* 和 *bar.c*，Subversion 把更新同步到本地工作副本．

通过命令 *svn update*，服务器把修改应用到本地的 工作副本，同时在被更新项目的旁边显示一个字母，表示 Subversion 对文件 采取了什么操作．为了明白这些字母的涵义，查看 `svn help update` 的输出或参考 svn 参考手册—Subversion 命令行客户端 的 svn update (up)．

修改

现在用户可以开始工作，修改工作副本里的资料．工作副本支持的修改类型分 为两种：文件修改 (*file changes*) 和 目录修改 (*tree changes*)．在修改文件时不需要告知 Subversion，用户可以使用任意一种自己喜欢的工具来修改文件，例如编辑 器，字处理程序，图形工具等．Subversion 可以自动检测到哪些文件发生了 变化，处理二进制文件和处理文本文件一样简单高效．目录修改涉及到目录结构 的变化，例如添加和删除文件，重命名文件和目录，复制文件和目录．目录修改 要使用 Subversion 的命令完成．文件修改和目录修改只有在提交后才会更新 到仓库中．

对符号链接进行版本控制

在类 Unix 系统中，Subversion 还能对特殊的文件类型— 符号链接 (*symbolic link*，或者叫作 “软链接 ”) 进行版本控制．软链接是一种透明的 引用，引用到文件系统中的其他文件或目录，对软链接的操作会间接地作用到 它所引用的文件或目录上．

如果把一个软链接提交到仓库中，Subversion 会自动加以识别．如果在不支持软链接的 Windows 操作系统中检出工作副本，Subversion 就会 创建一个同名的普通文件，文件内容是软链接所指向的对象的路径，虽然该 文件不能当作一个软链接使用，但 Windows 用户仍然可以编辑该文件．

下面是最常用到的 5 个改变目录结构的 Subversion 子命令：

`svn add FOO`

这个命令把文件，目录或软链接 *FOO* 添加 到需要进行版本控制的名单中，在下一次提交时，*FOO* 就会正式添加到仓库里．如果 *FOO* 是一个目录，那么目录内的所有内容都会 被添加到仓库中．如果只想添加 *FOO* 它自己，就带上选项 `--depth=empty`．

`svn delete FOO`

上面的命令从工作副本中删除文件，目录或符号链接 *FOO*，在下一次提交时，*FOO* 就会从仓库中删除.（当然，没有什么东西可以从仓库中被完全地删除干净 —它们只是从版本号 HEAD 中删除，用户可以 从更早的版本中看到被删除的文件）.[1]

`svn copy FOO BAR`

从 *FOO* 复制出一个 *BAR*，并把 *BAR* 添加到 需要进行版本控制的名单中. *BAR* 被提交到仓库 后，Subversion 会记录它是由 *FOO* 复制得到的. 除非带上选项 `--parents`，否则 *svn copy* 不会创建父目录.

`svn move FOO BAR`

这条命令等价于 `svn copy FOO BAR; svn delete FOO` ，也就是从 *FOO* 复制出一个 *BAR*，然后再删除 *FOO*. 除非带上选项 `--parents`，否则 *svn move* 不会创建父目录.

`svn mkdir FOO`

该命令等价于 `mkdir FOO; svn add FOO`，也就是创建一个新目录 *FOO*，并把它添加到仓库中.

在没有工作副本的情况下修改仓库

Subversion 确实 支持在没有显式的提交操作 下，马上把目录修改提交到仓库中. 除了工作副本中的路径，*svn mkdir*，*svn copy* 和 *svn delete* 还可以接受仓库的 URL 作为参数. 前面 我们也提到过，*svn import* 总是直接修改仓库. 我们会在 “在没有工作副本的情况下工作”一节 介绍 如何在没有工作副本的情况下提交目录修改.

执行基于 URL 的操作既有好处也有坏处，比较明显的好处是速度快：仅仅为了执行一个很简单的操作而检出工作副本，未免也太麻烦了. 坏处 是用户一次只能执行一个或一种类型的操作. 使用工作副本最大的好处是 它可以作为修改的“暂存区”，在提交之前用户可以检查 修改是否正确. 暂存区的修改既可以简单，也可以很复杂，它们都 会被当作一个单独的修改提交到仓库中.

审查修改

工作副本修改完成后，就要把它们都提交到仓库中，不过在提交之前，应该查看 一下自己到底修改了哪些东西. 通过检查修改，用户可以写出更准确的 提交日志 (*log message*，和修改一起存放到仓库中的一段文本，该文本以人类可读的形式描述了本次 修改的相关信息). 在审查修改时，用户可能会发现自己无意中修改了一个不相 关的文件，因此在提交之前需要撤消它的修改. 用户可以使用命令 *svn status* 查看修改的整体概述，用命令 *svn diff* 查看修改的细节.

看，没网络!

如果仓库是通过网络访问的，那么即使没有网络连接，用户也可以执行 *svn status*，*svn diff* 和 *svn revert*，这就方便用户在离线的情况下管理和 审查未完成的修改.

为了完成这些功能，Subversion 为每一个正在被版本控制的文件，在工作 副本的管理区中 (1.7 前的版本有多个管理区) 缓存一份未修改的原始文 件，这就允许 Subversion 在不需要连接网络的情况下，查看或撤消本地 的修改. 这些缓存（称为 基文本 (*text-base*)) 允许 Subversion 把用户的修改 压缩后再向服务器提交. 这样做有很大的好处 — 就算用户的网络访问速度很快，但是和发送整个文件相比，只发送差异会快得多.

[1] 如果你希望在 HEAD 中重新看到被删除的文件，参考 “恢复已删除的文件”一节.

查看修改的整体概述

为了看到修改的整体概述，使用命令 *svn status*，它可能是用户最常用到的一个 Subversion 命令.

因为 *cvs status* 的输出非常庞杂，而 *cvs update* 不仅执行更新操作，还会报告本 地的修改情况，所以大多数 CVS 用户更喜欢用 *cvs update* 来查看修改情况. 对 Subversion 来 说，更新和状态报告这两个功能是完全分离的，更多的细节在 “状态与更新的区别”一节.

如果在工作副本的根目录不加任何参数地执行 `svn status`，Subversion 就会检查并报告所有 文件和目录的修改.

```
$ svn status
?       scratch.c
A       stuff/loot
A       stuff/loot/new.c
D       stuff/old.c
M       bar.c
$
```

在默认的输出模式下，*svn status* 先打印 7 列 字符，然后是几个空白字符，最后是文件或目录名. 第一列字符报告文件或 目录的状态，其中最常的几种字符或状态是：

`? item`

文件，目录或符号链接 *item* 不在版本 控制的名单中.

`A item`

文件，目录或符号链接 *item* 是新增的，在下一次提交时就会加入到仓库中.

`C item`

文件 *item* 有未解决的冲突，意思是说从 服务器收到的更新和该文件的本地修改有所重叠，Subversion 在处理 这些重叠的修改时发生了冲突. 用户必须解决掉冲突后才能向仓库 提交修改.

`D item`

文件，目录或符号链接 *item* 已被删除，在下一次提交时就会从仓库中删除 *item*.

`M item`

文件 *item* 的内容被修改.

如果给 *svn status* 传递一个路径名，那么命 令只会输出和该路径相关的状态信息：

```
$ svn status stuff/fish.c
D       stuff/fish.c
```

svn status 支持选项 `--verbose (-v)`，带上该选项后，命令会输出当前目录中每一项的 状态，即使是未被修改的项目：

```
$ svn status -v
```

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

这是 *svn status* 的 “长格式” (long form) 输出. 第一列字符的含义不变, 第二列显示该项在工作副本 中的版本号, 第三和第四列显示该项最后一次被修改的版本号和作者.

前面执行的几次 *svn status* 都不需要和仓库 通信 — 它们只是根据工作副本管理区里的数据和文件当前的内容 来报告各个文件的状态. 有时候用户可能想知道在上一次更新之后, 哪些 文件在仓库中又被更新了, 为此, 可以给 *svn status* 带上选项 `--show-updates` (`-u`), 这样 Subversion 就会和仓库通信, 输出工作副本中已过时的项目:

```
$ svn status -u -v
M      *        44        23    sally     README
M               44        20    harry     bar.c
       *        44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
A                0         ?     ?        stuff/things/bloo.h
Status against revision:   46
```

注意带有星号的那 2 行, 如果此时执行 `svn update`, 就会从仓库收到 *README* 和 *trout.c* 的更新. 除此之外我们还可以知道, 在本地被修改的文件当中, 至少有一个在仓库中 也被更新了 (文件 *README*), 所以用户必须在提交前把 仓库的更新同步到本地, 否则仓库将会拒绝针对已过时文件的提交, 关于这点我们会在后面介绍更多的细节.

除了我们介绍的例子, *svn status* 还可以显示更 丰富的信息, 关于 *svn status* 更详细的介绍, 查看 `svn help status` 的输出或阅读 svn 参考手册—Subversion 命令行客户端 的 svn status (stat, st)

查看修改的细节

查看修改的另一个命令是 *svn diff*, 它会输出文件 内容的变化. 如果在工作副本的根目录不加任何参数地执行 `svn diff`, Subversion 就会输出工作副本中人类 可读的文件的变化. 文件的变化以 *标准差异* (*unified diff*) 格式进行输出, 这种格式把文件 内容的变化描述成 “块” (hunk) 或 “片断” (snippet), 其中每一行文本都加上一个单字符前缀: 空格表示该行没有变化; 负号 (`-`) 表示该行被删除; 正号 (`+`) 表示该行是新增的. 在 *svn diff* 的语境中, 这些冠以正负号的行显示了修改 前的行和修改后的行分别是什么样子的.

这是一个执行 `svn diff` 的例子:

```
$ svn diff
Index: bar.c
===================================================================
--- bar.c (revision 3)
+++ bar.c (working copy)
@@ -1,7 +1,12 @@
```

```
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>

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

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

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

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

svn diff 在比较了工作副本中的文件和基文本后 再输出它们之间的差异. 在命令的输出中, 新增的文件其每一行都被冠 以正号; 被删除的文件其每一行都被冠以负号. *svn diff* 的输出格式和程序 *patch* 以及 Subversion 1.7 引入的子命令 *svn patch* 兼容. 处理补丁的命令 (例如 *patch* 和 *svn patch*) 可以读取并应用 补丁文件 (*patch files*, 简称 “补丁”). 利用补丁, 用户就可以在不提交的情况下, 把工作副本的修改分享给其他 人, 创建补丁的方式是把 *svn diff* 的输出重定向到 补丁文件里:

```
$ svn diff > patchfile
$
```

Subversion 默认使用它自己内部的差异比较程序来生成标准差异格式 的输出. 如果用户想要其他格式的差异输出, 就用选项 `--diff-cmd` 指定一个外部的差异比较程序, 如果需要 的话, 还可以用选项 `--extensions` 向差异比较程序 传递其他额外的参数. 例如, 用户想用 GNU 的程序 *diff* 对文件 *foo.c* 进行差异比较, 还要求 *diff* 在比较时忽略大小写, 按照上下文差异格式来 产生输出:

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i" foo.c
…
$
```

修正错误

假设用户在查看 *svn diff* 的输出时发现针对某 一文件的修改都是错误的，也许这个文件就不应该被修改，也许重新开始 修改文件会更加容易．为了撤消现在的修改，用户可以再次编辑文件，手动 地复原成原来的样子，又或者是从其他地方找到一个原始文件，把改错的 文件覆盖掉，还可以用 `svn patch --reverse-diff` 或 `patch -R` 逆向应用补丁，除此之外可能还有 其他办法．

幸运的是 Subversion 提供了一种简便的方法来撤消工作副本中的 修改，用到的命令是 *svn revert*:

```
$ svn status README
M       README
$ svn revert README
Reverted 'README'
$ svn status README
$
```

在上面的例子里，Subversion 利用缓存在基文本中的内容，把文件 回滚到修改前的原始状态．需要注意的是，*svn revert* 会撤消 任何 一个未提交的修改，例如用户可能不 想往仓库中添加新文件：

```
$ svn status new-file.txt
?       new-file.txt
$ svn add new-file.txt
A         new-file.txt
$ svn revert new-file.txt
Reverted 'new-file.txt'
$ svn status new-file.txt
?       new-file.txt
$
```

或者是用户错误地删除了一个本不该删除的文件：

```
$ svn status README
$ svn delete README
D         README
$ svn revert README
Reverted 'README'
$ svn status README
$
```

svn revert 提供了一个很好的补救机会，否则的话，用户就得花费大量的时间，自己一点一点地手工撤消修改，又或 者采用一个更麻烦的做法，直接删除工作副本，然后重新从服务器上检出一个 干净的工作副本．

解决冲突

我们已经看过 *svn status -u* 如何预测是否 有冲突，但是解决冲突仍然需要由用户自己来完成．冲突可以在用户把 仓库中的修改合并到本地工作副本的任何时候发生，到目前为止用户已经 知道的命令中，*svn update* 就有可能产生冲突— 该命令的唯一功能就是把仓库中的更新合并到本地工作副本．那么当发生冲突 时 Subversion 如何通知用户，以及用户应该如何处理它们?

假设用户在执行 `svn update` 后看到了如下 输出:

```
$ svn update
Updating '.':
U    INSTALL
G    README
Conflict discovered in 'bar.c'.
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options:
```

不用担心左边有 `U` (Updated, 更新) 或 `G` (merGed, 合并) 的文件, 这表示 它们成功地吸收了来自仓库的更新. `U` 表示该文件不包含本地修改, 只是用仓库中的修改更新了文件内容. `G` 表示该文件含有本地修改, 但是这 些修改和来自仓库的修改没有冲突.

再下来几行就比较有趣了. 首先, Subversion 报告说在把仓库的修改 合并到文件 *bar.c* 时, 发现其中一些修改和本地未 提交的修改产生了冲突. 原因可能是其他人和用户都修改了同一行, 无论是 因为什么, Subversion 在发现冲突时会马上把文件置成冲突状态, 然后询问 用户他想怎么办. 用户可以从 Subversion 给出的几个选项中选择一个, 如果想看完整的选项列表, 就输入 `s`:

```
…
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options: s

  (e)  - change merged file in an editor  [edit]
  (df) - show all changes made to merged file
  (r)  - accept merged version of file

  (dc) - show all conflicts (ignoring merged version)
  (mc) - accept my version for all conflicts (same)  [mine-conflict]
  (tc) - accept their version for all conflicts (same)  [theirs-conflict]

  (mf) - accept my version of entire file (even non-conflicts)  [mine-full]
  (tf) - accept their version of entire file (same)  [theirs-full]

  (m)  - use internal merge tool to resolve conflict
  (l)  - launch external tool to resolve conflict  [launch]
  (p)  - mark the conflict to be resolved later  [postpone]
  (q)  - postpone all remaining conflicts
  (s)  - show this list (also 'h', '?')
Words in square brackets are the corresponding --accept option arguments.

Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options:
```

先简单地介绍一下每一个选项.

`(e) edit [edit]`

使用环境变量 `EDITOR` 定义的编辑器打开 发生冲突的文件.

`(df) diff-full`

按照标准差异格式显示基础修订版和冲突的文件之间的差异.

`(r) resolved`

编辑完成后, 告诉 *svn* 用户已经解决了冲突, 现在应该接受文件的当前内容.

`(dc) display-conflict`

显示冲突的区域, 忽略合并成功的修改.

`(mc) mine-conflict [mine-conflict]`

丢弃从服务器收到的, 与本地冲突的所有修改, 但是接受不会产生 冲突的修改.

`(tc) theirs-conflict [theirs-conflict]`

丢弃与服务器产生冲突的所有本地修改, 但是保留不会产生冲突 的本地修改.

`(mf) mine-full [mine-full]`

丢弃从服务器收到的该文件的所有修改, 但是保留该文件的 本地修改.

`(tf) theirs-full [theirs-full]`

丢弃该文件的所有本地修改, 只使用从服务器收到的修改.

`(m) merge`

打开一个内部文件合并工具来解决冲突, 该选项从 Subversion 1.8 开始支持.

`(l) launch`

打开一个外部程序来解决冲突, 在第一次使用该选项之前需要完成 一些准备工作.

`(p) postpone [postpone]`

让文件停留在冲突状态, 在更新完成后再解决冲突.

`(s) show all`

显示所有的, 可以用在交互式的冲突解决中的命令.

我们将对以上命令进行更为详细的说明, 说明中将按照功能对命令进行 分组.

交互式地查看冲突差异

在决定如何交互地解决冲突之前, 有必要看一下冲突的内容, 其中有两 个命令可以帮到我们. 第一个是 `df`:

```
…
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all 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
…
```

差异内容的第一行显示了工作副本之前的内容 (版本号 BASE), 下一行是用户的修改, 最后一行是从服务器 收到的修改 (通常 是版本号 HEAD).

第二个命令和第一个比较类似, 但是 `dc` 只会显示冲突区域, 而不是文件的所有修改. 另外, 该命令显示冲突区域 的格式也稍有不同, 这种格式允许用户更方便地比较文件在三种状态下的 内容: 原始状态; 带有用户的本地修改, 忽略服务器的冲突修改; 带有服 务器的修改, 忽略用户的本地修改.

审查完这些命令提供的信息之后, 用户就可以采取下一步动作.

交互式地解决冲突差异

交互式地解决冲突的主要方法是使用一个内部文件合并工具, 该工具 询问用户如何处理每一个冲突修改, 而且允许用户有选择地合并和编辑修改. 除此之外还有其他几种方式用于交互式地解决冲突—其中两种允许用 户使用外部编辑器, 有选择地合并和编辑修改, 另外几种允许用户简单地选择 文件版本. 内部合并工具集合了所有解决冲突的方式.

看完引起冲突的修改后, 接下来就要解决这些冲突. 我们要介绍的第 一个命令是 `m` (merge), 从 Subversion 1.8 开 始支持, 该命令允许用户从众多选项中选择一个来解决冲突:

```
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options: m
Merging 'Makefile'.
Conflicting section found during merge:
(1) their version (at line 24)                |(2) your version (at line 24)
-----------------------------------------------+-----------------------------------------------
top_builddir = /bar                            |top_builddir = /foo
-----------------------------------------------+-----------------------------------------------
Select: (1) use their version, (2) use your version,
        (12) their version first, then yours,
        (21) your version first, then theirs,
        (e1) edit their version and use the result,
        (e2) edit your version and use the result,
```

```
(eb) edit both versions and use the result,
(p) postpone this conflicting section leaving conflict markers,
(a) abort file merge and return to main menu:
```

从上面可以看到，使用内部文件合并工具时，用户可以循环遍历文件中 的每一个冲突区域，对每一个冲突区域用户都可以选择一个不同的选项，或者 推迟解决该冲突.

如果用户想用一个外部编辑器来选择本地修改的某些组合， 此时可用 用命令 `e` (edit) 来手动地编辑带有冲突标记的 文件，该命令会打开一个文本编辑器（参考 “使用外部编辑器”一节). 文件编辑完毕后，如果用户感到满意，就要用命令 `r` (resolved) 告诉 Subversion 文件的冲突已经解决了.

不管别人怎么说， 使用文本编辑器编辑文件来解决冲突是一种比较低 级的方法 （见 “手动地解决冲突”一节)，因此，Subversion 提供了一个命令 `l` (launch) 来打开 精美的图形化合并工具（见 “外部合并工具”一节).

还有两个稍微折衷一点的选项，命令 `mc` (mine-conflict) 和 `tc` (theirs-conflict) 分别告诉 Subversion 选择用户的本地修改或从服务器收到的修改作为冲突 获胜的一方. 但是和 “mine-full” 以及 “theirs-full” 不同的是，这两个命令会保留不产生冲突的本地修改和从服务器收到的修改.

最后，如果用户决定只想使用本地修改，或者是只使用从服务器收到 的修改，可以分别选择 `mf` (mine-full) 与 `tf` (theirs-full).

推迟解决冲突

这节的标题看起来好像是在讲如何避免夫妻之间爆发冲突，可实际上 本节还是在介绍和 Subversion 相关的内容. 如果用户在更新时遇到了冲 突，但是还没有准备好立即解决，这时可以选择 `p` (postpone) 来推迟解决. 如果用户早就准备好不想交互式地解决冲突，可以给 *svn update* 增加一个参数 `--non-interactive`，此时发生冲突的文件会被自动 标记为 `C`.

从 Subversion 1.8 开始，内部的文件合并工具允许用户推迟解决 某些特定的冲突，但仍然可以解决其他冲突. 于是，用户可以以冲突 区域为单位（而不仅仅是以文件为单位）来决定哪些冲突可以推迟解决.

`C` (Conflicted) 表示来自服务器的修改和用户的本地修改有所重叠，用户在更新完成后必须 手动加以选择. 如果用户推迟解决冲突，*svn* 通常 会从三个方面帮助用户解决冲突：

- 如果在更新过程中产生了冲突，Subversion 就会为含有冲突的 文件打印一个字符 `C`，并记住 该文件处于冲突状态.
- 如果 Subversion 认为文件是支持合并的，它就会把 *冲突标记* (*conflict markers*)——一段给冲突划分 边界的特殊文本——插入到文本中来显式地指出重叠区域 (Subversion 使用属性 `svn:mime-type` 来判断 一个文件是否支持基于行的合并，见 “文件内容类型”一节).
- 对每一个产生冲突的文件，Subversion 都会在工作副本中生成 三个额外的文件，这些文件不在版本控制的名单中：

  filename.mine

  该文件的内容和用户执行更新操作前的文件内容相同，它 包含了当时所有的本地修改（如果 Subversion 认为该文件不 支持合并就不会创建 *.mine*).

  filename.rOLDREV

  该文件的内容和版本号 `BASE` 对应 的文件内容相同，也就是在执行更新操作前工作副本中未修改 的版本，*`OLDREV`* 是基础版本号.

filename.rNEWREV

该文件的内容和从服务器收到的版本相同，*NEWREV* 等于更新到的版本号（如果没有额外指定的话，就是 HEAD).

例如，Sally 修改了文件 *sandwich.txt*，但 是还没有提交. 同时，Harry 提交了同一文件的修改. 在提交前 Sally 执行了更新操作，结果产生了冲突，她选择推迟解决冲突：

```
$ svn update
Updating '.':
Conflict discovered in 'sandwich.txt'.
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options: p
C    sandwich.txt
Updated to revision 2.
Summary of conflicts:
  Text conflicts: 1
$ ls -1
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

此时，直到这三个临时文件被删除之前，Subversion 不会允许 Sally 提交 *sandwich.txt*:

```
$ svn commit -m "Add a few more things"
svn: E155015: Commit failed (details follow):
svn: E155015: Aborting commit: '/home/sally/svn-work/sandwich.txt' remains in conflict
```

如果用户选择推迟解决冲突，只有在冲突解决之后，Subversion 才 会重新允许用户提交修改，其中要用到的命令是 *svn resolve*. 该命令接受一个 `--accept` 选项，它指明了用户想要如何解决冲突. 在 Subversion 1.8 以前，`--accept` 是命令 *svn resolve* 的必填选项，但是现在它是可选的. 如果不带 `--accept` 地执行 *svn resolve*, Subversion 就会进入交互式地冲突 解决步骤，这部分内容我们已经在上一节— “交互式地解决冲突差异”一节—介绍过了. 下面我们会介绍如何使用选项 `--accept`.

选项 `--accept` 指示 Subversion 使用预先定义 好的几种方法之一来解决冲突. 如果用户想要用上一次检出时的版本，就 写成 `--accept=base`; 如果用户只想保留自己的修改，就写成 `--accept=mine-full`; 如果用户只想保留从 服务器收到的更新，就写成 `--accept=theirs-full`. 除了刚才介绍的几个，还有其他一些选项值，参考 svn 参考手册—Subversion 命令行客户端 的 `--accept` *ACTION*.

如果用户想要自己选择哪些修改进入最终版本，那就自己手动编辑 文件，修改冲突区域（带有冲突标记的区域），然后使用选项 *--accept=working* 告诉 Subversion 把文件的 当前内容作为冲突解决后的状态.

svn resolve 删除三个临时文件，将用户指定的 文件版本作为冲突解决后的最终版. 命令执行成功后 Subversion 不再认为 文件处于冲突状态：

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

手动地解决冲突

第一次尝试手动解决冲突会让不少人感到紧张，但只要多练几次，就会像 骑自行车一样简单.

这里有一个例子. 由于沟通上的误会，你和你的同事，Sally，同时修改 了 *sandwich.txt*，Sally 先提交了修改，结果当你 更新工作副本时发生了冲突，现在你需要手动编辑文件来解决冲突. 首先先 看一下发生冲突后的文件内容：

```
$ 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
```

分别由小于号，等号和大于号组成的行是冲突标记，它们不是冲突数据 的一部分，用户通常只需要确保在提交前把它们都删除掉即可. 前两个标记之 间的文本是用户的本地修改.

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

后两个标记之间的内容是 Sally 提交的修改：

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

通常情况下你不能直接删除冲突标记和 Sally 的修改— 否则的话当她收到三明治时就会感到一头雾水，此时你应该向她说明意大 利熟食店不出售泡洋白菜丝. 假设 *sandwich.txt* 修改完毕后的内容是：

```
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
Salami
Mortadella
```

```
Prosciutto
Creole Mustard
Bottom piece of bread
```

使用命令 *svn resolve* 移除文件的冲突状态后，接下来就可以提交修改了：

```
$ 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."
```

通常情况下，如果用户还没有编辑好文件就不要用 *svn resolve* 告诉 Subversion 你已经解决好了 冲突，因为临时文件一旦被删除，即使文件中还含有冲突标记，Subversion 依然会允许用户提交修改.

如果用户在编辑含有冲突的文件时感到困惑，应该看一下 Subversion 创建的那三个临时文件，甚至可以用第三方的交互式文件合并工具来查看 它们.

只使用从服务器收到的更新

如果在更新时产生了冲突，而你想要完全丢弃自己的修改，就执行 `svn resolve --accept theirs-full` *`CONFLICTED-PATH`* ，此时 Subversion 就会丢弃用户的本地修改，并 删除临时文件：

```
$ svn update
Updating '.':
Conflict discovered in 'sandwich.txt'.
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options: p
C    sandwich.txt
Updated to revision 2.
Summary of conflicts:
  Text conflicts: 1
$ 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'
$
```

使用 svn revert

如果用户决定丢弃当前的所有修改（无论是在冲突后，还是在任何时候），就 用 *svn revert*:

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

注意，含有冲突的文件被回滚后不需要再对它使用 *svn resolve*.

提交修改

终于，所有的编辑都完成了，从服务收到的更新也已合并完成，现在你 已经准备好向仓库提交修改.

svn commit 把本地的所有修改发往仓库. 提交时 用户需要输入一段日志来描述本次修改，日志被附加到新的版本号上. 如果 日志比较简短，可以用选项 `--message (-m)` 直接在命令行上输入日志:

```
$ svn commit -m "Corrected number of cheese slices."
Sending        sandwich.txt
Transmitting file data .
Committed revision 3.
```

如果用户已经事先把日志写到了某个文本文件中，希望 Subversion 在 提交时直接从该文件中读取日志，这可以通过选项 `--file (-F)` 实现:

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

如果用户在提交时没有指定选项 `--message (-m)` 或 `--file (-F)`, Subversion 就会自动打开用户指定的编辑器（见 “通用配置选项”一节 的 `editor-cmd`) 来编写日志.

如果用户在编写日志时突然又不想提交了，那就不保存地退出编辑 器; 如果已经保存过，那就删除全部的提交日志，再保存一遍，然后 退出编辑器:

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

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

仓库不知道也不关心用户的提交是否有意义，它只能确保没有人趁你不 注意时修改了同一文件. 如果确实有人这么做了，整个提交就会失败，并打印 一条错误消息说其中某些文件过时了:

```
$ svn commit -m "Add another rule"
Sending        rules.txt
Transmitting file data .
svn: E155011: Commit failed (details follow):
svn: E155011: File '/home/sally/svn-work/sandwich.txt' is out of date
…
```

（错误消息的具体内容取决于网络协议和服务器，但是基本内容都是类似 的.）

此时用户需要执行 `svn update`, 解决可能 的冲突，然后再次尝试提交.

本节介绍的内容覆盖了 Subversion 的基本工作周期. 为了方便用户 使用仓库和工作副本，Subversion 还提供了很多特性，但是在大部分情况下，Subversion 的日常使用只会用到我们目前所介绍的这些命令. 下面我们还将 会介绍几个较常用到的命令.

检查历史

Subversion 仓库就像一台时间机器，它记录了用户提交的每一次修改，允许用户查看文件和目录以前的版本，以及它们的元数据．只要一个命令，用户就可以检出仓库在以前任意一个时间点或版本号的版本（或者回滚工作 副本的版本号）．不过，有时候用户可能只是想看一下过去的历史，而不是想真正地回到过去．

下面几个命令提供了检索历史数据的功能：

svn diff

从行的级别上查看修改的内容

svn log

和版本号绑定的日志消息，及其日期，作者，以及受影响的文件 路径．

svn cat

根据给定的版本号，输出文件在该版本下的内容．

svn annotate

根据给定的版本号，查看该版本下的文件的每一行的最后一 次修改信息．

svn list

根据给定的版本号，列出仓库在该版本下的文件与目录清单．

查看历史修订的细节

我们已经介绍过 *svn diff*—按照标准差异格式 显示文件的变化，前文我们是用它显示工作副本的本地修改．

实际上，*svn diff* 有三种用法：

- 查看本地修改
- 比较工作副本和仓库
- 比较仓库的版本号

查看本地修改

如果不带选项地执行 `svn diff`，命令就会拿文件的当前内容和存放在 *.svn* 中的原始 文件作对比：

```
$ svn diff
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
```

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

比较工作副本和仓库

如果带上选项 `--revision(-r)`, 命令就把工作副本和仓库中指定的版本号作对比:

```
$ svn diff -r 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

比较仓库的版本号

如果用选项 `--revision(-r)` 传递了一对用冒号隔开的版本号, 命令就会比较这两个版本号的差异.

```
$ 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
$
```

如果要比较某个版本号与前一个版本号, 比较方便的做法是用选项 `--change(-c)`:

```
$ svn diff -c 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
```

```
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

最后，即使本地机器上没有工作副本，*svn diff* 也可以比较仓库的版本号，方法是在命令行中指定 URL:

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

生成历史修改列表

为了查看某个文件或目录的历史修改信息，使用命令 *svn log*，它显示的信息包括提交修改的作者，版本号，时间和日期，以及日志消息（如果有的话）：

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

注意，*svn log* 默认按照时间逆序来打印消息，如果用户只想查看某段范围内的日志，或者是单个版本号的日志，又或者是想 改变打印顺序，就带上选项 `--revision(-r)`:

表 2.1. 常见的日志请求

命令	描述
`svn log -r 5:19`	按照时间顺序打印从版本号 5 到 19 的日志
`svn log -r 19:5`	按照时间逆序打印从版本号 5 到 19 的日志
`svn log -r 8`	显示版本号 8 的日志

也可以只查看单个文件或目录的日志，例如：

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

…

上面的命令只会显示和 *foo.c* 相关的版本号的 日志.

为什么 svn log 没有显示我刚提交的日志?

如果在提交之后马上不带参数地执行 `svn log`, 用户就会发现输出的日志消息中没有包含最近的几次提交, 这种情况是由 *svn commit* 和 *svn log* 的 共同作用的结果. 首先, 用户向仓库提交修改时, *svn* 只更新了被提交的文件和目录的版本号, 所以父目录依然是 较旧的版本号 (原因见 “更新和提交是分开的”一节), *svn log* 默认从目录的当前版本号开始获取历史消息, 所以用户就看不到刚提交的日志. 解决办法是用 *svn update* 更新工作副本, 或通过选项 `--revision(-r)` 显式地向 *svn log* 传递一个版本号.

如果用户想在日志消息中看到更多的细节, 就带上选项 `--verbose(-v)`. 因为 Subversion 允许用户移动和复制文件或目录, 所以如果能在日志中看到文件路径的变化 就方便多了. 带上选项 `--verbose(-r)` 后, *svn log* 的输出中就会包含被修改的文件路径:

```
$ 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 还支持选项 `--quiet(-q)`, 它会阻止打印日志消息主体, 如果和 `--verbose(-v)` 一起使用, 那么 *svn log* 只会打印被修改的文件路径.

为什么 svn log 什么都没输出?

Subversion 用了一段时间后, 大多数用户可能会碰到下面这种情况:

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

乍看起来好像是 Subversion 出错了, 但是别忘了版本号是整个 仓库统一进行编号. *svn log* 针对仓库中的文件 路径进行操作, 如果用户在执行命令时没有提供路径参数, *svn log* 就把当前工作副本作为默认目标. 于是, 如果用户上在某个工作副本目录中查看某个版本号的日志, 而在这个版本 号中, 当前工作目录及其子目录与子目录都没有被修改, 所以 Subversion 什么都没打印. 如果确实想看该版本号的日志, 就把仓库的顶层 URL 作为 参数传递给 *svn log*, 例如 `svn log -r 2 ^/`.

从 Subversion 1.7 开始, 用户还可以让 *svn log* 产生标准差异格式的输出, 就像 *svn diff*. 如果给 *svn log* 加上选项 `--diff`, 用户就可以在行的级别上看到本次修订的具体修改内容, 于是, 用户可以同 时从高层的语义修改和底层的基于行的变化来查看文件的修改历史.

从 Subversion 1.8 开始，*svn log* 支持选项 `--search` 和 `--search-and`. 这两个 选项允许用户指定搜索模式字符串，从而过滤 *svn log* 的输出：只有当版本号的作者，日期，日志消息或被修改的文件路径与搜索 模式匹配时，才会输出该日志.

浏览仓库

利用 *svn cat* 和 *svn list*, 用户可以查看任意一个版本号下的文件和目录，而无须修改工作副本，实际上，在使用这两个命令时甚至都不需要工作副本.

显示文件的内容

如果用户只想查看文件旧版本的内容，可以用 *svn cat*:

```
$ svn cat -r 2 rules.txt
Be kind to others
Freedom = Chocolate Ice Cream
Everything in moderation
Chew with your mouth open
$
```

还可以把输出重定向到一个文件中：

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

显示每一行的修改属性

和 *svn cat* 比较类似的命令是 *svn annotate* 有点类似，但是 *svn annotate* 的输出更丰富—除了文件的内容外，还会输出每一行 最后一次被修改时的作者，版本号及其日期（可选）.

如果参数是工作副本中的文件，*svn annotate* 会根据文件的当前内容输出每一行的属性：

```
$ svn annotate rules.txt
     1      harry Be kind to others
     3      sally Freedom = Responsibility
     1      harry Everything in moderation
     -          - Chew with your mouth closed
     -          - Listen when others are speaking
```

在上面的例子里，某些行的属性没有打印出来，原因是这几行在工作 副本中被修改了. 利用这个特点，我们也可以通过 *svn annotate* 判断出文件的哪些行被修改了. 用户可以用版本号关键词 BASE（见 “版本号关键字”一节）查看文件的未修改版本的输出：

```
$ svn annotate rules.txt@BASE
     1      harry Be kind to others
     3      sally Freedom = Responsibility
     1      harry Everything in moderation
     1      harry Chew with your mouth open
```

选项 `--verbose` (`-v`) 使得 *svn annotate* 在输出中增加每一行的版本号的 提交日期（这会显著增加输出内容的宽度，所以我们不在这里展示添加 了选项 `--verbose` 后的运行效果）.

和 *svn cat* 一样，*svn annotate* 也能针对文件的旧版本进行操作，这个功能有时候会很有 帮助—如果用户已经找到了文件中某一行的最后一次修改的版本号，他可能还想知道在此之前是谁最后一次修改了这一行：

```
$ svn blame rules.txt -r 2
     1      harry Be kind to others
     1      harry Freedom = Chocolate Ice Cream
     1      harry Everything in moderation
     1      harry Chew with your mouth open
```

和 *svn cat* 不同的是，*svn annotate* 的正常运行要求文件必须是人类可读的，以行为单位的文本 文件，如果 Subversion 认为文件不是人类可读的（根据文件的 `svn:mime-type` 属性—见 “文件内容类型”一节），*svn annotate* 就输出一条错误消息：

```
$ svn annotate images/logo.png
Skipping binary file (use --force to treat as text): 'images/logo.png'
$
```

就像错误消息中提示的那样，用户可以通过增加选项 `--force` 来禁止 Subversion 去检查文件是否是人类 可读的．如果用户强制要求 *svn annotate* 去读取 非人类可读的文件，命令就会输出一堆混乱的信息：

```
$ svn annotate images/logo.png --force
     6      harry \211PNG
     6      harry ^Z
     6      harry
     7      harry \274\361\MI\300\365\353^X\300…
```

根据执行命令时的心情，用户可能不会使用标准形式 (*svn annotate*)，转而使用 *svn blame* 或 *svn praise*，后面两种形式是 标准形式的别名，它们是完全等价的．

和许多获取信息的命令一样，*svn annotate* 也接受仓库的 URL 作为参数，这样即使没有工作副本，用户也可以照常 执行命令．

列出被版本控制的文件

命令 *svn list* 可以列出仓库目录中的文件，而 不用把它们下载到本地：

```
$ svn list http://svn.example.com/repo/project
README
branches/
tags/
trunk/
```

如果想得到更详细的信息，添加选项 `--verbose` (`-v`)：

```
$ svn list -v http://svn.example.com/repo/project
  23351 sally                Feb 05 13:26 ./
  20620 harry             1084 Jul 13  2006 README
  23339 harry                Feb 04 01:40 branches/
  23198 harry                Jan 23 17:17 tags/
  23351 sally                Feb 05 13:26 trunk/
```

从左到右分别表示文件或目录最后一次被修改时的版本号，作者，文件大小（仅针对文件），日期以及文件或目录的名字．

不带参数的 *svn list* 默认使用当前工作 目录对应的仓库 URL 作为参数，而非本地的工作副本目录. 毕竟，如果用户想要列出本地目录中的文件，应该使用 Shell 命令 *ls* (或其他非 Unix 系统的等价命令).

获取老的仓库快照

用户可以用带有选项 `--revision (-r)` 的 *svn update* 命令，把整个工作副本回退到旧版本: [2]

```
# Make the current directory look like it did in r1729.
$ svn update -r 1729
Updating '.':
…
$
```

许多 Subversion 新手会尝试利用上面的 *svn update* 例子来 “撤消” (undo) 已经提交的修改，但是这不可能 实现，因为如果被修改的文件的版本号比较旧，那就无法成功提交. 关于如何 “撤消” 提交请参考 “恢复已删除的文件”一节

如果用户想以较老的快照为基础创建一个新的工作副本，只要稍微修改 一下 *svn checkout* 的命令行即可；对于 *svn update*，可以给它添加一个选项 `--revision (-r)`. 由于 “限定版本号与实施版本号”一节 介绍的原因，用户可能想 把目标版本号作为 Subversion 扩展 URL 语法的一部分.

```
# Checkout the trunk from r1729.
$ svn checkout http://svn.example.com/svn/repo/trunk@1729 trunk-1729
…
# Checkout the current trunk as it looked in r1729.
$ svn checkout http://svn.example.com/svn/repo/trunk -r 1729 trunk-1729
…
$
```

如果用户想构建一个发布版，其中包含了所有的被版本控制的文件和 目录，命令 *svn export* 可以完成这项工作. *svn export* 在本地创建一份仓库的完整或部分副本，但是没有 *.svn* 目录. 命令的基本语法和 *svn checkout* 相同:

```
# Export the trunk from the latest revision.
$ svn export http://svn.example.com/svn/repo/trunk trunk-export
…
# Export the trunk from r1729.
$ svn export http://svn.example.com/svn/repo/trunk@1729 trunk-1729
…
# Export the current trunk as it looked in r1729.
$ svn export http://svn.example.com/svn/repo/trunk -r 1729 trunk-1729
…
$
```

有时候你需要的只是清理一下

既然我们已经讲到了使用 Subversion 时经常碰到的日常工作，现在将介绍 一些和工作副本相关的管理性任务.

[2] 看到了吗? 我们早就告诉过你 Subversion 是一台时间机器.

删除工作副本

服务端的 Subversion 不会跟踪工作副本的状态或存在情况，所以工作 副本不会影响服务器的工作负载．同样，删除工作副本时也不需要告诉服务 器．

如果用户下次可能还要用到工作副本，那么在下次使用之前，直接把工作 副本留在磁盘上也不会产生什么问题，不过在开始使用之前，记得用 *svn update* 更新一下工作副本．

然而，如果用户已经确定自己以后不会再用到工作副本，为了节省磁盘 空间，你也可以用操作系统提供的删除命令把工作副本删除掉．但是在删除 之前我们建议执行一下 *svn status*，然后查看带有前缀 ? 的列表中是否有重要的文件．

从中断中恢复

当 Subversion 修改工作副本时—修改文件或文件的管理状态— 它会尽量保证操作能够安全地执行．在修改工作副本之前，Subversion 把 它的意向操作记在一个私有的 “待完成列表” (to-do list) 中，然后开始执行操作，在执行过程中 Subversion 会去获取工作副本中相 关部分的锁，这可以避免其他客户端在工作副本处于中间状态时对它进行 访问，最后，Subversion 释放锁并清理待完成列表．从结构上来看，它有点 像日志文件系统．如果 Subversion 的一个操作被中断了（例如进程被杀死或 机器崩溃），待完成列表将保留在磁盘上，这就允许 Subversion 后面可以再 打开列表，做完未完成的工作，把工作副本恢复到一致的状态．

上面介绍的正是 *svn cleanup* 的功能：*svn cleanup* 在工作副本中搜索未完成的工作，操作完成 时移除工作副本的锁．如果 Subversion 告诉你工作副本中的某些部分是被 “锁住” 的，执行 *svn cleanup* 就可以解决 该问题．*svn status* 也会显示工作副本的加锁状态，被加锁的路径其左边有一字符 L：

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

不要把工作副本的管理锁和用户创建的锁相混淆，后者是为了实现 并发版本控制的 加锁-修改-解锁 模型，见 “锁” 的多种涵义．

处理目录冲突

到目前为止我们只在文件内容的级别上讨论冲突，如果你和你的同事在 同一文件上的修改相互重叠，那么 Subversion 就会要求你在合并了这些修改 之后才能提交．[3]

如果其他人把你正在编辑的文件移动到其他地方或删除了，那这时候又会发生 什么事？发生这种事的原因可能是同事之间沟通不及时，一个人认为文件应该 被删除，而另一个人还想接着修改该文件，也可能是你的同事想重新规划目录 布局．如果你正在编辑的文件已经移动到了其他位置，那么这些修改可能需要 应用到移动后的文件中．这种冲突的级别是在目录树结构上，而不是在文件的 内容上，称为 目录冲突 (*tree conflicts*)．

[3] 当然，你 可以 把仍然含有冲突标记的文件标记为已解决并提交它们，但是现实中几乎不 会有人这么做．

Subversion 1.6 以前的目录冲突

在 Subversion 1.6 以前的版本中，目录冲突会产生意想不到的结果. 例如，如果一个文件在本地被修改了，但是在仓库中被重命名了. 执行 *svn update* 时 Subversion 会执行以下步骤:

- 检查将要被重命名的文件.
- 删除重命名前的文件，如果文件在本地被修改了，就保留在 磁盘上，但不再是仓库或工作副本的一部分.
- 添加重命名后的文件.

出现这种情况时，用户可能还没有意识到有些修改被留在了未被跟踪的 文件上，如果这时候向服务器提交，这些修改就不会被提交到仓库中. 如果 文件很多，出现这种情况的概率就会增大，处理起来也会相当枯燥.

从 Subversion 1.6 开始，这种以及其他类似的情况都会被当作冲突 看待.

和文件内容的冲突一样，只有在目录冲突解决之后才能向仓库提交修改.

目录冲突示例

假设有一个软件项目的代码目录结构如下所示:

```
$ svn list -Rv svn://svn.example.com/trunk/
     13 harry                 Sep 06 10:34 ./
     13 harry              27 Sep 06 10:34 COPYING
     13 harry              41 Sep 06 10:32 Makefile
     13 harry              53 Sep 06 10:34 README
     13 harry                 Sep 06 10:32 code/
     13 harry              54 Sep 06 10:32 code/bar.c
     13 harry             130 Sep 06 10:32 code/foo.c
$
```

后来，在版本号 14, 你的同事 Harry 把 *bar.c* 重命名为 *baz.c*, 但是你并不知情. 此时你正忙于 编写另外一套修改，其中就牵涉到 *bar.c*:

```
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c (revision 13)
+++ 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 13)
```

```
+++ 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";
 }
$
```

提交失败时你开始意识到有人已经修改了 *bar.c*:

```
$ svn commit -m "Small fixes"
Sending        code/bar.c
Transmitting file data .
svn: E155011: Commit failed (details follow):
svn: E155011: File '/home/svn/project/code/bar.c' is out of date
svn: E160013: File not found: transaction '14-e', path '/code/bar.c'
$
```

此时应该执行 *svn update*, 命令不仅把 Harry 的修改同步到本地工作副本, 还产生了一个目录冲突:

```
$ svn update
Updating '.':
   C code/bar.c
A    code/baz.c
U    Makefile
Updated to revision 14.
Summary of conflicts:
  Tree conflicts: 1
$
```

在上面的例子中, *svn update* 在第四列放置一个 大写字母 C 表示该条目有冲突. *svn status* 可以显示冲突的其他细节:

```
$ svn status
M       code/foo.c
A  +  C code/bar.c
      >   local edit, incoming delete upon update
Summary of conflicts:
  Tree conflicts: 1
$
```

注意 *bar.c* 如何又被自动地添加到工作副本中, 如果用户想保留 *bar.c*, 就不需要再额外执行一次 *svn add*.

由于 Subversion 是用一个复制操作和一个删除操作实现移动, 而且在 更新时很难将这两个操作联系在一起, 所以 Subversion 的警告信息只是说 在本地被修改的文件已经在仓库中被删除了, 这个删除可能是移动操作的一 部分, 也可能就是一次单纯的删除操作. 准确地判断仓库在语义上发生了什 么变化显得尤为重要—只有这样才能让自己的修改适应项目的整体 轨迹. 为了弄清楚冲突发生的原因, 你可以阅读日志, 和同事沟通, 在行的 级别上查看修改等.

在这个例子里, Harry 的提交日志提供了所需要的信息.

```
$ svn log -r14 ^/trunk
------------------------------------------------------------------------
```

```
r14 | harry | 2011-09-06 10:38:17 -0400 (Tue, 06 Sep 2011) | 1 line
Changed paths:
   M /Makefile
   D /code/bar.c
   A /code/baz.c (from /code/bar.c:13)

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

svn info 显示了冲突条目的 URL. *左边* (*left*) 的 URL 显示了 冲突的本地端来源, *右边* (*right*) 的 URL 显示了冲突的服务器端来源, 这些 URL 指出了我们应该从哪个版本号开始搜索导致冲突的修改.

```
$ svn info code/bar.c
Path: code/bar.c
Name: bar.c
URL: http://svn.example.com/svn/repo/trunk/code/bar.c
…
Tree conflict: local edit, incoming delete upon update
  Source  left: (file) ^/trunk/code/bar.c@4
  Source right: (none) ^/trunk/code/bar.c@5

$
```

bar.c 已经成为目录冲突的受害者, 在冲突解决 之前无法提交:

```
$ svn commit -m "Small fixes"
svn: E155015: Commit failed (details follow):
svn: E155015: Aborting commit: '/home/svn/project/code/bar.c' remains in confl
ict
$
```

为了解决这个冲突, 用户要么同意, 要么不同意 Harry 提交的重命名 修改.

如果用户同意重命名, 那么 *bar.c* 就成了多余的 了, 你可能想要删除 *bar.c* 并把目录冲突标记为已 解决, 但是请等一下, 文件上还有你的修改! 在删除 *bar.c* 之前你必须决定它上面的修改是否需要应用到 其他地方, 比如重命名后的文件 *baz.c*. 不妨假设你 的修改需要 “跟随重命名” (follow the move), 但是 Subversion 还没有聪明到能够替你完成这件工作[4], 所以你必须手动地迁移修改.

在我们的例子里, 你完全可以手动地再修改一次 *baz.c*—毕竟只修改了一行, 但是这种做法只适用 于修改很少的情况, 我们再介绍一种更具有通用性的方法. 先用 *svn diff* 创建一个补丁文件, 然后修改补丁文件的头 部信息, 使其指向重命名后的文件, 最后再应用修改后的补丁.

```
$ svn diff code/bar.c > PATCHFILE
$ cat PATCHFILE
Index: code/bar.c
===================================================================
--- code/bar.c (revision 14)
```

[4]在某些情况下, Subversion 1.5 和 1.6 会 替你完成这件事, 但 这种有点随意的功能已经在 Subversion 1.7 被移除了.

```
+++ 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";
 }
$ ### Edit PATCHFILE to refer to code/baz.c instead of code/bar.c
$ cat PATCHFILE
Index: code/baz.c
===================================================================
--- code/baz.c (revision 14)
+++ 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";
 }
$ svn patch PATCHFILE
U         code/baz.c
$
```

现在 *bar.c* 上的修改已经成功地转移到了 *baz.c* 上, 用户现在可以删除 *bar.c* 并告诉 Subversion 把工作副本的当前内容 作为冲突解决的结果.

```
$ svn delete --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 14)
+++ 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 14)
```

```
+++ 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";
 }
$
```

但是如果你不同意重命名, 那又该如何? 如果用户已经确定 *baz.c* 上的修改已经进行了保存或者可以丢弃, 那也可以 直接删除 *baz.c* (别忘了撤消 Harry 对 *Makefile* 的修改). 因为 *bar.c* 已经准备好添加到仓库中, 所以接下来只需 要把冲突标记为已解决即可:

```
$ svn delete --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 14)
+++ 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 14)
+++ 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 14)
+++ code/baz.c (working copy)
```

```
@@ -1,4 +0,0 @@
-const char *bar(void)
-{
-    return "Me neither!\n";
-}
Index: Makefile
===================================================================
--- Makefile (revision 14)
+++ Makefile (working copy)
@@ -1,2 +1,2 @@
 foo:
- $(CC) -o $@ code/foo.c code/baz.c
+ $(CC) -o $@ code/foo.c code/bar.c
```

恭喜，你已经解决了你的第一个目录冲突！现在你可以提交修改，并告诉 Harry 由于他的修改，你做了很多额外的工作.

小结

本章我们介绍了 Subversion 客户端的大部分命令，剩下的命令中比较 重要的几个主要用于分支与合并（见 第 4 章 分支与合并），以及属性（见 “属性”一节）. 然而，你可能想 快速浏览一下 svn 参考手册—Subversion 命令行客户端，看看 Subversion 到底提供 了多少个命令，这些命令又如何帮助你提高工作效率.

第 3 章 高级主题

如果读者是从头开始，一章一章地阅读本书，那么你应该拥有了足够的知识 去使用 Subversion 客户端工具完成最常见的版本控制操作．你已经知道了如何 从 Subversion 仓库检出工作副本，如何用 *svn commit* 和 *svn update* 提交和接收修改，甚至运行 *svn status* 已经成为了你的下意识动作．总之，你已经准备 好在一个典型的应用环境中使用 Subversion.

但是 Subversion 远远不止 “常见的版本控制操作”，除了 和中央仓库沟通文件和目录的变化外，它还具备很多功能.

本章将要介绍的 Subversion 特性， 用户在自己的日常工作中可能不会用到， 但是却很重要． 本章假设读者已经熟悉了 Subversion 基本的文件与目录的版本 控制功能，如果读者还不了解这方面的内容，先阅读 第 1 章 基本概念 和 第 2 章 基本用法 这两章．一旦 读者消耗了本章的内容，你将会成为一位强大的 Subversion 用户.

版本号指示器

我们已经在 “版本号”一节 说过，Subversion 的版本号非常直观—随着提交的不断增多，表示版本号的整数 也不断增大，但是用不了多久，用户就再也记不清每个版本号包含了哪些修改．幸运的是，Subversion 的典型工作流程不太经常要求用户提供版本号，对于 那些确实需要版本号的操作而言，用户可以从提交日志中看到所需的版本号，或者 使用在特定语境下可以表示特定版本号的关键字.

在引用版本号时，在数字的左边增加一个前缀 “r”（例如 `r314`）是 Subversion 社区约定俗成的做法，很多 Subversion 的相关工具都鼓励 这种写法．在需要提供裸版本号的大多数场合中，你也可以使用 *rNNN* 形式的语法.

但是在少数情况下，用户必须及时精确地描述出版本号，可是手上却没有合适的参数．所以除了用整数指定版本号外，*svn* 还支持另外两种指定版本 号的形式：版本号关键字 (*revision keywords*) 和版本号日期.

在指定版本号范围时，可以混合使用不同形式的版本号指示器，例如你可 以把 `-r`*REV1*`:`*REV2* 中的 *REV1* 写成 版本号关键字，把 *REV2* 写成整数，或者把 *REV1* 写成日期，把 *REV2* 写成版本号关键字，如此等等．每个版本号指示器被单独求值，所以你可以把任意形式的版本号指示器放在冒号的两边.

版本号关键字

Subversion 支持理解的版本号关键字有很多个，可以用这些关键字替换 选项 `--revision` (`-r`) 后面的整数，这些关键字会被 Subversion 解释成特定的版本号：

`HEAD`

仓库中最近的（或最年轻的）版本号.

`BASE`

工作副本中的某一项目的版本号，如果该项在本地被修改了，则该 版本号引用的是修改前的项目.

`COMMITTED`

等于或早于 BASE 并且离它最近的一个版本号，在该版本号中项目被修改了.

PREV

项目最后一次被修改时的版本号的前一个版本号，从技术上讲它 就是 COMMITTED-1.

从它们的描述可以看出，PREV, BASE 和 COMMITTED 只能引用工作 副本中的路径，而 HEAD 既可以引用工作副本中的路径，也可以引用仓库的 URL.

下面是一些版本号关键字的使用示例:

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

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

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

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

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

$ svn update -r PREV foo.c
# rewinds the last change on foo.c, decreasing foo.c's working revision

$ svn diff -r BASE:14 foo.c
# compares the unmodified version of foo.c with the way foo.c looked
# in revision 14
```

版本号日期

版本号没有透露出一丝一毫与版本控制系统外部世界相关的信息， 但是有 时候你需要把真实世界的时间与版本控制历史的时间联系起来. 为此，选项 --revision(-r) 也接受日期形式的 参数，日期用一对花括号 ({ 和 }) 包裹起来，Subversion 接受标准的 ISO-8601 格式的日期与时间，以及其他 一些形式，下面是一些例子.

```
$ svn update -r {2006-02-17}
$ svn update -r {15:30}
$ svn update -r {15:30:00.200000}
$ svn update -r {"2006-02-17 15:30"}
$ svn update -r {"2006-02-17 15:30 +0230"}
$ svn update -r {2006-02-17T15:30}
```

```
$ svn update -r {2006-02-17T15:30Z}
$ svn update -r {2006-02-17T15:30-04:00}
$ svn update -r {20060217T1530}
$ svn update -r {20060217T1530Z}
$ svn update -r {20060217T1530-0500}
…
```

如果版本号日期参数中含有空格，那么大多数 shell 都会要求用引号 包围参数，或者对参数中的空格进行转义，某些 shell 可能还会要求对花 括号进行转义，具体的细节可以查阅你所用的 shell 的文档.

如果用户指定了一个日期，Subversion 就把该日期解析成最近的版本号，然后再针对该版本号进行操作：

```
$ svn log -r {2006-11-28}
------------------------------------------------------------------------
r12 | ira | 2006-11-27 12:31:51 -0600 (Mon, 27 Nov 2006) | 6 lines
…
```

是 Subversion 早了一天吗

如果用户指定了一个日期作为版本号，但没有指定一天中的时间（例如 `2006-11-27`），你可能希望 Subversion 得到的是 11 月 27 号这一天的最后一个版本号，可事实上是 Subversion 得到的是 26 号或者更早的版本号. 记住 Subversion 要找的是离那天 最近 的仓库的版本号，如果没有指定时间，例如 `2006-11-27`，Subversion 默认时间是 `00:00:00`，所以不会出现 27 号那天的版本号.

如果你希望 27 号的版本号也在搜索的范围里，那就在参数中指定时间（`{"2006-11-27 23:59"}`），或者干脆指定后一天（`{2006-11-28}`）.

还可以指定一段日期范围，此时 Subversion 会找到这段时间内的所有 版本号，包括开始日期和结束日期：

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

Subversion 能够根据版本号日期得到版本号是因为版本号的时间戳被维护 成一个串行化的序列—版本号越年轻，时间戳也就越年轻. 但是时间戳 是存放在版本号的 `svn:date` 属性内（见 “属性”一节），而该属性是可修改的，且未被纳 入版本管理，所以说打乱版本号的时间戳是有可能的. 如果时间戳被打乱了，大多数操作都不会受到影响，毕竟它们都是使用版本号的整数编号作为版本 号的标识，但是当你用日期指定版本号范围时，Subversion 返回的数据很可 能不是你所想要的. 导致版本号时间戳不再有序的常见操作是把多个仓库的 历史组合成单独的一个.

限定版本号与实施版本号

我们经常在自己的系统中对文件和目录进行复制，移动，重命名和替换，但是版本控制系统不能按照相同的方式操作它所管理的文件与目录. Subversion 的文件管理非常灵活，几乎和操作未版本化的文化一样灵活. 但是这种灵活性也意味着在仓库的生命周期中，一个版本 控制对象可能会有多个路径，而一个路径在不同时间可能表示完全不同的版本控制 对象，当用户同这些路径与对象交互时会产生一定的复杂度.

如果对象的版本历史里包含了″地址上的变化″, Subversion 自己就会注意 到这点. 比如说用户要求查看上周被重命名的一个文件的版本历史, Subversion 会提供全部的相关日志—重命名发生时的版本号, 再加上重命名前与重命名后 的相关版本号. 所以说在大部分情况下, 用户都不需要考虑对象的地址变化可能 带来的影响, 但是在少数情况下, Subversion 需要你的帮助来消除歧义.

最简单的一种场景是一个文件或目录从仓库中被删除后, 又有一个同名的 文件或目录被添加到仓库中, 被删除的对象和新增的对象之间毫无关系, 只是 碰巧路径相同, 假设都是 */trunk/object*, 那么向 Subversion 询问 */trunk/object* 的历史是表示什么 意思? 是在问当前对象的历史, 还是那个被删除的对象的历史? 或者是在问 该路径上存在过的 *所有* 对象的操作历史? 为了得到 自己想要的信息, Subversion 需要一些提示.

由于移动操作, 对象的历史变得更加复杂. 比如说你有一个目录叫作 *concept*, 它包含了几个初期的软件项目. 慢慢地, 软件开始成型, 你开始考虑为项目取一个名字[1]. 假设你要取的软件名字是 Frabnaggilywort, 把目录重命名成 软件的名字是很合理的操作, 于是 *concept* 被重命名 为 *frabnaggilywort*. 项目接着进行, Frabnaggilywort 发布了 1.0 版, 很多用户都下载了并在日常工作中使用它.

故事听起来还不错, 但是还没结束. 企业家的脑子里经常会有新想法出现, 于是你又创建了一个新目录 *concept*, 循环再次开始. 实际上, 在几年内循环会重复进行多次, 每一次都以创建 *concept* 开始, 如果想法逐渐地明朗起来, *concept* 很可能会被重新命名; 如果想法被否定了, *concept* 就会被删除. 更有甚者, 用户还有可能把 *concept* 改名一段时间后, 又改回到 *concept*.

在这种场景下, 指挥 Subversion 操作这些重复使用的路径就好像在指挥 一个摩托车手, 从芝加哥的 West Suburbs 向东行驶到 Roosevelt Road, 再向 左驶入主街. 在短短的 20 分钟里, 你会穿过 Wheaton, Glen Ellyn 和 Lombard 的 “主街”, 但它们并非是同一个地方, 我们的 摩托车手—也就是 Subversion—需要更多的细节才能把事情做对.

幸运的是, Subversion 允许用户精确地指定他想去的是哪一个主街, 其中 用到的特性是 限定版本号 (*peg revision*), 它的目的是确定一条唯一的历史线. 因为在任意一个给定的时刻（或者说给定的版本号）一条路径上至多只能有一个版 本控制对象, 所以说结合使用路径与限定版本号就可以明确地识别一条特定的 历史线. 限定版本号使用 *at 语法* (*at syntax*) 在 Subversion 的命令行客户端工具上 指定, 之所以叫作 “at 语法” 是因为指定版本号的方式是在路径 的末尾加上符号 `@`, 然后再写上版本号.

但是本书多次提到的 `--revision (-r)` 到底 是什么? 这个版本号或版本号集合叫作 *实施版本号* (*operative revision*) 或 *实施版本号范围* (*operative revision range*). 一旦用路径和限定版本号确定一条特定的历史线, Subversion 就 对实施版本号执行用户请求的操作. 用芝加哥的道路进行类比, 如果我们要去 Wheaton 的 606 N. 主街[2], 可以把 “主街” 看成路径, 把 “Wheaton” 看成限定 版本号, 这两项信息确定了一条唯一的路径, 避免我们走弯路. 现在我们把 “606 N.” 作为实施版本号, 最终我们得到了一个精确的目的地.

[1] “ 你不应该给它取名字, 因为一旦取了名字, 你就开始喜欢它了. ”—Mike Wazowski

[2] 把 606 N. 主街作为 Wheaton 的历史中心应该是比较恰当的.

限定版本号算法

提供给客户端命令行工具的路径和版本号参数如果可能含有歧义，Subversion 就会运行限定版本号算法来消除歧义，下面是一个用来说明的 示例：

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

如果 *OPERATIVE-REV* 比 *PEG-REV* 老，算法的执行过程是：

1. 定位由版本号 *PEG-REV* 识别的 *item*，有且仅有一个对象.
2. 反向追踪对象的历史（还要考虑重命名操作带来的影响），直到版本号 *OPERATIVE-REV* 里的祖先.
3. 对祖先执行用户所请求的操作，无论当时这个祖先位于何处，叫什么 名字.

但是如果 *OPERATIVE-REV* 比 *PEG-REV* 年轻 的话又会如何？这会给定位 *OPERATIVE-REV* 中的路径的理论 问题增加一些复杂度，因为在 *PEG-REV* 和 *OPERATIVE-REV* 之间，路径的历史可能会多 次发生分叉（由于复制操作）. 不仅如此，Subversion 不会为了高效地正向 追踪对象的历史而记录足够多的信息. 所以在这种情况下的算法会有一些差别：

1. 定位由版本号 *OPERATIVE-REV* 识别的 *item*，有且仅有一个对象.
2. 反向追踪对象的历史（还要考虑重命名操作带来的影响），直到版本号 *PEG-REV* 里的祖先.
3. 检查对象在 *PEG-REV* 和 *OPERATIVE-REV* 中的位置（路径）是否相同，如果是，说明至少这两个位置是直接相关的，那就在 *OPERATIVE-REV* 的位置上执行用户所请求的操作. 否则的话相关性无法 建立，输出错误信息，表示无法找到可用的路径（也许某一天 Subversion 对这种情况会处理得更加灵活与优雅）.

注意，即使用户没有显式地给出限定版本号或实施版本号，它们仍然存在. 为了方便用户，工作副本里的项目的限定版本号默认是 BASE ，仓库 URL 默认是 HEAD. 如果没有显式 指定实施版本号，则默认与限定版本号相同.

比如说用户在很久以前就创建了仓库，在版本号 1 添加了第一个目录 *concept*，用户后来在目录里放了一个介绍概念的文件 *IDEA*. 几次提交后，项目的代码逐渐成型，在版本号 20 用户把 *concept* 重命名为 *frabnaggilywort* . 在版本号 27，用户又有了一个新主意，所以在项目根目录下又 创建了目录 *concept*，里面也放了一个描述概念的文件 *IDEA*. 然后又过了 5 年，期间提交了几千次修改.

几年后，用户想知道文件 *IDEA* 在版本号 1 中是什么 样子，但是 Subversion 需要知道用户是在询问 当前 文件 在版本号 1 时的内容，还是在问版本号 1 中文件 *concept/IDEA* 的内容. 当然这两个问题的答案是不一样的，利用限定版本号，用户 就可以向 Subversion 说明他想问的是哪一个问题. 为了确定当前的 *IDEA* 在版本号 1 时的内容，用户执行了：

```
$ svn cat -r 1 concept/IDEA
svn: E195012: Unable to find repository location for 'concept/IDEA' in revision 1
```

当然，在这个例子里，当前的文件 *IDEA* 在版本号 1 时并不存在，于是 Subversion 报了一个错误. 上面的命令实际上是以下显式 指定持勾版本号命令的简写形式：

```
$ svn cat -r 1 concept/IDEA@BASE
svn: E195012: Unable to find repository location for 'concept/IDEA' in revision 1
```

命令的执行结果是预料之中的.

敏锐的读者可能想知道是否是限定版本号的语法导致了问题，因为工作副本 路径或 URL 本身可能就带有符号 @，毕竟 *svn* 怎么知道 `news@11` 是表示一个目录的普通名字，还是表示 “*news* 的版本号 11”？谢天谢地，*svn* 总是当成后一种情况，方法是在路径的末尾添加一 个 @ 符号，例如 `news@11@`. *svn* 只关心参数中的最后一个 @，即使省略了 @ 后面的版本号也是合法的. 这个方法也适用 于以 @ 结尾的路径—你可以用 `filename@@` 表示一个名为 `filename@` 的文件.

再考虑另一个问题—在版本号 1 中，占用路径 *concept/IDEA* 的文件的内容是什么？我们可以用一个 带有显式限定版本号的命令来回答这个问题.

```
$ svn cat concept/IDEA@1
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

注意在上面的命令中我们并没有提供实施版本号，这是因为如果没有指定 实施版本号，Subversion 默认使用限定版本号作为实施版本号.

命令的执行结果看来是正确的， 输出的文本甚至提到了 “frab a naggily wort”， 所以它描述的软件应该就是现在的 Frabnaggilywort，实际上我们还可以通过组合显式的限定版本号和显式的实施版本号来验证这一点. 我们已经知道在 `HEAD` 里，项目 Frabnaggilywort 位于目录 *frabnaggilywort*，于是我们希望看到 `HEAD` 的 *frabnaggilywort/IDEA* 在版本号 1 中 的内容.

```
$ svn cat -r 1 frabnaggilywort/IDEA@HEAD
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

限定版本号和实施版本号也不需要如此琐碎，举例来说， *frabnaggilywort* 已经从 `HEAD` 删除，但是我们知道它在版本号 20 时还是存在的，而且我们想知道其中存放的 *IDEA* 在版本号 4 和版本号 10 之间的差异，可以使用 限定版本号 20，结合上文件 *IDEA* 的版本号 20 的 URL，然后使用 4 和 10 作为实施版本号范围.

```
$ svn diff -r 4:10 http://svn.red-bean.com/projects/frabnaggilywort/IDEA@20
Index: frabnaggilywort/IDEA
===================================================================
--- frabnaggilywort/IDEA (revision 4)
+++ frabnaggilywort/IDEA (revision 10)
@@ -1,5 +1,5 @@
-The idea behind this project is to come up with a piece of software
-that can frab a naggily wort.  Frabbing naggily worts is tricky
-business, and doing it incorrectly can have serious ramifications, so
-we need to employ over-the-top input validation and data verification
-mechanisms.
+The idea behind this project is to come up with a piece of
+client-server software that can remotely frab a naggily wort.
+Frabbing naggily worts is tricky business, and doing it incorrectly
+can have serious ramifications, so we need to employ over-the-top
```

```
+input validation and data verification mechanisms.
```

幸运的是，大多数人都不会碰到这么复杂的情况，但是如果遇到了，记住 限定版本号可以帮助 Subversion 消除歧义.

属性

我们已经详细地描述了 Subversion 如何存放和检索存放在仓库中的不同 版本的文件和目录，介绍这些最基本的功能用了一整章的篇幅. 如果 Subversion 对版本控制的支持就到此为止，从版本控制的角度来看它的功能已经很完整了.

但 Subversion 并没有停下脚步.

作为目录和文件版本控制的补充，Subversion 提供了为每一个文件和目录添加，修 改和删除版本化元数据的接口. 我们把这些元数据称为 *属性* (*properties*)，属性可看作是一张两列的表格，附加到 工作副本的每个项目上，表格把属性的名字映射到任意值. 一般来说，属性的名字 和值可以是任意的，唯一的要求是属性名只能使用 ASCII 字符. 属性最好的地方 是它们也是被版本控制的对象，就像文件的内容那样，用户可以修改，提交和撤销 属性的修改. 用户执行提交和更新操作时，属性的修改也会被发送和接收— 用户的工作流程不会因为属性的加入而发生变化.

Subversion 保留了一组名字以 `svn:` 开始的属性，所以你在创建自己的属性时应该避免使用以 `svn:` 开始 的名字，否则的话，未来 Subversion 的新版本可能会采用同名的属性来 支持新特性，而属性的含义可能与你创建时的完全不同.

除了文件和目录，属性还可以出现在其他地方，每一个版本号都是一个实体，可以在它上面附加任意的属性，唯一的要求是属性名只能使用 ASCII 字符. 同文件和目录的属性相比，最大的不同是版本号的属性不会被版本控制，也就是 说如果版本号的属性被删除或修改了，Subversion 没有能力把它们恢复到以前 的值.

关于属性的使用， Subversion 并没有很特别的策略， 唯一的要求是用户不 要使用以 `svn:` 开始的属性名， 这是保留给 Subversion 使用的名字空间，Subversion 使用的属性包括版本化的和未版本化的. 文件和 目录上特定的版本化属性具有特殊的意义或效果，或提供了版本号的一些信息. 在提交时，特定的版本号属性被自动地附加到版本号上，属性包含了与版本号 有关的信息. 大多数属性会在谈到相关的主题时再介绍，Subversion 的预定义 属性的完整列表见 “Subversion 的保留属性”一节.

虽然 Subversion 自动地把属性 (`svn:date`, `svn:author`, `svn:log` 等) 附加到 版本号，但它并 不 假设在这之后这些属性仍然存在，用户及其使用的工具也应如此. 版本号的属性可以通过 API 或客户端工具删除 (如果仓库的钩子允许)，删除后并不影响 Subversion 的正常功能. 所以在 编写与 Subversion 仓库交互的脚本时，不能假定任意版本号的属性是存在的.

本节将介绍 *svn*—不仅是对 Subversion 用户，也对 Subversion 自身—对属性的支持. 读者将会学到与属性相关的 *svn* 子命令，以及属性如何影响用户的工作检验.

为什么需要属性?

Subversion 使用属性存放和文件，目录，版本号相关的额外信息，读者 可能也会发现属性的类似用法. 你会发现，如果在数据附近能有个地方保存 自定义元数据将会是一项非常有用的特性.

假设你想要设计一个网站，其中存放了很多数字照片，在显示时会给照片 加上标题和日期. 因为你的照片经常发生变化，所以你希望网站能够尽量地自动 处理由于照片变动而产生的影响. 照片可以很大，你希望在网站上可以显示照片 的缩略图.

你可以用传统的文件实现缩略图，也就是说你可以把照片 *image123.jpg* 及其缩略图 *image123-thumbnail.jpg* 放在同一个目录里．如果你 希望照片及其缩略图能使用相同的文件名，也可以把缩略图放在不同的目录里，例如 *thumbnails/image123.jpg*. 你可以按照类似的方 法存放标题和日期．这里最大的问题是每增加一个新图片，网站的文件数量都 会成倍地增加．

现在考虑如果利用 Subversion 的文件属性来部署网站． 设想有一个图片 文件 *image123.jpg*， 带有属性 `caption`, `datestamp` 和 `thumbnail`. 使用属性后的工作副本看起来更容量管理 —实际上，普通的浏览器只能看到图片文件，但是你的自动化管理脚本 可以知道得更多．脚本可以使用 *svn* (更好的做法是用 Subversion 的语言绑定—见 “使用 API”一节) 获取图片的属性信息，而不必读取索引文件或处理路径．

Subversion 对属性的名字和值有一些限制，如果属性的值很大，或者在 单个的文件或目录上设置了很多的属性，对于这两种情况 Subversion 处理 起来非常笨拙．Subversion 通常会把单个项目的所有属性及其值同时加载到内存中，如果属性过多，性能就会受到影响，甚至导致命令失败．

自定义版本号属性也经常用到，一种常见的用法是为版本号添加一个包含 问题跟踪 ID 的属性，表示该版本号修复了这个问题．其他一些用法还可以 是为版本号附加一个更友好的名字—人们很难记住版本号 1935 是一个 经过充分测试的版本，但是如果给版本号 1935 添加一个属性 `test-results`, 属性值是 `all passing`, 这样一来就方便多了．用户可以通过 *svn commit* 的选项 `--with-revprop` 为新提交的版本号附加属性 `test-results`:

```
$ svn commit -m "Fix up the last remaining known regression bug." \
             --with-revprop "test-results=all passing"
Sending        lib/crit_bits.c
Transmitting file data .
Committed revision 912.
$
```

可搜索性 (或者说, 为什么 不 使用属性)

与 Subversion 属性相关的所有工具—说得更准确点就是所有可 用的接口—都有一个比较严重的问题: 虽然 *设置* 一个自定义属性非常简单, 但是 *搜索* 自定义属性就 完全是另一回事了.

为了定位一个自定义版本号属性， 通常需要线性访问仓库的所有版本号， 询问每一个版本号 “你有没有我要找的属性?”. 为了帮助 搜索, 给命令 *svn log* 带上选项 `--with-all-revprops`, 再把输出模式调成 XML. 在下面 的输出中可以看到自定义属性 `testresults` 的信息:

```
$ svn log --with-all-revprops --xml lib/crit_bits.c
<?xml version="1.0"?>
<log>
<logentry
   revision="912">
<author>harry</author>
<date>2011-07-29T14:47:41.169894Z</date>
<msg>Fix up the last remaining known regression bug.</msg>
<revprops>
<property
   name="testresults">all passing</property>
</revprops>
</logentry>
…
$
```

查找自定义的版本化属性也很痛苦, 经常要在整个工作副本中递归地调用 *svn propget*. 也许递归访问不会像线性访问那样糟糕, 但在性能和成功率上仍然还有很大的提升空间, 尤其是当搜索是从仓库的根 目录开始时.

由于这个原因, 用户可能会选择—特别是版本号属性—把元 数据直接写到日志消息里, 为了方便从日志消息里解析出元数据, 要使用一 些策略驱动 (可能还会通过程序施加强制性) 的书写格式, 例如下面的 Subversion 日志消息格式就很常见:

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

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

但是问题仍然存在, Subversion 没有日志消息模版机制, 所以用户必须 自己协商出一套统一的, 嵌入版本号元数据的日志消息格式, 在使用时也只 能由用户自己来保证格式的一致性.

操作属性

命令 *svn* 提供了几种用于添加或修改文件和目录 属性的方法. 如果属性的值比较短, 而且是人类可读的, 那么添加新属性的 最简单的方法是在子命令 *svn propset* 的命令行参 数上指定属性名和值:

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

```
$
```

Subversion 对于属性值给予了很大的灵活性，如果属性值包含多行文本，甚至是二进制格式，此时用户就不太可能把值写在命令行参数上，为了解决 这个问题，*svn propset* 支持选项 `--file` (`-F`)，该选项指定了一个包含 属性值的文件的名字.

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

对属性名有一些限制条件，属性名必须以字母，冒号 (`:`) 或下划线 (_) 开始，接下来的字符，除了前面介绍的，还可 以用数字，连字符 (`-`)，句点 (`.`). [3]

除了 *propset*, *svn* 还提供了 子命令 *propedit*. *propedit* 使用 预先配置的外部编辑器（见 “通用配置选项”一节) 来添加或修改属性. 执行 *svn propedit* 时，命令在一个临时文件上打开 编辑器，临时文件的内容是属性的当前值（如果是添加新属性，内容就是空的），然后用户就可以按照自己的需要在编辑器里修改属性值，修改完成后保存临时 文件，最后退出编辑器. 退出编辑器后，如果 Subversion 检测到属性原来的 值被修改了，它就把修改后的值当作属性的新值. 如果用户没有修改便退出 编辑器，属性值就保持不变：

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

应该注意到，和 *svn* 的其他子命令一样，属性操作 可以同时施加到多个路径上，这就允许用户用一个命令修改整个文件集合的属性，例如我们可以这样做：

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

如果用户不能方便地获取属性值，那么属性的添加和删除就没什么大用处，所以 *svn* 提供了两个子命令用于显示文件和目录上的 属性名和值. 命令 *svn proplist* 列出指定路径上的属性 的名字，一旦知道了属性名，就可以用命令 *svn propget* 分别地获取各个属性的值，它根据指定的属性名和一个路径（或多个路径）打印 出属性的值.

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

执行命令 *svn proplist* 时如果加上选项 `--verbose` (`-v`)，命令就会同时列出 所有属性的名字和值.

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

[3] 如果读者熟悉 XML，就会发现这很像 XML “Name” 语法的 ASCII 子集.

```
  license
    ================================================================
    Copyright (c) 2006 Red-Bean Software.  All rights reserved.

    Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions
    are met:

    1. Redistributions of source code must retain the above copyright
    notice, this list of conditions, and the recipe for Fitz's famous
    red-beans-and-rice.
    …
```

最后一个与属性相关的子命令是 *propdel*. 因为 Subversion 允许为属性设置空值, 所以用户不能想当然地认为用 *svn propedit* 和 *svn propset* 把属性值设置成空值, 就能实现完全删除属性的效果, 比如说下面的命令不会产 生用户想要的效果 (用户想要的效果是删除属性 `license`):

```
$ 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

$
```

为了完全删除属性, 需要使用子命令 *propdel*, 它的使用语法和其他属性命令类似:

```
$ svn propdel license calc/button.c
property 'license' deleted from 'calc/button.c'.
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright
    (c) 2006 Red-Bean Software
$
```

还记得那些非版本化的版本号属性吗? 用户也可以用我们刚刚介绍过的 *svn* 的子命令去修改它们, 只要加上选项 `--revprop` 和欲修改的版本号. 因为版本号是全局的, 所以 只要用户已经位于欲修改的版本号的工作副本中, 就不需要为命令指定目标路径, 否则的话, 可以在命令行上提供目标路径的 URL 参数. 例如, 用户可能想修改 一个已存在的版本号的提交日志, [4] 如果你的当前工作目录是工作副本的一 部分, 可以不带目标路径地执行命令 *svn propset*:

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

即使用户没有检出仓库的工作副本, 仍然可以通过提供仓库的根 URL 来 修改属性:

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

[4] 修改提交日志的拼写错误, 语法问题和其他的一般性错误可能是 `--revprop` 最常见 的应用场景.

```
              http://svn.example.com/repos/project
property 'svn:log' set on repository revision '11'
$
```

需要注意的是只有在仓库管理员配置后用户才能修改非版本化属性（见“修正提交日志消息”一节）. 这是因为如果属性是非 版本化的，用户一不小心就有可能弄丢信息. 仓库管理员可以采取一定的措施 防止信息丢失，在默认情况下，修改非版本化属性是被禁止的.

如果可以的话，用户应该尽量使用 *svn propedit*，而不是 *svn propset*. 虽然这两个命令的执行结果是 一样的，但是 *svn propedit* 允许用户看到将要被修改 的属性的当前值，这可以帮助他们确认自己是否正在按照自己想要的那样 操作. 另外，在编辑器中修改具有多行文本的属性，要比在命令行上修改方便 得多.

属性和 Subversion 工作流程

既然读者已经熟悉了所有与属性相关的 *svn* 子命令，现在来看属性修改将会如何影响 Subversion 的工作流程. 我们已经说过，文件和目录的属性是被版本控制的，就像文件内容那样，因此 Subversion 也 支持属性的合并—或干净利落地，或带有冲突.

和文件内容一样，属性修改一开始只是本地的，只有用 *svn commit* 提交后，属性的修改才会持久化. 属性的修改 也能轻易地撤消—命令 *svn revert* 可以撤消所有 文件和目录的本地修改，包括属性修改，内容修改，以及其他所有的本地修改. 你也可以用 *svn status* 和 *svn diff* 获取文件和目录的属性状态.

```
$ svn status calc/button.c
 M      calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
___________________________________________________________________
Added: copyright
## -0,0 +1 ##
+(c) 2006 Red-Bean Software
$
```

注意，子命令 *status* 把 `M` 显示 在了第二列，而不是第一列，这是因为我们修改的是 *calc/button.c* 的属性，而不是内容. 如果我们同时修改 了内容和属性，我们就会同时在第一列和第二列看到 `M`（我们在 “查看修改的整体概述”一节 介绍了 *svn status*）.

属性冲突

和文件内容一样，本地的属性修改有可能和其他人提交的修改产生冲突，发生冲突后，Subversion 把条目设置成冲突状态.

```
$ svn update calc
Updating '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.
Summary of conflicts:
  Property conflicts: 1
$
```

发生冲突后，Subversion 也会在冲突对象的同一个目录下创建一个以 *.prej* 作为后缀名的文件，其中包含了冲突的细节，用户应该检查这个文件的内容，从而决定应该如何解决冲突. 在冲突解决之 前，冲突条目的 *svn status* 输出信息的第二列会有 一个字母 C, 如果试图提交修改，提交操作将会失败.

```
$ 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'.
$
```

为了解决冲突，只需要确保发生冲突的属性包含了正确的值，然后用命令 *svn resolve --accept=working* 告诉 Subversion 你已经手动解决了冲突问题.

读者可能已经注意到了 Subversion 的属性差异输出并不是一种标准的 格式，用户仍然可以用 *svn diff* 并把它的输出重定向到补丁文件里，但 *patch* 会忽略属性的补丁— *patch* 的一条规则是忽略所有不能理解的内容，这就意味 着如果用户用的是 *patch*, 为了完整地打上 *svn diff* 生成的补丁，用户必须手工地打上和属性相关的 修改.

Subversion 1.7 从两个方面改善了这个问题，首先，属性的差异输出至少 是机器可读的—这是对 1.7 版之前的属性显示的改进. 然后 Subversion 1.7 引入了新命令 *svn patch*, 专门用来处理 *svn diff* 的输出中带有的额外信息，并把这些信息应用 到工作副本中. 对于属性来说，使用 Subversion 1.7 及以后版本的 *svn diff* 生成的补丁，如果其中包含了属性差异，那么 *svn patch* 可以自动地把这些差异应用到工作副本. 关于 *svn patch* 的更多信息，见 svn 参考手册—Subversion 命令行客户端 的 svn patch.

svn diff 在报告属性的变化时有一个例外，那就是 特殊的 `svn:mergeinfo` 属性—该属性用于跟踪合 并信息—的变化会以一种更适合人类阅读的方式呈现出来，这对于那些 需要阅读合并信息的用户来说特别有帮助. 不过，补丁程序（包括 *svn patch*) 仍然会忽略与 `svn:mergeinfo` 相关的补丁. 这样做看起来好像是有问题的，但事实上并非如此，因为属性 `svn:mergeinfo` 由 *svn merge* 单独 进行管理，关于合并跟踪的更多信息，见 第 4 章 分支与合并.

继承的属性

Subversion 1.8 引入了继承属性这个概念. 将一个属性设置成可继承的 并没有什么很特别的地方, 实际上, 所有版本化的属性都是可继承的! 1.8 前 的版本化属性和 1.8 后的版本化属性的主要区别是后者支持在一个目标路径 的 父路径 (*parents*) 上搜索 属性, 即使这些父路径在工作副本里不存在.

有些命令可以显示出一般的属性继承, 首先 *svn proplist* 和 *svn propget* 可以检索 URL 的或工作副本路径的父路径 上的所有属性, 方法是带上选项 `--show-inherited-props`. 读者可能会觉得这是选项 `--recursive` 的反面—选项 `--recursive` 向 “下” 递归到目标的子目录里, 而 `--show-inherited-props` 是向 “上” 看 目标的父目录. 命令 *svnlook propget* 和 *svnlook proplist* 按照类似的方法使用选项 `--show-inherited-props`.

举个例子, 在工作副本的根目录递归地调用 *propget* , 发现子命令的目标路径及其中的一个子目录 *site* 都设置了属性 `svn:auto-props`:

```
$ svn pg svn:auto-props --verbose -R .
Properties on '.':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native

Properties on 'site':
  svn:auto-props
    *.html = svn:eol-style=native
```

如果我们把子目录 *site* 作为子命令的目标路径, 然后使用选项 `--show-inherited-props`, 我们将会看到属性 `svn:auto-props` 存在于目标路径 和 它的父路径上, 父路径的属性是 “被继承的”:

```
$ svn pg svn:auto-props --verbose --show-inherited-props site
Inherited properties on 'site',
from '.':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native

Properties on 'site':
  svn:auto-props
    *.html = svn:eol-style=native
```

在上一个例子里, 工作副本的根目录对应仓库的根目录, 但即使没有这种 对应, 属性也可以从工作副本的外面继承. 现在检出上一个例子的 *site* 目录, 使它成为工作副本的根目录:

```
$ svn co http://svn.example.com/repos site-wc
A    site-wc/publish
A    site-wc/publish/ch2.html
A    site-wc/publish/news.html
A    site-wc/publish/ch3.html
A    site-wc/publish/faq.html
```

```
A    site-wc/publish/index.html
A    site-wc/publish/ch1.html
 U   site-wc
Checked out revision 19.

$ cd site-wc
```

当我们在一条工作副本路径上检查继承的属性时将会看到，一个属性继承 自工作副本里的父目录，一个属性继承自仓库里的父路径，该路径在工作副本 的根目录的 “上层”：

```
$ svn pg svn:auto-props --verbose --show-inherited-props publish
Inherited properties on 'publish',
from 'http://svn.example.com/repos':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native

Inherited properties on 'publish',
from '.':
  svn:auto-props
    *.html = svn:eol-style=native
```

用户只能从他拥有读权限的仓库路径上继承属性—见 “内建的认证与授权”一节 和 “授权选项”一节. 如果用户对某条 父路径没有读权限，看起来的效果就像是父路径上没有设置属性.

前面已经说过，*svnlook proplist* 和 *svnlook propget* 也支持选项 `--show-inherited-props`，但它们不是以工作副本路径或 URL 作为目标路径，而是以仓库的路径作为目标路径：

```
$ svnlook pg repos svn:auto-props /site/publish --show-inherited-props -v
Inherited properties on '/site/publish',
from '/':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native

Inherited properties on '/site/publish',
from '/site':
  svn:auto-props
    *.html = svn:eol-style=native
```

当工作副本被首次检出或者更新时，从工作副本根目录上层继承而来的属性 会被缓存在工作副本的管理数据库里，这样的话在查看继承的属性时就不用再 访问仓库了，同时也允许那些不要求访问仓库的子命令（例如 *svn add*）在保持 “无连接” 的同时，仍然可以访问到从工作 副本之外的路径继承而来的属性. 但同时也意味着在最近一次更新之后，来自工 作副本根目录上层的继承属性可能已经发生了变化，使得本地缓存变成过时了的. 所以如果用户要求继承的属性始终是最新的，最好更新一下工作副本或直接询问 仓库.

到这里读者可能会想 “看起来挺有趣的，但这有什么好处呢？”对于属性继承本身来说是没多大用处，在 1.8 之前，Subversion 所有的保留属性 `svn:*`（还可能包括所有的用户自定义属性）都只能应用到它们所在的路径上，至多再加上直接子路径[5]. Subversion 使用继承属性完成另一些更有趣的事情，比如说用属性 `svn:auto-props` 设置自动属性，用属性 `svn:global-ignores` 实现全局的忽略模式—关于这些特殊属性的更多信息和使用方法，见 “自动属性设置”一节 和 “忽略未被版本控制的项”一节.

目前可继承的属性中起主要作用的是 `svn:auto-props` 和 `svn:global-ignores`，但这并不意味着故事就此结束. 在 Subversion 未来的版本里，我们期待继承属性能加入更多的特性，例如日志消息模版，与此同时，请尽情按照你自己的喜好去使用继续属性. 用户想要应用到整个仓库的任意一段版本化元数据（或仓库中的某个较大的部分）都可以轻易地存放到仓库的根目录（或适当的子目录）的属性上. 我们相信某些用户和管理员使用继承属性的方式可能连我们都未必想像得到.

自动属性设置

属性是 Subversion 最强大的特性之一，它是本章和其他章节介绍的众多 Subversion 特性—文本差异比较，合并支持，关键字替换和换行符转换等—的关键基础. 为了充分发挥属性的作用，它们必须被设置到正确的文件和目录上，不幸的是，这个步骤在日常工作中常常被人遗忘，尤其是因为即使属性设置不当通常也不会造成很明显的错误（至少和文件添加失败比起来，不是很明显）. 为了帮助用户更好地使用属性，Subversion 提供了几个简单但很有用的特性.

每当用户使用 *svn add* 和 *svn import* 向仓库添加文件时，Subversion 自动地在文件上设置一些常见的属性. 首先，如果操作系统的文件系统支持可执行权限位并且文件具有可执行权限，Subversion 就自动在文件上设置 `svn:executable` 属性（关于这个属性的更多信息，见 “文件的可执行性”一节）.

然后，Subversion 会试图判断文件的 MIME 类型. 如果用户为 `mime-types-files` 设置了一个运行时配置参数，Subversion 就会尝试根据文件的后缀名为文件搜索一个对应的 MIME 类型映射，若找到的话，它就把文件的 `svn:mime-type` 属性设置成找到的 MIME 类型. 如果用户没有为 `mime-types-files` 设置运行时配置参数，或者根据后缀名没有找到对应的类型映射，Subversion 就使用启发式的算法来判断文件的 MIME 类型. 取决于编译时的配置，Subversion 1.7 可以利用文件扫描函数库[6]检测文件的类型. 如果前面的都失败了，Subversion 就使用它非常基本的启发式算法来判断文件是否包含非文本数据，如果是，就自动地把文件的 `svn:mime-type` 属性设置成 `application/octet-stream`（最一般的 MIME 类型，表示“这是字节的集合”）. 当然，如果 Subversion 的判断不正确，又或者是用户想把 `svn:mime-type` 设置成更精确的值—比如 `image/png` 或 `application/x-shockwave-flash`—可以自由地修改或删除属性 `svn:mime-type`（关于 Subversion 如何使用 MIME 类型的更多信息，见本章后面的 “文件内容类型”一节）.

有很多文件使用的是 UTF-16 编码，虽然在语义上文件的内容是纯文本的，但是 UTF-16 使用的字节在 ASCII 字符的范围之外，因此 Subversion 更倾向于把它们归类为二进制文件，用户在给这些文件进行差异比较，合并和关键字替换时也会因此遇到一些小麻烦.

借助运行时配置系统（见 “运行时配置区域”一节），Subversion 提供了一种更加灵活的自动属性设置功能，它允许用户创建文件名模式到属性名和值的映射. 再说一次，这些映射会影响 *svn add* 和 *svn import*，除了会覆盖由 Subversion 判断出的默认 MIME 类型，还可能添加额外的属性或自定义属性. 例如，用户想创建一个映射，这个映射是说每次添加一个 JPEG 文件时—文件的名字符合模式 `*.jpg`—Subversion 都应该自动地把这个文件的 `svn:mime-type` 属性设置为 `image/jpeg`. 又或者说匹配模式 `*.cpp` 的文件都应该把 `svn:eol-style` 设置成 `native`，把 `svn:keywords` 设置成 `Id`. 关于运行时配置如何支持自动属性的更多细节，见 “通用配置选项”一节.

[5]有一个例外是 `svn:mergeinfo` 属性，它是可继承的—见 “合并信息和预览”一节

[6]当前比较常用的函数库是 libmagic

虽然借助运行时配置系统来支持自动属性设置非常方便，但 Subversion 管理员可能更希望看到这样一种情况：当客户端在一个从特定服务器检出的 工作副本上工作时，可以自动地顾及到某些属性定义. Subversion 1.8 及其 之后的客户端版本通过可继承属性 `svn:auto-props` 实现这个功能.

属性 `svn:auto-props` 可以像运行时配置系统那样， 自动地为新增的文件设置属性， 属性 `svn:auto-props` 的值应该和运行时配置选项 `auto-props` 的值相同 （也 就是任意数量的键值对， 格式是 FILE_PATTERN = PROPNAME=VALUE[;PROPNAME=VALUE ...]). 和运行时选项 `auto-props` 一样，如果使用了选项 `--no-auto-props`, 属性 `svn:auto-props` 就会被忽略，但是有所不同的是，即使配置选项 `enable-auto-props` 被设置为 `no`, 属性 `svn:auto-props` 也不会被禁止.

举例来说，你检出了主干的工作副本，想在其中添加一个新文件（假设 运行时配置系统禁止了自动属性）：

```
$ svn st
?       calc/data.c

$ svn add calc/data.c
A         calc/data.c

$ svn proplist -v calc/data.c
Properties on 'calc/data.c':
  svn:eol-style
    native
```

可以看到，当 *data.c* 被版本控制后，文件自动 设置了属性 `svn:eol-style`. 因为运行时配置选项 `auto-props` 是禁止了的，所以属性 `svn:auto-props` 肯定来自 *data.c* 的 父路径. 执行带上选项 `--show-inherited-props` 的命令 *svn propget* 可以看到，事实的确是如我们所想的那样：

```
$ svn propget svn:auto-props --show-inherited-props -v calc
Inherited properties on 'calc',
from 'http://svn.example.com/repos':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native
```

属性 `svn:global-ignores` 及其对应的运行时配置 选项 `global-ignores` 是一起起作用，但属性 `svn:auto-props` 和运行时选项 `auto-props` 的关系就不这样，如果运行时选项 `auto-props` 在一个模式上设置了一个自动属性，而 属性 `svn:auto-props` 也在 同一个 模式上设置了自动属性，那么属性的设置就会覆盖运行时配置选项的设置. 从一个路径继承而来的自动属性[7]也只会覆盖从其他路径继承的 同一个 模式. 覆盖的先后顺序是：

- 在 `svn:auto-props` 上定义的，针对某一模式的 自动属性会覆盖运行时配置选项 `auto-props` 上设置 的同一模式的自动属性.
- 对于一个给定的模式而言，如果它的自动属性继承自多个父路径的 `svn:auto-props` 属性，那么在路径上最近的父路径 的自动属性会覆盖其中父路径.

[7]用户只能从他拥有读权限 路径上继承属性，所以说如果管理员在较高层的父路径上（例如仓库的 根目录）设置了属性 `svn:auto-props`, 他就应该 确保所有用户都能读取该路径或者期望的自动属性设置不会失效.

- 对一个给定的模式而言，如果在路径的 `svn:auto-props` 属性上显式地设置了一个自动属性，那它就会覆盖从其他路径 继承而来的相同模式上的自动属性.

举例来说，假设你有一个如下所示的运行时配置：

```
[miscellany]
enable-auto-props = yes
[auto-props]
*.py  = svn:eol-style=CR
*.c   = svn:eol-style=CR
*.h   = svn:eol-style=CR
*.cpp = svn:eol-style=CR
```

你想添加 *calc* 目录中的三个文件：

```
$ svn st
?       calc/data-binding.cpp
?       calc/data.c
?       calc/editor.py
```

先看一下 *calc* 的 `svn:auto-props` 属性：

```
$ svn propget svn:auto-props -v --show-inherited-props calc
Inherited properties on 'calc',
from 'http://svn.example.com/repos':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native

Inherited properties on 'calc',
from '.':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:keywords=Author Date Id Rev URL
```

添加这三个文件，然后检查它们的自动属性：

```
$ svn add calc --force
A         calc/data-binding.cpp
A         calc/data.c
A         calc/editor.py
```

文件 *data-binding.cpp* 只有一个匹配的模式，也就是运行时配置选项里的 `*.cpp = svn:eol-style=CR`，显然文件的属性 `svn:eol-style` 被设置为 `CR` ：

```
$ svn proplist -v calc/data-binding.cpp
Properties on 'calc/data-binding.cpp':
  svn:eol-style
    CR
```

文件 *editor.py* 既匹配运行时配置选项里的一 条模式，也匹配属性 `svn:auto-props` 里的模式，根据前 面介绍的覆盖顺序，显式设置在 *calc* 上的属性值 (`*.py = svn:eol-style=native`) 的优先级较高，所以 属性 `svn:eol-style` 被设置为 `native`：

```
$ svn proplist -v calc/editor.py
Properties on 'calc/editor.py':
  svn:eol-style
    native
```

文件 *data.c* 同时匹配运行时配置选项和继承属性 `svn:auto-props` 的模式．自动属性 `svn:keywords` 只被定义了一次，在 *calc* 上定义，所以 *data.c* 自动获取了该属性．*calc* 上的 `svn:auto-props` 没有 为 `svn:eol-style` 定义值，所以最近的父路径 `http://svn.example.com/repos` 提供了这个值：

```
$ svn proplist -v calc/data.c
Properties on 'calc/data.c':
  svn:eol-style
    native
  svn:keywords
    Author Date Id Rev URL
```

自动属性的覆盖只发生在 相同的 模式上，如果 新增的文件同时匹配多个模式，那就无法确定最终应用的是哪一个自动属性．比如说用户想把文件 *foo.cpp* 添加到目录 *bar*，而 *bar* 的属性 `svn:auto-props` 的值是：

```
*.c*  = svn:eol-style=native
*.cpp = svn:eol-style=native;svn:keywords=Author Date Id Rev URL
```

因为 *foo.cpp* 同时匹配两个不同的模式，所以 我们没办法事先确定属性 `svn:keywords` 是否被设置到 *foo.cpp* 上．

`svn:auto-props` 最后一个需要注意的地方是它 （以 及类似的 `svn:global-ignores`，见 “忽略未被版本控制的项”一节) 只是向理解属性 的客户端工具提供了一个建议，较老的客户端会忽略这些属性，选项 `--no-auto-props` 会忽略它们，用户可能会选择手动地修改 或删除自动属性—有很多方法可以旁路掉包含在 `svn:auto-props` 里的推荐属性．因此，管理员仍然需要使用钩子脚本验证文件和 目录上的属性是否符合管理员的策略，并拒绝与策略不兼容的提交（钩子脚本见 “忽略未被版本控制的项”一节)．

Subversion 的保留属性

本节将对 Subversion 所有的保留属性做一个简单的总结，包括版本化的 的属性（和文件，目录关联）与非版本化的属性（和版本号关联）．

版本化的属性

这些是 Subversion 保留给自己用的版本化属性：

`svn:auto-props`

该属性包含了一系列的自动属性定义，如果被设置在一个目录上，那么自动属性定义会应用到目录内的所有文件，见 “自动属性设置”一节．

`svn:executable`

如果该属性被设置到一个文件上，那客户端就会给 Unix 工作副本里 的文件设置上可执行权限，见 “文件的可执行性”一节.

`svn:mime-type`

如果属性出现在一个文件上，那么属性值指出了文件的 MIME 类型，当更新时，属性可以帮助客户端判断是否可以安全地对文件进行基于行 的合并操作. 另外，当用户通过网页浏览器获取文件时，该属性还会影 响文件的具体行为. 更多的信息参考 “文件内容类型”一节.

`svn:ignore`

如果该属性出现在一个目录上，属性值是一个未被版本化的文件 模式列表，符合模式的文件会被 *svn status* 和 其他子命令忽略，见 “忽略未被版本控制的项”一节.

`svn:global-ignores`

如果该属性出现在一个目录上，属性值是一个未被版本化的文件 模式列表，符合模式的文件会被 *svn status* 和 其他子命令忽略，但是和 `svn:ignore` 不同的是，这些模式会应用到目录内 *所有的* 子目录及其 子文件，而不仅仅是目录的直接子文件，见 “忽略未被版本控制的项”一节.

`svn:keywords`

如果该属性出现在一个文件上，属性的值指出了客户端应该如何 扩展文件内的特定关键字，见 “关键字替换”一节.

`svn:eol-style`

如果该属性出现在一个文件上，则属性的值指出了客户端应该如何 处理工作副本和导出目录里的文件的行终止符，见 “行结束标记”一节 和 svn export.

`svn:externals`

如果该属性出现在一个目录上，则属性的值是一个包含了多个路径 和 URL 的列表，这些路径和 URL 都是客户端需要检出的内容，见 “外部定义”一节.

`svn:special`

如果该属性出现在一个文件上，则表示该文件不是一个普通的文件，可能是一个符号链接或其他特殊的对象[8]

`svn:needs-lock`

如果该属性出现在一个文件上，客户端就会把工作副本里的这个 文件设置成只读，也就是提醒用户在编辑文件之前需要加锁，见 “锁通信”一节.

`svn:mergeinfo`

Subversion 使用该属性跟踪合并信息，更多的细节见 “合并信息和预览”一节，除非你 *真得* 知道自己在做什么，否则不要 编辑该属性.

[8]在写到这里时，符号链接是已知的唯一一个 “特殊” 对象，在以后 的版本里可能会出现更多种类的特殊对象.

未版本化的属性

下面是保留给 Subversion 使用的未版本化的（或版本号）属性，它们中 的大部分都会出现在仓库的每个版本号上，属性携带了关于修改的起因与 本质.

`svn:author`

如果设置了该属性，则属性包含了创建此版本号用户名，如果没有该属性，那么版本号是匿名提交的.

`svn:autoversioned`

如果设置了该属性，则说明版本号是通过自动版本化特性创建的，见 “自动版本控制”一节.

`svn:date`

包含了版本号创建时的 UTC 时间，使用 ISO 8601 格式，时间来自 服务器 的机器时钟，而不是客户 端的时钟.

`svn:log`

包含了描述版本号的日志消息.

Subversion 的某些辅助工具—*svnrdump* 和 *svnsync*—也会使用未版本化的属性完成记帐工作，这些属性只会出现在仓库的版本号 0 上，关于 *svnrdump* 和 *svnsync* 的更多信息，见 第 5 章 仓库管理. 下面是由 *svnrdump* 和 *svnsync* 创建并管理的属性.

`svn:rdump-lock`

为 *svnrdump load* 访问仓库临时施加互斥 性，通常只有在 *svnrdump load* 活动时— 或者在 *svnrdump* 不能干净地与仓库断开连接时 —该属性才会被观察到（只有当这个属性出现在版本号 0 上时，它才是有意义的）.

`svn:sync-currently-copying`

包含了源仓库中已经被 *svnsync* 镜像备份 的版本号（只有当这个属性出现在版本号 0 上时，它才是有意义的）.

`svn:sync-from-uuid`

包含了由 *svnsync* 创建的镜像的源仓库的 UUID（只有当这个属性出现在版本号 0 上时，它才是有意义的）.

`svn:sync-from-url`

包含了由 *svnsync* 创建的镜像的源仓库目录 的 URL（只有当这个属性出现在版本号 0 上时，它才是有意义的）.

`svn:sync-last-merged-rev`

包含了最近一次被成功地镜像备份的源仓库的版本号（只有当这 个属性出现在版本号 0 上时，它才是有意义的）.

`svn:sync-lock`

为 *svnsync* 的镜像操作临时添加仓库访问 的互斥性，通常只有在 *svnsync* 活动时— 或者在 *svnsync* 不能干净地与仓库断开连接时，只有当这个属性出现在版本号 0 上时，它才是有意义的）.

文件的可移植性

对于经常需要在不同的操作系统中工作的用户来说，比较幸运的一点是 Subversion 命令行工具在所有系统中的表现几乎都是相同的，如果用户已经知道 了如何在一种系统中使用 *svn*，那他也就知道了如何在其他 系统中使用 *svn*.

然而，其他软件或存放在 Subversion 仓库里的文件并不都是这样．比如说 在一台 Windows 机器上，对于 “文本文件” 的定义和 Linux 机器 类似，除了一点—标记一行结束的字符序列不同．除此之外，Unix 平台（和 Subversion）支持符号链接，而 Windows 不支持；Unix 平台根据文件系统权限 来判断文件的可执行性，而 Windows 是根据文件的扩展名．

Subversion 并不想把整个世界都统一到公共的定义和实现上，当用户要在多 种不同的操作系统中管理文件与目录时，它所能做的只是尽量减少用户的麻烦．本节介绍 Subversion 如何帮助用户在多种不同的平台中使用 Subversion.

文件内容类型

和许多应用程序一样，Subversion 也会使用 MIME（多用途互联网邮件扩展 类型，Multipurpose Internet Mail Extensions）内容类型．属性 `svn:mime-type` 除了可以作为文件内容类型的存放位置，它的值还决定了 Subversion 的某些行为特征．

识别文件类型

很多程序都会根据文件的名字—尤其是扩展名—对文件内容 的类型和格式作出一些假定，比如说文件名以 *.txt* 结 尾的文件通常被认为是人类可读的，也就是说用户可以通过简单的阅读来理解 文件的内容，而不需要经过复杂的译解过程．另一方面，文件名以 *.png* 结尾的文件通常会被当作可移植网络图像 (Portable Network Graphics, PNG) 格式—它们不是人类可读的类型，只有理解 PNG 格式的软件才能把文件内容以适当的形式展现出来．

不幸的是，有些扩展名的意义会随着时间而发生变化．当个人计算机首次出 现时，如果有一个名为 *README.DOC* 的文件，那就几乎可 以确定它是个纯文本文件，就像现在的 *.txt* 文件，但是 在 90 年代中期，同样名字的文件很可能是一个 Microsoft Word 文档，它采用 了一种私有的，人类不可读的文件格式．不过这种变化并非一蹴而就—曾经 有一段时间，当计算机用户看到一个 *.DOC* 文件时，常常 想不清楚文件的格式到底是什么类型．[9]

计算机网络在文件名与文件类型的关系上更加疑惑．信息跨网络地传输，并且由服务器端的脚本动态生成，本质上不能算作真正的文件，也就不存在 文件名．比如说，网页服务器需要通过其他途径告诉浏览器它们正在下载的 文件是什么类型，这样的话针对文件，浏览器可以采取更加智能的做法，比 如用一个对应的程序打开文件，或提示用户应该把下载的文件放到哪个目录 下．

最终总算有一个描述数据流内容的标准出现了．1996 年，RFC 2045 发布，它是描述 MIME 的 5 篇 RFC 文档的第一篇．文档介绍的概念包括 媒体类型及其子类型，并推荐了一种表示这些类型的语法．如今，MIME 媒体 类型—或 “MIME 类型” —被广泛地应用在邮件程序，网页服务器等软件中，作为一种解决文件内容格式混乱的事实标准．

比如说，Subversion 提供的一项特性是在更新工作副本时，支持基于行 的文件内容合并，但是二进制文件没有 “行” 的概念，于是，如果文件的 `svn:mime-type` 属性被设置成非文本 MIME 类型（非文本的 MIME 类型通常不以 `text/` 开始，但是也有 例外），Subversion 就不会对文件执行合并操作．作为替代，如果被更新的二 进制文件含有本地修改，那文件就不会被更新，Subversion 会另外创建两个 新的文件，其中一个的扩展名是 *.oldrev*，对应文件的 BASE 版本号；另一个的扩展名是 *.newrev*，对应更新 后的版本号．这样做是为了避免对不支持合并的文件进行合并而带来的错误．

[9] 雪上加霜的是，当时还 有一款叫作 WordPerfect 的软件也使用 *.DOC* 作为他们的私有文件格式的扩展名！

另外，为了能够以行为单位显示修改，文件必须能被划分成“行”，如果 *svn diff* 和 *svn annotate* 的目标文件的 MIME 类型是非文本的，这 两个命令默认会报错. 如果用户的文件是 XML 文件，它们的 `svn:mime-type` 被设置成 `application/xml`，虽然它们是人类可读的文本文件，但 Subversion 仍然会把它们看成是非文本 文件，幸好，为命令添加选项 `--force` 可以强制 Subversion 不管文件的 MIME 类型，直接执行操作.

如果属性 `svn:mime-type` 的值不能说明文件内容 是文本的，这将会给其他属性造成意想不到的影响. 例如，二进制文件没有 行的概念，所以 Subversion 不允许为二进制文件设置属性 `svn:eol-style`. 如果命令的目标是单一的文件，那么就很 容易看出来—*svn propset* 会报错退出，但是，如果用户递归地执行属性设置命令，可能就没那么明显了：如果 Subversion 觉得某个文件不适合设置给定的属性，它就会悄无声息地跳过该文件.

Subversion 提供了多种机制，用于自动地设置属性 `svn:mime-type`，详细的介绍见 “自动属性设置”一节.

另外，如果文件设置了属性 `svn:mime-type`，响应 GET 请求时，Subversion Apache 模块将会使用属性的值填充 HTTP 头部的 `Content-type` 字段. 如果用户使用浏览器查看仓库的内容，这可以提示浏览器应该如何显示文件.

文件的可执行性

在很多操作系统里，一个文件是否可以执行取决于该文件是否设置了 可执行权限位. 该位默认是不开启的，如果用户需要可执行权限，必须显式地 开启它. 但是记住应该为哪些检出的文件设置可执行位是一件很麻烦的事情，所以 Subversion 提供了属性 `svn:executable`，如果文件 设置了该属性，Subversion 就会在工作副本里打开文件的可执行位.

该属性对不支持可执行位的文件系统是没有效果的，比如 FAT32 和 NTFS.[10] 另外，尽管该属性 没有预定义的值，在设置属性时，Subversion 强制把它的值设置为 `*`. 最后，该属性只对文件有效，对目录不起作用.

行结束标记

除非属性 `svn:mime-type` 进行了额外说明，否则 Subversion 总是假设文件的内容是人类可读的. 一般来说，Subversion 会根据 自己的知识来判断是否可以对文件进行基于上下文的差异比较，如果不能的话，就 按字节比较差异.

Subversion 默认情况下并不关心文件的 *行结束 (EOL) 标记* (*end-of-line (EOL) markers*) 类型. 不幸的是，如何结束一行，不同的操作系统有着不同的约 定. 比如说，Windows 软件使用一对 ASCII 控制字符表示一行的结束— 一个回车符 (`CR`) 后面再跟一个换行符 (`LF`)；而 Unix 系统中的软件只用单一的换行符 (`LF`) 表示一行的结束.

如果文件的行结束标记与操作系统的 *本地的行结束风格* 不同，有些软件可能无法正确地处理这种文件. 所以在典型情况下，Unix 程序把来自 Windows 的文件里的回车符 (`CR`) 当 成一个普通字符（通常显示成 `^M`），而 Windows 程序会 把来自 Unix 系统的文件显示成一段很长的行，因为它们找不到用来结束一行 的回车符 (`CR`).

如果用户要在不同的操作系统之间分享文件，如此敏感的 EOL 标记可不 是什么好事. 比如说有一个源代码文件，开发人员可能会同时在 Unix 和 Windows 系统中编辑它，如果所有开发人员使用的工具都能保留文件原来的行 结束风格，那就不会产生什么问题.

可惜的是，如果文件的行结束标记和本地不同，很多程序要么不能正确地读 取并显示文件，要么在保存时，把文件的行结束标记转换成本地风格. 如果是前 一种情况，开发人员在开始编辑文件之前，需要使用一种格式转换工具（比如 *dos2unix* 及其伙伴 *unix2dos*）把 文件的行结束标记转换成本地风格. 如果是后一种情况就不用在编辑之前转换文件 格式. 但是两种

[10] Windows 的文件系统使用文件的扩展名（比如 *.EXE*, *.BAT* 和 *.COM*）表示文件是可执行的.

情况都会导致文件的每一行都发生变化！在提交修改之前，用户 有两种选择，一是使用格式转换工具把文件的行结束标记转换成与原来一样的 风格，二是直接提交—使用新的行结束标记.

这种情况的结果是既浪费了时间，也提交了很多没必要的修改. 浪费时间 已经足够烦人了，更糟糕的是一次提交修改了文件的每一行，这会给后面的历史 查询带来很大的麻烦—是哪几行修改解决了问题，或者是哪一行修改引入 了语法错误.

问题的解决办法是使用属性 `svn:eol-style`. 如果属 性的值是有效的，Subversion 将根据属性值对文件进行特殊的处理，这样文件 的行结束风格就不会随着操作系统的变化而变化. 属性的有效值包括：

`native`

文件的行结束标记是操作系统的本地风格，换句话说，如果一个用户 在 Windows 操作系统上检出了工作副本，文件的 `svn:eol-style` 被设置成 `native`，则文件将使用 `CRLF` 作为行结束标记. 而 Unix 用户 检出的文件的行结束标记是 `LF`.

注意，不管操作系统是什么类型，Subversion 仓库中存放的文件 的行结束标记总是 `LF`，这对用户来说是透明的.

`CRLF`

无论是什么操作系统，文件总是使用 `CRLF` 作为行结束标记.

`LF`

无论是什么操作系统，文件总是使用 `LF` 作为 行结束标记.

`CR`

无论是什么操作系统，文件总是使用 `CR` 作为行结束标记. 这种行结束标记用得很少.

忽略未被版本控制的项

在一个长时间使用的工作副本里，除了被版本控制的文件和目录外，常常 还有很多未被版本控制的文件与目录，而且它们将来也不会被添加到仓库里，这 些文件可能是文本编辑器的备份文件，或编译器产生的目标文件，对它们进行版 本控制是没有意义的，用户随时都有可能把它们删除.

希望工作副本不受这些杂质影响是不可能的. 实际上这是 Subversion 的一个 特性，那就是对操作系统来说工作副本就是一个普通的目录，与未被版本化的 目录相比并没有本质上的区别. 不过工作副本里的未被版本化的文件和目录会给 用户产生一定的困扰. 比如说，命令 *svn add* 和 *svn import* 默认会递归地执行，命令并不知道目录中的哪些 文件是用户想要的，哪些是不想要的. 命令 *svn status* 默认报告工作副本里的每一个项目的状态—包括未被版本控制的文件与目 录—如果未被版本控制的项目很多，命令的输出就比较扰人.

于是，Subversion 提供了几种方式告诉 Subversion 哪些文件是可以忽略的. 其中一种要用到 Subversion 的运行时配置系统（见 “运行时配置区域”一节），会受到配置影响的通常是在特定 计算机上执行的 Subversion 操作，或计算机上的某些特定用户. 另外两种方式 用到了 Subversion 的目录属性，与版本化目录的联系更为紧密，因此它会影响 到版本化目录的所有工作副本. 上面说的两种机制都会用到 文件模式 (*file patterns*)（用于匹配文件名的字符串，包含了字面字符与通配符）来决定应该忽略哪些 文件.

Subversion 运行时配置系统提供了一个选项— `global-ignores` —选项的值是空白符分隔的文件名模式集. 如果文件的名字 与集合中的某个模式匹配, 那这个文件对 Subversion 来说相当于是不存在的, 命令 *svn add*, *svn import* 和 *svn status* 就会忽略它. 如果工作副本里有永远不会 被版本控制的文件 (比如 Emacs 的备份文件 `*~` 和 `.*~`), 这个特性就会非常有用.

Subversion 的文件名模式

文件名模式（也叫作 *globs* 或 *shell 通配符模式* (*shell wildcard patterns*)) 是一个字符串，这个字符串用于匹配一个文件名，比较常见的 用法是从一大堆文件中，选出具有类似性质的子集，而不用列出每个文件名字. 模式中的字符分为两种：普通字符—按照字面值进行匹配，例如字母 a 就是匹配字母 a; 通配符— 在匹配时被解释成和字面值不同的含义.

文件名模式的语法有很多种，Subversion 用的是 Unix 系统中最常见的一 种语法，这种语法被实现成系统函数 `fnmatch`. 下面简 单介绍该语法支持的通配符：

`?`

匹配任意 一个 字符

`*`

匹配 0 个或多个字符组成的字符串

`[`

开始定义一个字符类，`]` 表示定义结束，字 符类可以匹配任意一个类中的字符

Unix shell 支持相同的文件名模式语法，下面是 shell 的一些使用例子：

```
$ 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
$
```

完整的文件名模式比我们这里介绍的要更加复杂，但是对大多数 Subversion 用户来说，这里介绍的基本用法已经足够了.

如果被版本控制的目录上设置了属性 `svn:ignore`，属性值应该是一个文件名模式列表，各项之间用换行符分开，Subversion 根据文件名模式列表判断相同目录内的哪些文件是可以忽略的。属性 `svn:ignore` 不会覆盖运行时配置选项 `global-ignores` 的值，而是作为一种补充。与 `global-ignores` 不同的是，属性 `svn:ignore` 里的模式只能作用在该属性所在的目录上，不会递归作用到子目录上。属性 `svn:ignore` 的一个常用目的是告诉 Subversion 去忽略每个用户的工作副本中可能都会有的文件，例如编译器的输出文件—对于本书而言，就是 HTML, PDF, PostScript 文件，或其他 DocBook XML 转换过程中产生的临时文件和输出文件。

Subversion 1.8 提供了一个比 `svn:ignore` 更强大的属性—`svn:global-ignores`。和 `svn:ignore` 相同的是，`svn:global-ignores` 只能设置到目录上，属性值是文件名模式集合。[11] `svn:global-ignores` 定义的文件名模式会添加到运行时配置选项 `global-ignores` 与属性 `svn:ignore` 定义的模式上。与 `svn:ignore` 不同的是，`svn:global-ignores` 是可继承的[12]，它会递归地作用到目录内的*所有*路径上，而不仅仅是目录的直接子文件。

Subversion 的忽略文件名模式只对未被版本控制的文件和目录起作用，一旦文件和目录被版本控制了，忽略文件名模式就不会对它们产生影响。也就是说，不要以为某个被版本控制的文件名符合忽略模式，在你提交时，Subversion 就会忽略它的修改—Subversion *总是*会注意到所有被版本控制的对象。

CVS 用户的忽略模式

Subversion 的 `svn:ignore` 属性的语法和功能非常像 CVS *.cvsignore* 文件。实际上，如果用户想从 CVS 的工作副本迁移到 Subversion 的工作副本中，为了迁移忽略模式，可以把 *.cvsignore* 作为命令 *svn propset* 的输入：

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

但是，CVS 和 Subversion 在忽略模式的使用方式上有所区别。两个系统使用忽略模式的时机不同，在忽略模式的作用目标上也有细微的差别，另外，Subversion 无法识别 ! 的特殊意义，对 CVS 来说，! 表示不要忽略这种模式。

运行时配置选项 `global-ignores` 里的忽略模式更倾向于个人化[13]，并且和工作副本相比，更贴近用户的个人需求。所以，本节的余下部分主要关注 `svn:ignore`，`svn:global-ignores` 及如何使用它们。

假设某个工作副本的 *svn status* 输出是：

```
$ svn status calc
 M      calc/button.c
?       calc/calculator
?       calc/data.c
?       calc/debug_log
?       calc/debug_log.1
?       calc/debug_log.2.gz
?       calc/debug_log.3.gz
```

在上面的例子里，用户已经修改了 *button.c*，但是工作副本里还有一些未被版本控制的项目：刚从源代码编译出的 *calculator* 程序，一个叫做 *data.c* 的源代码文件，还有几个用于调试的日志文件。假设用户已经知道编译系统总是会输出一

[11] `svn:global-ignores` 的各个文件名模式之间可能用空白符分隔（就像运行时配置选项 `global-ignores`），而不仅仅是换行符（属性 `svn:ignore` 的各个文件名模式之间只能用换行符分隔）。

[12] 当然，只有 1.8 或更新版本的 Subversion 客户端才能认识 `svn:global-ignores` 的意义与可继承性

[13] 抛开个人化不说，如果用户没有显式地设置 `global-ignores`，Subversion 就会使用默认值，见 “通用配置选项”一节

个目标文件 *calculator* [14], 而且测试程序总是会留下一些 调试日志文件, 除了用户自己的工作副本, 该项目所有的工作副本都有可能出现 这些文件. 用户非常清楚地知道, 当他执行 *svn status* 时, 并不想看到这些他不感兴趣的文件, 而且他也相信其他人也对它们不感兴趣. 于是, 用户决定为目录 *calc* 设置属性 `svn:ignore` :

```
$ svn propget svn:ignore calc
calculator
debug_log*
$
```

属性设置完毕后, 目录 *calc* 包含了未被提交的本地 修改. 注意看 *svn status* 的输出发生了什么变化:

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

现在, 命令的输出变得干净多了! 编辑器产生的目标文件 *calculator* 和日志文件仍然留在工作副本里, Subversion 只是不再提醒用户这些文件的存在. 输出变干净后, 用户就能更容易地关注到更重要的 事情上—例如用户可能忘记把源代码文件 *data.c* 添加到仓库里.

当然, 减少垃圾信息只是一个选择, 如果用户确实想看到所有的文件, 包括 正常情况下会被忽略的文件, 可以给 *svn status* 加上选项 `--no-ignore`:

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

被隐藏的未被版本控制的项目再度显示出来, 但是在项目的左边加上了字母 `I` (Ignored). 请等一下, 为什么 *wip.1.diff* 也有 `I`? *calc* 的属性 `svn:ignore` 里并没有匹配 *wip.1.diff* 的模式, 那么它为什么会被忽略?[15] 答案是继承的属性 `svn:global-ignores`. 执行带上选项 `--show-inherited-props` 的命令 *svn propget* , 就可以看到属性 `svn:global-ignores` 被设置 在了工作副本的根目录上, 果然在这个属性里找到了匹配 *wip.1.diff* 的模式:

```
$ svn pg svn:global-ignores calc -v --show-inherited-props
Inherited properties on 'calc',
from '.':
  svn:global-ignores
    *.diff
    *.patch
```

之前提过, *svn add* 和 *svn import* 也会用到忽略模式列表, 这两个操作都会要求 Subversion 开始管理文件与目录. 在递归的添加操作或导入操作中, Subversion 不会要求用户去选择目录 中的哪些文件应该被版本控制, 而是使用忽略模式—包括全

[14]这不就是构建系统存在的意义吗?
[15]假设用户的运行时配置选项 `global-ignores` 里也没有匹配的模式

局的，每个目录 与继承的—来决定哪些文件应该被忽略. 同样，用户也可以用选项 `--no-ignore` 告诉 Subversion 不会忽略任意一个文件.

即使已经设置了属性 `svn:ignore` 和 `svn:global-ignores`, 使用 shell 的通配符可能还是会产生一些 问题. 在 Subversion 接收到参数之前，shell 会把通配符扩展成显式的文件 名列表，所以执行 `svn SUBCOMMAND *` 相当于执行 `svn SUBCOMMAND file1 file2 file3` …. 如果是 *svn add*, 效果就像是给命令带上选项 `--no-ignore`, 所以最好使用 `svn add --force .` 完成文件的批量添加. 显式地指定目标文件确保了当前目录不 会因为已经处于版本控制之下而被忽略，选项 `--force` 使得 Subversion 在遍历目录，添加未被版本控制的文件时，仍然遵循属性 `svn:ignore`, `svn:global-ignores` 和运行时配置选项 `global-ignores` 的值. 如果你不想 递归地添加所有项目，别忘了给命令 *svn add* 加上选项 `--depth files`.

关键字替换

Subversion 支持把 关键字 (*keywords*)—跟文件有关的一段有用的动态信息—替换成文件 的内容. 关键字提供了与文件最后一次修改有关的信息，但是每次文件被修改时，这个信息都会发生变化，更重要的是，文件刚被修改后，除了版本控制系统，对 任何一个企图保持数据最新的过程都是一场混乱，如果把工作交给用户，就很容易 造成信息过时.

比如说用户有一个文档，他想显示文档最后一次被修改的日期. 他可以要求 文档的每一个作者在他们提交修改之前，在文档中记录一下本次修改的日期. 但 是很快就会出现，总有人会忘记记录修改日期. 更好的做法是让 Subversion 去 完成记录时间的操作，比如说在每次提交时，把文档中的关键字 `LastChangedDate` 替换成当时的日期. 通过在文档中放置一个 关键字锚点 (*keyword anchor*), 用户可以控制关键字的插入位置. 锚点就是一段简单的文本，格式是 `$KeywordName$` .

如果只想单纯地往文件中添加关键字锚点并不会产生什么特别的效果，除非 用户显式要求 Subversion, 否则的话它决不会执行文本替换操作，毕竟用户有可 能只是想写一篇介绍如何使用关键字的文档[16], 此时用户当然不希望 Subversion 把 示例中的关键字锚点都替换掉.

为了告诉 Subversion 是否要替换某个文件中的关键字， 我们要再次使用 与属性有关的子命令. 设置在文件上的属性 `svn:keywords` 决定了文件中的哪些关键词将会被替换，属性值是空格分隔的关键字名或别名列表.

举个例子，假设用户一个叫作 *weather.txt* 的文件，文件的内容是:

```
Here is the latest report from the front lines.
$LastChangedDate$
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

如果文件上没有设置属性 `svn:keywords`, Subversion 就不会对文件做什么特别的操作. 现在开启关键字 `LastChangedDate` 的替换.

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

文件 *weather.txt* 此时含有未被提交的属性修改，但文件的内容并没有发生变化 (除非用户在设置属性之前又修改了文件). 注意 文件还包含了关键字 `Rev` 的锚点，但 `svn:keywords` 的属性值并没有包含关键字 `Rev` . 如果文件中没有要被替换的关键字，或者关键字没有出现在 `svn:keywords` 的属性值里，Subversion 就不会真正地替换 关键字.

[16] …或者是书中 的一节…

属性修改提交后，Subversion 会立刻更新工作副本里的文件，将其中的关键 字替换成对应的文本. 关键字锚点将会出现替换后的文本，替换的结果仍然包含 关键字的名字以及两边的美元符 ($). 因为 `svn:keywords` 的属性值里没有包含对应的关键字，所以 `Rev` 没有被替换.

注意我们把属性 `svn:keywords` 设置成 `Date Author`，而关键字锚点则写成了 `$LastChangedDate$`，但仍然得到了正确的结果，这是因为 `LastChangedDate` 是 `Date` 的别名.

```
Here is the latest report from the front lines.
$LastChangedDate: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

如果其他人向 *weather.txt* 提交了新的修改，自己 工作副本里的文件不会自动更新—直到用户显式地更新了工作副本，此时，*weather.txt* 的关键字会被重新替换，以反应最新的修改 时间.

作为锚点出现在文件里的关键字都区分大小写：用户必须使用大小写正确的 关键字. 同样也要注意属性 `svn:keywords` 的值也区分大 小写. 为了保持向后兼容，某几个关键词是不区分大小写的，但不建议用户使用 这个特性.

Subversion 定义了几个支持替换的关键字，下面列出这些关键字，其中一些 关键字拥有别名：

`Date`

这个关键字描述了仓库中的文件已知的最后一次被修改的时间，格式类似 于 `$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $` . 它的别名是 `LastChangedDate`. 和关 键字 `Id` 不同 (`Id` 使用 UTC 时间)，`Date` 会按照本地时区显示日期.

`Revision`

这个关键字描述了仓库中的文件已知的最后一次被修改的版本号，显 示格式 类似于 `$Revision: 144 $`，它的别名有 `LastChangedRevision` 和 `Rev`.

`Author`

这个关键字描述了仓库中的文件已知的最后一次是被谁修改的，显示 格式类似 于 `$Author: harry $`，它的别名是 `LastChangedBy`.

`HeadURL`

这个关键字描述了仓库中的文件的最新版本的完整 URL 路径， 显示格式 类似于 `$HeadURL: http://svn.example.com/repos/trunk/calc.c $`，它的别名是 `URL`.

`Id`

这个关键字是几个关键字的组合，它显示的内容类似于 `$Id: calc.c 148 2006-07-28 21:30:43Z sally $`，例子的意思是文件 *calc.c* 最后一次修改是在 2006 年 7 月 28 日，版本号 148，作者是 `sally`. `Id` 使用 UTC 时间，而 `Date` 使用本地时区.

`Header`

这个关键字和 `Id` 类似，但是增加了 `HeadURL` 的内容，看起来就像 `$Header: http://svn.example.com/repos/trunk/calc.c 148 2006-07-28 21:30:43Z sally $`.

在介绍关键字时，（隐式的或显式的）用到了形容词 “已知的”，这是因为关键字替换是一个客户端操作，客户端只能知道最近一次更新工作副本 时从仓库中获取的信息. 如果工作副本一直得不到更新，即使仓库中的文件已经 修改了，工作副本里的关键字也不会被替换成更新的信息.

> **$GlobalRev$ 在哪儿?**
>
> 新手常常弄不清楚 `$Rev$` 的工作方式. 因为 Subversion 有一个全局增长的版本号，很多新用户以为 `$Rev$` 显示的是版本号的全局最大值. 但实际上 `$Rev$` 被替换成文件最后一次被 修改 时的版本号，只要理解了这点就不会再有疑惑. 但是还有一个问题没有解决 —缺少了关键字的支持，用户应该如何在文件中自动获取全局最新的版本 号?
>
> 完成这项工作需要借助另一个工具，*svnversion*. *svnversion* 遍历工作副本，然后输出最新的版本号. 利用 这个命令，再加上一些额外的工具，用户就可以把版本号信息插入到文件中，关于 *svnversion* 的更多信息，见 svnversion 参考手册—Subversion 工作副本版本信息.

除了前面几个预定义的关键字，Subversion 1.8 允许用户定义新的关键字. 为了定义一个关键字，给属性 `svn:keywords` 的值添加 新的记号，记号的格式是 *`MyKeyword= FORMAT`*，其中 *`MyKeyword`* 是关键字的名字（关键字锚点需要），*`FORMAT`* 是一个格式化的字符串，替换文件中的关 键字时会根据格式字符进行替换.

格式化字符串支持的格式控制符有以下这些:

`%a`

由 `%r` 指定的版本号的作者.

`%b`

文件的 URL 的基本名 (basename).

`%d`

由 `%r` 指定的版本号的日期的短格式.

`%D`

由 `%r` 指定的版本号的日期的长格式.

`%P`

文件相对于仓库根目录的路径.

`%r`

已知的文件最后一次被修改时的版本号（和用来替换 `Revision` 的版本号相同）.

`%R`

仓库根目录的 URL.

%u

文件的 URL.

%_

一个空格符（定义关键字的字符串中不能包含字面空格）.

%%

一个百分号 (%).

%H

等价于 %P%_%r%_%d%_%a.

%I

等价于 %b%_%r%_%d%_%a.

可以看到，很多单独的格式控制字符所表示的信息与预定义的关键字所表示 的信息相同，但是自定义关键字允许用户得到更灵活和更丰富的信息. 比如说，用户希望有一个关键字能被替换成文件在仓库里的相对路径，以及最后一次修改 文件的版本号，此时就需要自定义一个关键字：

```
$ svn pset svn:keywords "PathRev=%P,%_r%r" calc/button.c
property 'svn:keywords' set on 'button.c'
$
```

接下来用户要把关键字锚点插入到文档的适当位置，在这个例子里，关键字 锚点要写成 $PathRev$. 提交修改后，文件中原来显示 $PathRev$ 的文本，变成了 $PathRev: trunk/calc/button.c, r23 $.

如果关键字替换后的内容超过了 255 个字符，Subversion 会自动截断过 长的部分. 自定义关键字的名字如果超过了 255 个字符，Subversion 会自动 忽略超出的部分.

用户还可以为替换后的字符串指定一个固定的长度. 在关键字名字后面加 两个冒号 (::)，然后是一定个数的空格，这样就指定了 一个固定长度. 当 Subversion 准备替换关键字时，如果发现锚点指定了一个固 定长度，Subversion 就只会替换空格部分. 如果替换后的字符串不够长，不足 的部分就会用空格填充；如果替换后的字符串不过长，字符串就会被截断，并在截断的地方放置一个 # 字符.

比如说，你有一个文档，文档把 Subversion 的关键字按照表格的样式进行 排版，如果使用原来形式的关键字替换语法，替换前的文件内容看起来就像：

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

现在看起来表格的格式还挺工整的，但是提交后（开启了关键字替换功能），文件的内容就变成了：

```
$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
```

替换后的效果令人感到失望，用户可能会忍不住手工地调整每一行没对齐的 文本，但是实际上只要关键字的值占用相同的宽度，格式就不会被打乱. 如果版本 号增长到比较长的位数（例如从 99 增长到 100)，或者有一个名字很长的用户提 交了修改，文件的版式就得重新调整. 如果用户使用的 Subversion 版本大于等于 1.2，就可以使用具有固定长度的关键字语法，为了使用这种关键字语法，把文件 的内容改成：

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

提交修改，这次 Subversion 会注意到文件中使用了具有固定长度的关键字语 法，替换后，字段的长度保持不变—较短的 `Rev` 和 `Author` 使用空格填充不足的部分，较长的 `Date` 被井字符截断：

```
$Rev:: 13             $:  Revision of last commit
$Author:: harry       $:  Author of last commit
$Date:: 2006-03-15 0#$:  Date of last commit
```

固定长度的关键字替换在以下场景非常方便：(1) 文件把数据放在长度固定 的字段里；(2) 除了格式的本地应用程序外，其他程序难以修改某些数据字段 的存放大小. 当然，如果涉及到二进制文件格式，用户必须非常小心，关键字替换（无论是长度是否固定）不能破坏格式的完整性. 虽然这听起来很容易，但是对于 现在流行的大多数二进制格式而言，实际做起来可能会非常困难，绝不是稍微用 点心就能对付过去的.

注意关键字字段的长度以字节为单位，在处理多字节字符时可能会出问题. 比如说，用户名如果包含 UTF-8 字符，截断可能会发生在组成一个字符的多个 字节之间. 从字节来看这可能只是一个非常普通的截断，但是在解释成 UTF-8 字符就会产生错误，很可能会产生乱码. 有些应用程序在打开这种含有错误编 码的文件时会认为整个文件已经损坏，拒绝对文件进行进一步操作. 所以在限 制关键字的长度时，注意避免在字符当中发生截断.

稀疏目录

默认情况下，大多数 Subversion 操作在处理目录时会采用递归的方式，比如说，*svn checkout* 会检出仓库指定区域内的所有文件与目录. Subversion 1.5 引入了一个新特性：稀疏目录 (*sparse directories*，或 浅检出 (*shallow checkouts*)). 和完整的递归 操作相比，新特性允许用户更加轻浅地检出工作副本—或工作副本的一部分，以后仍然还能访问到原来未被检出的文件与子目录.

举个例子，假设我们有一个仓库，仓库中存放的是拥有宠物的家庭成员（这个 例子确实有点奇怪），普通的 *svn checkout* 操作会得到 一整棵目录树的工作副本：

```
$ 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.
$
```

现在我们再次执行检出操作，不过这次要求 Subversion 只检出最上层的目 录，不包括其中的文件与子目录：

```
$ svn checkout file:///var/svn/repos mom-empty --depth empty
Checked out revision 1
$
```

注意我们这次给命令 *svn checkout* 加了一个选项 `--depth`. 很多子命令都支持这个选项， 选项的意义类似于 `--non-recursive` (`-N`) 和 `--recursive` (`-R`). 实际上，Subversion 希望选项 `--depth` 最终能超过并替换掉这两个旧选项. 对新手来说，`--depth` 拓宽了用户能够指定的操作深度，增加了 一些原来不支持（或支持地不一致）的深度. 下面是用户可以使用的几种深度值：

`--depth empty`

只包含操作的直接目标，不包括其中的文件或子目录.

`--depth files`

只包含操作的直接目标及其中的直接子文件.

`--depth immediates`

包括操作的目标自身，及它的直接子文件与直接子目录，子目录为空.

`--depth infinity`

包括目标自身，及它的所有子文件与子目录，子目录的子文件与 子目录，等等.

当然，如果仅仅是把两个选项合并成一个选项，那就没必要花费整整一节的笔墨 介绍它，幸运的是远不止选项合并这么简单. 深度的概念不仅延伸到 Subversion 客户端执行的操作，同时还描述了工作副本的 周围深度 (*ambient depth*)，它是工作副本为项目记录的深度. 深度的关键之处在于它是 “粘着” (sticky) 的，工作副本记住了用户为每一个项目 指定的深度，在用户显式地修改之前，项目的深度不会发生变化. 默认情况下，不管文件的深度设置是什么样的，Subversion 的命令只会操作工作副本中已有 的项目.

可以用命令 *svn info* 查看工作副本的周围深度，如果周围深度是除了无限递归外的其他内容，*svn info* 就会显示一行描述深度值的信息：

```
$ svn info mom-immediates | grep "^Depth:"
Depth: immediates
$
```

前面的两个例子演示了深度值 `infinity`（*svn checkout* 的默认行为）和 `empty` 的效果，现在看一下其他深度的例子：

```
$ 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.
$
```

和 `empty` 相比，这些深度会得到更多的内容，但和 `infinity` 相比，会得到更少的内容.

我们已经介绍了 *svn checkout* 如何利用选项 `--depth`，但读者会看到除了 *checkout*，还有很多子命令也支持 `--depth`. 在这些命令中，指定深度 将操作的作用域限制在某一层次上，非常类似老选项 `--non-recursive` 和 `--recursive` 的行为. 这就意味着当我们操作 一个处在某个深度上的工作副本时，可以执行一个深度更浅的操作. 实际上，我 们可以更一般地说：对于一个给定的，处于任意周围深度（深度可以是混合的）的工作副本，和一个指定了操作深度（或使用默认值）的 Subversion 命令，命令 将保持工作副本的周围深度不变，同时将操作的作用域限制在所给定（或默认的）的深度上.

除了选项 `--depth`，命令 *svn update* 和 *svn switch* 还支持第二种与深度有关的选项 `--set-depth`，它可以修改工作副本中项目的粘着深度. 现在看 一下如何使用 `svn update --set-depth` *`NEW-DEPTH TARGET`*，把原来深度为 `empty` 的工作副本逐渐加深：

```
$ svn update --set-depth files mom-empty
Updating 'mom-empty':
A    mom-empty/kittie1.txt
A    mom-empty/doggie1.txt
Updated to revision 1.
$ svn update --set-depth immediates mom-empty
Updating 'mom-empty':
A    mom-empty/son
A    mom-empty/daughter
Updated to revision 1.
$ svn update --set-depth infinity mom-empty
Updating '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.
$
```

随着深度的不断加深，每次更新，仓库都会给我们传来更多的数据.

在上面的例子里，我们都是在工作副本的根目录执行操作，改变周围深度，其 实我们可以独立地修改工作副本的 任意 子目录的周围 深度. 认真使用这项特性就可以在工作副本中只保留感兴趣的部分，而忽略那些 不重要的部分（所以称为 “稀疏” 目录），下面的例子展示了典型 的用法：

```
$ 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
Updating 'mom-empty/son':
A    mom-empty/son
Updated to revision 1.
$ svn update --set-depth empty mom-empty/daughter
Updating 'mom-empty/daughter':
A    mom-empty/daughter
Updated to revision 1.
$ svn update --set-depth infinity mom-empty/daughter/granddaughter1
Updating '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.
$
```

幸运的是，在一个工作副本里出现如此复杂的周围深度并不会使用户与工作 副本的交互也变得复杂. 用户仍然可以像以往那样修改文件，显示修改，撤消或 提交修改，而不用给相关命令提供新的选项 (包括 `--depth` 和 `--set-depth`). 当没有指定深度时，*svn update* 也能正常工作—命令根据各个项目的粒着深度更新工作副本里 已有的文件和目录.

读者心里可能在想 “那么，我什么时候会用到稀疏目录呢?” 用到稀疏目录的一种场景是仓库的布局比较特殊，尤其是许多相关的项目模块都 在同一个仓库中分别占据一个单独的目录 (例如 *trunk/project1* , *trunk/project2*, *trunk/project3* 等)，但是用户可能只关心其中的部分模块— 比如说项目的主要模块及其依赖模块. 用户可以分别检出他所关心的各个模块的 工作副本，但是这些工作副本之间是互不相交的，如果想同时对它们执行同一个操作 就会很麻烦，必须多次切换目录. 另一种选择是利用稀疏目录特性，检出一个只 包含了感兴趣的模块的工作副本. 首先为模块的公共父目录检出一个深度为 `empty` 的工作副本，然后按照深度 `infinity` 更新感兴趣的模块目录，就像我们在上一个例子中展示的那样. 可以把稀疏目录看成是工作副本中项目的选入系统.

浅检出的原始实现 (Subversion 1.5) 就已经很不错了，但是它不能缩减 工作副本项目的深度，Subversion 1.6 解决了这个问题. 比如说在一个深度 原来是 `infinity` 的工作副本里执行 *svn update --set-depth empty*, 工作副本就会删除除了顶层目录外的所有文件与目录 [17] Subversion 1.6 还 为选项 `--set-depth` 引入的一个新的值: `exclude` . 如果给命令 *svn update* 带上选项 `--set-depth exclude` 会造成被更新的目标从工作副本中完全 删除—如果目标是一个目录，那么目录也会被完全删除，而不是留下一个 空目录. 如果工作副本中用户想保留的东西要比不想保留的东西多，那 `--set-depth exclude` 就能提供很大的方便.

考虑一个包含了几百个子目录的目录，用户想要从工作副本中忽略其中一个 子目录，如果是用 “增量” 的方法得到稀疏目录， 首先先检出一 个深度为 `empty` 的工作副本， 然后显式地把每一个子目录 的深度设置成 `infinity` (使用 `svn update --set-depth infinity`), 除了那个用户 不感兴趣的子目录.

```
$ svn checkout http://svn.example.com/repos/many-dirs --depth empty
…
$ svn update --set-depth infinity many-dirs/wanted-dir-1
…
$ svn update --set-depth infinity many-dirs/wanted-dir-2
…
```

[17] 删除操作是安全的，Subversion 会保留 修改过的或未被版本控制的项目.

```
$ svn update --set-depth infinity many-dirs/wanted-dir-3
…
### and so on, and so on, ...
```

这可能会非常枯燥，尤其是工作副本中还不存在存根目录供用户处理．另一个问题是如果有人在顶层目录下创建了一个新的子目录，当用户更新 工作副本时将看不到这个新的子目录，这应该不是你想要的效果．

从 Subversion 1.6 开始，你有了另一种选择．首先检出一个完整的目录，然后在不兴趣的目录上执行 `svn update --set-depth exclude`

```
$ svn checkout http://svn.example.com/repos/many-dirs
…
$ svn update --set-depth exclude many-dirs/unwanted-dir
D         many-dirs/unwanted-dir
$
```

和第一种方法相比，使用第二种方法后在工作副本里留下的数据是相同的，但是如果有新的子目录被提交到仓库中，更新工作副本时仍然可以看到．第二种 方法的缺点是一开始要检出后来不用的子目录，如果子目录过于庞大，大到磁盘 无法容纳（可能这就是用户不想把它检出到工作副本里的原因）．

虽然这个功能—从工作副本中排除已有的项目—由 *svn update* 完成，但读者可能已经注意到 `svn update --set-depth exclude` 的输出和通常的 *svn update* 的不太一样．排除是一个完全客户端的操作，但命令的输出却不太符合这个事实．

如果出现这种情况，你可能需要一个折衷的方法．首先，使用 `--depth immediates` 检出顶层目录，然后用 `svn update --set-depth exclude` 排除不感兴趣的子目录，最后，把剩下的子目录的深度设置成 `infinity`，因为子目录都已 经出现在本地了，所以应该会容易一点．

```
$ svn checkout http://svn.example.com/repos/many-dirs --depth immediates
…
$ svn update --set-depth exclude many-dirs/unwanted-dir
D         many-dirs/unwanted-dir
$ svn update --set-depth infinity many-dirs/*
…
$
```

再说一次，这种方法得到的工作副本里的数据和前两种方法完全相同，当有 新的文件或目录提交到顶层目录时，更新操作按深度 `empty` 把文件或目录更新到本地，接下来你可以决定针对新出现的项目应该采取什么 操作：是把深度扩展到 `infinity`，还是把它排除．

锁

Subversion 的数据合并算法与 复制–修改–合并 模型之间的关系就像水和 船：水能载舟，亦能覆舟—尤其是当 Subversion 尝试解决冲突时，合并 算法的表现至关重要．Subversion 自身只提供了一种合并算法：三路差异比较算 法的智能足够在行的级别上处理数据．作为补充，Subversion 允许用户指定外部 的差异比较工具（在 “外部三路差异比较工具”一节 和 “外部合并工具”一节 介绍），这些外 部工具可能比 Subversion 工作得更好，比如在单词或字符的级别上比较差异．但是这些工具和 Subversion 的算法通常只能处理文本文件，在面对非文本文件 时，现实就残酷多了．如果用户无法找到支持非文本文件合并的工具，复制–修改–合并 模型就不再适用．

介绍一个现实生活中可能会遇到的例子. Harry 和 Sally 是同一个项目的 图片设计师, 为汽车保险部门设计一款海报. 海报的中心是一辆汽车, 海报的格式 是 PNG. 海报的布局已经基本确定, Harry 和 Sally 将一辆 1967 年淡蓝色 Ford Mustang 照片放在海报中央, 车的左前侧保险杠有一点凹陷.

项目计划有所改动, 导致车身的颜色需要修改, 于是 Sally 把工作副本更 新到 HEAD, 打开图片编辑软件, 将车身的颜色改成樱桃红. 同时, Harry 觉得车的毁坏程度应该更严重一些, 这样效果更好, 于是他也把自己 的工作副本更新到 HEAD, 在车挡风玻璃上增加了一些裂痕. 就在 Harry 提交修改后, Sally 也提交了自己的修改, 显然, Subversion 会拒绝 Sally 的提交.

现在麻烦来了. 如果 Harry 和 Sally 编辑的是文本文件, 此时 Sally 只 要更新工作副本, 然后就可以再次尝试提交, 最差的情况不过是两人都修改了文件 的同一区域, 而 Sally 必须手工地解决冲突. 但海报不是文本文件, 它是二进制 的图片, 没有哪一款软件可以聪明到能够把两张图片合并成一张, 得到一辆 樱桃红的, 挡风玻璃上有裂痕的汽车.

如果 Harry 和 Sally 是串行地修改图片, 那事情就会顺利很多—比如 Sally 修改车身颜色并提交后, Harry 再去添加裂痕, 或者是 Sally 等到 Harry 添加裂 痕后再去修改车身颜色. “复制-修改-合并 解决方案”一节 已经说过, 如果 Harry 和 Sally 之间进行了充分的沟通, 这种问题大部分都可 以迎刃而解. 但是版本控制系统也是一种沟通的形式, 由软件来保证工作的串行 化并不是一件坏事, 反而效果更好, 效率更高. 正是基于这点考虑, Subversion 实现了 加锁-修改-解锁 模型. Subversion 的 锁定 特性和其他版本控制系统的 “保留检出” 比较类似.

Subversion 的锁定特性是为了最大程度地减少时间和精力的浪费. 允许用户 独占地修改仓库中的文件, 保证了用户在不支持合并的修改上所花费的精力不会 被浪费—他的修改总能提交成功. 并且, Subversion 把对象正在被锁定的 事实告诉给了其他用户, 其他用户就可以知道该对象正在被修改, 也就不会把时 间浪费在无法成功提交与合并的修改上.

当我们谈到 Subversion 的锁定特性时, 实际上说的是多种不同行为的集合, 包括锁定文件的的能力 [18] (获得独占修改文件的权利), 解锁一个文件 (放弃独占修改 文件的权利), 查看哪些文件被哪些用户锁定, 为锁定的文件添加注释 (强烈建议) 等, 所有的这些都会在本节进行详细介绍.

“锁” 的多种涵义

本节及书中的其他地方, 单词 “锁 (lock) ” 和 “ 加锁 (locking)” 描述一种用于实现用户间互斥的机制, 避免提交 产生碰撞. 不幸的是, Subversion 还有其他种类的 “锁” 需要 注意.

管理锁 (*administrative locks*), 由 Subversion 内部使用, 用于防止多个客户端在 操作同一个工作副本时产生碰撞. *svn status* 输出中 第三列的 `L` 指的就是管理锁, 命令 *svn cleanup* 会删除管理锁, 见 “有时候你需要的只是清理一下”一节.

使用老的 Berkeley DB 作为仓库后端的管理员需要了解 *数据库锁*, 它由 Berkeley DB 数据库内部使用, 防止多个程序在访问数 据库时产生碰撞. 这种类型的锁如果在遭遇错误出现了不应该出现的持久化 状态, 将导致仓库 “楔形化” (见 “Berkeley DB 恢复”一节.

另外, *svnsync* 锁 可以让命令 *svnsync* 的多个实例在操作仓库的同一个镜像时保持互斥. 这种锁类型通过版本号的 `svn:sync-lock` 属性实现, *svnsync* 的介绍见 “使用 svnsync 复制仓库”一节.

最后还有一个 *svnrdump* 锁. 它和 svnsync 锁非常 类似, 只是它和命令 *svnrdump load* 相关, *svnrdump* 的介绍见 “使用 svnrdump 迁移仓库数据”一节.

除非出现问题, 否则的话用户可以不用考虑后两种锁. 贯穿全书, “ 锁” 都指的是第一种锁, 除非显式地或者从上下文可以明显地推断 出锁的其他类型.

[18] Subversion 目前不支持锁定目录.

创建锁

在 Subversion 仓库里，一个 锁 (*lock*) 就是一段元数据，它赋予了一个用户独占修改文件的权利，该 用户被认为是 锁的持有者 (*lock owner*). 仓库 负责管理锁，具体来说就是锁的创建，实施和删除. 如果有一个提交试图修改或 删除被锁定了的文件（或删除文件的某个父目录），仓库就会要求客户端提供 2 项信息—一是执行提交操作的客户端已被授权为锁的所有者，二是提供了 锁令牌，表明客户端知道它用的是哪一个锁.

为了演示锁的创建，我们再以海报设计作为例子. Harry 决定修改一个 JPEG 图片，为了防止其他用户在他完成修改之前向该文件提交修改，他使用命令 *svn lock* 锁定了仓库中的文件：

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

上面的例子展示了一些新东西. 首先，Harry 向命令 *svn lock* 传递了选项 `--message` (`-m`)，和命令 *svn commit* 类似，*svn lock* 支持注释—借助选项 `--message (-m)` 或 `--file` (`-F`)—注释描述了锁定文件的原因. 然而，和 *svn commit* 不同的是 *svn lock* 不 会通过启动文本编辑器来要求用户输入注释，注释是可选的，但是为了更好地 与其他用户沟通，建议输入注释.

第二，尝试加锁成功了，这就是说文件之前未被锁定，而且 Harry 工作副本 里的文件是最新的. 如果 Harry 工作副本里的文件是过时了的，仓库将会拒绝 加锁请求，要求 Harry 执行 *svn update* 并重新执行加 锁命令. 如果文件已经处于加锁状态，加锁命令也会失败.

如果加锁成功，*svn lock* 会输出确认信息，从现在开始，文件已经被锁定的事实会体现在 *svn status* 和 *svn info* 的输出信息里.

```
$ svn status
     K  banana.jpg

$ svn info banana.jpg
Path: banana.jpg
Name: banana.jpg
Working Copy Root Path: /home/harry/project
URL: http://svn.example.com/repos/project/banana.jpg
Relative URL: ^/banana.jpg
Repository Root: http://svn.example.com/repos/project
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 2198
Node Kind: file
Schedule: normal
Last Changed Author: frank
Last Changed Rev: 1950
Last Changed Date: 2006-03-15 12:43:04 -0600 (Wed, 15 Mar 2006)
Text Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Properties Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Checksum: 3b110d3b10638f5d1f4fe0f436a5a2a5
Lock Token: opaquelocktoken:0c0f600b-88f9-0310-9e48-355b44d4a58e
Lock Owner: harry
Lock Created: 2006-06-14 17:20:31 -0500 (Wed, 14 Jun 2006)
Lock Comment (1 line):
```

```
Editing file for tomorrow's release.

$
```

svn info 在执行时不会与仓库通信，但是它仍然可 以显示锁令牌，说明了一个很重要的信息：它们被缓存在工作副本里．锁令牌 的存在非常重要，它向工作副本提供了使用锁的授权．并且， *svn status* 在文件名的旁边显示一个 K (locKed 的缩写)，表示该文件存在锁令牌．

关于锁令牌

锁令牌不是认证令牌 (authentication token)，而是 授权 令牌 (authorization token)．令牌不是 一个受保护的秘密，实际上，任意一个用户都可以用 `svn info` *`URL`* 发现锁的一个独一无二的令牌．只有当一个 锁令牌处在工作副本里时它才是特殊的，这说明了锁是在这个特定的工作 副本里被创建出来的，仅仅验证锁的所有者并不能完全避免意外．

比如说，你在办公室的电脑上锁定了一个文件，但是还没有提交修改就下 班回家了，如果你想在家里完成提交就会失败，因为仅仅被授权为锁的所有 者并不能保证提交成功．换句话说，锁令牌阻止了一部分的 Subversion 相关 软件破坏另一部分的工作．(在我们的例子里，如果你确实需要从另一个工作 副本提交修改，就必须先 *破坏* 锁，然后重新锁定文件)．

因为 Harry 已经锁定了文件 *banana.jpg*，所以 Sally 不能提交和 *banana.jpg* 有关的修改：

```
$ svn delete banana.jpg
D         banana.jpg
$ svn commit -m "Delete useless file."
Deleting       banana.jpg
svn: E175002: Commit failed (details follow):
svn: E175002: Server sent unexpected return value (423 Locked) in response to
DELETE request for '/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-df4224ddc
35d/banana.jpg'
$
```

修改完香蕉的黄色阴影后，Harry 可以向仓库提交修改，这是因为他被授 权为锁的拥有者，而且工作副本包含了正确的锁令牌：

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

注意提交完成后，*svn status* 显示锁令牌不再出现 在工作副本里，这是 *svn commit* 的标准行为— 它搜索工作副本 (如果提供了目标列表，则搜索该列表) 的本地修改，并将所 有遇到的锁令牌作为提交事务的一部分发送给服务器，如果提交成功，仓库中 所有涉及到的锁都会被释放—*即使是未被提交的文件上的锁 也会被释放*．这是为了防止粗心的用户持锁时间过长．如果 Harry 随意地把目录 *images* 下的 30 个文件都锁定了 (因为他 不确定哪些文件需要修改)，而他只修改了其中 4 个文件，当他执行完 `svn commit images` 后，所有 30 个文件的锁都会被 释放．

为 *svn commit* 添加选项 `--no-unlock` 就不会在提交成功后自动释放锁，适用选项的场景是用户需要多次 提交修改. 你可以通过运行时配置选项 `no-unlock`（见 “运行时配置区域”一节）把不自动释放锁设置成默认行为.

当然，锁定文件后并不要求一定要向该文件提交修改才能释放锁，用户可以在 任何时候用命令 *svn unlock* 释放文件上的锁：

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

发现锁

如果由于其他用户锁定了文件而导致提交失败，获取有关锁的信息非常方便，最简单的方式是执行 `svn status -u`:

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

在这个例子里，Sally 不仅可以看到 *foo.h* 是过时了的，而且他打算提交的两个文件中，有一个在仓库中是被锁定了的. 字符 `O` 表示 “其他” (“Other”), 意思是说 文件被其他用户锁定了，如果 Sally 试图提交，*raisin.jpg* 上的锁会阻止提交成功. Sally 想知道是 谁，在什么时候，因为什么原因锁定了文件，*svn info* 可以 回答他的问题：

```
$ svn info ^/raisin.jpg
Path: raisin.jpg
Name: raisin.jpg
URL: http://svn.example.com/repos/project/raisin.jpg
Relative URL: ^/raisin.jpg
Repository Root: http://svn.example.com/repos/project
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.
$
```

svn info 除了可以检查工作副本里的项目外，也可以 检查仓库里的项目. 如果传递给 *svn info* 的参数是一个 工作副本路径，那么缓存在工作副本里的所有信息都会显示出来；只要显示的 信息中提到了锁，那就说明工作副本持有一个锁令牌（如果文件是被其他用户 或者是在另一个工作副本里锁定的，那么在一个工作副本路径上执行 *svn info* 将不会显示关于锁的任何信息）. 如果传递给 *svn info* 的是一个 URL，输出的信息反映了仓库中的 项目的最新版，信息中提到关于锁的任何信息都是在描述项目的当前加锁情况.

在我们的例子里，Sally 可以看到 Harry 在 2 月 16 日锁定了文件 *raisin.jpg*，原因是 “Need to make a quick tweak to this image”. 现在已经 6 月了，Sally 怀疑 Harry 忘记给文件解锁，她可能会打电话给 Harry，向他抱怨，让他马上释放锁. 如 果联系不到 Harry，她可能会强行地破坏锁，或者让管理员来帮她解决.

破坏与窃取锁

锁并非是不可侵犯的—在 Subversion 的默认配置状态下，除了创建锁的 用户可以释放锁之外，任意一个用户也可以释放锁. 如果释放锁的用户不是锁 的创建者，我们把这种行为叫作 *破坏锁* (*breaking the lock*).

对于管理员来说，破坏锁非常简单. 命令 *svnlook* 和 *svnadmin* 可以直接从仓库中显示与移除锁 (关于 *svnlook* 和 *svnadmin* 的更多信息，见 “管理员工具箱”一节).

```
$ 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'.
$
```

Subversion 还允许用户通过网络破坏其他用户的锁，为了破坏 Harry 设置 在 *raisin.jpg* 上的锁，Sally 要给 *svn unlock* 加上选项 `--force`:

```
$ svn status -u
M               23   bar.c
M    O          32   raisin.jpg
        *       72   foo.h
Status against revision:     105
$ svn unlock raisin.jpg
svn: E195013: '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: warning: W160039: Unlock failed on 'raisin.jpg' (403 Forbidden)
$ svn unlock --force http://svn.example.com/repos/project/raisin.jpg
```

```
'raisin.jpg' unlocked.
$
```

在上面的例子里，Sally 第一次尝试解锁失败了，因为她直接在工作副本 的 *raisin.jpg* 上执行 *svn unlock*，而她的工作副本里并没有锁令牌．为了直接从仓库中删除锁，她需要向 *svn unlock* 传递一个 URL 参数．增加 URL 参数后的第 一次尝试失败了，因为她没有被授权为锁的所有者（而且她也没有锁令牌）．但 是增加了选项 `--force` 后，锁成功的被打开（破坏）了．

仅仅把锁破坏掉可能还不够．Sally 除了要打开 Harry 忘记打开的锁之外，她 还想重新锁定文件，以便自己对文件进行编辑．她可以先用带上选项 `--force` 的 *svn unlock* 把锁打开，然后再 用 *svn lock* 锁定文件．但是在两个命令之间可能会有其他用户锁定了文件．更简单的做法是 *窃取* (*steal*) 锁，它是把锁的破坏与重新加锁合并成一个 原子操作，具体的做法是给 *svn lock* 加上选项 `--force`:

```
$ svn lock raisin.jpg
svn: warning: W160035: Path '/project/raisin.jpg' is already locked by user 'h
arry' in filesystem '/var/svn/repos/db'
$ svn lock --force raisin.jpg
'raisin.jpg' locked by user 'sally'.
$
```

无论锁是被破坏还是被窃取，Harry 都会感到惊讶．Harry 的工作副本仍然包 含原来的锁令牌，但是锁却不存在了，此时把锁令牌称为 *失效的* (*defunct*)，锁令牌对应的锁要么被 破坏（不在仓库里），要么被窃取（被另一把不同的锁替换掉）．不管是哪一种 情况，Harry 都可以用 *svn status* 查看详情：

```
$ svn status
     K  raisin.jpg
$ svn status -u
     B          32   raisin.jpg
Status against revision:     105
$ svn update
Updating '.':
  B  raisin.jpg
Updated to revision 105.
$ svn status
$
```

如果仓库的锁被破坏了，`svn status --show-updates (-u)` 会在文件的旁边显示字符 `B` (Broken)．如果有一把新锁出现在原来的位置上，显示的就是字符 `T` (sTolen)．最后，*svn update* 会从工作副本中移除所有的失效锁．

加锁策略

不同的系统对锁的严格程度都有不同的理解．有些人认为应该不惜代价地 维护锁的持有，只有管理员或创建锁的原始用户才能释放锁，他们的观点是 如果任何人都可以破坏锁，就会产生混乱，也就实现不了加锁的目的．另一些 人认为锁是一个重要的沟通工具，如果用户频繁地破坏其他人的锁，那就说明 是团队管理出现了问题，而这种问题与软件无关．

Subversion 默认使用比较温和的做法，但允许管理员通过钩子脚本，创建 更严格的加锁策略．特别地，钩子 pre-lock 和 pre-unlock 允许管理员决定什么时候才能允许创建 锁与释放锁，取决于一个锁是否已经事先存在，这两个钩子还可以决定一个 特定的用户是否可以破坏或窃取锁．还可以使用钩子 post-lock 和 post-unlock 在加锁与解锁后发送通知邮件．关于钩子的更多信息，见 “实现仓库钩子”一节．

锁通信

我们已经介绍了如何使用 *svn lock* 和 *svn unlock* 完成锁的创建，释放，破坏与窃取. 锁实现了文件 的串行提交，但是我们应该如何防止浪费时间?

比如说，Harry 锁定了一个图片文件，然后开始编辑. 同时在几英里之外，Sally 也想编辑同一个文件，她忘了执行 `svn status -u`, 所以她完全不知道 Harry 已经锁定了 她要编辑的文件. 她花了几个小时完成了图片的修改，当她试图提交修改时，发现文件被锁定或者工作副本里的文件过时了. 无论如何，她的修改无法与 Harry 的修改合并，两人中必须有一个人要放弃他的工作成果.

Subversion 的解决办法是提供了一种机制，这种机制会提醒用户在开始 修改文件之前，要先锁定文件，这种机制是一个特殊的属性 `svn:needs-lock`. 如果文件设置了该属性（属性值并不重要），Subversion 将试图使用文件系统的权限把文件设置成只读—除非用户显式 地锁定了文件. 如果提供了锁令牌（*svn lock* 的运行结果），文件的权限就变成可读写，如果锁被释放了，文件再次变成只读.

如果图片文件设置了属性 `svn:needs-lock`, 当 Sally 打开并开始修改图片时就会注意到有些地方不对劲：很多程序在以读 写方式打开文件时，如果发现文件是只读的，将会向用户发出警告，并阻止用户 向只读文件保存修改. 这将会提醒 Sally 应该在修改之前先锁定文件，到那时 她就会发现锁已经预先被别人锁定了：

```
$ /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: warning: W160035: Path '/project/raisin.jpg' is already locked by user 'h
arry' in filesystem '/var/svn/repos/db'
$ 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.
$
```

建议用户和管理员把属性 `svn:needs-lock` 设置到 所有不支持合并的文件上，这是养成良好的加锁习惯和避免浪费时间的主要 方法.

注意 `svn:needs-lock` 是一个通信工具，与加锁系统 独立工作. 换句话说，无论是否设置了这个属性，文件都可以被锁定，相反，设置了这个属性，仓库也不会要求在提交时必须提供锁.

不幸的是，这种机制并非毫无缺点. 即使文件设置了属性 `svn:needs-lock`, 只读提醒也可能不会起作用. 有时候应用 程序不够规范，会 “劫持” 只读文件，然后悄无声息地允许用户修改并 保存文件. Subversion 对这种情况无能为力—目前为止还没有什么方法可以 完全替代人与人之间的交流[19]

[19]除非我们可以像瓦肯人那样做到心灵 融合.

外部定义

有时候，构造一个由多个不同的检出所组成的工作副本是很有用的，比如说，用户可能想把多个来自不同位置的子目录放到一个目录里，这些子目录甚至来自不 同的仓库. 用户当然可以手工实现—用 *svn checkout* 创建出嵌套的工作副本结构. 但是如果每个用户都有这种需求，那么所有的用户 都得自己手工构造.

幸运的是，Subversion 支持 外部定义 (*externals definitions*), 外部定义是一个本地目录到仓库目录 URL 的映射. 用户使用属性 `svn:externals` 批量地声明外 部定义，创建或修改属性的命令是 *svn propset* 或 *svn propedit* (见 “操作属性”一节). 属性 `svn:externals` 可以设置在任意一个被版本控制的目录上，属性的值描述了外部 仓库的位置，以及检出到本地时得到的本地目录.

`svn:externals` 的方便之处是一旦目录设置了该属性， 所有检出该目录的用户都会受益. 换句话说， 如果有一个用户已经用外部定义 构造好了一个嵌套的工作副本结构，其他用户就不用再重新做一遍— 当原始的工作副本检出完毕后，Subversion 还会自动检出外部的工作副本.

外部定义里的目标子目录事先 不能 存在— Subversion 在检出外部的工作副本时会自动创建它们.

`svn:externals` 是版本化的属性，如果用户需要 修改一个外部定义，使用普通的属性修改子命令即可. 如果提交了属性 `svn:externals` 的修改，下一次执行 `svn update` 时，Subversion 将会根据修改后的外部定义更新 检出的项目，同样的事情也会发生在其他用户的工作副本里.

因为属性 `svn:externals` 的值由多行文本组成，所以我 们强烈建议用户使用 *svn propedit* (而不是 *svn propset*) 修改属性.

Subversion 1.5 之前的外部定义的格式是一个多行表格，每一行包括子 目录 (相对于设置了属性的目录)，可选的版本号标志，以及一个完全限定的 Subversion 仓库 URL 的绝对路径. 外部定义的一个例子是:

```
$ 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
```

如果用户检出了目录 *calc*, Subversion 会继续检出 外部定义里的项目.

```
$ 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
…
```

从 Subversion 1.5 开始，`svn:externals` 支持一 种新的格式，外部定义仍然是多行文本，但某些信息的顺序与格式发生了变化. 新的语法更加贴近 *svn checkout* 的参数：首先是可选的 版本号标志，然后是外部仓库的 URL，本地目录的相对路径. 注意，这次我们没有 说 “完全限定的 Subversion 仓库的 URL 的绝对路径”，这是因为 新的格式支持相对 URL 和带有限定版本号的 URL. 上面的例子在 Subversion 1.5 里的写法是：

```
$ 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
```

带有限定版本号（见 “限定版本号与实施版本号”一节）的写法是：

```
$ 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
```

用户应该在经过慎重地考虑后再决定要不要在外部定义中显式地指定版本号， 这意味着用户必须决定应该在什么时候抓取哪一个版本的快照到外部目录里. 显式地指定版本号除了可以避免向用户没有权限的仓库提交修改外，当用户把 工作副本回退到之前的版本时，外部定义属性也会回退到当时的版本，外部工作 副本也会根据外部定义进行相应地更新. 对于软件项目而言，这可能会造成旧版 代码构建失败.

对大多数仓库而言，三种格式的外部定义的最终效果都是一样的，它们都有 同样的好处，也都有同样的麻烦. 因为用到了 URL 的绝对路径，如果移动或 复制一个带有外部定义的目录，这并不会对外部定义的检出造成影响（虽然本地 目标子目录的绝对路径会随着目录的重命名而发生变化）. 在某些情况下，这会给 用户造成困扰，比如说，用户有一个名为 *my-project* 的顶层目录，用户在它的其中一个子目录（*my-project/some-dir* ）上设置了外部定义，外部定义指向的是另一个子目录（*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

$
```

现在用户用 *svn move* 重命名 *my-project*，但外部定义仍然指向 *my-project* 的子目 录，即使这个目录已经不存在了.

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

Committed revision 12.
$ svn update
Updating '.':

svn: warning: W200000: Error handling externals definition for 'renamed-projec
t/some-dir/subdir':
svn: warning: W170000: URL 'http://svn.example.com/projects/my-project/externa
l-dir' at revision 12 doesn't exist
At revision 12.
svn: E205011: Failure occurred processing one or more externals definitions
$
```

另外，当使用绝对的 URL 路径时，如果仓库支持多种 URL 模式，这也会产 生问题. 比如说仓库服务器允许任意用户通过 `http://` 或 `https://` 访问仓库，但只允许通过 `https://` 提交修改. 如果用户的外部定义使用了 `http://` 形式的 URL，用户将无法从外部定义创建的工作副本里提交修改. 另一方面，如果仓库服务器只支持 `https://` 形式的 URL，但客户端 只支持 `http://`，那么它将无法检出外部项目. 还要注意，如果用户重定位了工作副本（通过命令 *svn relocate*），外部定义检出的工作副本并 不会 被重定位.

在改善这些问题方面，Subversion 1.5 前进了一大步. 前面已经说过新的 外部定义支持 URL 的相对路径，Subversion 1.5 提供了多种指定相对 URL 路径 的语法.

`../`

相对于设置了 `svn:externals` 的目录的 URL

`^/`

相对于 `svn:externals` 所在的仓库的根 目录

`//`

相对于设置了属性 `svn:externals` 的目录 的 URL 模式

`/`

相对于 `svn:externals` 所在的仓库的根 URL

`^/../REPO-NAME`

相对于和定义了 `svn:externals` 的仓库 处于同一个 `SVNParentPath` 位置下的兄弟仓库

如果把绝对的 URL 路径改成相对路径，之前的例子就可以写成：

```
$ svn propget svn:externals calc
^/sounds third-party/sounds
```

```
/skinproj@148 third-party/skins
//svn.example.com/skin-maker@21 third-party/skins/toolkit
$
```

Subversion 1.6 为外部定义添加了两个增强功能，首先，利用引号和转义字 符，外部工作副本的路径可以包含空格，在此之前如何处理路径中的空格是一 件很麻烦的事情，因为空格被用作外部定义的字段分隔符，现在只需要用双引号 包裹路径，或者用反斜杆 (\) 转义路径中会引起问题的字符．如果外部定义的 URL 部分包含空格，此时应该使用标准的 URL 编码表示空格．

```
$ svn propget svn:externals paint
http://svn.thirdparty.com/repos/My%20Project "My Project"
http://svn.thirdparty.com/repos/%22Quotes%20Too%22 \"Quotes\ Too\"
$
```

Subversion 1.6 还支持为文件设置外部定义．外部文件 (*file externals*) 的配置方式与目录相同，外部文件 将以版本化文件的形式出现在工作副本里．

比如说仓库中有一个文件 */trunk/bikeshed/blue.html*，现在你想把文件在版本号 40 时的版本放在 */trunk/www/* 的工作副本里，作为 *green.html*．

实现这个要求的外部定义是：

```
$ svn propget svn:externals www/
^/trunk/bikeshed/blue.html@40 green.html
$ svn update
Updating '.':

Fetching external item into 'www'
E    www/green.html
Updated external to revision 40.

Update to revision 103.
$ svn status
    X   www/green.html
$
```

可以看到，把文件抓取到工作副本里时，Subversion 在外部文件的左边显示字 符 E，执行 *svn status* 时，在外部文 件的左边显示字符 X．

外部目录可以把工作副本检出到任意深度的子目录内，中间缺失的目录会 被自动创建，而外部文件只能检出到已存在的目录内．

使用 *svn info* 检查外部文件时，可以看到外部文件 URL 与版本号：

```
$ svn info www/green.html
Path: www/green.html
Name: green.html
Working Copy Root Path: /home/harry/projects/my-project
URL: http://svn.example.com/projects/my-project/trunk/bikeshed/blue.html
```

```
Relative URL: ^/trunk/bikeshed/blue.html
Repository Root: http://svn.example.com/projects/my-project
Repository UUID: b2a368dc-7564-11de-bb2b-113435390e17
Revision: 40
Node kind: file
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 40
Last Changed Date: 2009-07-20 20:38:20 +0100 (Mon, 20 Jul 2009)
Text Last Updated: 2009-07-20 23:22:36 +0100 (Mon, 20 Jul 2009)
Checksum: 01a58b04617b92492d99662c3837b33b
$
```

因为外部文件是作为版本化的文件出现在工作副本里，它们可以被修改，如 果引用的是版本号 HEAD 的文件，还可以提交修改，提交后的修改不仅会出现在 外部文件时，还包括被引用的文件．然而在我们的例子里，外部文件被指定了一 个较旧的版本号，所以无法提交成功：

```
$ svn status
M   X   www/green.html
$ svn commit -m "change the color" www/green.html
Sending        www/green.html
svn: E155011: Commit failed (details follow):
svn: E155011: File '/trunk/bikeshed/blue.html' is out of date
$
```

定义外部文件时要始终牢记这点：如果外部定义指向的是一个特定版本号的 文件，将无法提交外部文件的修改．如果用户希望可以提交外部文件的修改，就不要指定除了 HEAD 之外的其他版本号，这与没有指定 版本号是同样的效果．

不幸的是，Subversion 对外部定义的支持仍然不够理想．外部文件与外部 目录都还有不足之外需要完善．比如说外部定义的本地子目录不能包含父目录指示 符 `..`（例如 *../../skins/myskin*）．外部文件不能引用其他仓库的文件，不能直接对外部文件进行移动或删除（但可被 复制），而是应该修改 `svn:externals`．

或许最令人失望的是由外部定义创建的工作副本与主工作副本（属性 `svn:externals` 所在的工作副本）之间是分离的，而且 Subversion 也只能操作非正交的工作副本．也就是说如果你想要提交一个或多个外部工作副本 里的修改，你只能显式地在每个外部工作副本里执行 *svn commit* —在主工作副本内提交并不会影响外部工作副本．

我们已经介绍了 `svn:externals` 旧格式的缺点，以及 Subversion 1.5 的新格式如何改善这些缺点，但是在使用新的格式时注意不要 引入新的问题．举个例子，最新的客户端仍然支持旧的外部定义格式，1.5 版以前 的客户端却不支持新格式．如果用户把所有的外部定义格式都更新成新格式，那 就相当于强迫所有的用户都要把客户端更新成最新版．同时还要注意外部定义里的 `-r`*NNN* 部分—旧格式把 它作为限定版本号，而新格式把它作为实施版本号（除非显式指定，否则使用 HEAD 作为限定版本号，限定版本号与实施版本号的区别见 “限定版本号与实施版本号”一节）．

外部工作副本是一个完全自给自足的工作副本，和普通的工作副本没有 任何区别．这是一个很方便的特性，允许用户独立地对外部工作副本进行操作，而不会受到主工作副本（属性 `svn:externals` 所在的工 作副本）的影响，但要注意不要无意间修改工作副本而产生一些微妙的问题．比如说外部定义可能指定了外部工作副本的版本号，如果用户直接在外部工作 副本里执行 *svn update*，Subversion 将允许操作执行，这会导致外部工作副本的版本号与外部定义指定的版本号不一致．如果主工作 副本期望外部工作副本具有特定的内容，那么使用 *svn switch* 把外部工作副本（或其中的一部分）切换到另一个 URL 也会造成 类似的问题．

除了 *svn checkout, svn update, svn switch* 和 *svn export* 外（这些子命令实际上管理了 互不相交的 (*disjoint*)（或者说互相分离的），检出了外部 定义的子目录），*svn status* 也可以识别外部工作副本. *svn status* 为外部 工作副本所在的子目录显示字符 X, 然后递归地显示外部 工作副本内的各个项目的状态. 为子命令添加选项 `--ignore-externals` 将会禁止子命令处理外部定义.

变更列表

对开发人员而言，有时候可能会遇到这样一种情况：在某些代码上完成 多个不同的修改. 这并非由于糟糕的工作计划，因为开发人员常常在阅读某一部分的代码时，发现另 一部分代码的问题，又或许是开发人员把一个大修改拆分成几个逻辑性更强的小 修改，而这几个小修改还没有全部完成. 很多时候，这些小修改不能完全包含在一 个模块里，修改之间也不能安全地隔开，修改可能有重叠，或修改了同一模块 的不同文件，或修改了同一个文件的不同行.

开发人员可以采用不同的方法对这些在逻辑上分开的修改进行组织. 有的人 使用单独的工作副本保存未完成的修改，其他人可能会创建短期的特性分支，还 有的人会使用 *diff* 和 *patch* 来备份 与还原未提交的修改，每一个修改都对应一个补丁文件. 每一种方法都有各自的 优缺点，而且修改的细节会在很大程度上影响对修改进行区分的方法.

Subversion 提供了一种新方法：变更列表 (*changelists*). 变更列表基本上就是一些应用到工作副本文件上 的任意标签（每个文件上最多只能有一个标签），用来表示多个互相关联的文件的 共同目的，经常使用谷歌软件的用户对此比较熟悉. 比如说 谷歌邮箱 [http://mail.google.com/] 并没有提供传统的基于文件夹 的邮件组织形式，用户可以把任意的标签应用到邮件上，如果有多个邮件的标签 相同，就可以说它们是同一个组的，查看具有类似标签的一组邮件变成了一个简单 的用户界面技巧. 很多 Web 2.0 网站也提供了类似的机制，比如 YouTube [http://www.youtube.com/] 和 Flickr [http://www.flickr.com/] 的“标签” (tag), 以及博文的 “类别” (categories). 人们已经明白数据的 组织方式非常重要，但是如何对数据进行组织应该是一个很灵活的概念. 旧的 “文件与文件夹” 范式对某些应用程序来说过于刻板.

Subversion 允许用户通过向文件打标签来创建变更列表，如果一个文件被打 上标签，说明该文件和这个变更列表是相关的，用户还可以删除标签，把命令的 操作限定到具有特定标签的文件上，具体的细节将在本节进行介绍.

创建与修改变更列表

命令 *svn changelist* 用于创建，修改和删除变更列表，更准确地说这个命令可以设置或清除某个特定的工作副本文件上的变更列表关 联. 当用户第一次用某个变更列表为文件打标签时，变更列表才被创建出来；当用户把最后一个标签从文件上移除时，对应的变更列表被删除. 下面用一个 例子来解释这些概念.

Harry 正在解决计算器程序中数字运算过程的几个问题，他已经修改了几 个文件：

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

在测试的过程中，Harry 发现他的修改暴露了用户接口实现 *button.c* 里的一个问题，Harry 决定在另一个单独的提交中把 这个问题也解决掉. 在一个只包含了少量文件和修改的小工作副本里，Harry 可 以不依靠 Subversion 就可以对两个逻辑上不相关的修改进行组织，但是今天 他想试用一下 Subversion 的变更列表.

Harry 先创建一个变更列表，并关联两个已被修改的文件，具体的做法是 用命令 *svn changelist* 向这两个文件分配一个任意的 变更列表名：

```
$ svn changelist math-fixes integer.c mathops.c
A [math-fixes] integer.c
A [math-fixes] mathops.c
$ svn status

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

可以看到, *svn status* 的输出反映了新的分组.

现在 Harry 着手修改用户接口的问题. 因为他知道将要修改哪个文件, 所 以他也向这个文件分配了一个变更列表, 不幸的是, Harry 错误地向第三个文件 分配了和前两个文件一样的变更列表:

```
$ svn changelist math-fixes button.c
A [math-fixes] button.c
$ svn status

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

幸好 Harry 很快就发现了错误, 现在他有两个选择, 一是删除与 *button.c* 关联的变更列表, 然后分配一个新的变更列表:

```
$ svn changelist --remove button.c
D [math-fixes] button.c
$ svn changelist ui-fix button.c
A [ui-fix] button.c
$
```

二是直接向 *button.c* 分配一个新的变更列表, 此时 Subversion 会先移除 *button.c* 原来的变更列表:

```
$ svn changelist ui-fix button.c
D [math-fixes] button.c
A [ui-fix] button.c
$ svn status

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

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

现在 Harry 的工作副本里有了两个不同的变更列表, *svn status* 会根据它们的变更列表对输出进行分组. 虽然 Harry 还没有修改 *button.c*, 但 *svn status* 仍然 会输出与它有关的信息, 这是因为 *button.c* 被分配了 一个变更列表. 任何时候都可以向文件添加或删除变更列表, 无论它们是否含有 本地修改.

接下来, Harry 解决了 *button.c* 的用户接口问题.

```
$ svn status

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

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

变更列表用作操作过滤器

我们在上一节看到的 *svn status* 对变更列表的分组 效果还不错, 但还不是很有用. 除了 *svn status*, 通过 选项 `--changelist`, 还有很多操作都会理解变更列表.

如果提供了选项 `--changelist`, Subversion 命令将会把 操作的作用域限定到具有特定变更列表的文件上. 假如说 Harry 想查看变更列表 `math-fixes` 里的文件的修改, 他可以在 *svn diff* 的后面显式地列出变更列表 `math-fixes` 的所有 文件.

```
$ 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)
…
$
```

如果文件比较少的话还可以接受, 但是如果变更列表包含了 20 个或 30 个 文件, 那就有点麻烦了. 不过既然它们都属于同一个变更列表, 可以用变更列 表替换文件列表:

```
$ 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)
…
$
```

准备提交时，Harry 可以再次使用选项 `--changelist` 把提交操作的作用域限定到具有特定变更列表的文件上. 他可以像下面这样提交 用户接口的修改：

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

实际上 *svn commit* 还提供了另一个和变更列表相关 的选项：`--keep-changelists`. 一般情况下，在文件提交后，变更列表就会从文件上移除，但是如果提供了选项 `--keep-changelists` ，Subversion 就会把变更列表保留在提交了的文件上. 在任何一种 情况下，提交某个变更列表的文件时，不会对其他变更列表产生影响.

```
$ svn status

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

选项 `--changelist` 只是作为命令的操作目标的过滤 器，它不会向命令添加更多的操作目标. 举个例子，命令 `svn commit /path/to/dir` 的操作目标是目录 */path/to/dir* 及其子文件，如果再向命令添加一个变更列表，那么只有位于 */path/to/dir* 目录内的，且分配了 相应变更列表的文件才会被当作提交的目标—提交不会包含其他位置（例如 */path/to/another-dir*) 的文件，即使它们 拥有和命令行相同的变更列表.

命令 *svn changelist* 也支持选项 `--changelist`, 这允许用户方便地重命名或删除变更列表：

```
$ svn changelist math-bugs --changelist math-fixes --depth infinity .
D [math-fixes] integer.c
A [math-bugs] integer.c
D [math-fixes] mathops.c
A [math-bugs] mathops.c
$ svn changelist --remove --changelist math-bugs --depth infinity .
D [math-bugs] integer.c
D [math-bugs] mathops.c
$
```

最后，用户可以一次指定多个 `--changelist` 选项，此时受命令影响的文件将是它们的并集.

变更列表的限制

变更列表是组织工作副本文件的好工具，但是它也有一些限制. 变更列表 是特定的工作副本的产物，这就意味着变更列表不能被传送给仓库，或与其他 用户分享. 只能在文件上分配变更列表—Subversion 目前还不支持在目录 上使用变更列表. 最后，在工作副本的一个文件上最多只能分配一个变更列表，如果用户发现自己需要在一个文件上分配多个变更列表，那只能算你倒霉了.

网络模型

在某些情况下，用户需要了解 Subversion 客户端如何与服务器通信. Subversion 的网络层是抽象的，也就是说无论服务器是什么类型，Subversion 表现出的行为总是类似的. 不管是用 HTTP 协议 (`http://`) 与 Apache HTTP 服务器通信，还是传统的 Subversion 协议 (`svn://`)，基本网络模式都是相同的. 本节将介绍 Subversion 网络 模式的基本概念，包括 Subversion 如何管理授权与认证.

请求与响应

Subversion 客户端的大部分时间都用在工作副本的管理上，当它需要从远 程仓库获取信息时，客户端生成并向服务器发送网络请求，服务器再用适当的回答 响应该请求. 网络协议的细节对用户是透明的—客户端试图访问一个 URL，根据 URL 模式，客户端将使用某种特定的协议与服务器通信.

运行 `svn --version` 查看客户客户端支持的 URL 模式与协议.

当服务器接收到客户端发来的请求时，它经常会要求客户端阐明自己的身份. 服务器向客户端发送一个认证消息，客户端提供 `证书` (*credentials*) 进行响应，认证一旦完成，服务器便 向客户端返回它所请求的信息. 这与 CVS 系统不同，CVS 系统的客户端先向服 务器提供证书 (“登录”)，然后再发送请求. 而在 Subversion 中，服务器在适当的时候向客户端索要证书，在此之前客户端不会主动向服务 器发送证书，这样的话某些操作就更加方便. 比如说如果服务器被配置成允许 任何用户读取仓库，当客户端试图检出工作副本时 (*svn checkout*)，服务器将不会要求客户端提供证书.

如果客户端发起的请求将会产生一个新的版本号（例如 *svn commit*)，Subversion 就使用请求中的，经过认证的用户名作为新的版本号 的作者，具体来说就是把经过认证的用户名作为新版本号的 `svn:author` 属性值（见 “Subversion 的保留属性” 一节). 如果客户端未经过认证（也就 是说服务器没有向客户端发送认证请求），新版本号的 `svn:author` 属性值将为空.

客户端证书

很多 Subversion 服务器都会要求认证. 有时候匿名的读操作是允许的，但是写操作必须经过认证，还有些服务器要求读写都需要认证. 不同的 Subversion 服务器选项支持不同的认证协议，但是从用户的视角来看，可以 把认证简单地理解为用户名与密码. Subversion 客户端提供了几种不同的方法 来检索和存放用户的认证证书，包括交互性地提示用户输入用户名与密码，以及 存放在磁盘上的加密或未加密过的数据缓存.

对安全比较敏感的读者可能在想 “在磁盘上缓存密码? 这可是个 馊主意，千万不要这么干!” 不用担心—并没有听起来的这么 糟糕. 下面将介绍 Subversion 使用的几种证书缓存类型，什么时候用到它们，以及如何禁止它们的全部或部分功能.

缓存证书

Subversion 提供了一种方法，用于避免用户每次都要输入用户名与密码. 在默认情况下，每当客户端成功地响应服务器的认证请求时，认证证书都会 被缓存到磁盘上，并把服务器的主机名，端口与认证域的组合作为键值. 这个 缓存在将来会被自动查阅，这就避免了用户再次输入认证证书. 如果在缓存中 没有找到合适的证书，或者是缓存的证书认证失败，此时客户端就会提示用户 输入用户名与密码.

Subversion 开发人员承认在磁盘上缓存认证证书有可能成为安全隐患，为了解决这个问题，Subversion 会利用操作系统环境，把信息泄漏的风险 降到最低.

- 在 Windows 操作系统中，Subversion 客户端把密码存放在 *%APPDATA%/Subversion/auth/* 目录内．在 Windows 2000 及之后的系统里，磁盘上的密码会使用标准的 Windows 加密服务进行加密．因为密钥由 Windows 管理，且绑定到用户个人的 登录证书，所以只有用户才能解密缓存的密码．（如果用户的 Windows 帐户密码被管理员重置，那么所有缓存的密码都不能再被解密，此时 Subversion 就认为缓存密码不存在，在需要时重新提示用户输入.）

- 类似的，在 Mac OS X 系统中，Subversion 用登录名作为键值存放 所有仓库的密码（由 keychain 服务进行管理），键值由登录密码进行 保护．用户可以施加额外的策略，例如每当 Subversion 要使用密码时，就要求用户输入帐户密码.

- 类 Unix 系统没有标准的 “keychain” 服务，但 Subversion 仍然知道如何用 “GNOME Keyring”, “ KDE Wallet” 和 “GnuPG Agent” 服务安全地 存放密码．把未加密的密码存放在 *~/.subversion/auth/* 之前，Subversion 会询问用户是否要这么做．注意缓存 区 *auth/* 仍然受到权限的保护，只有用户（目录的所有者）才能读取其中的数据．操作系统的文件权限保护避免了密码 被系统中的其他非管理员用户看到，当然前提是其他用户不能直接接触 存储设备或备份.

当然，这些机制并不能完全解决问题，对于那些为了追求安全而不惜牺牲 便利的用户来说，Subversion 提供了多种方式用于禁止证书缓存.

禁止密码缓存

用户在执行一个要求认证的操作时，Subversion 默认把密码加密后缓存 在本地，在某些操作系统中，Subversion 可能无法进行加密，在这种情况 下 Subversion 将会询问用户是否以明文地方式缓存证书：

```
$ svn checkout https://host.example.com:443/svn/private-repo
--------------------------------------------------------------------------
ATTENTION!  Your password for authentication realm:

   <https://host.example.com:443> Subversion Repository

can only be stored to disk unencrypted!  You are advised to configure
your system so that Subversion can store passwords encrypted, if
possible.  See the documentation for details.

You can avoid future appearances of this warning by setting the value
of the 'store-plaintext-passwords' option to either 'yes' or 'no' in
'/tmp/servers'.
--------------------------------------------------------------------------
Store password unencrypted (yes/no)?
```

如果用户贪图方便，不想每次都输入密码，那就输入 `yes` ．如果用户担心以明文的方式缓存密码不太安全，而且不想每次都被询问是否要缓存密码，你可以永久地禁止密码明文缓存，或者为每一个 服务器设置单独的策略.

用户在考虑如何使用 Subversion 的缓存功能时，可能需要咨询一下 公司制度—很多公司对于如何存放员工的认证证书都有很严格的规定.

为了永久地禁止以明文方式缓存密码，在本地配置文件 *servers* 的 `[global]` 部分添加一行 `store-plaintext-passwords = no`. 为了对特定的服务器 禁止明文密码缓存，在配置文件 *servers* 的适当 位置添加同样的一行（具体的细节见 第 7 章 定制自己的 *Subversion* 体验 的 “运行时配置选项”一节).

为了禁止特定的 Subversion 命令缓存密码，向该命令添加选项 `--no-auth-cache`. 为了永久地禁止缓存，在本地的 Subversion 配置文件中添加一行 `store-passwords = no`.

删除已缓存的证书

有时候用户想从缓存中删除特定的证书，为了实现这个目标，你需要 进入到 *auth/* 目录，然后手动地删除对应的缓存文件. 每一个证书都对应一个单独的文件，如果查看文件的内容，你将会看到关键字 和值，关键字 `svn:realmstring` 描述了文件与哪一个 服务器关联.

```
$ 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
```

一旦找到了对应的缓存文件，直接删除即可.

命令行认证

所有的 Subversion 命令都支持选项 `--username` 和 `--password`, 选项的作用分别是指定用户名与密码，这样 Subversion 就不会再提示用户输入这两项信息. 有了这两个选项，就 可以很方便地在脚本里执行 Subversion 命令，而不用依赖缓存的认证证书. 除此之外，如果 Subversion 已经缓存了认证证书，但你知道这不是你想使 用的那个（比如多个人使用相同的登录名登录操作系统，但每个人所使用的 Subversion 认证证书却不一样），可以用这两个选项重新指定用户名与密码. 用户可以不指定选项 `--password`, 只让 Subversion 从 命令行参数中获取用户名，但它仍然会提示用户输入与用户名对应的密码.

认证小结

关于 Subversion 的认证行为最后再讲一点，尤其是 `--username` 和 `--password` 这两个选项. 很多客户端子 命令都支持这两个选项，但是要注意使用它们并不会 自动 地把证书发送给服务器，前面已经说过，只有当服务器认为需要 证书时，

才会主动向客户端 “索要” 证书，客户端不能随心所 欲地向服务器 “推送” 证书. 如果在命令行选项上指定了用户 名和 (或) 密码，只有当服务器需要时，它们才会被递送给服务器. 使用这 两个选项的最典型情况是用户想要明确地指定用户名，而不是让 Subversion 自己猜一个 (例如登录操作系统的用户名)，又或者是避免出现交互式的提示 信息 (例如命令是在脚本里执行的).

服务器的一个常见的配置错误是从来不向客户端发起认证请求， 如果 用户在命令行上指定了选项 `--username` 和 `--password`，会发现它们不起作用，新的版本号总是 匿名提交的.

下面几点介绍了当客户端收到一个认证请求时所做的操作.

1. 首先，客户端检查用户是否在命令行上输入了证书 (选项 `--username` 和 (或) `--password`)，如果 是，客户端将使用它们响应服务器的认证请求.

2. 如果命令行参数没有提供证书，又或者是提供的证书是无效的，客户 端就在运行时配置的 *auth/* 目录查找服务器的 主机名，端口和认证域，检查是否有合适的证书缓存. 如果有就使用缓存 的证书响应请求.

3. 最后，如果前面的认证都失败了，客户端就会提示用户输入用户 名与密码 (除非指定了选项 `--non-interactive` 或其他等效的设置).

如果客户端成功地用上面的任意一种方法满足了服务器的认证请求，它就试图把证书缓存在本地磁盘上 (除非用户禁止了缓存).

在没有工作副本的情况下工作

“Subversion 的工作副本” 一节 已经说过，Subversion 的工作 副本是一种暂存区，暂存用户的私有修改，当修改完成，准备共享给其他用户时，就把修改提交到仓库中. 于是，用户的大部分时间都是在用客户端与工作副本打 交道，即使是不处理工作副本的操作 (例如 *svn log*)，也 经常使用工作副本里的文件或目录作为操作的目标文件.

明确地说，从工作副本里提交是修改文件的典型方式，幸运的是这并不是唯一 的选择，如果修改相对比较简单，用户甚至可以在不检出工作副本的前提下提交 修改，本节就是介绍与此有关的内容.

远程客户端命令行操作

为了完成一些相对较小的修改，Subversion 的客户端命令行工具的很 多操作都可以在没有工作副本的前提下，直接对仓库 URL 发起. 其中的部分 内容在本书的其他地方介绍，但是为了方便读者，我们在这里详尽地列出了 它们.

最明显的远程类提交操作应该是命令 *svn import*，我们在 “导入文件和目录” 一节 介绍如何快速地把 一个目录导入到仓库中时，提到了这个命令.

当目标参数是 URL 时，命令 *svn mkdir* 和 *svn delete* 也可以是远程操作，这允许用户在没有工作副本的前 提下，在仓库中添加新的目录或 (递归地) 删除文件. 每次执行这两个命令时，客户端与服务器的通信过程类似于把工作副本里新增的目录或删除的文件提交 给服务器的过程. 如果认证没有问题，并且没有发生冲突，服务器就在一个单独 的版本号里完成添加或删除.

你可以用两个 URL 作为 *svn copy* 或 *svn move* 的参数—一个是源，另一个是目标—直接向 仓库提交文件的复制或移动. 如果是在工作副本里执行，这两个操作将会是耗时 最长的操作之一，如果使用仓库的 URL 进行远程操作，它们就可以在常数时间 内完成. 实际上，在创建分支时，人们经常使用 *svn copy* 远程操作，这部分内容将在 “创建分支”一节 介绍.

和普通的 *svn commit* 一样，上面介绍的几个远程 操作都接受用户输入一段日志，描述本次操作做了什么，输入日志的方式可以 用选项 `--file(-F)` 或 `--message(-m)`, 如果这两个选项都没有指定，客户端就会提示用户输入日志消息.

最后，很多与版本号属性相关的操作都可以直接对仓库发起. 实际上，这 里谈到的版本号属性比较独特，因为它们不是存放在工作副本里，所以它们 必须 在不与工作副本交互的情况下修改. 关于如何管理 Subversion 属性的更多信息，见 “属性”一节.

使用 svnmucc

客户端命令行工具的远程提交操作的一个缺点是用户每次提交只能执行一 个操作—或者说一种类型的操作. 比如说在一个工作副本内，为了用一个 全新的目录替换掉旧目录，先执行 *svn delete*, 再执行 *svn mkdir*—是一个很自然的操作. 当用户提交这两 个操作的执行结果时，仓库将创建一个新的版本号，该版本号完整地记录了这两 个操作. 但是客户端命令行的远程操作不能在单个版本号中完成这两步操作 —*svn delete URL* 会创建一个新的版本号并删除目录; *svn mkdir URL* 会在第二个版本号中完成目录的创建.

幸运的是，Subversion 另外提供了一个工具，用于把多个远程操作放在一 个提交中完成，这个工具是 *svnmucc*—Subversion 多 URL 命令行客户端 (Multiple URL Command Client):

```
$ svnmucc --help
Subversion multiple URL command client
usage: svnmucc ACTION...

  Perform one or more Subversion repository URL-based ACTIONs, committing
  the result as a (single) new revision.

Actions:
  cp REV URL1 URL2       : copy URL1@REV to URL2
  mkdir URL              : create new directory URL
  mv URL1 URL2           : move URL1 to URL2
  rm URL                 : delete URL
  put SRC-FILE URL       : add or modify file URL with contents copied from
                           SRC-FILE (use "-" to read from standard input)
  propset NAME VAL URL   : set property NAME on URL to value VAL
  propsetf NAME VAL URL  : set property NAME on URL to value from file VAL
  propdel NAME URL       : delete property NAME from URL
…
```

svnmucc 很多年前就已经包含在 Subversion 的源代 码树中（那时候称为 *mucc*), 但是直到 1.8, *svnmucc* 才享受到完全的支持，成为 Subversion 客户端命令行工 具套装的正式成员.

svn 可以做到的转换，*svnmucc* 都可以做到，但不同的是，*svnmucc* 的功能并不是把操作 切分成多个子命令. 用户可以在一条命令行上 (或者在一个文件中，通过选项 `--extra-args(-X)` 把文件传递给 *svnmucc*) 输入多个操作及其参数，*svnmucc* 支持的某些操作模仿了对应的客户端命令行. 读者可能已经注意到 *svnmucc* 帮助信息中 列出的操作，例如 `cp`, `mkdir`, `mv`

和 `rm`, 和我们在 “远程客户端命令行操作”一节 提到的操作非常 类似，但是请记住，它们之间最关键的区别是用户可以在 *svnmucc* 的一次调用中，执行任意多的操作，所有的这些 操作只会产生一个新的版本号.

如果使用 *svnmucc* 完成本节开头的远程目录替换操 作，一个示例是:

```
$ svnmucc rm http://svn.example.com/projects/sandbox \
          mkdir http://svn.example.com/projects/sandbox \
          -m "Replace my old sandbox with a fresh new one."
r22 committed by harry at 2013-01-15T21:45:26.442865Z
$
```

可以看到，*svnmucc* 在一个版本号中完成了两步操作，而在没有工作副本的情况下，*svn* 会产生两个新的版本号.

svnmucc 和 *svn* 的另一个区 别是如果用户没有在命令行提供日志消息（通过选项 `--message(-m)` 或 `--file(-F)`), *svnmucc* 将不会提示用户输入提示日志，转而使用一段常备的日志（相对来说没什么价值）.

svnmucc 的作用不仅仅是混合 *svn* 的操作，它还增加了一些其他命令行工具不支持的功能. 例如用户可以使用操作 *put* 添加或修改仓库里的文件，把来自本地文件或标准输入 的内容复制到仓库的文件中. *svnmucc* 还提供了 *propset*, *propsetf* 和 *propdel* 这三种操作，用于设置或删除文件和目录的属性（属性值既可以显式地在命令行指定，也可以从本地文件中读取），而其他客户端 命令行工具还不支持这些操作（其他命令行工具只能直接操作工作副本里的文件 的属性）.

svnmucc 可以做哪些事，以及什么事应该由它来做 —这两者的区别非常重要，先来两句名言:

> “给予越多，期望越多.”
>
> —耶稣

> “能力越大，责任越大.”
>
> —Ben, 蜘蛛侠 Peter Parker 的叔叔

不使用工作副本的坏处是丧失了冲突检测的能力. 当按照典型的方式使用 *svn* 时，被提交的修改是相对于仓库中文件的特定基础 版本，这样用户就不会无意中覆盖其他用户对相同文件提交的修改. 服务器知道 被用户修改的文件的版本，也知道在该版本之后是否有其他用户修改了文件，有了这些，当用户的提交会破坏其他用户的修改时，Subversion 就会拒绝提交，强迫用户合并其他用户已提交的修改，并重新考虑自己的修改. 因为 *svnmucc* 没有用到工作副本，也就绕过了冲突检测，使得 *svnmucc* 提交的每个修改都是相对于仓库中的最新 版本，希望这种情况不是用户正想看到的样子.

幸运的是，*svnmucc* 也希望用户在使用它时能更加地 谨慎，方法是使用选项 `--revision(-r)`. 通过这个选项，用户可以明确地指定相对于提交的基础版本号，该版本号应该是 用户在提交前看到的最新的版本号.

强烈建议用户总是为 *svnmucc* 加上选项 `--revision(-r)`, 并指定正确的 版本号.

说明选项 `--revision(-r)` 重要性 的最好方式是展示如何正确地使用命令 *svnmucc put*. 假设 Harry 想在没有工作副本的情况下修改仓库里的文件 *README*（仅针对修改 *README* 这个操作而言，我们假设使用工作副本不会带来其他额外的好处，例如在提交 前执行工作副本里的一个脚本，以保证 Harry 的修改是符合要求的），他要确定的第一件事是针对文件的哪个版本进行 修改. 在典型的情况下，用户总是想修改文件的最新版，于是 Harry 查询文件 最后一次被提交的版本号，并把该版本号对应的文件内容抓取到本地的一个临时 文件中.

```
$ svn info http://svn.example.com/projects/sandbox/README
Path: README
URL: http://svn.example.com/projects/sandbox/README
Relative URL: ^/sandbox/README
Repository Root: http://svn.example.com/projects
Repository UUID: 13f79535-47bb-0310-9956-ffa450edef68
Revision: 22
Node Kind: file
Last Changed Author: sally
Last Changed Rev: 14
Last Changed Date: 2012-09-02 10:34:09 -0400 (Sun, 02 Sep 2012)

$ svn cat -r 14 http://svn.example.com/projects/sandbox/README \
      > README.tmpfile
$
```

现在 Harry 有了 *README* 的一个副本, 副本的内容 和它最后一次被提交时的内容相同. 他按照自己的想法对副本进行了修改, 修改 完成后, 他打算提交到仓库中.

如果 Harry 单纯地使用 `svnmucc put…`, 将仓库中 *README* 的内容替换成本地修改后的版本, 那他就是在滥用 *svnmucc* 的能力. 要是在他提交前一 毫秒, Sally 也提交了 *README* 的修改, 那会怎样? 和 *svn* 相比, *svnmucc* 不会为了同时 保留两个用户的修改而尝试在服务器端做一些内容合并操作, *svnmucc* 会直接使用指定的内容替换掉文件的最新版本. Harry 不会察觉到这些, 但 Sally 可能会大发雷霆.

```
$ svnmucc put README.tmpfile \
          http://svn.example.com/projects/sandbox/README \
          -m "Tweak the README file."
r24 committed by harry at 2013-01-21T16:21:23.100133Z
$
Message from sally@shell.example.com on pts/2 at 16:26 ...
We need to talk.  Now.
EOF
```

Harry 应该回想被自己修改的本地文件来自哪个版本号, 然后把该版本号 通过选项 `--revision(-r)` 传递给命令 *svnmucc*, 这样服务器就有机会检查被用户修改的文件是否 过旧, 如果是的话就拒绝提交.

```
$ svnmucc -r 14 put README.tmpfile \
          http://svn.example.com/projects/sandbox/README \
          -m "Tweak the README file."
svnmucc: E170004: Item '/sandbox/README' is out of date
$
```

和 *svnmucc* 的其他其他选项一样, 选项 `--revision(-r)` 的作用域是整个命令 —命令中指定的每一个操作. 这就使得用户在使用 *svnmucc* 时, 具有了和下面这种情形同样的保护措施 —检出整个仓库的工作副本 (并且工作副本具有一致的版本号), 修改工作 副本里的文件, 最后提交工作副本的所有修改.

svnmucc 是 Subversion 客户端工具集的有益补充, 它的完整手册见 svnmucc 参考手册—Subversion 多 URL 命令行客户端.

小结

读完本章后，读者应该牢固地掌握了 Subversion 的某些特性，虽然这些 特性不可能每次都会被用到，但了解它们总是会派上用场. 读者应该继续往下阅读，在下面的内容里，你将会学到分支，标签与合并，学习完这些知识后，读者基本上 就算是完全掌握了 Subversion 的客户端操作. 虽然我们的律师不允许我们向你做 出任何承诺，但了解这些知识会让你更加潇洒.

第 4 章 分支与合并

> “君子务本 (It is upon the Trunk that a gentleman works.)”
>
> —孔子

分支与合并是版本控制的基础功能，从概念上解释非常简单，但是它的复杂性 和各种细微差别值得我们用整整一章进行介绍. 我们将会介绍这些操作背后的基本 思想，以及 Subversion 在实现上的某些独特之处. 如果读者对 Subversion 的基本 概念（见 第 1 章 基本概念）还不了解，在阅读本章之前建议读者先 了解它们.

什么是分支

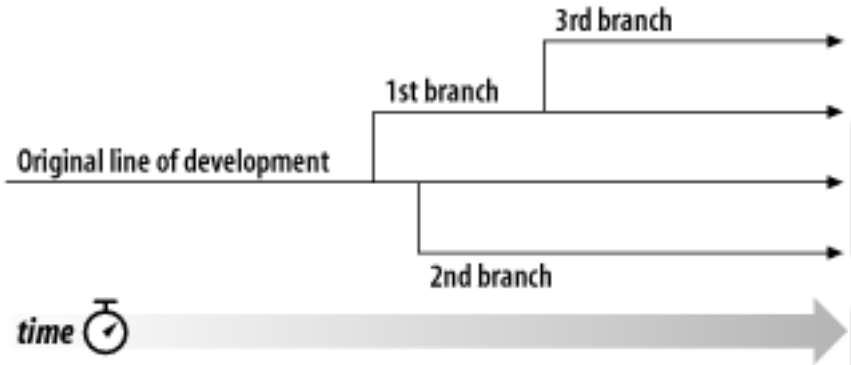
假设你的工作是为公司的某个部门维护文档—比如说一本手册. 一天，另一个部门也请你替他们维护同一份文档，但需要根据他们的部门情况，对手册的 某些部分作一些修改.

对于这种情况你应该怎么处理? 最容易想到的做法是为另一个部门创建一份 文档的副本，然后单独地对这两份文档进行维护. 每当部门要求对文档进行修改 时，你就把修改写到相应的文档里.

你应该会经常对两个副本做相同的修改，比如说你在第一个副本时发现了一 个打字错误，同样的错误在第二个副本里也应该存在，毕竟两份文档的大部分内容 都是一样的.

这是分支的基本概念—顾名思义，它是一条独立存在的开发线，如果回溯地 足够深，将会看到它和其他分支共享相同的历史. 一个分支的生命总是开始于复 制操作，从那儿开始产生自己的历史（见 图 4.1 “分支示意图”）.

图 4.1. 分支示意图



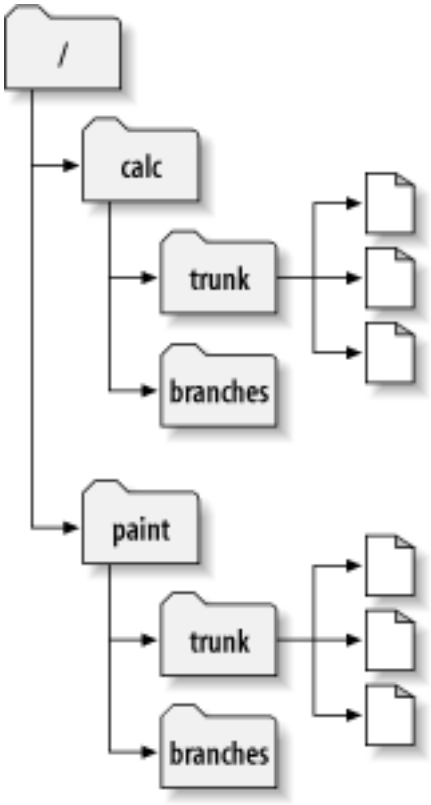
Subversion 提供了很多命令用于帮助用户维护文件与目录的并行分支，这些 命令允许你通过复制来创建分支，并记住这些副本之间是有关的. 它们还可以帮助 你把一个分支的修改复制到其他分支上. 最后，它们可以把工作副本的某些部分 映射到不同的分支上，这样你就可以在日常工作中 “混合搭配” 不同的开发线.

使用分支

阅读到这里，读者应该理解了每一次提交是如何创建一个新的文件系统树 状态（称为 “版本号”），如果还不理解，读者应该回去阅读 “版本号”一节 的内容.

再次回顾 第 1 章 基本概念 的例子：你和你的同事—Sally —共享一个包含了两个项目的仓库，这两个项目是 *paint* 和 *calc*. 如 图 4.2 “仓库的起始布局” 所示，每一个项目都包含了 子目录 *trunk* 和 *branches*. 读 者很快就会明白如此布局的原因.

图 4.2. 仓库的起始布局



假设 Sally 和你都有一份 “calc” 项目的工作副本，更确切 地说，你们每个人都有一份 */calc/trunk* 的工作副本. 项目的所有材料都放在子目录 *trunk* 内，而不是直接放到 */calc* 里，因为开发团队把 */calc/trunk* 作为开发 “主线” (main line).

现在团队要求你为软件项目实现一个比较大的特性，这项工作的时间会比较 长，而且会影响到项目内的所有文件. 首先想到的第一个问题是你不想干扰 Sally —她目前正在解决软件的几个小问题. Sally 的工作依赖于这样一个事实，那就是项目的最新版 (存放在 */calc/trunk*) 总是可用的. 如果你开始一点一点地提交你的修改，那肯定会影响到 Sally 的工作，甚至包括团队内的其他成员.

一种可能的办法是在你完成全部的修改之前，不向仓库提交修改，也不更新 工作副本，这种情况会持续几周，但是这会产生很多问题. 首先这不太安全，如果 你的工作副本或机器遭到破坏，之前所有的工作都会白费. 第二，不够灵活，除非 你手动地把你的修改复制到其他工作副本或机器中，否则的话你就只能在一个固定 的工作副本上工作，如果要把半成员品分享给其他人也很麻烦. 一种比较良好的 软件工程做法是允许团队中的其他成员审查你的修改，如果别人不能看到你在中间阶段的提交，你将得不到别人的反馈，甚至在错误的方向上努力多日，直到别人 注意到你的工作. 最后，当你完成所有的修改时，你可能会发现很难把你的修改 合并到仓库里. Sally (或其他人) 可能在你工作的过程中向仓库提交了很多修改，几周后，当你最终执行 *svn update* 时，这些修改很难合并 到你的工作副本里.

更好的做法是在仓库中创建一个属于你自己的分支 (或一条开发线)，这样 你就可以保存尚未完成的工作，也不会干扰到其他人，还可以与其他人分享你的 工作进度. 下面我们将会介绍具体的步骤.

创建分支

创建分支非常简单—就是用命令 *svn copy* 在 仓库中为项目目录树创建一个副本. 因为项目的源代码放在 */calc/trunk*, 所以你要复制的就是这个目录. 那么新副本应该 放在哪里? 分支在仓库里的存放位置由项目自己来决定. 最后, 你的分支需要 一个名字, 用于和其他分支区分开. 分支的名字对 Subversion 而言并不重要 —你可以根据工作的特点为分支取一个你认为最好的名字.

假设团队规定分支存放在目录 *branches* 内 (这是 最常见的情况), 而 *branches* 是项目主干的兄弟目录 (在我们这个例子里, 存放分支的目录就是 */calc/branches*). 虽然没什么创意, 但你还是想把新的分支叫做 `my-calc-branch`, 这就意味着你将会创建一个新目录 */calc/branches/my-calc-branch*, 新目录的生命周期 以 */calc/trunk* 的副本作为开始.

读者应该已经见过如何在工作副本中用命令 *svn copy* 复制出一个新文件或目录, 除了工作副本, 它还可以完成 远程复制 (*remote copy*)—复制操作会 立刻提交到仓库中, 产生一个新的版本号, 完全不需要工作副本的参与. 从 命令的形式上看, 只是从一个 URL 中复制出新的一个:

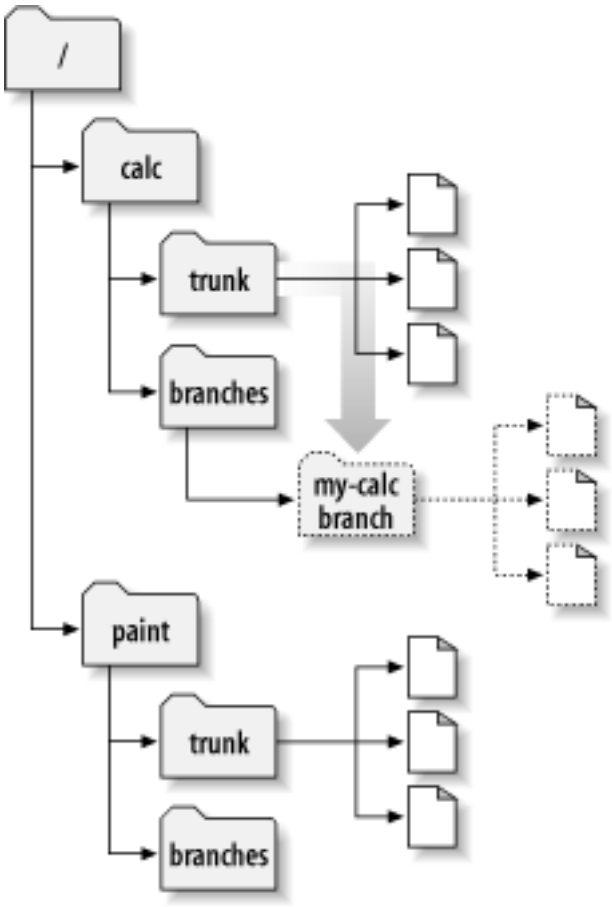
```
$ svn copy ^/calc/trunk ^/calc/branches/my-calc-branch \
           -m "Creating a private branch of /calc/trunk."

Committed revision 341.
$
```

上面的命令立刻在仓库中产生了一次提交, 在版本号 341 创建了一个新 目录, 它是目录 */calc/trunk* 的拷贝, 如图 图 4.3 “创建了分支后的仓库” 所示. [1] 当然, 使用 *svn copy* 复制工作副本里的目录来创建分支 也是可以的, 但我们不推荐这种做法, 因为可能会很慢. 在客户端复制目录是 一个线性时间复杂度的操作, 实际上它需要递归地复制目录内的每一个文件和 子目录, 这些文件和子目录都存放在本地磁盘上. 而远程复制是一个时间 复杂度为常量的操作, 大多数用户都是采用这种方式创建分支. 另外, 工作副 本中的目录可能含有混合的版本号, 虽然不会产生有害的影响, 但是在合并时 可能会产生不必要的麻烦. 如果用户选择通过复制工作副本中的目录来创建 分支, 在复制前应该确保被复制的目录不含有本地修改和混合的版本号.

[1] Subversion 不支持在不同的仓库间复制, 当命令 *svn copy* 和 *svn move* 的参数 带有 URL 时, 这些 URL 必须都在同一个仓库内.

图 4.3. 创建了分支后的仓库



廉价拷贝

Subversion 的设计非常特殊，用户复制一个目录时，不必担心仓库会 增长过大—实际上 Subversion 不会复制任何数据，作为替代，它 创建了一个新的目录项，将其指向一个 *已存在* 的 目录树. 如果你是一名有经验的 Unix 用户，马上就能看出来这和硬链接是 同样的概念. 随着文件和目录的修改不断增多，Subversion 会继续尽可能地 利用这种硬链接思想，只有在必要时（消除对象的不同版本之间的歧义）才会 真正地复制数据.

你会经常听到 Subversion 用户谈论 “廉价拷贝”. 无论 目录有多大，Subversion 都只需要一段极小的，常量的时间和空间就能完成 复制操作. 实际上，这个特性也是 Subversion 处理提交操作的基础: 每一 个版本号都是前一个版本号的 “廉价拷贝”，只有少数几项 被修改了.（关于这部分的更多内容，请登录到 Subversion 官网，阅读 Subversion 设计文档中的 “冒泡 (bubble up)” 方法）.

当然，这些复制和共享数据的内部机制对用户而言都是透明的，他们只能 看到目录被复制了. 我们的重点是复制在时间和空间上都很廉价，如果用户是 在仓库内创建分支（通过执行命令 `svn copy` *`URL1 URL2`*），操作消耗 的时间是常量的，而且非常快. 只要用户有需求，可以随意地创建分支.

在分支上工作

创建完分支后，用户就可以检出它的工作副本，然后开始工作：

```
$ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch
A    my-calc-branch/doc
A    my-calc-branch/src
A    my-calc-branch/doc/INSTALL
A    my-calc-branch/src/real.c
A    my-calc-branch/src/main.c
A    my-calc-branch/src/button.c
A    my-calc-branch/src/integer.c
A    my-calc-branch/Makefile
A    my-calc-branch/README
Checked out revision 341.

$
```

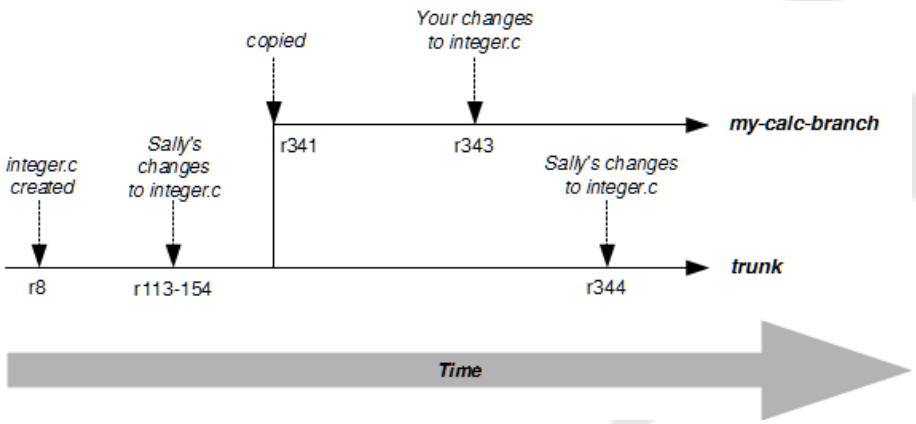
和其他工作副本相比，这个工作副本并没有什么特别的地方，它只不过是映射 到了仓库的另一个目录. 而 Sally 在更新时将不会看到在这个工作副本里提 交的修改，因为她的工作副本映射的是 */calc/trunk*.（记得看一下本章后面的 “遍历分支”一节，它是 创建分支工作副本的另一种办法）

假设分支创建后又过了一周，期间提交了下面这些修改：

- 在版本号 342 修改了文件 */calc/branches/my-calc-branch/src/button.c*
- 在版本号 343 修改了文件 */calc/branches/my-calc-branch/src/integer.c*
- Sally 在版本号 344 修改了文件 */calc/trunk/src/integer.c*.

现在，文件 *integer.c* 产生了两条独立的开发线，如 图 4.4 “一个文件历史的分叉” 所示.

图 4.4. 一个文件历史的分叉



当用户查看文件 *integer.c* 副本的修改历史时，事 情开始变得有趣起来：

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

```
$ svn log -v src/integer.c
------------------------------------------------------------------------
r343 | user | 2013-02-15 14:11:09 -0500 (Fri, 15 Feb 2013) | 1 line
Changed paths:
   M /calc/branches/my-calc-branch/src/integer.c

* integer.c:  frozzled the wazjub.
------------------------------------------------------------------------
r341 | user | 2013-02-15 07:41:25 -0500 (Fri, 15 Feb 2013) | 1 line
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

Creating a private branch of /calc/trunk.
------------------------------------------------------------------------
r154 | sally | 2013-01-30 04:20:03 -0500 (Wed, 30 Jan 2013) | 2 lines
Changed paths:
   M /calc/trunk/src/integer.c

* integer.c:  changed a docstring.
------------------------------------------------------------------------
…
------------------------------------------------------------------------
r113 | sally | 2013-01-26 15:50:21 -0500 (Sat, 26 Jan 2013) | 2 lines
Changed paths:
   M /calc/trunk/src/integer.c

* integer.c: Revise the fooplus API.
------------------------------------------------------------------------
r8 | sally | 2013-01-17 16:55:36 -0500 (Thu, 17 Jan 2013) | 1 line
Changed paths:
   A /calc/trunk/Makefile
   A /calc/trunk/README
   A /calc/trunk/doc/INSTALL
   A /calc/trunk/src/button.c
   A /calc/trunk/src/integer.c
   A /calc/trunk/src/main.c
   A /calc/trunk/src/real.c

Initial trunk code import for calc project.
------------------------------------------------------------------------
```

注意到 Subversion 在追溯分支 `my-calc-branch` 中的文件 *integer.c* 的历史时，即使到达了创建分支 的时间点，也仍然会继续往下追踪. 在历史中显示的是分支被创建的事件，这 是因为当 */calc/trunk/* 中所有的文件都被复制时，自然也就复制了 *integer.c*. 现在再看一下 Sally 在 她的副本上执行同样的命令会输出什么内容：

```
$ pwd
/home/sally/calc
```

```
$ svn log -v src/integer.c
------------------------------------------------------------------------
r344 | sally | 2013-02-15 16:44:44 -0500 (Fri, 15 Feb 2013) | 1 line
Changed paths:
   M /calc/trunk/src/integer.c

Refactor the bazzle functions.
------------------------------------------------------------------------
r154 | sally | 2013-01-30 04:20:03 -0500 (Wed, 30 Jan 2013) | 2 lines
Changed paths:
   M /calc/trunk/src/integer.c

* integer.c:  changed a docstring.
------------------------------------------------------------------------
…
------------------------------------------------------------------------
r113 | sally | 2013-01-26 15:50:21 -0500 (Sat, 26 Jan 2013) | 2 lines
Changed paths:
   M /calc/trunk/src/integer.c

* integer.c: Revise the fooplus API.
------------------------------------------------------------------------
r8 | sally | 2013-01-17 16:55:36 -0500 (Thu, 17 Jan 2013) | 1 line
Changed paths:
   A /calc/trunk/Makefile
   A /calc/trunk/README
   A /calc/trunk/doc/INSTALL
   A /calc/trunk/src/button.c
   A /calc/trunk/src/integer.c
   A /calc/trunk/src/main.c
   A /calc/trunk/src/real.c

Initial trunk code import for calc project.
------------------------------------------------------------------------
```

Sally 看到了她提交的版本号 344, 但没有看到版本号 343. 对 Subversion 而言, 这两个提交影响的是存放在仓库中不同位置上的不同文件, 而 Subversion 的输出 *确实* 表明了这两个文件共享一段 相同的历史—在创建分支 (版本号 341) 之前, 它们是同一个文件. 也就是因为这个原因, 所以你和 Sally 都能看到版本号 8 到版本号 154 的提交 历史.

分支背后的关键概念

在阅读完这一节后, 读者应该牢记以下两点. 第一, 在 Subversion 内部是 没有分支这个概念的—它只知道如何复制. 当用户复制一个目录时, 产生 的新目录被称为 “分支” 完全是用户赋予它的意义, 用户也可以 从其他角度看待它, 但是对于 Subversion 而言, 它只是一个含有额外历史信息 的普通目录.

第二，Subversion 的分支作为 普通的文件系统目录 存在于仓库中，这和其他版本控制系统不太一样，其他版本控制系统创建分支的 典型做法是为文件集添加处于额外维度的 “标签”. Subversion 不关心分支目录的存放位置，但是大多数开发团队都遵循传统做法：把所有的 分支都放在 *branches/* 目录内，当然，用户也可以制订 自己的策略.

基本合并

现在你和 Sally 并行地在两个分支上进行开发：你在自己的私有分支上工作，Sally 在项目的主干（开发主线）上工作.

如果项目有很多开发人员，大多数人都会检出主干的工作副本. 如果有人需要 完成一个长期的修改，而这个修改的中间成果很可能会扰乱主干，那么比较标准 的做法是为它创建一个私有分支，把修改都提交到这个分支上，直到所有的相关 工作都完成为止.

有了分支后，好消息是你和 Sally 的工作不会互相干扰，但坏消息是分支 容易偏离主干过远. 记住，“缓慢爬行” 策略的问题是当你完成 分支上的工作时，把分支上的修改合并到主干上而不产生大量的冲突，几乎是不 可能的.

因此在工作的过程中，你和 Sally 会继续分享修改，哪些修改值得分享完全由你 来决定，Subversion 允许用户有选择地在分支之间 “复制” 修改. 当你在分支上的工作全部完成时，分支上的整个修改集合就可以被复制到主干上. 用 Subversion 的行话来讲，把一个分支上的修改复制到其他分支上— 这 种操作称为 合并 (*merging*)，完成这种操作的命令是 *svn merge*.

在下面的例子里，我们假设 Subversion 客户端和服务器端的版本都是 1.8 或更新的版本. 如果客户端或服务器端的版本小于 1.5，事情就会变得很复杂：旧版的 Subversion 不会自动跟踪修改，这就迫使用户必须手工实现类似的效果，而这种过程相对来说比较痛苦，具体来说，用户必须按照合并语法，详细地指定 被复制的版本号范围（见本章后面的 “合并语法详解”一节），而且还要注 意哪些修改已经合并，哪些没有. 因此，我们 强烈 建 议用户不要使用 1.5 版本之前的 Subversion 客户端与服务器端.

合并跟踪

Subversion 1.5 增加了 合并跟踪 (*merge tracking*) 特性，在引入这个特性之前，为了跟踪合并，要求用户手工执行一些笨拙的操作，或使用外部工具，后面发布的 Subversion 在合并跟踪上增加了很多功能，也修复了很多问题，所以我们才建议读者总是 使用最新版的 Subversion，无论是在客户端，还是在服务器端. 记住，即使 服务器端的版本是 1.5-1.7，你仍然可以使用 1.8 版的客户端，这对于合并 跟踪来说非常重要，因为与合并跟踪有关的绝大多数功能增强和问题修复都是在客户端实现.

变更集

在继续之前，我们需要提醒读者后面的内容会经常讨论到 “修改”. 对版本控制系统有经验的用户经常混用 “修改” (change) 和 “变更集” (changeset) 这两个概念，但我们必须弄清楚 Subversion 是怎么理解 变更集 (*changeset*) 的.

每个人对变更集的理解似乎都有所不同，至少在变更集对版本控制系统的 意义上都有不同的期待. 从我们的角度来说，变更集只是一个带有独特的名 字的修改集合. 修改可能包括文件的修改，目录结构的修改，或元数据的修改. 更一般的说，变更集只是带有名字的补丁.

在 Subversion 中，一个全局的版本号 *N* 确定了仓库中的一棵目录树：它是仓库在第 *N* 次提交后的样子. 同时它还确定了一个隐式的变更集：如果用户对目录树 *N* 和 *N*-1 进行 比较，就可以得到与第 *N* 次提交对应的补丁. 正因为如此，版本号 *N* 不

仅可以表示一棵 目录树，还可以表示一个变更集. 如果用户使用了一个问题跟踪系统来管理 问题，用户就可以使用版本号指代修复问题的特定补丁—例如，“这个问题在 r9238 中解决”，然后其他人就可以执行 `svn log -r 9238` 查看修复问题的提交日志，再用 `svn diff -c 9238` 查看补丁的具体内容. Subversion 命令 *svn merge* 也可以使用版本号作为参数（读者马上就 会看到）. 通过指定参数，用户可以把一个分支上的特定的变更集合并到另一个 分支上：为 *svn merge* 添加参数 `-c 9238` 就可以把变更集 r9238 合并到你的工作副本里.

保持分支同步

继续我们的例子，假设自从你开始在自己的私有分支上工作后，时间过了一周，你要添加的新特性还未完成，但你知道在你工作的同时，团队里的其他人会 继续向项目的主干 */trunk* 提交修改. 最好把主干上 的修改复制到你自己的分支上，以便确保他们的修改能够与你的分支契合，这可 以通过 *自动同步合并* (*automatic sync merge*) 完成，自动同步合并的目的是为了让分支与祖先 分支上的修改保持同步. “自动” 合并的意思是用户只需要提供 合并所需的最小信息（也就是合并的源以及被合并的工作副本目标），至于哪些 修改需要合并则交由 Subversion 决定—在自动合并中，不需要通过 选项 `-r` 或 `-c` 向 *svn merge* 传递变更集.

经常保持分支与开发主线同步可以降低分支被合并到主干上时发生冲突 的概率.

Subversion 知道分支的历史，也知道它是在什么时候从主干上分离出来. 为了执行一个同步合并，首先要确保分支的工作副本是 “干净的” —也就是没有本地修改. 然后只需要执行：

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

$ svn merge ^/calc/trunk
--- Merging r341 through r351 into '.':
U    doc/INSTALL
U    src/real.c
U    src/button.c
U    Makefile
--- Recording mergeinfo for merge of r341 through r351 into '.':
 U   .
 $
```

命令 `svn merge` *`URL`* 告诉 Subversion 把 *`URL`* 上的所有未被合并 的修改都合并到当前工作副本上（在典型的情况下，也就是你的工作副本的 根目录）. 注意到我们用的是带有脱字符 (^) 的语法 [2]，这样我们 就不用输入完整的主干 URL 地址. 还要注意输出信息中的 “ Recording mergeinfo for merge…”，这是说合并正在 更新属性 `svn:mergeinfo`，我们会在本章后面的 “合并信息和预览”一节 介绍 `svn:mergeinfo`.

在本书及其他地方（包括 Subversion 邮件列表，讨论合并跟踪的文章等），你会经常听到一个术语 *合并信息* (*mergeinfo*)，其实它就是 属性 `svn:mergeinfo` 的缩写.

[2] 脱字符语法在 1.6 加入

在无需跟踪合并的情况下保持分支同步

用户可能无法使用 Subversion 的合并跟踪特性，也许是因为服务器端 的版本过老，也许是客户端过老. 这时候用户仍然可以执行合并操作，但 Subversion 需要用户手工完成历史计算工作，这些工作在支持合并跟踪的 版本中是自动完成的.

为了复制主干上最近的修改，你需要按照 “老式” 的方法 执行同步合并—指定你想合并的版本号范围.

继续使用前面的例子，用户已经知道从 */calc/trunk* 创建分支 */calc/branches/my-calc-branch* 的版本号 是 341:

```
$ svn log -v -r341
------------------------------------------------------------------------
r341 | user | 2013-02-15 07:41:25 -0500 (Fri, 15 Feb 2013) | 1 line
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

Creating a private branch of /calc/trunk.
------------------------------------------------------------------------
```

如果你已经准备好把主干的修改同步到分支上，首先指定 */calc/trunk* 被复制时的版本号作为起始版本号，然后把 */calc/trunk* 上最年轻的修改指定为结束 版本号，后者可以通过 `svn log -rHEAD` 找到:

```
$ svn log -q -rHEAD http://svn.example.com/repos/calc/trunk
------------------------------------------------------------------------
r351 | sally | 2013-02-16 08:04:22 -0500 (Sat, 16 Feb 2013)
------------------------------------------------------------------------

$ svn merge http://svn.example.com/repos/calc/trunk -r340:351
U    doc/INSTALL
U    src/real.c
U    src/button.c
U    Makefile
```

解决掉可能会出现的冲突后，就可以把合并后的修改提交到你的分支上. 现在，为了避免以后再浪费时间合并已经合并过的修改，用户最好把这次 合并的版本号范围记录下来，但是应该把它们记在哪里比较好? 最直接的地 方就是这次合并的提交日志消息:

```
$ svn ci -m "Sync the my-calc-branch with ^/calc/trunk through r351."
…
```

在下一次把 */calc/branches/my-calc-branch* 向 */calc/trunk* 同步时，步骤是一样的，除了把 起始版本号改成已经合并过的 最年轻的 版本号. 如果用户在提交日志里记录了关于合并的信息，就可以很方便地在提交日 志消息里找到起始版本号. 起始版本号确定后，就可以再次执行同步合并:

```
$ svn log -q -rHEAD http://svn.example.com/repos/calc/trunk
------------------------------------------------------------------------
r959 | sally | 2013-03-5 7:30:21 -0500 (Tue, 05 Mar 2013)
------------------------------------------------------------------------

$ svn merge http://svn.example.com/repos/calc/trunk -r351:959
…
```

执行完上面的例子后，分支的工作副本就包含了本地修改，而且这些修改 都是创建完分支后，主干上的修改的副本：

```
$ svn status
 M      .
M       Makefile
M       doc/INSTALL
M       src/button.c
M       src/real.c
```

这时候比较明智的操作是使用 *svn diff* 查看修 改的内容，并构建测试分支里的代码. 注意当前工作目录 (“ .”) 也被修改了，*svn diff* 显示它新增了 `svn:mergeinfo` 属性.

```
$ svn diff --depth empty .
Index: .
===================================================================
--- .   (revision 351)
+++ .   (working copy)

Property changes on: .
___________________________________________________________________
Added: svn:mergeinfo
   Merged /calc/trunk:r341-351
```

这个属性是非常重要的与合并相关的元数据，用户 不 应该直接修改它的值，因为后面的 *svn merge* 会用到该 属性 (关于合并元数据的更多内容，我们稍后就会进行介绍).

执行完合并后，可能会有冲突需要处理—就像执行完 *svn update* 那样—或者可能还需要进行一些小修改，保证 合并的结果是正确的 (记住，没有 *语法* 冲突并不表 示没有 *语义* 冲突!). 如果合并后产生了很多问题，用户总是可以用 `svn revert . -R` 撤消本地的所有 修改，然后就可以和同事讨论 “怎么回事”. 如果一切都很顺利，用户就可以把修改提交到仓库里：

```
$ svn commit -m "Sync latest trunk changes to my-calc-branch."
Sending        .
Sending        Makefile
Sending        doc/INSTALL
Sending        src/button.c
Sending        src/real.c
Transmitting file data ....
Committed revision 352.
```

现在，用户的私有分支就和主干 “同步” 了，用户也就不用 担心自己的工作和其他人的相差太远.

为何不用补丁?

Unix 用户可能会想: 何必要用 *svn merge*, 为什 么不直接使用 *svn patch* 或操作系统的 *patch* 命令完成同样的工作? 例如:

```
$ cd my-calc-branch

$ svn diff -r 341:351 ^/calc/trunk > my-patch-file

$ svn patch my-patch-file
U         doc/INSTALL
U         src/real.c
U         src/button.c
U         Makefile
```

对于这个例子而言, *svn merge* 与 *svn patch*, *patch* 相比, 的确没什么区 别, 但 *svn merge* 拥有 *patch* 所不具备的功能. *patch* 可使用的补丁格式很有限, 它只能修改文件内容, 却无法表示 目录结构 的变化, 例如文件和目录的添加, 删除或移动, *patch* 也不能 识别属性的修改. 如果 Sally 添加了一个新目录, *svn diff* 的输出也体现不出这一修改. *svn diff* 的 输出格式也有限制, 所以它也无法表示某些修改. 即使是 Subversion 自己的 *svn patch* 命令 (比操作系统的 *patch* 命令灵活得多) 也有类似的限制.

而命令 *svn merge* 通过直接修改工作副本来表示 目录结构和属性上的变化. 更重要的是 *svn merge* 会 把已经复制到分支上的修改记录下来, 从而 Subversion 可以精确地知道 每个位置都存在着哪些修改 (见 “合并信息和预览”一节). 这个特性 使得分支是可管理的, 否则的话用户必须手工记录哪些修改已经合并, 哪些 没有.

假设又过去了一周, 你在自己的分支上提交了更多的修改, 而你的同事也 在不断地修改主干. 再一次, 你想把主干上的修改合并到自己的分支上, 于是 执行下面的命令:

```
$ svn merge ^/calc/trunk
svn: E195020: Cannot merge into mixed-revision working copy [352:357]; try up\
dating first
$
```

这种情况可能不在用户的预料之中! 在自己的分支了工作了一周后, 你 发现工作副本包含了混合的版本号 (见 “版本号混合的工作副本”一节). 1.7 及之后版本的 *svn merge* 在默认情况下禁止向含有混合版本号的工作 副本合并, 简单来说, 这是属性 `svn:mergeinfo` 合并跟踪 方式的限制导致的 (见 “合并信息和预览”一节), 这些限制意味 着向一个含有混合版本号的工作副本合并将导致无法预料的内容与目录冲突[3]. 我们不希望产生任何不必要的冲突, 所以先更新工作副 本, 然后再尝试合并.

```
$ svn up
Updating '.':
At revision 361.

$ svn merge ^/calc/trunk
--- Merging r352 through r361 into '.':
U    src/real.c
U    src/main.c
```

[3]命令 *svn merge* 的选项 `--allow-mixed-revisions` 允许用户关闭这个限制, 但是这样做的前提是用户必须理解可能的后果, 以及动机要足够充分.

```
--- Recording mergeinfo for merge of r352 through r361 into '.':
 U   .
```

Subversion 知道主干上的哪些修改已经合并到了分支上，所以它只会合并 那些未合并过的主干修改. 如果构建和测试都没有问题，用户就可以用 *svn commit* 把分支的修改提交到仓库里.

子目录合并与子目录合并信息

在本章的大部分例子中，被合并的目标都是分支（见 “什么是分支”一节) 的根目录，虽然这是最常见的 情况，但是偶尔也需要直接合并分支的子目录，这种类型的合并称为 子目录合并 (*subtree merge*)，它的合并信息也相应地称为 子目录合并信息 (*subtree mergeinfo*). 子目录合并和子目录合并信息其实并没有什么特别的地方，唯一需要注意的一点是：一个分支上完整的合并记录可能不仅仅记录在分支根 目录的合并信息里，可能还要查看子目录的合并信息才能得到完整的合并信息. 幸运的是 Subversion 会替用户完成这些操作，用户几乎不需要直接参与，用一个简单的例子解释一下：

```
# We need to merge r958 from trunk to branches/proj-X/doc/INSTALL,
# but that revision also affects main.c, which we don't want to merge:
$ svn log --verbose --quiet -r 958 ^/
------------------------------------------------------------------------
r958 | bruce | 2011-10-20 13:28:11 -0400 (Thu, 20 Oct 2011)
Changed paths:
   M /trunk/doc/INSTALL
   M /trunk/src/main.c
------------------------------------------------------------------------

# No problem, we'll do a subtree merge targeting the INSTALL file
# directly, but first take a note of what mergeinfo exists on the
# root of the branch:
$ cd branches/proj-X

$ svn propget svn:mergeinfo --recursive
Properties on '.':
  svn:mergeinfo
    /trunk:651-652

# Now we perform the subtree merge, note that merge source
# and target both point to INSTALL:
$ svn merge ^/trunk/doc/INSTALL doc/INSTALL -c 958
--- Merging r958 into 'doc/INSTALL':
U    doc/INSTALL
--- Recording mergeinfo for merge of r958 into 'doc/INSTALL':
 G   doc/INSTALL

# Once the merge is complete there is now subtree mergeinfo on INSTALL:
$ svn propget svn:mergeinfo --recursive
Properties on '.':
  svn:mergeinfo
    /trunk:651-652
Properties on 'doc/INSTALL':
```

```
  svn:mergeinfo
    /trunk/doc/INSTALL:651-652,958

# What if we then decide we do want all of r958? Easy, all we need do is
# repeat the merge of that revision, but this time to the root of the
# branch, Subversion notices the subtree mergeinfo on INSTALL and doesn't
# try to merge any changes to it, only the changes to main.c are merged:
$ svn merge ^/subversion/trunk . -c 958
--- Merging r958 into '.':
U    src/main.c
--- Recording mergeinfo for merge of r958 into '.':
 U   .
--- Eliding mergeinfo from 'doc/INSTALL':
 U   doc/INSTALL
```

你可能会感到奇怪，为什么上面的例子里我们只合并了 r958，但 *INSTALL* 却含有 r651-652 的合并信息，这是由于合并 信息的继承性，合并信息的继承性我们会在 合并信息继承 介绍. 另外还要注意 *doc/INSTALL* 上的子目录合并信息 被移除了，或者说被 “省略” 了，这被称为 合并信息 省略 (*mergeinfo elision*)，当 Subversion 检测到多余的子目录合并信息时，就会发生这种现象.

在 Subversion 1.7 版之前，合并操作会无条件地更新目标所有的子 目录合并信息，对于拥有大量子目录合并信息的用户而言，即使是相对比较 简单的合并（例如只合并了一个文件），也会影响所有子目录的合并信息，甚至包括那些不是受影响路径的父目录的目录，在某种程度上会让人感到 困惑和沮丧. Subversion 1.7 以及之后的版本解决了这个问题，方法是只 更新受影响路径（也就是通过应用差异而被修改，添加或删除的路径，见 “合并语法详解”一节) 的父目录的子目录合并信息，有一个例外是目标的合并信息 总是会被更新，即使被应用的差异不会产生任何修改.

重新整合分支

如果用户完成了分支上的所有工作，也就是说新特性已经完成，你已经准备好 把分支合并到主干上（这样的话团队中的其他成员就可以分享你的工作成果），合并 的步骤很简单，首先把分支与主干同步，就像之前做过的那样[4]

```
$ svn up # (make sure the working copy is up to date)
Updating '.':
At revision 378.

$ svn merge ^/calc/trunk
--- Merging r362 through r378 into '.':
U    src/main.c
--- Recording mergeinfo for merge of r362 through r378 into '.':
 U   .

$ # build, test, ...
```

[4] 从 Subversion 1.7 开始，用户并非一定要像例子中演示的那样，每次都将分支 的根目录与主 干进行同步. 如果 分支已经通过一系列的子目录合并，在效果上已经实现了与主干的同步，后面的再整合合并也可以正常工作. 但是请 读者扪心自问，如果分支在效果上已经同步了，那为什么还要再做子目录合并呢? 此时再做子目录合并只会带来无谓的复杂性.

```
$ svn commit -m "Final merge of trunk changes to my-calc-branch."
Sending        .
Sending        src/main.c
Transmitting file data .
Committed revision 379.
```

现在，使用 *svn merge* 把分支上的修改合并到主干 上，这种类型的合并称为 “自动再整合” (automatic reintegrate) 合并，在执行合并之前，用户需要一份 */calc/trunk* 的工作 副本，可以用 *svn checkout* 或 *svn switch* （见 “遍历分支”一节) 获取.

术语 “再整合” 来自子命令 *merge* 的选项 `--reintegrate`. 该选项在 Subversion 1.8 被 废弃 (1.8 可以自动检测什么时候才需要执行再整合合并)，但是 1.5 到 1.7 版的 Subversion 客户端在执行再整合合并时都要求提供该选项.

在合并分支前，主干的工作副本不能含有本地修改，已切换的路径，或混合 的版本号（见 “版本号混合的工作副本”一节)，这种状 态不仅会带来很多方便，而且是自动再整合合并所要求的.

一旦准备好了一个整洁的主干工作副本，用户就可以把分支合并到主干上了：

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

$ svn update
Updating '.':
At revision 379.

$ svn merge ^/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
U    src/real.c
U    src/main.c
U    Makefile
--- Recording mergeinfo for merge between repository URLs into '.':
 U   .

$ # build, test, verify, ...

$ svn commit -m "Merge my-calc-branch back into trunk!"
Sending        .
Sending        Makefile
Sending        src/main.c
Sending        src/real.c
Transmitting file data ...
Committed revision 380.
```

恭喜，你在分支上提交的修改现在都已经合并到了开发主线. 应该注意的 是和你到目前为止所做的合并操作相比，自动再整合合并所做的工作不太一样. 之前我们是要求 *svn merge* 从另一条开发线（主干）上 抓取下一个变更集，然后把变更集复制到另一条开发线（你的私有分支）上. 这种操作非常直接，Subversion 每一次都知道如何从一次停止的地方开始. 在我们前面讲过的例子里，Subversion 第一次是把 */calc/trunk* 的 r341-351 合并到 */calc/branches/my-calc-branch*，后来它就继续合并下一段范围，r351-361，在最后一次同步，它又合并了 r361-378.

然而，在把 */calc/branches/my-calc-branch* 合并到 */calc/trunk* 时，其底层的数学行为是非常 不一样的．特性分支现在已经是同时包含了主干修改和分支私有修改的大杂烩，所以没办法简单地复制一段连续的版本号范围．通过使用自动再整合合并，你是在要求 Subversion 只复制那些分支特有的修改（具体的实现方式是比较最新版的分支 与主干，最终得到的差异就是分支所特有的修改）．

始终记住自动再整合合并只支持上面描述的使用案例，由于这个狭隘的 重点，除了前面提到的要求（最新的工作副本[5]，不含有混合的版本号，已切换的路径或本地修改）外，*svn merge* 的大部分选项都会使它不能正常工作，如果用户用到了除 `--accept`, `--dry-run`, `--diff3-cmd`, `--extensions`, `--quiet` 之外的其他非全局选项，将会得到一个错误．

既然你的私有分支已经合并到了主干上，现在就可以把它删除了：

```
$ svn delete ^/calc/branches/my-calc-branch \
             -m "Remove my-calc-branch, reintegrated with trunk in r381."
```

…

不过，分支的历史不是很重要吗？如果有人想查看分支的每一次修改，审查特性的演变怎么办？不用担心，虽然你的分支在 */calc/branches* 再也看不到了，但是它在仓库的历史 里依然存在．在 */calc/branches* 的 URL 上执行 一个简单的 *svn log*，就可以看到分支的全部历史．你的分支甚至可以某一时刻复活，你期待吗（见 “恢复已删除的文件”一节）．

分支被合并到主干后，如果选择不删除分支，你可能会继续从主干同步 修改，然后再次重新整合分支[6]．如果你这样做了，那么只有第一次重新整合之后发生的 修改 才会被合并到主干上．

合并信息和预览

Subversion 跟踪变更集的基本机制—也就是判断哪些修改已经合并到哪些 分支上—是在版本化的属性中记录数据．更确切地说，与合并相关的数据 记录在文件和目录的 `svn:mergeinfo` 属性中．（如果读者 还不了解 Subversion 的属性，见 “属性”一节.）

你可以像查看其他属性那样，查看属性 `svn:mergeinfo`：

```
$ cd my-calc-branch

$ svn pg svn:mergeinfo -v
Properties on '.':
  svn:mergeinfo
    /calc/trunk:341-378
```

虽然你可以像修改其他属性那样修改 `svn:mergeinfo`，但我们强烈建议你不要这么做，除非你真地知道自己在做什么．

在单个路径上的 `svn:mergeinfo` 数量可以变得 非常庞大，*svn propget --recursive* 和 *svn proplist --recursive* 在处理大量的子目录合并 信息时，命令的输出也会很多（见 “子目录合并与子目录合并信息”一节）．为了减少输出的内容，比较常用的方法是为命令添加选项 `--verbose`．

[5] 自动再整合合 并也支持目标是浅检出的目录（见 “稀疏目录”一节），但是如果受影响的路径 由于目录是稀疏的，而不出现在工作副本中，那么该路径就会被忽略— 这可能 不是 用户想要的结果！

[6] 只有 Subversion 1.8 允许 这样重用一个特性分支．较早的版本要求一个特性分支在被多次重新整合 之前，需要一些特殊的处理，更多的信息参见本书较早的版本：http://svnbook.red-bean.com/en/1.7/svn.branchmerge.basicmerging.html#svn.branchemerge.basicmerging.reintegrate

当用户执行 *svn merge* 时，Subversion 就会自动 更新属性 `svn:mergeinfo`，属性的值指出了给定路径上 的哪些修改已经复制到目录上. 在我们之前的例子里，修改的来源是 */calc/trunk*，被合并的目录是 */calc/branches/my-calc-branch*. 旧版的 Subversion 会悄无 声息地维护属性 `svn:mergeinfo`，合并后，用户仍然可 以用命令 *svn diff* 或 *svn status* 查看合并产生的修改，但是当合并操作修改属性 `svn:mergeinfo` 时不会显示任何提示信息. 而 Subversion 1.7 及以后的版本就 不再这样了，当合并操作更新属性 `svn:mergeinfo` 时，Subversion 会给出一些提示信息. 这些提示信息都是以 `--- Recording mergeinfo for` 开始，在合并的末尾输出. 不像其他的合并提示信息，这些信息不是在描述差异被应用到工作副本（见“合并语法详解”一节），而是在描述 为了跟踪合并而产生的 “家务” 变化.

Subversion 提供了子命令 *svn mergeinfo*，用于查看 两个分支间的合并关系，特别是查看目录吸收了哪些变更集，或者查看哪些变 更集它是有资格吸收的，后者提供了一种预览，预览随后的 *svn merge* 命令将会复制哪些修改到分支上. 在默认情况下，*svn mergeinfo* 将会输出两条分支之间的关系的图形化 概览. 回到我们先前的例子，用命令 *svn mergeinfo* 分析 */calc/trunk* 和 */calc/branches/my-calc-branch* 之间的关系：

```
$ cd my-calc-branch

$ svn mergeinfo ^/calc/trunk
    youngest common ancestor
    |         last full merge
    |         |        tip of branch
    |         |        |         repository path

    340               382
    |                 |
  -------| |-----------         calc/trunk
     \          /
      \        /
       --| |-----------         calc/branches/my-calc-branch
             |        |
             379      382
```

图中显示了 */cal/branches/my-calc-branch* 拷贝 自 */calc/trunk@340*，最近的一次自动合并是从分支到 主干的自动再整合合并，在版本号 380. 注意到图中 没有 显示我们在版本号 352, 362, 372 和 379 执行的自动同步合并，在每个 方向上只显示了最近的自动合并 [7] 这种默认输出对于获取两个分支之间的合并概览非常有用，如果想要清楚地看到分支上合并了哪些版本号，就增加选项 `--show-revs=merged`:

```
$ svn mergeinfo ^/calc/trunk --show-revs merged
r344
r345
r346
…
r366
r367
r368
```

同样地，为了查看分支可以从主干上合并哪些修改，就用选项 `--show-revs=eligible`:

```
$ svn mergeinfo ^/calc/trunk --show-revs eligible
```

[7] 这里的 “方向” 指的是从主干到分支（自动同步）或从分支到主干（自动再整合）的合并.

```
r380
r381
r382
```

可实施与不可实施的版本号

选项 `--show-revs` 列出的版本号只包含了在合并时（将）会产生实际修改的版本号，所以说当用户从 */calc/trunk* 合并一段连续的版本号（例如 r341-378) 到 */calc/branches/my-calc-branch* 时，只有那些包含在选项 `--show-revs=merged` 输出的列表中的版本号才真正地含 有针对 */cal/trunk* 的修改，对于合并而言，这些 版本号被称为 “可实施的版本号”，注意不要和选项 `-r` 使用的实施版本号搞混，实施版本号见 “限定版本号与实施版本号”一节. 反之，r341-378 中未出现在 选项 `--show-revs=merged` 输出列表中的版本号被称 为 “不可实施的版本号”.

命令 *svn mergeinfo* 需要一个 “源” URL（修改的来源），接受一个可选的 “目标” URL（合并修改 的目标）. 如果没有指定目标 URL，命令就把当前工作目录当成目标. 在上面 的例子里，因为我们要查询的是分支工作副本，命令假定我们想知道的是主干 URL 上的哪些修改可以合并到 */calc/branches/my-calc-branch* .

从 Subversion 1.7 开始，*svn mergeinfo* 也可以 描述子目录合并信息和不可继承的合并信息. 为了描述子目录合并信息，要加上选项 `--recursive` 或 `--depth`，而不可继承的合并信息本来就会被考虑到.

合并信息继承

如果某个路径具有属性 `svn:mergeinfo`，我们就说它 含有 显式的合并信息 (*explicit mergeinfo*). 显式的合并信息不仅描述了合并到属性所在的 目录上的修改，还描述了目录的子文件和子目录（因为子文件和子目录继承了 父目录的合并信息），例如：

```
# What explicit mergeinfo exists on a branch?
$ svn propget svn:mergeinfo ^/branches/proj-X --recursive
/trunk:651-652

# What children does proj-X have?
$ svn list --recursive ^/branches/proj-X
doc/
doc/INSTALL
README
src/main.c

# Ask what revs were merged to a file with no explicit mergeinfo
$ svn mergeinfo ^/trunk/src/main.c ^/branches/proj-X/src/main.c \
                --show-revs merged
651
652
```

执行的第一个命令中，只有 */branches/proj-X* 的根目录含有显式的合并信息，然而，如果我们用 *svn mergeinfo* 询问哪些修改被合并到 */branches/proj-X/src/main.c*，命令报告有两个 版本号被合并到 */branches/proj-X/src/main.c*，而这两个版本号也出现在 */branches/proj-X* 显式 的合并信息中. 这是因为 */branches/proj-X/src/main.c* 没有属于它自己的 显式合并信息，它的合并信息继承自最近的会被继承 的祖先路径，也就是 */branches/proj-X*.

在两种情况下合并信息会被继承. 第一种情况是如果一个路径含有显式的合并信息，该路径就不会继承任何合并信息，我们可以这样理解：一个路径 上的显式合并信息总是一个完整的合并记录. 一旦存在显式的合并信息，它就 会覆盖路径可能从其他地方继承而来的合并信息. 第二种情况是合并信息本 身是不可 继承的，这种特殊类型的合并信息 只能应用到设置了 属性 `svn:mergeinfo` 的目录上（不可继承的合并信息不 能设置在文件上），例如：

```
# The '*' decorator indicates non-inheritable mergeinfo
$ svn propget svn:mergeinfo ^/branches/proj-X
/trunk:651-652,758*

# Revision 758 is non-inheritable, but still applies to the path it is
# set on. Here the '*' decorator signals that r758 is only partially
# merged from trunk.
$ svn mergeinfo ^/trunk ^/branches/proj-X --show-revs merged
651
652
758*

# Revision 758 is not reported as merged because it is non-inheritable
# and applies only to ^/trunk
$ svn mergeinfo ^/trunk/src/main.c ^/branches/proj-X/src/main.c \
                --show-revs merged
651
652
```

在你自己的仓库中，你可能永远不需要考虑合并信息的继承，也不会遇到 不可继承的合并信息. 关于合并信息继承的完整讨论已经超出了本书的范围，如果读者还有问题没有想明白，可以阅读 “关于合并跟踪的最后一点内容”一节 列出的参考资料.

假设有一个分支同时包含了子目录合并信息和不可继承的合并信息:

```
$ svn pg svn:mergeinfo -vR
# Non-inheritable mergeinfo
Properties on '.':
  svn:mergeinfo
    /calc/trunk:354,385-388*
# Subtree mergeinfo
Properties on 'Makefile':
  svn:mergeinfo
    /calc/trunk/Makefile:354,380
```

从合并信息中可以看到 r385-388 只被合并到了分支的根目录上，但不 包括任何一个子文件. 还可以看到 r380 只被合并到了 *Makefile* 上. 如果用带上选项 `--recursive` 的 *svn mergeinfo* 查看从 */calc/trunk* 那里合并了哪些版本号到这个分支上，我们可以看到其中三个版本号带有星号 标记:

```
$ svn mergeinfo -R --show-revs=merged ^/calc/trunk .
r354
r380*
r385
r386
r387*
r388*
```

星号 * 表示该版本号只是被 部分地 合并到目标上（对于 `--show-revs=eligible`, 其星号的意义是 相同的）. 对于这个例子而言，它的意思是说如果我们尝试从 *^/trunk* 合并 r380, r387 或 r388, 将会产生更多的修改. 同样地，因为 r354, r385 和 r386 没有 被星号标记，所以再次合并这些版本号将不会产生任何修改. [8]

获取合并预览的另一种办法是使用选项 `--dry-run`:

```
$ svn merge ^/paint/trunk paint-feature-branch --dry-run
--- Merging r290 through r383 into 'paint-feature-branch':
U    paint-feature-branch/src/palettes.c
U    paint-feature-branch/src/brushes.c
U    paint-feature-branch/Makefile

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

选项 `--dry-run` 不会真正地去修改工作副本，它只会 输出一个真正的合并操作 将会 输出的信息. 如果嫌 *svn diff* 的输出过于详细，就可以用这个选项获得一个 比较 “高层的” 合并预览.

执行完合并，但是在提交之前，可以用 `svn diff --depth=empty /path/to/merge/target` 查看被合并的直接目标的修改，如果被合并的目标是目录，那么命令就只会输出属性的修改. 这是查看合并后属性 `svn:mergeinfo` 的变化的简便方法，从属性的 变化中可以看到这次合并合并了哪些版本号.

[8] 这是不可实施的 合并版本号的好例子.

当然，预览合并的最佳方法是执行合并．记住，执行 *svn merge* 并不是一个危险的操作（除非在合并前，工作副本含有本地修改，但我们已经强调过不要在这种情况下执行合并）．如果你不喜欢合并的结果，执行 `svn revert . -R` 就可以撤消合并产生的修改．只有在执行了 *svn commit* 后，合并的结果才会被提交到 仓库中．

撤消修改

人们经常使用 *svn merge* 撤消已经提交的修改．假设你正开心地在 */calc/trunk* 的工作副本上工作，突然发现版本号 392 提交的修改是完全错误的，它就不应该被提交．此时你可 以用 *svn merge* 在工作副本中 “撤消” 版本号 392 的修改，然后把用于撤消 r392 的修改提交到仓库中．你所要做 的只是指定一个 *逆* 差异（对于这个例子而言，指定 逆差异的命令行参数是 `--revision 392:391` 或 `--change -392`）．

```
$ svn merge ^/calc/trunk . -c-392
--- Reverse-merging r392 into '.':
U    src/real.c
U    src/main.c
U    src/button.c
U    src/integer.c
--- Recording mergeinfo for reverse merge of r392 into '.':
 U   .

$ svn st
M       src/button.c
M       src/integer.c
M       src/main.c
M       src/real.c

$ svn diff
…
# verify that the change is removed
…

$ svn commit -m "Undoing erroneous change committed in r392."
Sending        src/button.c
Sending        src/integer.c
Sending        src/main.c
Sending        src/real.c
Transmitting file data ....
Committed revision 399.
```

我们以前说过，可以把版本号当成一个特定的变更集，通过选项 `-r`，可以要求 *svn merge* 向工作副本应用一 个特定的变更集，或一段变更集范围．在上面这个例子里，我们是要求 *svn merge* 把变更集 r392 的逆修改应用到工作副本上．

记住，像这样撤消修改和其他 *svn merge* 操作一样，用户应该用 *svn status* 和 *svn diff* 确认修改的内容正是心里所期望的那样，检查没问题后再用 *svn commit* 提交．提交后，在 HEAD 上就再也看 不到 r392 的修改．

读者可能在想：好吧，其实并没有真正地撤消提交，版本号 392 的修改仍然 存在于历史中，如果有人检出了版本在 r392 到 r398 之间的 *calc*，他就会看到错误的修改，对吧？

说得没错，当我们谈论 “删除” 一个修改时，我们实际上说得是 把修改从版本号 HEAD 中删除，原始的修改仍然存在于仓库中 的历史中．在大多数时候，这种做法已经足够好了，毕竟大多数人只对项目的 HEAD 感兴趣．然而，在少数情况下，用户可能真地需要把 提交从仓库的历史中完全擦除（可能是不小心提交了一份机密文档）．这做起来并不 容易，因为 Subversion 的设计目标之一是不能丢失任何一个修改，版本号是以其他 版本号为基础的不可修改的目录树，从历史中删除一个版本号将会产生多米诺骨牌 效应，使后面的版本号产生混乱，甚至可能会使所有的工作副本失效．[9]

恢复已删除的文件

版本控制系统的一大好处是信息永远不会丢失．即使你删除了一个文件或 目录，虽然在版本号 HEAD 中已经看不到被删除的文件，但它们在早先的版本中仍然存在．新用户经常问的一个问题是 “怎样才 能找回以前的文件或目录?”

第一步是准确地指定你想要恢复的是哪一项条目．一种比较形象的比喻是 把仓库中的每个对象都想像成一个二维坐标，第一个坐标是特定的版本号目录 树，第二个坐标是目录内的路径，于是文件或目录的每一个版本都可以由一对 坐标唯一地确定．

首先，用户可能要用 *svn log* 找到他想恢复的二维 坐标，比较好的策略是在曾经含有被删除的项目的目录中运行 `svn log --verbose`，选项 `--verbose(-v)` 显示了在每个版本号中 被修改的所有项目，你所要做的就是找到那个删除了文件或目录的版本号．用户 可以依靠自己的肉眼寻找，也可以借助其他工具（例如 *grep* ）扫描 *svn log* 的输出．如果用户已经知道 待恢复的项目是在最近的提交中才被删除，那还可以用选项 `--limit` 限制 *svn log* 的输出．

```
$ cd calc/trunk

$ svn log -v --limit 3
------------------------------------------------------------------------
r401 | sally | 2013-02-19 23:15:44 -0500 (Tue, 19 Feb 2013) | 1 line
Changed paths:
   M /calc/trunk/src/main.c

Follow-up to r400: Fix typos in help text.
------------------------------------------------------------------------
r400 | bill | 2013-02-19 20:55:08 -0500 (Tue, 19 Feb 2013) | 4 lines
Changed paths:
   M /calc/trunk/src/main.c
   D /calc/trunk/src/real.c

* calc/trunk/src/main.c: Update help text.

* calc/trunk/src/real.c: Remove this file, none of the APIs
  implemented here are used anymore.
------------------------------------------------------------------------
r399 | sally | 2013-02-19 20:05:14 -0500 (Tue, 19 Feb 2013) | 1 line
Changed paths:
   M /calc/trunk/src/button.c
   M /calc/trunk/src/integer.c
   M /calc/trunk/src/main.c
```

[9]Subversion 已经计划在未来的某一天，能够实现永久地删除提交历史，但在 Subversion 实现之前，可以从 “svndumpfilter” 一节 找到变通 办法．

```
   M /calc/trunk/src/real.c

Undoing erroneous change committed in r392.
------------------------------------------------------------------------
```

在上面的例子里，我们假设要找的文件是 *real.c*，通过查看父目录的日志，可以看到 *real.c* 是在版本号 400 被删除．因此，*real.c* 的最后一个版本就是紧挨着 400 的前一个版本号，也 就是说你要从版本号 399 中恢复 */calc/trunk/real.c*.

这本来是最难的地方—调研．既然已经知道了要复原的是哪个项目，接下来你有两个选择．

其中一个选择是使用 *svn merge* “反向” 应用版本号 400（我们已经在 “撤消修改”一节 介绍了如何撤消修改）．命令的效果是把 *real.c* 重新添加到工作副本里，提交 后，文件将重新出现在版本号 HEAD 中．

然而对于我们这个例子而言，可能并不是最好的办法．反向应用版本号 400 不仅会添加 *real.c*，从版本号 400 的提交日志 可以看到，反向应用还会撤消 *main.c* 的某些修改，这应该不是用户想要的效果．当然，你也可以在逆合并完 r400 后，再手动地 对 *main.c* 执行 *svn revert*．但这种解决办法可扩展性不好，如果有 90 个文件在 r400 中被修改了，难道 也要一个个地执行 *svn revert* 吗?

第二种选择的目的性更强，不使用 *svn merge*，而是 用 *svn copy* 从仓库中复制特定的版本号与路径“坐标” 到工作副本里：

```
$ svn copy ^/calc/trunk/src/real.c@399 ./real.c
A         real.c

$ svn st
A  +    real.c

# Commit the resurrection.
…
```

状态输出中的加号表示这个项目不仅仅是新增的， 而且还带有历史信息， Subversion 知道它是从哪里复制来的． 以后对 *real.c* 执行 *svn log* 将会遍历到 r399 之前的历史，也就是说 *real.c* 并不是真正的新文件，它是已删除的原始文件 的后继，通常这就是用户想要的效果．然而，如果你不想维持文件以前的历史，还可以下面的方法恢复文件：

```
$ svn cat ^/calc/trunk/src/real.c@399 > ./real.c

$ svn add real.c
A         real.c

# Commit the resurrection.
…
```

虽然我们的例子都是在演示如何恢复被删除的文件，但同样的技术也可以 用在恢复目录上．另外，恢复被删除的文件不仅可以发生在工作副本中，还可 直接发生在仓库中：

```
$ svn copy ^/calc/trunk/src/real.c@399 ^/calc/trunk/src/real.c \
           -m "Resurrect real.c from revision 399."

Committed revision 402.

$ svn up
```

```
Updating '.':
A    real.c
Updated to revision 402.
```

高级合并

一旦用户开始频繁地使用分支与合并，很快就会要求 Subversion 把一个 特定的 的修改从一个地方合并到另一个地方. 为了完成 这项工作，用户要给 *svn merge* 传递更多的参数，下一节 将对命令的语法进行完整地介绍，同时还将讨论它们的典型应用场景.

精选

和术语 “变更集” 一样，术语 精选 (*cherrypicking*) 也经常出现在版本控制系统中. 精选指的是这样一种操作: 从分支中挑选 一个 特定的 变更集，将其复制到其他地方. 精选也可以指这样一种操作: 将一个特定的变更集 集合 (不一定是连续的) 从一个分支复制到另一个分支上. 这和典型的合并 场景相反 (典型的合并场景是自动合并下一段版本号范围).

为什么会有人只想复制单独的一个修改? 这种情况要比你想像的更常发生，假设你从 */calc/trunk* 创建了一个特性分支 */calc/branches/my-calc-feature-branch*:

```
$ svn log ^/calc/branches/new-calc-feature-branch -v -r403
------------------------------------------------------------------------
r403 | user | 2013-02-20 03:26:12 -0500 (Wed, 20 Feb 2013) | 1 line
Changed paths:
   A /calc/branches/new-calc-feature-branch (from /calc/trunk:402)

Create a new calc branch for Feature 'X'.
------------------------------------------------------------------------
```

在饮水机接水时，你听说 Sally 向主干上的 *main.c* 提交了一个很重要的修改，通过查看主干的提交历史，你发现在版本号 413, Sally 修正了一个很严重的错误，而这个错误也会影响你正在开发的新特性. 你的分支可能还没有准备好合并主干上的所有修改，但是为了能让工作继续 下去，你确实需要 r413 的修改.

```
$ svn log ^/calc/trunk -r413 -v
------------------------------------------------------------------------
r413 | sally | 2013-02-21 01:57:51 -0500 (Thu, 21 Feb 2013) | 3 lines
Changed paths:
   M /calc/trunk/src/main.c

Fix issue #22 'Passing a null value in the foo argument
of bar() should be a tolerated, but causes a segfault'.
------------------------------------------------------------------------

$ svn diff ^/calc/trunk -c413
Index: src/main.c
===================================================================
--- src/main.c  (revision 412)
+++ src/main.c  (revision 413)
@@ -34,6 +34,7 @@
```

```
…
# Details of the fix
…
```

就像上面例子中的 *svn diff* 查看 r413 那样，你也可以向 *svn merge* 传递相同的选项：

```
$ cd new-calc-feature-branch

$ svn merge ^/calc/trunk -c413
--- Merging r413 into '.':
U    src/main.c
--- Recording mergeinfo for merge of r413 into '.':
 U   .

$ svn st
 M      .
M       src/main.c
```

如果测试后没什么问题，就可以把修改提交到仓库中．提交后，Subversion 更新分支属性 `svn:mergeinfo`，以反映 r413 已经合并到 分支中，这可以避免今后自动同步合并时再去合并 r413（在同一分支内多次 合并同一修改通常会导致冲突）．还要注意合并信息 `/calc/branches/my-calc-branch:341-379`，这条信息 是早先 */calc/trunk* 在 r380 再整合合并 */calc/branches/my-calc-branch* 时记录的，当我们 在 r403 创建分支 *my-calc-feature-branch* 时，这条合并信息也被一并复制．

```
$ svn pg svn:mergeinfo -v
Properties on '.':
  svn:mergeinfo
    /calc/branches/my-calc-branch:341-379
    /calc/trunk:413
```

从下面 *mergeinfo* 的输出中可以看到，r413 并没有 被列为可合并的版本号，这是因为它已经被合并了：

```
$ svn mergeinfo ^/calc/trunk --show-revs eligible
r404
r405
r406
r407
r409
r410
r411
r412
r414
r415
r416
…
r455
r456
r457
```

上面的输出表示当分支要自动同步合并主干时，Subversion 将把合并分成 两步进行，第一步是合并所有可合并的修改，直到 r412，第二步是从 r414 开始 合并所有可合并的修改，直到 HEAD．因为我们已经合并了 r413，所以它会被跳过：

```
$ svn merge ^/calc/trunk
--- Merging r403 through r412 into '.':
U    doc/INSTALL
U    src/main.c
U    src/button.c
U    src/integer.c
U    Makefile
U    README
--- Merging r414 through r458 into '.':
G    doc/INSTALL
G    src/main.c
G    src/integer.c
G    Makefile
--- Recording mergeinfo for merge of r403 through r458 into '.':
 U   .
```

将一个新版本上的修改 回植 (*backporting*) 到另一个分支可能是精选修改最常见的需求，例如，当开发团队在维护软件的“发布分支” 时，就会经常遇到 这种情况（见 “发布分支”一节.)

在上面的例子里，读者是否注意到了 Subversion 如何合并两个不同的版 本号范围? *svn merge* 为了跳过变更集 413（分支已经包 含了变更集 413), 向工作副本应用了两个独立的补丁，这么做除了可能会使 冲突更难解决之外，本身并没有什么不对的地方. 如果第一段修改范围产生了 冲突，那么只有在冲突解决后才能接着应用第二段修改范围. 如果在出现冲突 后，用户选择推迟解决冲突，那么命令 *svn merge* 将会报错退出，在第二次运行合并命令之前，用户必须解决冲突.

提醒一句：*svn diff* 和 *svn merge* 在概念上非常类似，但在很多情况下它们使用不同的语法，详情 见 svn 参考手册—Subversion 命令行客户端. 比如说 *svn merge* 要求一个工作副本路径作为被合并的操作目标，如果没有指定，命令就假设是以 下两种情况之一：

- 被合并的是当前工作目录.
- 用户想把特定文件上的修改合并到当前工作目录的同名文件上.

如果用户在合并目录时没有指定目标路径，*svn merge* 就认为是第一种情况，尝试把修改合并到当前工作目录. 如果是合并文件，并 且这个文件（或者说名字相同的文件）在当前工作目录中存在，*svn merge* 就认为是第二种情况，尝试把修改合并到具有相同名字的 本地文件上.

合并语法详解

读者已经见过了 *svn merge* 的几个例子，后面还会 看到几个，如果你对合并的工作原理感到疑惑，不用太过自责，你不是唯一 一个有这种感觉的人. 很多用户（特别是版本控制的新手）一开始都会被命令 的语法和适用它们的场景搞蒙. 其实这个命令比你想像中的要简单很多，有一 个非常简单的办法可以帮助你理解 *svn merge* 如何工作.

困惑主要来自命令的 *名字*. 术语 *合并* (*merge*) 在某种程度上表示分支被组合起 来，或者说有一些神秘的混合数据正在产生. 事实并非如此，命令更恰当的名 字是 *svn diff-and-apply*, 因为新名字恰当地描述了 合并过程中所发生的事情：比较两个仓库目录，然后把差异应用到工作副本上.

如果你是用 *svn merge* 在分支之间复制修改，那么 通常情况下自动合并会工作得很好. 例如下面的命令

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

尝试把分支 *some-branch* 上的修改合并到当前 的工作目录上 (假定当前目录是工作副本或工作副本的一部分, 而且和 *some-branch* 有历史上的联系), 命令只会合并当前 目录还没有的修改. 如果用户在一周后再执行相同的命令, 命令就只会复制在上一次合并后新出现的修改.

如果用户想通过指定被复制的版本号范围, 最大程度地使用 *svn merge*, 则命令接受三个参数:

1. 一个初始的仓库目录 (通常被叫作比较的 *左侧* (*left side*))

2. 一个最终的仓库目录 (通常被叫作比较的 *右侧* (*right side*)

3. 接受差异的工作副本 (通常被叫作合并的 *目标* (*target*))

这三个参数一旦指定, Subversion 就比较两个仓库目录, 将比较产生的差异 作为本地修改应用到目标工作副本. 命令执行结束后, 得到的结果和用户手工编 辑文件或执行各种命令 (例如 *svn add* 和 *svn delete*) 得到的效果是等价的. 如果合并的结果没什么问题, 用户 就可以把它们提交到仓库中, 如果用户不喜欢合并的结果, 只要用 *svn revert* 就可以撤消所有的修改.

svn merge 允许用户灵活地指定这三个参数, 下面是 一些例子:

```
$ svn merge http://svn.example.com/repos/branch1@150 \
            http://svn.example.com/repos/branch2@212 \
            my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
```

第一种语法显式地指定了三个参数; 如果被比较的是同一 URL 的两个不同 版本号, 可以像第二种语法那样简写, 这种类型的合并称为 “二路 URL ”合并 (原因显而易见); 第三种语法说明工作副本参数是可选的, 如果省略工作副本参数, 默认是当前工作目录.

虽然第一个例子展示了 *svn merge* 的 “完整 ” 语法, 但使用时要小心, 它会导致 Subversion 不去更新元数据 `svn:mergeinfo`, 下一节将对此进行更详细的介绍.

没有合并信息的合并

如果可能的话, Subversion 都会去尝试生成合并的元数据, 从而帮助后面 调用的 *svn merge* 更加智能, 但在某些情况下, `svn:mergeinfo` 既不会被创建, 也不会被更新, 对这些情况要稍 微注意一点:

合并不相关的源

> 如果用户要求 Subversion 去比较两个完全不相关的 URL, 那么 Subversion 仍然会生成补丁并应用到工作副本上, 但不会创建或更新 合并元数据. 因为两个源之间没有公共的历史, 而将来的 “智能 ”合并需要这些公共历史.

合并外部仓库

> 虽然执行这样一条命令—`svn merge -r 100:200 http://svn.foreignproject.com/repos/trunk` —是可以的, 但生成的补丁依然缺少 合并元数据. 在撰写本书时, Subversion 还不支持在属性 `svn:mergeinfo` 内表示多个不同仓库的 URL.

使用`--ignore-ancestry`

如果向命令 *svn merge* 传递选项 `--ignore-ancestry`，这将导致 *svn merge* 按照和 *svn diff* 相同的方式生成 不含有历史的差异，更多的内容将在 “关注或忽略祖先”一节 介绍.

反向合并目标的修改历史

在本章的 “撤消修改”一节 我们 介绍了如何使用 *svn merge* 应用一个 “逆补丁 ”，从而回滚已提交的修改. 如果使用这项技术撤消某个对象的已 提交的修改（例如在主干上提交了 r5，之后又马上用 `svn merge . -c -5` 撤消 r5 的修改），这种类型的合并也不会更新 合并信息. [10]

自然历史和隐式合并信息

在讨论 合并信息继承 时我们说过，如果在一个路径上设置了属性 `svn:mergeinfo` ，我们就说这个路径含有 “显式的” 合并信息，与 些相对，一个路径上可以有 “隐式的” 合并信息. 隐式的合并 信息（或 自然历史 (*natural history*)）仅仅是把路径自己的历史（见 “检查历史”一节）解释成合并信息. 虽然隐式的合 并信息在很大程度上只是一个实现上的细节，但在理解合并跟踪上很有帮助.

如果说用户在 r100 创建了 *^/trunk*，在 r201 基于 *^/trunk@200* 创建了 *^/branches/feature-branch*，则 *^/branches/feature-branch* 的自然历史包含了该 分支历史曾经经历过的所有仓库路径与版本号：

/trunk:100-200
/branches/feature-branch:201

随着仓库中新的版本号的出现，分支的自然历史—即隐式的合 并信息—也在不断地包含这些版本号，直到分支被删除的那天. 当 仓库的 HEAD 增长到 234 时，下面的例子展示了 分支的隐式合并信息：

/trunk:100-200
/branches/feature-branch:201-234

隐式的合并信息不会显示在属性 `svn:mergeinfo` 中，但 Subversion 会表现得好像它确实把隐式的合并信息放到了 `svn:mergeinfo` 里，所以说当用户检出 *^/branches/feature-branch*，然后执行 `svn merge ^/trunk -c 58`，却什么也不会发生. 这是因为 Subversion 知道提交到 *^/trunk* 的 r58 已经存在于目标的自然 历史里，也就没必要再合并一次，毕竟避免重复的合并 是 Subversion 合并跟踪特性的主要目标！

关于合并冲突的更多内容

和 *svn update* 类似，*svn merge* 也是向工作副本应用修改，因此难免会产生冲突. 然而与 *svn update* 相比，由 *svn merge* 产生的冲突有 点不同，本节就是介绍这些不同之处.

在开始前假设用户的工作副本不含有本地修改，当用户执行 *svn update*，把工作副本更新到某个特定的版本号时，从服务器接收 的修改总能 “干净地” 应用到工作副本上. 服务器生成差异的 方式是比较两棵目录树：一个是工作副本的虚拟快照，另一个是用户指定的版本 号所对应的目录树. 因为比较的左侧等价于工作副本，所以生成的差异总能保证 正确地把工作副本更新到右侧.

[10] 有趣的是，像这样撤消一个版本号的修改后，我们就不能再用 `svn merge . -c 5` 重新合并 r5，因为合并信息记录的是 r5 已经合并过了. 为了忽略合并信息，必须加上选项 `--ignore-ancestry`

但是 *svn merge* 没有这种保证，而且冲突可能会更 混乱：高级用户可以要求服务器比较 任意 两个目录树，即使目录树和工作副本并不相关！这就意味着有很大的可能产生人为错误．用户有时候会比较两个错误的目录树，导致生成的差异不能被干净地应用到工 作副本上．命令 *svn merge* 会尽可能多地把修改应用到 工作副本，但某些修改可能根本就无法应用成功．合并错误的常见现象是出现了 意想不到的目录冲突：

```
$ svn merge ^/calc/trunk -r104:115
--- Merging r105 through r115 into '.':
   C doc
   C src/button.c
   C src/integer.c
   C src/real.c
   C src/main.c
--- Recording mergeinfo for merge of r105 through r115 into '.':
 U   .
Summary of conflicts:
  Tree conflicts: 5

$ svn st
 M      .
!     C doc
      >   local dir missing, incoming dir edit upon merge
!     C src/button.c
      >   local file missing, incoming file edit upon merge
!     C src/integer.c
      >   local file missing, incoming file edit upon merge
!     C src/main.c
      >   local file missing, incoming file edit upon merge
!     C src/real.c
      >   local file missing, incoming file edit upon merge
Summary of conflicts:
  Tree conflicts: 5
```

在上面的例子里，从现象来看，被比较的目录 *doc* 和 4 个 **.c* 文件在分支的两个快照中都存在，生成的 差异想去修改工作副本中对应路径上的文件内容．但这些路径在工作副本中都 不存在．无论真实的情况是什么，产生目录冲突最可能的原因是用户比较了两 个错误的目录树，或者是差异被应用到了错误的工作副本—这两种都是 用户最常犯的错误．当错误发生时，最简单的办法就是递归地撤消由合并产生 的所有本地修改 (`svn revert . --recursive`)，删除可能残留的未被版本控制的文件和目录，然后再用正确的参数执行 *svn merge*.

还要注意，即使在向不含有本地修改的工作副本合并，仍有可能产生 内容冲突．

```
$ svn st

$ svn merge ^/paint/trunk -r289:291
--- Merging r290 through r291 into '.':
C    Makefile
--- Recording mergeinfo for merge of r290 through r291 into '.':
 U   .
Summary of conflicts:
  Text conflicts: 1
```

```
Conflict discovered in file 'Makefile'.
Select: (p) postpone, (df) diff-full, (e) edit, (m) merge,
        (mc) mine-conflict, (tc) theirs-conflict, (s) show all options: p

$ svn st
 M      .
C       Makefile
?       Makefile.merge-left.r289
?       Makefile.merge-right.r291
?       Makefile.working
Summary of conflicts:
  Text conflicts: 1
```

为什么会发生这种冲突呢? 因为用户可以要求 *svn merge* 定义并应用任意一个老差异到工作副本中，而这个差异所包含的修改可能不能被干净地应用到文件中，即使这个文件不含有本地修改.

svn update 和 *svn merge* 的另 一个不同点是当冲突发生时，新创建的文件的名字. 在 “解决冲突”一节 我们已经看到更新操作可能会 创建形如 *filename.mine*, *filename.rOLDREV* 和 *filename.rNEWREV* . 的新文件. 当 *svn merge* 发生冲突时，它会 创建 3 个形如 *filename.working*, *filename.merge-left.rOLDREV* 和 *filename.merge-right.rNEWREV* 的新文件. 模式中的 “merge-left” 和 “merge-right” 分别指出了文件 来自比较的左侧和右侧，“rOLDREV” 描述了左侧的版本号，而 “rNEWREV” 描述了右侧的版本号. 无论是 *svn update* , 还是 *svn merge*, 这些文件名都可以帮助 用户分辨冲突的来源.

拦截修改

有时候，用户可能不想让某个特定的变更集被自动合并，比如说你所在的 团队的开发策略是在 */trunk* 完成新的开发工作，但 是，在向稳定分支回植修改时非常保守，因为稳定分支是面向发布的分支. 在比较极端的情况下，你可以手动地从主干精选修改—只精选那些足够 稳定的修改—再合并到分支上. 不过实际做起来可能没这么严格，大多数时 候你只想让 *svn merge* 把主干的大多数修改自动合并 到分支上，这时候就需要一种方法能够屏蔽掉一些特定的变更集，阻止它们 被自动合并.

为了拦截一个变更集，必须让 Subversion 认为变更集 已经 被合并了. 为了实现这点，在执行 *svn merge* 时添加选项 `--record-only`, 该选项使得 Subversion 更新 合并信息，就好像它真得执行了合并，但实际上文件内容并没有被修改.

```
$ cd my-calc-branch

$ svn merge ^/calc/trunk -r386:388 --record-only
--- Recording mergeinfo for merge of r387 through r388 into '.':
 U   .

# Only the mergeinfo is changed
$ svn st
 M      .

$ svn pg svn:mergeinfo -vR
Properties on '.':
  svn:mergeinfo
    /calc/trunk:341-378,387-388
```

```
$ svn commit -m "Block r387-388 from being merged to my-calc-branch."
Sending        .

Committed revision 461.
```

从 Subversion 1.7 开始，带有选项 `--record-only` 的合并是传递的，这就意味着除了在被合并的目标上记录被拦截的合并信息外，源的 `svn:mergeinfo` 属性上的任意修改都会被应用到目 标的 `svn:mergeinfo` 属性上. 举例来说，我们想要拦截 *^/paint/trunk* 上与特性 'paint-python-wrapper' 有关的修改被合并到分支 *^/paint/branches/paint-1.0.x* 上. 我们已经知道特性 'paint-python-wrapper' 已经在自己的分支上开发完成，并且在 r465 合并到了 */paint/trunk* 上：

```
$ svn log -v -r465 ^/paint/trunk
------------------------------------------------------------------------
r465 | joe | 2013-02-25 14:05:12 -0500 (Mon, 25 Feb 2013) | 1 line
Changed paths:
   M /paint/trunk
   A /paint/trunk/python (from /paint/branches/paint-python-wrapper/python:464)

Reintegrate Paint Python wrapper.
------------------------------------------------------------------------
```

因为 r465 是一个再整合合并，所以我们知道描述合并的信息被 记录了下来：

```
$ svn diff ^/paint/trunk --depth empty -c465
Index: .
===================================================================
--- .   (revision 464)
+++ .   (revision 465)

Property changes on: .
___________________________________________________________________
Added: svn:mergeinfo
   Merged /paint/branches/paint-python-wrapper:r463-464
```

如果只是简单地拦截 */paint/trunk* 的 r465 并 不能确保万无一失，因为其他人可能会直接从 */paint/branches/paint-python-wrapper* 合并 r462:464, 幸运的是选项 `--record-only` 的传递性质 可以防止这种情况发生. 选项 `--record-only` 把 r465 生成的 `svn:mergeinfo` 差异应用到工作副本上，从而 拦截住来自 */paint/trunk*直接合并和 */paint/branches/paint-python-wrapper* 的间接合并.

```
$ cd paint/branches/paint-1.0.x

$ svn merge ^/paint/trunk --record-only -c465
--- Merging r465 into '.':
 U   .
--- Recording mergeinfo for merge of r465 into '.':
 G   .

$ svn diff --depth empty
Index: .
===================================================================
--- .   (revision 462)
```

```
+++ .   (working copy)

Property changes on: .
___________________________________________________________________
Added: svn:mergeinfo
   Merged /paint/branches/paint-python-wrapper:r463-464
   Merged /paint/trunk:r465

$ svn ci -m "Block the Python wrappers from the first release of paint."
Sending        .

Committed revision 466.
```

现在，无论怎么尝试从 */paint/trunk* 合并特性都 不会产生任何实际的效果.

```
$ svn merge ^/paint/trunk -c465
--- Recording mergeinfo for merge of r465 into '.':
 U   .

$ svn st # No change!

$ svn merge ^/paint/branches/paint-python-wrapper -r462:464
--- Recording mergeinfo for merge of r463 through r464 into '.':
 U   .

$ svn st  # No change!

$
```

如果以后用户意识到自己实际上 需要 被拦截的 修改，那么有两种选择. 一种是用户可以撤消 r466, 撤消的方法见 “撤消修改”一节, 把撤消 r466 的修 改提交后，用户就可以再次从 */paint/trunk* 合并 r465. 另一种是在合并 r465 时带上选项 `--ignore-ancestry`, 这将导致命令 *svn merge* 忽略合并信息，直接应用所 请求的差异，见 “关注或忽略祖先”一节.

```
$ svn merge ^/paint/trunk -c465 --ignore-ancestry
--- Merging r465 into '.':
A    python
A    python/paint.py
 G   .
```

使用 `--record-only` 有一点危险，因为我们经常无 法分辨什么时候是 “我已经包含了这个修改”，什么时候是 “我没有这个修改，但目前还不想要它”. 使用 `--record-only` 实际上是在向 Subversion 撒谎，让它以为修改 已经被合并了. 记住修改是否被真正地合并是用户的责任，Subversion 无法 回答 “有哪些修改被拦截了” 这样的问题. 如果用户想跟踪被拦截的修改 (以后可能会放开对它们的拦截)，就要自己找地方记录，例如 记在某个文本文件上，或自定义的属性里.

对合并敏感的日志与注释

任意一个版本控制系统都需要支持的一项特性是能够查看是谁，在什么时 候，修改了什么地方，Subversion 完成这些功能的命令是 *svn log* 和 *svn blame*. 在单独的文件上执行 这两个命令时，它们不仅会显示影响文件的变更集历史，还可以精确地指出 每一行是哪个用户在什么时候修改的.

然而，当修改在分支间复制时，事情开始变得复杂起来. 比如说用 *svn log* 查询特性分支的历史，命令将会显示所有影响 过分支的版本号：

```
$ cd my-calc-branch

$ svn log -q
------------------------------------------------------------------------
r461 | user | 2013-02-25 05:57:48 -0500 (Mon, 25 Feb 2013)
------------------------------------------------------------------------
r379 | user | 2013-02-18 10:56:35 -0500 (Mon, 18 Feb 2013)
------------------------------------------------------------------------
r378 | user | 2013-02-18 09:48:28 -0500 (Mon, 18 Feb 2013)
------------------------------------------------------------------------
…
------------------------------------------------------------------------
r8 | sally | 2013-01-17 16:55:36 -0500 (Thu, 17 Jan 2013)
------------------------------------------------------------------------
r7 | bill | 2013-01-17 16:49:36 -0500 (Thu, 17 Jan 2013)
------------------------------------------------------------------------
r3 | bill | 2013-01-17 09:07:04 -0500 (Thu, 17 Jan 2013)
------------------------------------------------------------------------
```

但是这些日志完整地刻画了分支上的所有修改吗? 输出中没有明确指出的是 r352, r362, r372 和 r379 其实是从主干合并修改的结果. 如果你详细地查看 这几个日志将会发现我们没办法看到构成分支修改的多个主干变更集：

```
$ svn log ^/calc/branches/my-calc-branch -r352 -v
------------------------------------------------------------------------
r352 | user | 2013-02-16 09:35:18 -0500 (Sat, 16 Feb 2013) | 1 line
Changed paths:
   M /calc/branches/my-calc-branch
   M /calc/branches/my-calc-branch/Makefile
   M /calc/branches/my-calc-branch/doc/INSTALL
   M /calc/branches/my-calc-branch/src/button.c
   M /calc/branches/my-calc-branch/src/real.c

Sync latest trunk changes to my-calc-branch.
------------------------------------------------------------------------
```

我们知道被合并的修改来自主干，那么如何同时查看主干上的这些修改 历史? 答案是使用选项 `--use-merge-history` (`-g`), 展开被合并的 “子” 修改.

```
$ svn log ^/calc/branches/my-calc-branch -r352 -v -g
------------------------------------------------------------------------
r352 | user | 2013-02-16 09:35:18 -0500 (Sat, 16 Feb 2013) | 1 line
Changed paths:
   M /calc/branches/my-calc-branch
   M /calc/branches/my-calc-branch/Makefile
   M /calc/branches/my-calc-branch/doc/INSTALL
   M /calc/branches/my-calc-branch/src/button.c
   M /calc/branches/my-calc-branch/src/real.c
```

```
Sync latest trunk changes to my-calc-branch.
------------------------------------------------------------------------
r351 | sally | 2013-02-16 08:04:22 -0500 (Sat, 16 Feb 2013) | 2 lines
Changed paths:
   M /calc/trunk/src/real.c
Merged via: r352

Trunk work on calc project.
------------------------------------------------------------------------
…
------------------------------------------------------------------------
r345 | sally | 2013-02-15 16:51:17 -0500 (Fri, 15 Feb 2013) | 2 lines
Changed paths:
   M /calc/trunk/Makefile
   M /calc/trunk/src/integer.c
Merged via: r352

Trunk work on calc project.
------------------------------------------------------------------------
r344 | sally | 2013-02-15 16:44:44 -0500 (Fri, 15 Feb 2013) | 1 line
Changed paths:
   M /calc/trunk/src/integer.c
Merged via: r352

Refactor the bazzle functions.
------------------------------------------------------------------------
```

为 *svn log* 增加选项 `--use-merge-history (-g)`, 我们不仅可以看到 r352, 还可以看到通过 r352 从主干合并到分支的提交, 这些提交是 Sally 在主干上的工作. 这才是历 史的更完整的刻画!

命令 *svn blame* 也支持选项 `--use-merge-history (-g)`, 如果在执行命令时 没有带上该选项, 在查看 *src/button.c* 每一行的修改注释 时, 用户可能会对修改的负责人产生错误的印象:

```
$ svn blame src/button.c
…
   352    user    retval = inverse_func(button, path);
   352    user    return retval;
   352    user    }
…
```

用户 `user` 的确在 r352 提交了这 3 行修改, 但其中 2 行实际上来自 Sally 在 r348 的修改, 它们通过同步合并被合并到了分支中:

```
$ svn blame button.c -g
…
G    348    sally   retval = inverse_func(button, path);
G    348    sally   return retval;
     352    user    }
```

…

现在我们知道了是谁应该 真正地 为这 2 行修改 负责!

关注或忽略祖先

如果与 Subversion 开发人员交谈，你可能会经常听到一个术语：祖先 (*ancestry*). 这个术语描述了 仓库中两个对象间的一种关系：如果它们之间是相关的，那么其中一个对象就 是另一个对象的祖先.

比如说你在 r100 提交了文件 *foo.c* 的修改，那么 *foo.c@99* 就是 *foo.c@100* 的一个 “祖先”. 另一方面，如果你在 r101 删除了文件 *foo.c*, 然后在 r102 又提交了一个具有相同名字的文件，虽然从名字上看，*foo.c@99* 和 *foo.c@102* 是相关的，但实际上它们是两个完全不相关的对象，只是碰巧名 字相同罢了，它们之间不共享历史或 “祖先”.

介绍 “祖先” 是为了说明 *svn diff* 和 *svn merge* 之间的一个重要区别. *svn diff* 会忽略祖先，而 *svn merge* 对祖先非常敏感. 举例来说，如果用户要求 *svn diff* 去比较 *foo.c* 在 r99 和 r102 时的版本，命令将会盲目地比较这两个 版本，并输出以行为单位的差异. 但是如果用户要求 *svn merge* 去比较相同的两个对象，它将会注意到两个对象之间是不相关的，于是先删除旧文件，再添加新文件，从命令的输出信息可以看得很清楚：

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

大多数合并操作都会涉及比较两个在历史上相关的目录树，因此 *svn merge* 就把这种情况当成默认条件. 然而，在少数情况下用户 可能希望 *svn merge* 去比较两个不相关的目录树. 比如说 用户导入了两份源代码，分别表示软件的两个不同的供应商发布版（见 “供方分支” 一节), 如果用户要求 *svn merge* 去比较这两个目录树，将会看到第一个目录树被整 体删除，然后再整体添加第二个目录树. 对于这种情况，用户其实是希望 *svn merge* 只做基于路径的比较，完全忽略文件和目录 之间的任何关系. 添加选项 `--ignore-ancestry` 后，*svn merge* 的行为将变得和 *svn diff* 一样.（反之，添加 *--notice-ancestry* 后，命令 *svn diff* 的行为将变得和 *svn merge* 一样）

属性 `--ignore-ancestry` 还会禁止 合并跟踪, 这就 意味着当 *svn merge* 确定应该合并哪些版本号时，它就不会考虑，也不会更新 `svn:mergeinfo`.

合并与移动

开发人员的一个常见需求是对代码进行重构，特别是基于 Java 的软件 项目. 文件和目录被移来移去，经常会给项目的开发人员造成困扰. 听起来是 不是觉得这种场景很适合使用分支? 创建一个分支，尽管在分支里随意折腾，最后再把分支合并到主干上就行了，对吗?

可惜，现实情况还没有这么理想，这是 Subversion 目前还有待完善的地方. 其中的问题是 Subversion 的命令 *svn merge* 并没有人们 期望中的那样健壮，尤其是在处理复制和移动操作时.

使用 *svn copy* 复制一个文件时，仓库记住了新文件的 来源，但这项信息并不会传递给正在执行 *svn update* 或 *svn merge* 的客户端. 仓库不会告诉客户端 “把 工作副本中已有的这个文件复制到另一个位置”，相反，它会向客户端下发 一个全新的文件. 这可能会导致问题，尤其是和重命名有关的目录冲突. 重命名不仅涉及到 一个新的副本，还涉及到删除一个旧路径—一个不为人知的事实是 Subversion 没有 “真正的重命名”—*svn move* 只不过是 *svn copy* 和 *svn delete* 的组合而已.

比如说用户想对自己的私有分支 */calc/branch/my-calc-branch* 做一些修改，首先用户和 */calc/trunk* 做了一个自动同步合并，并在 r470 提交了合并：

```
$ cd calc/trunk

$ svn merge ^/calc/trunk
--- Merging differences between repository URLs into '.':
U    doc/INSTALL
A    FAQ
U    src/main.c
U    src/button.c
U    src/integer.c
U    Makefile
U    README
 U   .
--- Recording mergeinfo for merge between repository URLs into '.':
 U   .

$ svn ci -m "Sync all changes from ^/calc/trunk through r469."
Sending        .
Sending        Makefile
Sending        README
Sending        FAQ
Sending        doc/INSTALL
Sending        src/main.c
Sending        src/button.c
Sending        src/integer.c
Transmitting file data ....
Committed revision 470.
```

然后用户在 r471 把 *integer.c* 重命名为 *whole.c*，又在 r473 修改了 *whole.c* . 从效果上来看等价于创建了一个新文件（原文件的副本再加上 一些修改），再删除原文件. 同时在 */calc/trunk*, Sally 在 r472 提交了 *integer.c* 的修改：

```
$ svn log -v -r472 ^/calc/trunk
------------------------------------------------------------------------
r472 | sally | 2013-02-26 07:05:18 -0500 (Tue, 26 Feb 2013) | 1 line
Changed paths:
   M /calc/trunk/src/integer.c

Trunk work on integer.c.
------------------------------------------------------------------------
```

现在用户打算把自己的分支上的工作合并到主干上，你觉得 Subversion 会如何组合你和 Sally 的修改?

```
$ svn merge ^/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
   C src/integer.c
 U   src/real.c
A    src/whole.c
--- Recording mergeinfo for merge between repository URLs into '.':
```

```
 U   .
Summary of conflicts:
  Tree conflicts: 1

$ svn st
 M      .
      C src/integer.c
      >   local file edit, incoming file delete upon merge
 M      src/real.c
A  +    src/whole.c
Summary of conflicts:
  Tree conflicts: 1
```

实际情况是 Subversion 不会 把这些修改组合起来，而是产生一个目录冲突[11] 因为 Subversion 需要用户帮它算出你和 Sally 的哪些修改应该留在 *whole.c* 上，或者是重命 名操作是否应该保留.

用户解决完目录冲突后才能提交，这可能需要人工介入，见 “处理目录冲突”一节. 我们举这个例子的目的是提醒 用户，在 Subversion 改良之前，要小心对待从一个分支合并复制和重命名操 作到另一个分支，如果确实这样做了，要做好解决目录冲突的准备.

禁止不支持合并跟踪的客户端

如果用户只是把服务器端升级到 Subversion 1.5 及以后的版本，那么 1.5 版之前的客户端在 合并跟踪 方面会产生 问题，这是因为 1.5 版之前的客户端不支持这项特性. 当旧版客户端执行 *svn merge* 时，命令不会去更新属性 `svn:mergeinfo`，因此，随后的提交虽然是合并的结果，但 关于被复制的修改的信息不会告诉给服务器—这些信息就此丢失. 以后，如果新版客户端执行自动合并，很可能会因为合并重复的修改而产生大量冲突.

如果你和你的团队非常依赖 Subversion 的合并跟踪特性，你可能需要对 仓库进行配置，使得仓库禁止旧客户端提交修改. 最简单的配置方法是在 start-commit 钩子脚本里检查参数 “capabilities ”，如果客户端反映它支持 `mergeinfo` 功能，钩子脚本就允许客户端提交，否则的话就禁止该客户端提交修改，例 4.1 “合并跟踪的看门狗—钩子脚本 start-commit” 给出了 钩子脚本 `start-commit` 的一个示例：

例 4.1. 合并跟踪的看门狗—钩子脚本 start-commit

```
#!/usr/bin/env python
import sys
```

[11] 如果 Sally 没有提交 r472 的修改，那么 Subversion 将会注意到目标工作副本的 *integer.c* 和合并左端的 *whole.c* 其实是同一 个文件，此时合并将会成功，不会有冲突产生：

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

```
# The start-commit hook is invoked immediately after a Subversion txn is
# created and populated with initial revprops in the process of doing a
# commit. Subversion runs this hook by invoking a program (script,
# executable, binary, etc.) named 'start-commit' (for which this file
# is a template) with the following ordered arguments:
#
#   [1] REPOS-PATH   (the path to this repository)
#   [2] USER         (the authenticated user attempting to commit)
#   [3] CAPABILITIES (a colon-separated list of capabilities reported
#                     by the client; see note below)
#   [4] TXN-NAME     (the name of the commit txn just created)

capabilities = sys.argv[3].split(':')
if "mergeinfo" not in capabilities:
  sys.stderr.write("Commits from merge-tracking-unaware clients are "
                   "not permitted.  Please upgrade to Subversion 1.5 "
                   "or newer.\n")
  sys.exit(1)
sys.exit(0)
```

关于钩子脚本的更多信息，见“实现仓库钩子”一节.

关于合并跟踪的最后一点内容

最后要说的是 Subversion 的合并跟踪特性有一个复杂的内部实现，而 属性 `svn:mergeinfo` 是用户了解合并跟踪内部机制的 唯一窗口.

记录合并信息的时机和方式有时候会很难理解，另外，合并信息元数据的 管理也分成了很多种类型，例如“显式”与“隐式”的合并信息，“可实施”与“不可实施”的版本号，“省略”合并信息的特定机制，以及从父目录到子目录的“继承”.

我们决定只对这些主题进行简单的介绍，原因有以下几点. 首先对于一个普 通用户来说，细节过于复杂；第二，普通用户不需要 完全理解这些概念，实现上的细节对他们而言是透明的. 如果读者有兴趣，可以 阅读 CollabNet 的一篇文章：http://www.open.collab.net/community/subversion/articles/merge-info.html.

如果读者只想尽量避开合并跟踪的复杂性，我们有以下建议：

- 如果是短期的特性分支，遵循“基本合并”一节 描述的步骤.
- 避免子目录合并与子目录合并信息，只在分支的根目录执行合并，而不是在分支的子目录或文件上执行合并（见“子目录合并与子目录合并信息”一节).
- 不要直接修改属性 `svn:mergeinfo`，而是用 带有选项 `--record-only` 的命令 *svn merge* 向属性施加期望的修改，见“拦截修改”一节).
- 被合并的目标应该是一个工作副本，代表了一个 完整的 目录的根，这个目录则代表了某一时刻，仓库的一个单一 位置：

- 在合并前更新！不要使用选项 `--allow-mixed-revisions` 去合并含有混合版本号的工作副本.
- 不要合并带有 “已切换的” 子目录的目标（在 “遍历分支” 一节 介绍）.
- 避免合并含有稀疏目录的目标，类似地，也不要合并深度不是 `--depth=infinity` 的目标.
- 确保你对合并的源具有读权限，对被合并的目标具有读写权限.

当然，有时候你并不能完全按照上面所说的要求去做，此时也不用担心，只 要你知道这样做的后果就行.

遍历分支

命令 *svn switch* 切换一个已有的工作副本，使其映射 到另一个不同的分支. 虽然在使用分支时，该命令并不是必须的，但它提供了很 方便的快捷键. 在一个我们讲过的例子里，当用户创建完私有分支后，检出了该 分支的工作副本. 现在用户多了一种选择，用命令 *svn switch* 把 */calc/trunk* 的工作副本映射到新创建的分支：

```
$ cd calc
$ svn info | grep URL
URL: http://svn.example.com/repos/calc/trunk
Relative URL: ^/calc/trunk
$ svn switch ^/calc/branches/my-calc-branch
U    integer.c
U    button.c
U    Makefile
Updated to revision 341.
$ svn info | grep URL
URL: http://svn.example.com/repos/calc/branches/my-calc-branch
Relative URL: ^/calc/branches/my-calc-branch
$
```

“切换” 一个不含有本地修改的工作副本到另一个分支， 最终 得到的工作副本就像是从分支上检出的一样. 使用 *svn switch* 切换分支通常会更有效率，因为分支之间的差异通常很小，服务器只需要发送一 小部分数据，就可以让工作副本映射到一个新的分支.

svn switch 支持选项 `--revision (-r)`，因此用户还可以把工作副本切换到分支的其他版本，并 非只能是 `HEAD`.

当然，绝大多数项目都要比例子里的 *calc* 复杂得多，而且包含非常多的子目录，Subversion 用户在使用分支时经常遵循一些固定的 步骤：

1. 把项目的整个 “主干” 复制到一个新的分支目录.
2. 只把主干工作副本的 “一部分” 进行切换，以映射到另一 个分支.

换句话说，如果用户知道分支的工作只需要在某个特定的子目录内完成，他 就可以用 *svn switch*，只把这个子目录切换到分支上（用户甚至可以只切换一个文件！）. 通过这种方式，用户可以继续接收正常的 “主干” 更新到大部分的工作副本，但不会更新已切换的部分（除 非有人向分支提交了修改）. 这个特性给 “混合的工作副本” 添加了新的一个维度—工作副本不仅可以包含混合的版本号，甚至可以包含 混合的仓库位置.

典型情况下，已切换的子目录和 “被切换走的” 目录共享相同 的祖先，但 *svn switch* 也可以把子目录切换到一个不与 原目录共享祖先的仓库位置，方法是加上选项 `--ignore-ancestry` .

即使工作副本内包含了大量的已切换的子目录，这些子目录来自仓库中不同 位置，那么工作副本仍然可以正常工作. 更新工作副本时，各个子目录也会收到 正确的修改；提交时，本地修改仍然作为一个单一的原子修改，被提交到仓库中.

注意，虽然 Subversion 允许工作副本映射不同的仓库位置，但这些位置必须 在 同一个 仓库中. Subversion 还不支持跨仓库的交互，但以后可能会添加这一特性. [12]

如果管理员需要修改仓库的 URL，而仓库是通过 HTTP 进行访问，那么强烈 建议管理员在配置文件 *httpd.conf* 添加一个永久的 重定向（通过配置项 `RedirectPermanent`），将旧的 URL 重定向至新的 URL，这样的话，当用户仍然使用旧的 URL 访问仓库时，Subversion 客户端就会在错误信息中显示仓库的新 URL. 从 Subversion 1.7 开始，客户端可以自动地把工作副本重定向至新的 URL.

切换与更新

不知道读者有没有注意到 *svn switch* 和 *svn update* 的输出看起来好像是一样的? *svn switch* 实际上是 *svn update* 的超集.

执行 *svn update* 是在要求仓库比较两个目录树，仓库比较后，向客户端发送差异. *svn switch* 和 *svn update* 的唯一差别是后者永远是在比较同一个 仓库路径.

也就是说如果有一个 */calc/trunk* 的工作副本，*svn update* 总是自动地比较 */calc/trunk* 的工作副本与 HEAD 的 */calc/trunk*. 如果你把工作副本切换到另一个分支，*svn switch* 就把 */calc/trunk* 的工作副本与 另一个 处于版本号 HEAD 的分支目录进行比较.

换句话说，工作副本用 *svn update* 在时间上移动，用 *svn switch* 在时间 和 空间 上移动.

因为 *svn switch* 本质上是 *svn update* 的一个变种，所以它有着和 *svn update* 相同的行为：从服务器接收更新时，工作副本 的本地修改将被保留.

读者是否遇到过这种情形：在 */trunk* 的工作副本 中做了一些复杂的修改，结果却发现这些修改应该有一个自己的分支，这时候 用户可以这样做：

```
$ 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 ^/calc/branches/newbranch
At revision 353.
```

和 *svn update* 一样，*svn switch* 保留了本地修改. 现在你的工作副本已经映射到了新创建的分支，如果后面执行了 *svn commit*，修改将会被提交上新创建的分支中.

[12] 然而，如果服务器的 URL 发生了 改变，而用户不想丢掉已有的工作副本，那就可以用 *svn relocate* ，更多的信息和例子，见 svn 参考手册—Subversion 命令行客户端 的 svn relocate

标签

另一个常见的版本控制概念是标签. 标签是项目的一个 “快照”，它在 Subversion 中到处都是，因为每一个版本号都对应着提交后，文件系统的 一个快照.

然而，用户经常想给标签取一个更人性化的名字，例如 `release-1.0` ，而且用户只想对文件系统的某个子目录做快照，毕竟人们更容易记住 项目的发布版 1.0 是版本号为 4822 的特定子目录.

创建简单的标签

创建标签的命令是 *svn copy*. 如果用户想为 处于 HEAD 的 */calc/trunk* 创建一个快照，就执行：

```
$ 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.
```

这个例子假设目录 */calc/tags* 已经存在（如果不 存在，就先用 *svn mkdir* 创建它）. 复制完成后，新目录 *release-1.0* 就成为了 */calc/trunk* 在复制那一刻的永久快照. 当然，用户也 可以指定被复制的版本号，避免其他在用户没有觉察的时候，向仓库提交了新 的修改. 如果说用户已经知道版本号为 901 的 */calc/trunk* 正是自己想要的快照，就给命令 *svn copy* 添加选项 `-r 901`.

请等一下，这和创建分支的步骤不是一样的吗? 事实上的确如此. 对于 Subversion 而言，标签和分支没有区别，它们都是通过命令 *svn copy* 创建的目录. 之所以把复制出的目录叫作 “标签”的唯一原因是 *用户* 已经决定把该目录看成 标签—只要没有人往标签提交修改，它就永远保持创建时的样子. 如果用户 在创建标签后，往标签提交了新的修改，那么从用户的角度来看，它就变成了 一个分支.

如果读者是仓库的管理员，那么你有 2 种标签管理方法. 第 1 种是 “放任不管”：作为一种项目管理策略，一开始便规定好标签 的存放位置，确保所有用户都知道如何对待他们复制的目录（也就是确保他们不 会向标签提交修改）. 第 2 种更具有强制性：使用和 Subversion 配合的 访问控制脚本，禁止任何人在标签区提交修改，除了创建新的标签（见 第 6 章 服务器配置). 第 2 种方法通常是没有必要的，因 为如果用户不小心向标签提交了修改，总是可以用之前介绍的方法，撤消已经 提交的修改.

创建复杂的标签

有时候，用户可能需要更复杂的 “快照”，它不仅仅是单独 版本号下的单个目录.

举个例子，假设你的项目比我们的 *calc* 庞大得 多：项目内包含了大量的文件与目录. 在工作过程中，你可能需要创建一个含有指定特性和问题修正的工作副本，创建的方式可以是选择性地把文件或 目录退回到指定的版本（使用带有选项 `-r` 的 *svn update* 命令），把文件和目录切换到特定的分支（通过命令 *svn switch*），甚至是一连串的本地修改. 创建完毕后，你的工作副本就变成了一个大杂烩，但是在测试后，你确定这 正是你想要创建标签的目标.

是时候创建快照了，但复制 URL 在这里不起作用. 对于这种情况，用户 想要的是在仓库中，为当前状态下的工作副本创建一个快照. 幸运的是 *svn copy* 的 4 种用法中（见 svn 参考手册—Subversion 命令行客户端 的 svn copy (cp)），包含了把工作副本复制到仓库中的能力：

```
$ 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.
```

现在，仓库中就出现了一个新目录 */calc/tags/mytag* ，它是当前工作副本的快照—混合的版本号，URL，本地修改 等.

有些用户已经发现了这个特性的其他一些用法. 有时候用户的工作副本 可能包含了一堆本地修改，他想让其他用户审查一下，但这次不是用 *svn diff* 生成并发送补丁 (*svn diff* 无法体现目录或符号链接的变化)，而是用 *svn copy* 把 当前状态下的工作副本 “上传” 到仓库中的适当位置，例如你的 私有目录，然后其他用户就可以用 *svn checkout* 逐字 拷贝你的工作副本，或者用 *svn merge* 接收你做出的修改.

虽然这是上传工作副本快照的好办法，但要注意的是这 不是 一个创建分支的好办法，创建分支应该是它本身的事件，而这种 方法创建的分支混合了额外的修改，分支的创建和修改都在一个单独的版本号 里，这样的话我们以后就难确定哪一个版本号才是分支点.

分支维护

读者可能已经注意到 Subversion 非常灵活. 因为 Subversion 实现分支和 标签的底层机制是相同的 (目录复制)，而且分支和标签都是以普通的目录出现 在文件系统中，很多人觉得自己被 Subversion 吓到了: 这简直就是 过于 灵活了. 本节将介绍一些与管理相关的建议.

仓库布局

有一些标准的方式用于组织仓库内容. 大多数用户用目录 *trunk* 存放开发 “主线”，用目录 *branches* 存放分支，用目录 *tags* 存放标签. 如果 一个仓库只存放一个项目，人们通常会创建这些顶层目录:

```
/
  trunk/
  branches/
  tags/
```

如果在一个仓库中包含了多个项目，通常根据项目索引它们的布局，关 于 “项目根目录” 的更多内容，见 “规划仓库的组织方式”一节，下面就是一个典型的，包含了多个项目的仓库布局:

```
/
  paint/
    trunk/
    branches/
    tags/
  calc/
    trunk/
    branches/
```

tags/

当然，用户也可以完全忽略这些常见的布局，按照实际需要定义仓库的 布局．记住，无论你怎么选择，仓库布局并非一成不变，用户可以在任何时候 重新组织仓库的布局．因为分支和标签都是普通目录，所以用户可以随心所欲地 用命令 *svn move* 移动或重命名它们．从一种布局切换到 另一种布局只是服务器端的一系列移动而已，如果你不喜欢当前的仓库布局，可以任意修改目录结构．

记住，虽然移动目录很容易操作，但你仍然需要考虑其他用户．把目录调 来调去可能会让其他用户感到迷惑，如果有个用户的工作副本所对应的仓库目录 被其他用户用 *svn move* 移动其他地方去了，那么用户下 次执行 *svn update* 时，就会被告知工作副本对对应的仓库 目录已经不存在了，他必须用 *svn switch* 把工作副本切换 到仓库中的另一个位置．

数据的寿命

Subversion 模型具有的另一个优良特性是分支和标签的寿命是有限的，就像一个普通的被版本控制的条目．比如说用户最终在自己的分支中完成了工 作，当分支上所有的修改都被合并到 */calc/trunk* 后，就没必要再在仓库中保留自己的分支了．

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-branch \
             -m "Removing obsolete branch of calc project."

Committed revision 474.
```

上一小节刚刚讲过，如果工作副本所对应的仓库路径已经被删除了，此时 再更新工作副本将会收到错误：

```
$ svn up
Updating '.':
svn: E160005: Target path '/calc/branches/my-calc-branch' does not exist
```

对于这种情况，你所需要做的就是把工作副本切换到仓库中仍然存在的 其他路径：

```
$ svn sw ^/calc/trunk
D    src/whole.c
 U   src/real.c
A    src/integer.c
 U   .
Updated to revision 474.
```

现在你的分支就消失了，当然并非永远地消失：分支只是在 HEAD 上看不到了．如果用 *svn checkout*, *svn switch* 或 *svn list* 查看较 早的版本号，仍然可以看到你的旧分支．

如果浏览已删除的目录还不够，你还可能把它们再恢复回来．在 Subversion 中恢复数据非常容易，如果用户想把一个已删除的目录或文件恢复到 HEAD 中，只需要用 *svn copy* 把它从旧 版中复制出来即可：

```
$ svn copy ^/calc/branches/my-calc-branch@473 \
           ^/calc/branches/my-calc-branch \
           -m "Restore my-calc-branch."

Committed revision 475.
```

在我们的例子里，分支的寿命相对较短：创建分支的目的可能是为了 修正一个问题，或实现一个新的特性．当任务完成后，分支也就走到了生命的 尽头．在软件开发过程中，长期同时存在两条 “主要的” 分支并 不少见，比如说现在要发布 *calc*

的稳定版本，而开发 人员知道要花几个月的时间才能把潜在的问题修复殆尽，也不想向稳定版添加 新特性，所以你决定创建一个 “稳定” 分支，表示不想做过多的 修改：

```
$ svn copy ^/calc/trunk ^/calc/branches/stable-1.0 \
           -m "Creating stable branch of calc project."

Committed revision 476.
```

现在开发人员可以自由地向 */calc/trunk* 添加最 前沿的（或实验性的）新特性，同时达成一个约定：只有修复问题的修改才能 提交到 */calc/branches/stable-1.0*. 也就是说，当人 们在主干上工作的同时，精选修复问题的修改提交到稳定分支上. 即使在稳定 分支发布后，开发人员很可能也会维护很长一段时间—只要你还在为客户 支持这一发布版. 我们将会在下节介绍更多的相关内容.

常见的分支模式

分支和 *svn merge* 有着非常丰富的用法，本节介绍 其中最常见的几种.

版本控制最经常用在软件开发领域，所以本节先介绍 2 种在开发人员中最 常见的分支/合并模式. 如果读者使用 Subversion 不是为了软件开发，尽管跳过 本节，如果你是第一次使用版本控制工具的软件开发人员，请集中注意力，因为 这些模式经常被经验丰富的程序员看成是最佳的做法. 本节的内容不仅限于 Subversion，它们同样适用于其他版本控制系统，使用 Subversion 的术语进行 描述更有助于用户理解.

发布分支

大多数软件都有一个典型的生命周期：编码，测试，发布，如此循环往复. 这个过程有两个问题. 首先开发人员需要不断地往软件添加新特性，同时质保 团队也会不断地测试稳定版，开发与测试是同时进行的. 第二，开发团队通常 需要支持已发布的旧版，如果在最新版发现了一个问题，那么旧的发布版很 可能也有同样的问题，客户更希望直接修复这个问题，而不是等待新的版本发 布.

版本控制工具可以帮助开发人员解决这些问题，典型的步骤是：

1. 开发人员把所有的新工作都提交到主干上. 把 每天的修改—包含新特性，问题修复等—都提交到 */trunk*.
2. 复制主干到 “发布” 分支. 如果团队认为软件已经准备好发布（例如发布版 1.0），可能会复制 */trunk* 到 */branches/1.0*.
3. 团队继续并行工作. 一个团队开始对发布分支 进行严格的测试，其他团队继续在 */trunk* 上开发 新的工作（例如版本 2.0）. 如果有问题出现（无论是在 */trunk*，还是发布分支），修复问题，并把修改精选到拥有 相同问题的分支上. 但是这个过程有时候也会停下来，例如为了发布前的 最终测试而 “冻结” 分支.
4. 为分支打标签并发布. 当测试结束，*/branches/1.0* 被复制到 */tags/1.0.0* ，标签被打包并交付给客户.
5. 继续维护分支. 当团队在主干上为版本 2.0 工作时，修复问题的修改从 */trunk* 回植到 */branches/1.0*. 如果修改积累得足够多了，团队 可能决定发布 1.0.1: */branches/1.0* 被复制到 */tags/1.0.1*，标签被打包并交付给客户.

整体过程随着软件的成长而不断重复：当版本 2.0 完成时，创建了一个新 的 2.0 发布分支，再对该分支进行测试，打标签并发布. 几年后，仓库拥有 了大量的处于 “维护” 状态的发布分支，以及代表了最终交付 版的标签.

特性分支

特性分支 (*feature branch*) 是分支的一种类型，它曾是本章的主要例子（在这个例子中，你在分支上工作，而 Sally 在 */trunk* 上工作）. 这是一个临时分支，为了完成一 个复杂的修改，在完成前不能影响 */trunk* 的稳定性. 与发布分支不同（发布分支可能需要永远支持），特性分支被创建后，使用一 段时间，被合并到主干后就会被删除—它们的寿命是很有限的.

在什么样的情况下才需要创建一个特性分支—对于这个问题，不同的 项目，其策略也不尽相同. 有些项目甚至根本就不使用特性分支：*/trunk* 的提交就是一场大混战. 不使用特性分支的好处是操作 简单—开发人员不用了解分支或合并. 缺点是主干上的代码会经常处于不 可用状态. 还有些项目过度使用分支：没有一个修改是直接提交到主干上的，即使是非常简单的修改，也要创建一个短期分支，认真地审查修改后再合并到 主干，然后再删除分支. 这样做保证了主干上的代码总是可用的，但代价就是 极大的过程开销.

大多数项目走的是 “中间路线”：他们通常会坚持主干上的 代码应该总是可编译的，而且通过了所有的回归测试. 只有当修改含有大量不 稳定的提交时，才会创建特性分支. 一条很好的经验法则是：让一个程序员 单独工作几天，然后一次性提交所有修改（于是 */trunk* 总是可用的），如果这个提交所包含的修改过于庞大，以致于无法审查，那就 应该在特性分支中完成开发，然后再合并到主干上. 因为每次提交到分支上的 修改相对较小，它们可以轻易地被同行审议.

最后，是如何保持分支与主干 “同步”：我们之前已经警告 过，如果连续地在分支上工作几周，甚至几个月，同时主干上也有新的提交 出现，一直到两条开发线之间出现非常大的差异，此时再把分支合并到主干 上可能会成为一场恶梦.

解决问题最好的办法是定期从主干自动合并到分支，可以定一个标准，比如一周合并一次.

当用户终于准备好把 “同步的” 特性分支合并到主干上时，最后再为特性分支做一次自动同步合并，合并后分支和主干就是一样的了（除 了分支特有的修改）. 然后再执行自动再整合合并，把分支合并到主干上.

```
$ cd trunk-working-copy

$ svn update
Updating '.':
At revision 1910.

$ svn merge ^/calc/branches/mybranch
--- Merging differences between repository URLs into '.':
U    real.c
U    integer.c
A    newdirectory
A    newdirectory/newfile
 U   .
…
```

这种分支模式的另一种理解方式是：每周从主干向分支的同步就像在工作 副本中执行 *svn update*，最后从分支到主干的合并就像 在工作副本中执行 *svn commit*. 毕竟，工作副本就像是 一个非常浅的私有分支，一次只能保存一个修改.

供方分支

软件开发过程中可能会遇到这样一种情况，用户所维护的代码依赖其他人的数据，通常来说，项目会要求所依赖的数据尽量处于最新状态，但不能影响稳定性．只要某个团队所维护的数据会对另一个团队产生直接的影响，这种场景就会一直 存在．

比如说软件开发人员所开发的应用程序会用到一个第三方函数库，Subversion 和 Apache Portable Runtime (APR) 的关系即是如此，见 “Apache 可移植运行库” 一节．为了实现可移植性，Subversion 的源代码依赖于 APR 函数库，在 Subversion 的早期阶段，项目 总是紧紧追随 APR 的 API 更新．现在 Subversion 和 APR 都已经进入成熟期，所以 Subversion 只使用 APR 经过充分测试的稳定版 API．

如果你的项目依赖其他人的数据，有若干种方式可以用来同步这些数据．其中 最麻烦的一种是以口头或书面的方式通知项目的所有成员，将项目所需的第三方数 据更新到某个特定版本．如果第三方数据使用 Subversion 进行管理，就可以利用 Subversion 的外部定义，快速地将第三方数据更新到特定版本，并存放在工作 副本中（见 “外部定义” 一节）．

有时候用户可能需要使用自己的版本控制系统去维护第三方代码的定制化 修改．回到软件开发的例子中，程序员可能需要修改第三方函数库，以满足自己 的特殊需求．这些定制化修改可能包括新功能或问题修正，直到成为第三方函数 库的官方修改之前，它们只在内部维护．或者这些定制化修改永远不会发送给函数 库的官方维护人员，它们只是为了满足项目的需求而单独存在．

现在你面临一种非常有趣的情况．你的项目可以使用几种互不相交的方式存放 第三方数据的定制化修改，比如说使用补丁文件，或文件和目录的成熟的替代 版本．但是维护人员很快就会感到头疼，因此迫切需要一种机制，能够方便地把你的定制化修改应用到第三方代码上，并 且当第三方代码更新时能够迫使开发人员重新生成这些修改．

解决办法是使用 *供方分支* (*vendor branches*)．供方分支是一个存在于你自己的版本控制系统中的 目录，包含了由第三方提供的数据．被项目吸收的每一个供方数据版本都称为 一个 *供方物资* (*vendor drop*)．

供方分支有两个好处．首先，通过在自己的版本控制系统中存放当前支持 的供方物资，你就可以确认项目成员不必再担心他们是否使用了供方数据的正确 版本，只需要更新工作副本，他们就可以得到供方数据的正确版本．第二，因为 供方数据使用 Subversion 进行管理，所以用户可以方便地在仓库中存放自己的 定制化修改，而无需再使用某种自动的（或手动的）方法对定制化修改进行换 入换出．

不幸的是，在 Subversion 中并不存在一种管理供方分支的最佳方法．系统 的灵活性提供了多种不同的管理方法，每一种都有各自的优缺点，没有一种方法 可以当成 “万能钥匙”．在下面几节里，我们将从较高的层面介绍其中几种方法，所使用的例子也是依赖第三方函数库的 典型示例．

通常的供方分支管理过程

维护第三方函数库的定制化修改会牵涉到 3 个数据源：最新版的定制化 修改所基于的第三方函数库的版本，项目所使用的定制化版本（即实际上的 供方分支），以及第三方函数库的新版本．于是，管理供方分支（供方分支应 存放在用户自己的代码仓库中，包含了前面提到的 3 个数据源）本质上可以归结为 执行合并操作（指的是一般意义上的合并），但是其他开发团队可能会对其他 数据源—第三方函数库代码的全新版本—采取不同的策略，所以说 同样存在几种不同的方法去执行合并操作．

严格来说，有几种不同的方式用来执行这些合并操作，为简单起见，也为了 向读者展示一些具体的东西，我们假设只有一个供方分支，每当第三方函数库 发布新版本时，通过应用当前版本与最新版之间的差异，将分支更新到新的发 布版本．

另一种办法是为第三方函数库的每一个新版本都创建一个新的供方分支，并将当前原版函数库与定制版本（来自当前的供方分支）之间的差异应用到新的分支上. 这种方法并没有什么问题—我们只是觉得没必要在这里介绍所有的可能性.

下面几节介绍了在几种不同的场景中，如何创建并管理供方分支. 在下面的例子里，我们假设第三方函数库的名字是 libcomplex, 当前供方分支所基于的版本是 libcomplex 1.0.0, 分支的位置是 *^/vendor/libcomplex-custom*. 稍后读者将会看到如何把供方分支升级到 libcomplex 1.0.1, 同时保留定制化修改.

来自外部仓库的供方分支

先来看第一种管理供方分支的方法，该方法的适用条件是第三方函数库可以通过 Subversion 进行访问. 为了方便说明，我们假设函数库 libcomplex 存放在可以公开访问的 Subversion 仓库中，而且函数库的开发人员也使用了通常的发布步骤，即为每一个稳定的发布版创建一个标签.

从 Subversion 1.5 开始，*svn merge* 支持外部仓库合并 (*foreign repository merges*), 也就是合并的源与目标属于不同的仓库. 与旧版相比，Subversion 1.8 的 *svn copy* 的行为有所变化：如果从外部仓库复制目录到工作副本中，得到的目录将被工作副本收录，等待添加到仓库中. 这个特性叫做外部仓库复制 (*foreign repository copy*), 我们将用它引导供方分支.

现在开始创建我们的供方分支. 一开始先在仓库中创建一个存放所有供方分支的目录，然后检出该目录的工作副本.

```
$ svn mkdir http://svn.example.com/projects/vendor \
            -m "Create a container for vendor branches."
Committed revision 1160.
$ svn checkout http://svn.example.com/projects/vendor \
               /path/to/vendor
Checked out revision 1160.
$
```

利用 Subversion 的外部仓库复制特性，从供方仓库获取 libcomplex 1.0.0 的一份副本—包括文件和目录上所有的 Subversion 属性.

```
$ cd /path/to/vendor
$ svn copy http://svn.othervendor.com/repos/libcomplex/tags/1.0.0 \
           libcomplex-custom
--- Copying from foreign repository URL 'http://svn.othervendor.com/repos/lib\
complex/tags/1.0.0':
A    libcomplex-custom
A    libcomplex-custom/README
A    libcomplex-custom/LICENSE
…
A    libcomplex-custom/src/code.c
A    libcomplex-custom/tests
A    libcomplex-custom/tests/TODO
$ svn commit -m "Initialize libcomplex vendor branch from libcomplex 1.0.0."
Adding         libcomplex-custom
Adding         libcomplex-custom/README
Adding         libcomplex-custom/LICENSE
…
```

```
Adding         libcomplex-custom/src
Adding         libcomplex-custom/src/code.h
Adding         libcomplex-custom/src/code.c
Transmitting file data ....................................
Committed revision 1161.
$
```

如果用户使用的 Subversion 版本较旧，不支持外部仓库复制，那么与 此最类似的替代操作是导入（通过命令 *svn import*) 供方标签的工作副本，导入时需要加上选项 `--no-auto-props` 和 `--no-ignore`, 这样才能保证目录及其 所有的属性都能被完整地导入到你的仓库中.

有了基于 libcomplex 1.0.0 的供方分支后，我们就可以开始对 libcomplex 进行定制化修改，然后提交到分支上，并且可以开始在自己的应用程序中使用 修改后的 libcomplex.

一段时间后，官方发布了 libcomplex 1.0.1, 查看新版的修改日志后，我 们决定把自己的供方分支也升级到 1.0.1, 这时候需要用到 Subversion 的 外部仓库合并. 当前的供方分支是原始的 libcomplex 1.0.0 再加上我们的定 制化修改，现在我们需要把原始的 1.0.0 与 1.0.1 之间的差异应用到供方分 支，最理想的情况是被应用的差异不会影响到我们的定制化修改. 合并操作需要 使用 二路 URL 形式的 *svn merge*.

```
$ cd /path/to/vendor
$ svn merge http://svn.othervendor.com/repos/libcomplex/tags/1.0.0 \
            http://svn.othervendor.com/repos/libcomplex/tags/1.0.1 \
            libcomplex-custom
--- Merging differences between foreign repository URLs into '.':
U    libcomplex-custom/src/code.h
C    libcomplex-custom/src/code.c
U    libcomplex-custom/README
Summary of conflicts:
  Text conflicts: 1
Conflict discovered in file 'libcomplex-custom/src/code.c'.
Select: (p) postpone, (df) diff-full, (e) edit, (m) merge,
        (mc) mine-conflict, (tc) theirs-conflict, (s) show all options:
```

可以看到, *svn merge* 把 libcomplex 1.0.0 升级 到 1.0.1 的修改合并到了我们的工作副本. 在例子中，有一个文件发生了冲突, 应该是供方修改的区域与我们的定制化修改有所重叠. Subversion 安全地检测 到了冲突，并询问我们如何解决，使得定制化修改在 libcomplex 1.0.1 上仍然 有效.（关于冲突解决，见 “解决冲突”一节).

冲突一旦解决，并且测试和审查都没有问题后，就可以提交到供方分支上.

```
$ svn status libcomplex-custom
M       libcomplex-custom/src/code.h
M       libcomplex-custom/src/code.c
M       libcomplex-custom/README
$ svn commit -m "Upgrade vendor branch to libcomplex 1.0.1." \
             libcomplex-custom
Sending        libcomplex-custom/README
Sending        libcomplex-custom/src/code.h
Sending        libcomplex-custom/src/code.c
Transmitting file data ...
```

```
Committed revision 1282.
$
```

这就是当供方源是 Subversion 仓库时，管理供方分支的方式．这种方式 有几个值得注意的缺点，首先，外部仓库合并不能像同一仓库那样自动跟踪，这就意味着必须由用户记住供方分支已经做过哪些合并，以及下次升级时如何 构造合并．另外—对于其他形式的合并同样适用—源的重命名操作 会造成不小的麻烦，不幸的是目前并没有有效的办法缓解这个问题．

来自镜像源的供方分支

在上一节（“来自外部仓库的供方分支”一节）我们介绍了 当供方物资可通过 Subversion 进行访问时如何实现与维护供方分支．这是 一种比较理想的情况，因为 Subversion 非常擅长处理由它进行管理的数据 的合并．不幸的是，并不是所有的第三方函数库都可以通过 Subversion 进行 访问．很多时候，项目所依赖的函数库是通过非 Subversion 机制交付的，例如源代码的发布版压缩包．对于这种情况，我们强烈建议用户在把非 Subversion 信息导入 Subversion 时，尽量保持干净．下面我们将介绍另一 种供方分支管理方法，其中第三方函数库的发布版将以镜像的方式存放在我 们的仓库中．

首次创建供方分支非常简单，对于我们的例子而言，假设 libcomplex 1.0.0 是以代码压缩包的形式发布．为了创建供方分支，首先把 libcomplex 1.0.0 的压缩包解压到我们的仓库中，作为一个只读（只是一种惯例）的供方标签．

```
$ tar xvfz libcomplex-1.0.0.tar.gz
libcomplex-1.0.0/
libcomplex-1.0.0/README
libcomplex-1.0.0/LICENSE
…
libcomplex-1.0.0/src/code.c
libcomplex-1.0.0/tests
libcomplex-1.0.0/tests/TODO
$ svn import libcomplex-1.0.0 \
             http://svn.example.com/projects/vendor/libcomplex-1.0.0 \
             --no-ignore --no-auto-props \
             -m "Import libcomplex 1.0.0 sources."
Adding         libcomplex-custom
Adding         libcomplex-custom/README
Adding         libcomplex-custom/LICENSE
…
Adding         libcomplex-custom/src
Adding         libcomplex-custom/src/code.h
Adding         libcomplex-custom/src/code.c
Transmitting file data ....................................
Committed revision 1160.
$
```

注意，在导入时我们为命令增加了选项 `--no-ignore`，这样 Subversion 就不会遗漏任意一个文件或目录，同时还增加了选项 `--no-auto-props`，这样的话，Subversion 客户端就不会生成 供方物资中原本就没有的属性信息．[13]

供方发布物资进入我们的仓库后，接下来就可以用 *svn copy* 创建供方分支．

[13]严格来说，可以允许自动属性 工作，但其中的关键问题是确保每一个供方物资都能得到相同的对待．

```
$ svn copy http://svn.example.com/projects/vendor/libcomplex-1.0.0 \
           http://svn.example.com/projects/vendor/libcomplex-custom \
           -m "Initialize libcomplex vendor branch from libcomplex 1.0.0."
Committed revision 1161.
$
```

现在，我们拥有了基于 libcomplex 1.0.0 的供方分支，接下来就可以按照 项目的需要，对 libcomplex 进行定制化修改—修改完成后直接提交到 刚创建的供方分支里—然后再在自己的项目中使用定制过的 libcomplex.

一段时间后，发布了 libcomplex 1.0.1. 通过查看修改日志，我们打算 把供方分支升级到新版. 为了升级供方分支，我们需要把 1.0.0 和 1.0.1 之间的差异应用到供方分支中，而且不能影响定制化修改. 完成这项操作最 案例的方式是先把 libcomplex 1.0.1 作为 *libcomplex 1.0.0* 的增量版本 导入到我们的仓库中，然后使用 二路 URL 形式的 *svn merge*, 把差异应用到供方分支中.

事实证明，有多种方式都可以正确地把 libcomplex 1.0.1 添加到仓库中. [14] 我们在这里介绍的方法相对比较原始，但作为说明 已经足够了.

记住，我们希望 libcomplex 1.0.1 在我们这儿的镜像能和 1.0.0 的镜像 共享祖先，这样的话在把它们之间的差异合并到供方分支时，能产生最好的效果. 于是，首先通过复制 “供方标签” libcomplex-1.0.0 创建分支 libcomplex-1.0.1—它最终将变成 libcomplex-1.0.1 的副本.

```
$ svn copy http://svn.example.com/projects/vendor/libcomplex-1.0.0 \
           http://svn.example.com/projects/vendor/libcomplex-1.0.1 \
           -m "Setup a construction zone for libcomplex 1.0.1."
Committed revision 1282.
$
```

现在我们需要检出分支 libcomplex-1.0.1 的工作副本，然后把工作副本 中的代码升级到 1.0.1. 为了完成这些操作，我们将利用这样一个事实，就是 *svn checkout* 可以覆盖已存在的目录，并且如果提供了 选项 `--force`, 那么检出的目录和被覆盖的目标目录之间 的差异将作为本地修改，留在工作副本中.

```
$ tar xvfz libcomplex-1.0.1.tar.gz
libcomplex-1.0.1/
libcomplex-1.0.1/README
libcomplex-1.0.1/LICENSE
…
libcomplex-1.0.1/src/code.c
libcomplex-1.0.1/tests
libcomplex-1.0.1/tests/TODO
$ svn checkout http://svn.example.com/projects/vendor/libcomplex-1.0.1 \
               libcomplex-1.0.1 \
               --force
E    libcomplex-1.0.1/README
E    libcomplex-1.0.1/LICENSE
E    libcomplex-1.0.1/INSTALL
…
E    libcomplex-1.0.1/src/code.c
```

[14] 不正确的做法是再使用一次 *svn import*, 因为这将导致 libcomplex 1.0.0 和 libcomplex 1.0.1 不含有共同的祖 先.

```
E    libcomplex-1.0.1/tests
E    libcomplex-1.0.1/tests/TODO
Checked out revision 1282.
$ svn status libcomplex-1.0.1
M       libcomplex-1.0.1/src/code.h
M       libcomplex-1.0.1/src/code.c
M       libcomplex-1.0.1/README
$
```

可以看到，在 libcomplex 1.0.1 的目录中检出 libcomplex 1.0.0 的 代码，将得到一个包含了本地修改的工作副本—正是这些修改，把 libcomplex 1.0.0 升级到 libcomplex 1.0.1.

的确，这是一个非常简单的例子，升级操作只涉及到已有文件的修改. 在实际 工作中，第三方函数库的新版修改可能还包括添加或删除文件（目录），重命名文 件或目录等. 在这种情况下，把供方标签升级到新版会困难得多，作为训练，具体 的升级过程将留给读者完成. [15]

不管怎么，我们成功地把供方标签的工作副本升级到了 libcomplex 1.0.1, 然后提交修改.

```
$ svn commit -m "Upgrade vendor branch to libcomplex 1.0.1." \
             libcomplex-1.0.1
Sending        libcomplex-1.0.1/README
Sending        libcomplex-1.0.1/src/code.h
Sending        libcomplex-1.0.1/src/code.c
Transmitting file data ...
Committed revision 1283.
$
```

我们终于准备好了升级供方分支. 记住，我们的目标是把原始的 1.0.1 和 1.0.0 之间的差异应用到供方分支中. 下面展示了如何使用 二路 URL 形式的 *svn merge* 去更新供方分支的工作副本.

```
$ svn checkout http://svn.example.com/projects/vendor/libcomplex-custom \
               libcomplex-custom
E    libcomplex-custom/README
E    libcomplex-custom/LICENSE
E    libcomplex-custom/INSTALL
…
E    libcomplex-custom/src/code.c
E    libcomplex-custom/tests
E    libcomplex-custom/tests/TODO
Checked out revision 1283.
$ cd libcomplex-custom
$ svn merge ^/vendor/libcomplex-1.0.0 \
            ^/vendor/libcomplex-1.0.1
--- Merging differences between repository URLs into '.':
U    src/code.h
C    src/code.c
U    README
```

[15] 提示: `svn add --force /path/to/working-copy --no-ignore --no-auto-props` 可以方便地把任意一个新的供方物资条目 添加到仓库中.

```
Summary of conflicts:
  Text conflicts: 1
Conflict discovered in file 'src/code.c'.
Select: (p) postpone, (df) diff-full, (e) edit, (m) merge,
        (mc) mine-conflict, (tc) theirs-conflict, (s) show all options:
```

可以看到，*svn merge* 将必要的修改合并到工作副本 上，并将修改区域重叠的文件标记为冲突. Subversion 检测到冲突后，将允许 用户解决冲突（使用 “解决冲突”一节 介绍的方 法），使得我们的定制化修改在 libcomplex 1.0.1 中仍能正常工作. 冲突一旦 解决，并且审查与测试后都没出现什么问题，就可以提交了.

```
$ svn status
M       src/code.h
M       src/code.c
M       README
$ svn commit -m "Upgrade vendor branch to libcomplex 1.0.1."
Sending        README
Sending        src/code.h
Sending        src/code.c
Transmitting file data ...
Committed revision 1284.
$
```

到此为止，供方分支的升级工作就算完成了. 如果将来还要再次升级，仍然 可以按照本节介绍的步骤进行操作.

分支, 还是不分支?

分支还是不分支—这是一个有趣的问题. 本章非常深入地介绍了与分支 和合并有关的知识，这两者通常是 Subversion 用户困惑的主要来源. 虽然在分支 的创建和管理中，需要生搬硬套的内容并不复杂，但是有些用户经常就是否需要创 建分支而犹豫不决. 读者已经看到，Subversion 可以处理常见的分支管理场景，所以说决定是否需要为项目创建分支在技术上几乎不会产生什么影响，它的社会 影响反而占据更大的比重. 下面介绍一些在软件项目中使用分支的好处和代价.

使用分支最明显的好处是隔离性. 提交到分支上的修改不会影其他的开发线， 提交到其他开发线的修改也不会影响分支. 利用这点，开发人员就能安全地在分支 上开发新特性，对复杂的问题进行修正，对代码进行重写等. 无论 Sally 在自己 的分支上怎么折腾，Harry 和团队的其他人都可以不受阻碍地继承他们的工作.

利用分支，我们可以把相关的修改都组织到一个容易识别的集合中，比如说修 正某个问题的修改可能由多次提交组成，这些提交的版本号不是连续的. 用户可能 会用人类容易理解的语言描述它们: ″版本号 1534, 1543, 1587 和 1588", 很可 能还要在问题跟踪系统中再手工地生成这些号码. 如果修正问题的修改需要被移植 到其他版本中，则开发人员还要确保不会遗漏任何一个版本号. 但是，如果把这些 修改都提交到一个独一无二的分支中，那么在问题跟踪系统或移植修改时，只需要 引用分支的名字就能确定是哪些提交.

然而，使用分支的缺点是它的隔离性 会 与团队的 协作需求相抵触. 取决于同事的工作习惯，提交到分支上的修改可能不像主线上 的修改那样，得到非常充分的讨论，审查与测试. 分支的隔离性会鼓励用户抛弃 版本控制的 “最佳做法”，导致版本历史难以在事后进行审查. 在长期存在的分支上工作的开发人员有时候需要付出额外的努力，以确保分支的 演化方向与同事的保持一致. 对于有些分支而言，这些缺点都不算是问题，因为 它们只是试探性的分支，仅仅是在尝试代码库未来的发展方向，将来不会被整合 到主线上—单纯的规范不一定会扼杀远见. 但是有一个简单的 事实不容忽视，那就是代码及其修改如果能得到更多人的审查与理解，那么对项目 而言通常是有好处的.

并不是说使用分支在技术上一点坏处都没有. 如果我们从更抽象的层次来 看待分支, 就会发现 每次检出仓库的工作副本, 从某种意义上来说其实就是在创建分支, 这只是分支 的一种特殊类型, 它只存在于客户端主机, 不在仓库里, 用 *svn update* 把仓库的修改同步到分支上—非常像特殊情况下的, 简化版的 *svn merge*;[16] *svn commit* 等效于重新整合分支. 所以说从 这个角度来看, Subversion 用户其实一直都在和分支与合并打交道. 不用对 更新与合并之间的相似性过于惊讶, Subversion 短处最集中的地方—也就是 对文件和目录重命名的处理, 以及目录冲突的处理—都会给 *svn update* 和 *svn merge* 造成麻烦. 不幸的是 *svn merge* 的麻烦更大, 虽然 *svn update* 可以看成是 *svn merge* 的简化形式和 特例, 但一个真正的合并操作既不针对特例, 也不简单. 由于这个原因, 合并操作执行起来比更新更慢, 还要求显式的跟 踪 (通过本章讨论过的属性 `svn:mergeinfo`) 和历史分析计 算, 而且出错的机会也更多.

分支, 还是不分支? 归根结底还要看开发团队如何把握协作与隔离之间的 平衡.

小结

这一章我们讲了很多. 首先介绍了标签和分支的概念, 并说明了 Subversion 如何通过 *svn copy* 复制目录来实现这两个功能. 然后展示 了如何使用 *svn merge* 把修改从一个分支复制到另一个分支 上, 或回退错误的修改. 再然后介绍了如何使用 *svn switch* 创建具有混合位置的工作副本. 最后我们讨论了如何管理分支的组织与生存周期.

记住, Subversion 的分支和标签是很廉价的, 所以当你需要时请尽管使用!

为了方便读者, 下面的表格总结了与分支有关的常见操作及其对应的命令.

表 4.1. 分支与合并命令

操作	命令
创建一个分支或标签	`svn copy URL1 URL2`
把工作副本切换到另一个分支或标签	`svn switch URL`
将分支与主干同步	`svn merge trunkURL; svn commit`
查看合并历史或可合并的变更集	`svn mergeinfo SOURCE TARGET`
将分支合并至主干	`svn merge branchURL; svn commit`
合并一个特定的修改	`svn merge -c REV URL; svn commit`
合并一段范围的修改	`svn merge -r REV1:REV2 URL; svn commit`
从自动合并中拦截某个特定的修改	`svn merge -c REV --record-only URL; svn commit`
合并预览	`svn merge URL --dry-run`
放弃合并的结果	`svn revert -R .`
从历史中恢复文件	`svn copy URL@REV localPATH`
撤消已提交的修改	`svn merge -c -REV URL; svn commit`
查看对合并敏感的历史	`svn log -g; svn blame -g`

[16] 实际上, 你 可以 在工作副本中用 `svn merge -rLAST_UPDATED_REV:HEAD .`, 逐字地把仓库中的自从上一次更新后的修改合并到工作副本.

操作	命令
从工作副本创建一个标签	`svn copy . tagURL`
重新安排分支或标签的布局	`svn move URL1 URL2`
删除分支或标签	`svn delete URL`

第 5 章 仓库管理

Subversion 仓库是存放所有版本化数据的中心位置，因此受到系统管理员的额外照顾也是很正常的. 仓库的维护工作很少，更重要的是理解如何正确地配置它，这样才能避免出现潜在的问题，并解决实际发生的问题.

本章将介绍如何创建与配置 Subversion 仓库，还将通过几个例子，介绍 应该在什么时候，怎么用 Subversion 提供的工具维护仓库. 在这过程中，我 们将解决一些常见的问题和误区，并对如何管理仓库的数据提出一些建议.

如果读者只是作为一名普通用户访问仓库中的数据（通过 Subversion 客户端），完全可以跳过本章. 但如果你是（或者想成为）一名 Subversion 仓库管理员[1]，那么本章就是为 你而写的.

仓库的定义

在进入与仓库管理有关的主题之前，先给出仓库的定义. 它是什么样的? 感觉怎么样? 它喜欢喝热茶还是冰茶，加不加糖或柠檬? 作为一名管理员，人们 期望你能同时从字面和系统层面理解仓库的组成—仓库看起来是什么 样的，被 Subversion 以外的工具操作时如何反应；还要从逻辑层面理解在 仓库 内部 数据是如何表示的.

如果站在文件浏览器（例如 Windows 资源管理器）或基于命令行的文件 系统导航工具来看，Subversion 仓库只是一个包含了众多数据的普通目录，其中一些子目录包含了人类可读的配置文件，还有些子目录包含的是人类不可 读的文件. 在设计 Subversion 时，开发人员非常注意模块化与层次化，因此 简单地浏览一个典型仓库，对于展示仓库的基本组件来说已经足够了.

```
$ ls repos
conf/  db/  format  hooks/  locks/  README.txt
```

下面是关于目录中的文件及其子目录的简单介绍.（不要拘泥于术语的具体 意思—更详细的内容将在本章的其他地方进行介绍）

conf/

存放配置文件的目录.

db/

该目录包含了与所有版本化数据相关的数据. [2]

format

该文件描述了仓库的内部组织结构.（目录 *db/* 有时候也会有一个叫做 *format* 的文件，这个 *format* 仅仅是在描述 *db/* 的内容，这两个 *format* 之间并没有关系.）

hooks/

该目录包含了钩子脚本模板和已安装的钩子脚本.

[1] 说得可能过于严肃了，其实我们说的是那些对工作 副本以外的神秘领域感兴趣的读者.

[2] 严格地讲，Subversion 并没有强制要求版本化数据一定要存放在该目录内，还存在 一些其他的后端存储实现（尽管是有专利的）不把数据存放在该目录内.

locks/

Subversion 用该目录存放仓库的锁文件，锁文件用于管理仓库的并发 访问.

README.txt

包含了一小段内容的文本文件，文件的内容仅仅是为了提醒计算机 用户该文件所在的目录是一个 Subversion 仓库.

在 Subversion 1.5 之前，仓库还有一个子目录 *dav*, *mod_dav_svn* 使用该目录存放与 WebDAV *活动* (*activities*)—高层的 WebDAV 协议概念到 Subversion 提交事务的映射—有关的信息. Subversion 1.5 修改了这一行为，它把活动目录的所有权和配置目录位置的 能力移交给了 *mod_dav_svn*. 如今，新的仓库不需要 再保留子目录 *dav*, 除非 *mod_dav_svn* 正在被使用，并且还没有把活动数据库 存放位置设置到其他地方. 更多信息见 “mod_dav_svn 配置指令”一节

当然，当我们通过 Subversion 库函数访问仓库时，这些文件与目录就成为 了一种虚拟的，版本化的文件系统的实现，并且搭配了可定制的事件触发器. 这个文件系统对文件和目录的概念都 有自己的理解，和真正的文件系统（例如 NTFS, FAT32, ext3 等）非常类似，但它又比较特殊—它把文件和目录悬挂在版本号上，安全地记录用户曾经 对它们做出的修改，并保证这些修改是永远可访问的. 这就是存放用户所有 版本化数据的地方.

文件系统

在设计 Subversion 的开始阶段，开发人员使用 Berkeley DB (BDB) 实现 虚拟版本化文件系统的后端存储机制. 选择 Berkeley DB 基于多种理由，包 括它的开源授权，事务支持，可靠性，高性能，简单的 API, 线程安全，支持 游标等.

后来，引入了新的后端存储机制—*FSFS*.[3] 这个所谓的“文件系统的文件系统” 并不是一个在不透明的 数据库容器中实现的版本化文件系统，而是一个由大量文件组成的集合，这些文件更加透明，存放在操作系统的文件系统中. 随着软件的成熟，FSFS 最终成为了 Subversion 默认的后端存储机制. FSFS 并没有因此停止前进的 脚步，后来，几乎在每一项指标上—从性能，到可扩展性，再到可靠性 —FSFS 都超越了 Berkeley DB.

如今，如果用户使用的是开源的 Subversion 产品，那么一般情况下使用 FSFS 作为仓库的后端存储. 实际上从 Subversion 1.8 开始，官方不再推荐 使用 Berkeley DB 作为仓库的后端存储，仍然使用 Berkeley DB 的仓库在新 版的 Subversion 1.x 下可以继续工作，但与 Berkeley DB 相关的开发工作 不再继续—包括新特性或扩展. Subversion 只想提供一种单一的仓库 后端存储机制，而 FSFS 赢得了青睐.

必要时，本书仍然会介绍如何管理基于 BDB 的仓库，但本章的大部分内容 假设仓库后端存储用的是 FSFS. 关于如何管理基于 BDB 仓库的更完整的信息，请参考 附录 D, *传统的 Berkeley DB 后端存储* 或本书的旧版.

仓库部署策略

得益于 Subversion 仓库在整体设计和技术上的简洁，创建与配置仓库 是一项相对来说比较简单的任务. 需要做一些初步决定，但是设置仓库所涉及 到的实际工作非常基础，多做几次就能得心应手.

在开始创建仓库之前需要考虑的事情有：

[3] 虽然经常被读作 “fuzz-fuzz”, 但本书 假定读者把它念作 “eff-ess-eff-ess”.

- 仓库会存放哪些数据，如何组织这些数据?
- 仓库会放在哪儿，如何访问仓库?
- 你需要哪些类型的访问控制?

本节将帮助读者回答上面的几个问题.

规划仓库的组织方式

虽然 Subversion 允许用户随意地移动目录和文件，而不丢失任何信息，甚至还可以把版本历史的整个集合从一个仓库移动到另一个仓库，但这样做 难免会影响到那些需要经常访问仓库的人，他们希望仓库及其文件的位置能 够保持稳定. 所以说在创建新的仓库之前，试着为将来考虑，提前做好规划. 提前认真地规划好仓库及其所管理的数据，可以避免将来出现很多不必要的 麻烦.

假设仓库管理员需要为几个项目提供版本控制支持，首先需要决定的 是用一个单独的仓库存放这些项目，还是每个项目一个仓库，还是介于两者 之间.

把多个项目放在一个单独的仓库里有一定的好处，最明显的就是减少了 重复劳动. 一个单独的仓库意味着只有一个钩子集合，需要例行备份的东西 只有一个，如果 Subversion 发布了不兼容旧版的新版本，只有一个仓库需 要转储和加载等. 另外，用户还可以方便地在项目之间移动数据，而不会丢失 任何历史信息.

缺点是不同的项目对仓库的事件触发的需求可能不同，例如把提交通知发到不同 的邮件列表，对于如何构成一次合法的提交，其要求也会有所不同. 当然，这些问 题并非无法克服—只是说所有的钩子脚本都依赖仓库的布局，而不是假定整个 仓库只和单独的一组人有关系. 另外还要注意 Subversion 版本号的作用域是整个 仓库，虽然这些数字并没有任何特殊的魔力，但还是有人不喜欢看到这样一种情况：虽然他们的项目最近都没有提交什么修改，但是由于其他项目的活动，他们自己的 项目的版本号仍然在不断增长. [4]

还有一种折衷一点的方式可以采用. 比如说根据项目之间的相关性进行 分组，把相关性强的项目放在一个单独的仓库里. 这样的话项目之间共享数 据就会更加方便，如果有新的版本号产生，开发人员至少会知道这些新的版 本号和仓库中的项目都有或多或少的联系.

项目的组织方式确定后，接下来就要考虑仓库内部的目录结构. 因为 Subversion 在创建分支和标签时使用的是常规的目录拷贝操作（见 第 4 章 分支与合并), Subversion 社区建议管理员为每一个项目 的根目录—包含了所有与项目相关的数据的“顶层” 目录 —选择一个仓库位置，然后在项目根目录下创建 3 个子目录：*trunk* 是开发主线；*branches* 存放所有的分支；*tags* 存放所有的标签. [5]

比如说，仓库的目录结构可能是下面这样：

```
/
  calc/
    trunk/
    tags/
```

[4]在考虑如何实现合理的软件开发 标准时，无论是完全没有重视，还是考虑不周，全局的版本号数字都不应该 成为需要担心的事情，而且在决定如何管理项目与仓库时，也 *没必要* 去顾虑虑全局版本号.

[5]*trunk*, *tags* 和 *branches* 这 3 者有时被称为 “TTB 目录”

```
    branches/
  calendar/
    trunk/
    tags/
    branches/
  spreadsheet/
    trunk/
    tags/
    branches/
  …
```

注意，项目根目录在仓库中的位置并不重要．如果每个仓库只包含一个项 目，那么最常见的安排就是把仓库的根目录作为项目的根目录．如果一个仓库 内包含了多个项目，管理员可能会对它们进行分组，例如把目标类似或共享代 码的项目放在同一个子目录内，或者只是根据字母顺序进行分组．一个例子是：

```
/
  utils/
    calc/
      trunk/
      tags/
      branches/
    calendar/
      trunk/
      tags/
      branches/
    …
  office/
    spreadsheet/
      trunk/
      tags/
      branches/
    …
```

管理员完全可以按照自己的喜好决定仓库的布局，Subversion 对仓库的布局 并没有特殊的要求—在它看来，一个目录只是个目录而已．总而言之，仓库的布局应该满足项目及其开发人员的需求．

为了使讨论更加完整，再介绍一种非常常见的仓库布局．在这种布局里，*trunk*, *tags* 和 *branches* 位于仓库的根目录，而各个项目是它们各 自的子目录，例如：

```
/
  trunk/
    calc/
    calendar/
    spreadsheet/
    …
```

```
tags/
  calc/
  calendar/
  spreadsheet/
  …
branches/
  calc/
  calendar/
  spreadsheet/
  …
```

这种布局并没有什么错误的地方，但是对用户来说可能不够直观．如果项 目很多，并且每个项目都比较复杂，开发人员也很多，那么开发人员可能更希 望每个仓库只包含一两个项目．但上面的做法倾向于弱化项目的独特性，更 希望把整个项目集合看作一个单一的实体．尽管这只是一个社会议题，但我们 更希望读者使用一开始就推荐的仓库布局，因为如果在一个单一的仓库路径上 单独记录了一个项目的整个历史，那么查询（修改或迁移）单个项目的整个历 史—过去的，现在的，打过标签的和创建过分支的—就更加方便．

确定仓库的托管方式和位置

在创建你的 Subversion 仓库之前，需要回答的一个问题是仓库应该放在 哪里．这个问题关系到很多其他的选择，例如仓库的访问方式（是通过 Subversion 服务器进行访问，还是直接访问），哪些用户可以访问（是只允许 公司的内部员工，还是开放给整个互联网），围绕着 Subversion 还提供了哪 些服务（仓库浏览接口，基于邮件的提交通知等），数据的备份策略等．

本书在 第 6 章 服务器配置 介绍服务器的配置，但我 们在这里要说明的是有些问题的答案可能会影响仓库的位置．比如说特定的部 署场景可能会要求从多台计算机，通过远程文件系统访问仓库，或者是使用多 个仓库，而这些仓库的数据是在地理上分布的同步数据，从而为分散在世界各 地的用户提供更有效的访问．介绍 Subversion 的每一种部署方式不仅不可能，也超出了本书的范围，我们只希望读者能根据本章的内容以及读者能找到的其他 参考资料，评估自己的部署方式．

仓库的访问控制

Subversion 的访问控制几乎全由 Subversion 服务器进程进行管理，可 供选择的 Subversion 服务器在 第 6 章 服务器配置 介绍，在 “基于路径的授权” 一节 详细介绍基于路径 的访问控制．除了用户级的访问控制外，用户可能还希望仓库能被主机上的程序 访问．操作系统层面的用户与用户组权限对仓库也有影响．第 6 章 服务器配置 的内容将帮助读者解决与访问权限有关的 问题．

创建与配置仓库

在本章的开头部分（“仓库部署策略” 一节），我们 看到了几个在创建与配置仓库之前需要确定的几个问题．现在，我们终于要真枪 实弹地开始干了！本节读者将看到如何创建一个 Subversion 仓库，并对它进行 配置，使得当特定的事件发生时，仓库将执行预定的操作．

创建仓库

创建 Subversion 仓库是一件非常简单的工作，用到的命令是 *svnadmin create*.

```
$ # Create a repository
$ svnadmin create /var/svn/repos
$
```

假设父目录 */var/svn* 已存在，并且管理员对父 目录拥有写权限，上面的命令在 */var/svn/repos* 创建了一个新的仓库，使用的是默认的后端存储类型 (FSFS). 你还可以利用 选项 `--fs-type` 显式地指定后端存储类型，该选项接受 的参数是 `fsfs` 或 `bdb`.

```
$ # Create an FSFS-backed repository
$ svnadmin create --fs-type fsfs /var/svn/repos
$

# Create a legacy Berkeley-DB-backed repository
$ svnadmin create --fs-type bdb /var/svn/repos
$
```

执行完这个简单的命令，你就拥有了一个新的 Subversion 仓库. 取决于 用户的访问方式，管理员可能还需要调整仓库目录的文件系统权限. 与系统 管理有关的基础知识不在本书的讨论范围之内，所以这方面的内容就当作训练 留给读者.

传递给 *svnadmin* 的路径参数只是一个普通的文件 系统路径，而不是一个 URL（就像 *svn* 访问仓库时用 到的 URL 参数）. *svnadmin* 和 *svnlook* 都是服务器端的工具—它们只在存放着 仓库的主机上使用，用于查看或修改仓库的某些部分，实际上它们也无法 跨网络执行任务. Subversion 新手的一个常见错误是试图为这两个工具传递 URL（即使是 `file://` 也不行）.

存放在仓库子目录 *db/* 内的就是版本化文件系统的 实现. 新仓库的版本化文件系统的生命开始于版本号 0，根据定义版本号 0 不 包含任何数据，只有最顶层的根目录. 初始时，版本号 0 也有一个版本号属性，`svn:date`，属性的值是创建仓库的日期.

仓库既然创建好了，接下来就可以对它进行改造.

虽然仓库的一部分—例如配置文件和钩子脚本—需要手工 查看和修改，但管理员不应该（也不需要）手工修改仓库的其他部分. *svnadmin* 工具应该足以应付仓库的任意修改，或者管理 员也可以用第三方工具对仓库进行调整. 不要 为了篡改 版本控制历史而手工修改仓库里的数据文件！

实现仓库钩子

钩子 (*hook*) 是 一个由某些仓库事件触发的程序，仓库事件包括但不限于新版本号的产生或 是某些非版本化属性被修改了. 有些钩子（称为 “前置钩子”）在仓库的操作执行之前运行，从而实现提前报告即将发生的事情，或阻止即将 发生的事情. 还有些钩子（称为 “后置钩子”）在仓库事件完成 之后运行，可以用来执行一些检查—但不修改—仓库的任务. 每一个钩子都能得到足够的信息，包括发生了什么事件，被修改的仓库，以及触发 事件的用户.

默认情况下，子目录 *hooks* 包含了各种 钩子的模板：

```
$ 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
$
```

Subversion 仓库支持的每个钩子都有一个模板，通过查看这些模板脚本的 内容，用户可以看到每个脚本所执行的操作，以及传递给脚本的参数. 模 板展示了用户可能会用钩子完成哪些操作，借助 Subversion 工具的配合，钩子可以完成一些常见的操作. 为了安装一个能实际工作的钩子，用户只 需要把可执行程序或脚本放到仓库目录的 *hooks* 子目录里，按照钩子的要求对文件进行命名后，Subversion 就可以根据文件名（例如 *start-commit* 或 *post-commit*）决定在什么时候执行它.

如果存放仓库的主机是 Unix 操作系统，这就意味着用户必须提供一个脚 本或程序（可以是 shell 脚本，Python 程序，已编译的二进制程序，或其他 形式的可执行文件），文件的名字必须严格按照钩子的规定进行命名. 当然，子目录 *hooks* 里的模板不仅仅是为了向用户展示钩子的 典型内容—在 Unix 系统中，安装钩子最简单的方法是复制一个适当的 模板文件，复制出的新文件要删掉扩展名 *.tmpl*，然后 对脚本的内容进行必要的修改，并确保脚本具有可执行权限. 然而，Windows 操作系统是根据文件的扩展名判断文件是否是可执行的，因此管理员所提供的 钩子，其文件名（不包括扩展名部分）是钩子的名字，而扩展名是 Windows 规定的可作为可执行程序的几种扩展名之一，例如二进制可执行文件用 *.exe*，批处理文件用 *.bat*.

Subversion 执行钩子时的用户身份， 和访问仓库的进程的所有者的身份 是相同的， 在大多数情况下， 仓库经由一个 Subversion 服务器进程进行访问，因此执行钩子的用户身份和 Subversion 服务器进程的所有者是一致的. 为 了能让这个用户执行钩子，操作系统必须允许该用户执行钩子脚本，这还意味 着将被钩子直接或间接访问到的程序或文件（包括 Subversion 仓库里的文件），都是以该用户的身份进行访问. 总之，要注意与权限有关的问题可能会导致 钩子无法正常工作.

Subversion 目前支持多种钩子，详细的信息见 Subversion 仓库钩子参考手册. 作为一个仓库管理员，你需要决定仓库 应该实现哪些钩子（通过提供具有适当名字与权限的钩子文件），以及如何实现. 在做这个决定时，始终在心里面记着仓库的部署方式，比如说如果你是通过服务 器配置来决定哪些用户可以向仓库提交修改，那就没必要再在钩子里实现相同的 访问权限控制.

钩子脚本环境配置

默认情况下，Subversion 在空环境中执行钩子脚本，空环境指的是没有 设置任何环境变量的运行环境，甚至连 `$PATH`（在 Windows 系统中则是 `%PATH%`）都没有设置. 正是因为 这个原因，很多管理员都曾遇到过这种问题：自己手工执行钩子程序是没问题 的，但由 Subversion 执行时却无法正常工作. 传统的解决办法是在钩子脚本 中手工设置钩子所需的所有环境变量.

Subversion 1.8 为钩子脚本的运行环境管理引入了一种新方法— 钩子脚本环境配置文件. 如果 Subversion 服务器进程在仓库的子目录 *conf/* 内找到了一个名为 *hooks-env* 的文件，它就把该文件当成 INI 格式的 配置文件进行解析，将解析到的选项名和值作为环境变量，添加到钩子脚本 的运行环境中.

hooks-env 的语法非常简单直观： 每一节的名字 就是钩子脚本的名字 （例如 `[pre-commit]` 和 `[post-revprop-change]`），节内的配置项被看成是环境 变量的名字到值的映射. 另外还有一个特殊的 `[default]` 节，它所配置的环境变量对所有的钩子脚本都起作用（除非又被各节自己的 设置显式地覆盖了）. *hooks-env* 配置文件的例子 如 例 5.1 “hooks-env（配置钩子脚本环境）” 所示.

例 5.1. hooks-env（配置钩子脚本环境）

```
# All scripts should use a UTF-8 locale and have our hook script
# utilities directory on the search path.

[default]
LANG = en_US.UTF-8
```

```
PATH = /usr/local/svn/tools:/usr/bin


# The post-commit and post-revprop-change scripts want to run
# programs from our custom synctools replication software suite, too.

[post-commit]
PATH = /usr/local/synctools-1.1/bin:%(PATH)s

[post-revprop-change]
PATH = /usr/local/synctools-1.1/bin:%(PATH)s
```

例 5.1 “hooks-env (配置钩子脚本环境)” 还展示了 Subversion 配置文件解析器灵活的字符串替换语法. 在这个例子里, 选项 `PATH` 的值—拉取自文件的 `[default]` 部分—会替换掉其他地方的占位 符文本 `%(PATH)s`. 关于这种语法的更多信息, 见 Subversion 运行时配置目录内的 *README.txt* 文件. (关于运行时配置目录的更多信息, 见 “运行时配置区域”一节.)

当然, 在每一个仓库的 *conf/* 目录内都放置 一份完全一样的钩子环境配置文件, 做起来可能会有点笨拙, 特别是当它 们都需要修改时. 为此, Subversion 允许管理员为配置信息指定一个可 选的位置 (可能是共享的).

钩子脚本的常见用法

仓库的钩子脚本可以提供非常多的用处, 但大多数都可以简单地归类成 以下几种: 通知, 验证和重做.

通知脚本用于通知用户发生了某些事情. 最常见的用法是把每次提交的 修改以邮件的形式发送给项目的每一位成员, 用到的钩子是 post-commit 和 (或) post-revprop-change. 除此之外还有很多其他形式的 通知, 例如整合了问题跟踪的脚本, 或者是充当 IRC 机器人的脚本, 当仓库 发生变化时向大家通报.

站在验证的角度, 人们经常使用钩子 start-commit 和 pre-commit, 根据不同的标准去禁止或允许提交, 常用的判断标准有提交的作者, 描述 提交的日志消息的格式和 (或) 内容, 甚至是所提交的修改的底层细节. 同样地, 钩子 pre-revprop-change 充当的是版本号属性修改的看门狗, 考虑到版本号属性不属于版本控制的范畴, pre-revprop-change 在保护 版本号属性免受破坏性修改这一点上非常重要.

Subversion 1.5 发布后, 有一类验证用得非常广泛, 那就是验证提交 修改的客户端软件. Subversion 从 1.5 开始支持合并跟踪 (关于合并跟踪 的详细介绍, 见 第 4 章 分支与合并), 管理员需要提供 一种方法, 以便确保一旦仓库的用户开始使用新的合并跟踪特性, 那么他们 *所有的* 合并都要被跟踪. 为了避免用户向仓库提交 未被跟踪的合并, 管理员使用钩子 start-commit 检查客户端公示的特性字 符串, 如果客户端没有宣称支持合并跟踪, 那么提交就会被拒绝, 从而迫使 用户升级他们的客户端软件! 例 5.2 “要求客户端必须支持合并跟踪的钩子 start-commit” 展示了如何用钩子 start-commit 实现这个功能.

例 5.2. 要求客户端必须支持合并跟踪的钩子 start-commit

```
#!/usr/bin/env python
import sys

# sys.argv[3] is a colon-delimited capabilities list
if 'mergeinfo' not in sys.argv[3].split(':'):
  sys.stderr.write("""\
```

```
ERROR: Commits to this repository must be made using Subversion
clients which support the merge tracking feature.  Please upgrade
your client to at least Subversion 1.5.0.
""")
  sys.exit(1)
```

从 Subversion 1.8 开始，向 Subversion 1.8 服务器提交修改的客户端，除了提供它自己的特性字符串外，还会通过 短暂事务属性 (*ephemeral transaction properties*) 提供关于 它自己的额外信息. 短暂事务属性本质上是版本号属性，在提交时由客户端将短暂事务属性设置到提交事务上，服务器端在事务成为最终的版本号 之前，删除该属性. 查看短暂事务属性的方法和查看设置在提交事务上的其 他非版本化属性的方法相同，需要用到的钩子是 start-commit 和（或）pre-commit.

下面是 Subversion 当前提供并已实现的短暂事务属性：

`svn:txn-client-compat-version`

携带了客户端支持的 Subversion 函数库版本字符串，这对于判 断客户端是否支持仓库数据处理所要求的最小特性集非常有用.

`svn:txn-user-agent`

携带了 “用户代理” (user agent) 字符串，它描述 了发起提交的客户端程序. Subversion 库函数定义了该字符串的起始部分的内容，但是使用了 Subversion API 的第三方程序（例如 GUI 客户端）可以向字符串附加自定义的信息.

大多数客户端是在提交过程的早期阶段传送短暂事务属性，从而可以 被钩子 start-commit 检查，但 Subversion 的某些配置会使得这些属性 直到提交过程的后期才会被设置上. 管理员应该考虑同时在钩子 start-commit 和 pre-commit 上执行基于短暂事务属性的检查工作. 利用钩子 start-commit 过滤掉无效的客户端，如果无法在 start-commit 完成检查工作，就在 pre-commit 完成.

前面已经说过，就在事务变成最终的版本号之前，短暂事务属性将会被 删除，因此有些管理员希望这些属性上的信息能够永久保留. 我们的建议是 在钩子 pre-commit 里，把属性上的值复制到新的属性上. 实际上，Subversion 发布的源代码所提供的脚本 *persist-ephemeral-txnprops.py*（在 *tools/hook-scripts/*）做的正是这件事.

钩子的第三种常见用途是重做. 如果管理员只是在做一个简单的备份，又或者是远程仓库镜像备份，钩子脚本都能起到非常重要的作用. 关于仓库 备份的更多内容，见 “仓库备份”一节 和 “仓库复制”一节.

搜索或自己编写钩子脚本

读者应该可以想到，从 Subversion 社区或其他地方都能找到大量可以 随意使用的钩子程序和脚本. 实际上，Subversion 发布的源代码就提供了 几个适用性很广泛的钩子脚本，脚本文件放在 *tools/hook-scripts/*. 然而，如果读者无法找到 满意的钩子脚本，就可能需要自己编写. 关于如何使用 Subversion 的公共 API 进行软件开发，见 第 8 章 嵌入 *Subversion*.

钩子脚本几乎可以做任何事情， 但作者应该对脚本的功能有所控制. 虽然说使用钩子脚本去自动纠正被提交的文件中的错误，缺陷或违反策略 的做法—是一件很有诱惑力的事情，但这样做会导致问题. Subversion 在客户端缓存了仓库的部分数据，如果钩子脚本按照这种方式 修改了提交事务，那么客户端缓存的数据就成了过时了的，且难以察觉，过时的缓存会产生不可预料的后果. 虽然通过钩子脚本添加新的提交事务属 性通常没什么问题，但除此之外，一个提交事务的其它信息都应该看成是只 读的. 管理员不应该为了优化事务的载荷而对事务进行修改，更好的做法是 在钩子 pre-commit 里 验证 事务，拒绝不满足要 求的事务. 作为回报，仓库的用户将逐渐养成良好的提交习惯.

FSFS 配置

从 Subversion 1.6 开始，FSFS 提供了几个配置参数，管理员可利用它 们对仓库的性能和磁盘使用进行调整. 管理员可以在仓库目录的 *db/fsfs.conf* 里找到所有的配置参数及其说明.

仓库维护

维护 Subversion 仓库可能会令人望而却步，大部分是由于后端存储所固 有的复杂度. 完成任务的关键在于如何充分地利用工具—它们是什么，什么时候使用它们，以及如何使用. 本节将会介绍如何使用 Subversion 提供的 管理工具完成常见的仓库维护工作，例如数据迁移，升级，备份和清理.

管理员工具箱

Subversion 提供了几个用于创建，检查，修改和修复仓库的工具，下面简 单介绍一下这些工具.

svnadmin

命令 *svnadmin* 是仓库管理员最优秀的助手. 除了可以 用来创建仓库外，它还提供了几种维护操作. *svnadmin* 的使用方法和其他的 Subversion 命令行工具非常类似:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type 'svnadmin help <subcommand>' for help on a specific subcommand.
Type 'svnadmin --version' to see the program version and FS modules.

Available subcommands:
   crashtest
   create
   deltify
…
```

在本章的前面 (“创建仓库”一节) 我们已经介绍了子命令 *svnadmin create*, 稍后就会介绍 *svnadmin* 的其他子命令. 关于 *svnadmin* 完整的子命令列表，以及它们的功能，见 svnadmin 参考手册—Subversion 仓库管理工具.

svnlook

svnlook 用于查看仓库的各个版本号和 事务 (*transactions*, 正在生成过程中的版本号)，不会向仓库执行写操作. *svnlook* 经常被钩子使用，用于报告仓库即将提交或刚刚提交的修改，用到的钩子分别是 pre-commit 和 post-commit. 管理员还可以用 *svnlook* 诊断问题.

svnlook 的语法非常直观:

```
$ 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.
…
```

svnlook 的大部分子命令都能根据一个给定的版本 号或事务，输出与前一个版本号的不同之外或事务本身的信息，指定版本号和 事务的选项分别是 `--revision (-r)` 和 `--transaction (-t)`. 如果没有 指定 `--revision (-r)` 和 `--transaction (-t)`, *svnlook* 将查看仓库最年轻的版本号（即 HEAD). 如果版本号 19 是仓库 */var/svn/repos* 目前最年轻的版本号，则下面的 2 个命令是等价的：

```
$ svnlook info /var/svn/repos
$ svnlook info /var/svn/repos -r 19
```

有一个例外是子命令 *svnlook youngest* 不接受 任何选项，只是打印出仓库最年轻的版本号：

```
$ svnlook youngest /var/svn/repos
19
$
```

始终记住，管理员只能浏览未提交的事务，但这种事务在大多数仓库中 都不存在，因为事务要么是已提交了的（此时应该使用选项 `--revision (-r)`），要么被中止并 删除.

svnlook 的输出既能被人类理解，也能被程序解析. 例如，假设 *svnlook info* 的输出是：

```
$ svnlook info /var/svn/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
$
```

根据输出顺序，*svnlook info* 的输出由以下几 部分构成：

1. 作者，然后是换行符
2. 日期，然后是换行符
3. 日志消息的字符个数，然后是换行符
4. 日志消息本身的内容，然后是换行符

打印的内容是人类可读的，例如对于日期来说，它是以文本形式表示，而不是某种很费解的形式（例如把日期表示成距离某个特殊日期的 纳秒数）. 打印的内容同时也能被程序解析，因为日志消息可能有很 多行，在长度上也没有限制，所以 *svnlook* 在打印日 志消息本身的内容之前，先打印消息的长度，这就允许处理 *svnlook* 输出的程序针对日志消息做出更明智的决策，例如为日志消息分配多少内存，或者是应该跳过多少字节才能完全跳过日志 消息.

svnlook 还能执行很多查询：只显示上面提到的 信息的部分内容，递归地列出被版本控制的目录树，报告哪些路径在给定 的版本号或事务中被修改了，显示文件和目录的内容和属性的变化，等等. *svnlook* 完整的参考手册，见 svnlook 参考手册—Subversion 仓库检查工具.

svndumpfilter

虽然 *svndumpfilter* 不是管理员最常使用的工具 之一，但它却提供了一种非常特别且有用的功能—它可以作为一种基 于路径的过滤器，快速地修改 Subversion 仓库历史的数据流.

svndumpfilter 的使用语法是:

```
$ 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)
```

svndumpfilter 包含了 2 个子命令: *svndumpfilter exclude* 和 *svndumpfilter include*, 它们允许管理员从历史数据流中排除或保留指定的 路径. 在本章的 “过滤仓库历史”一节 读者将会学习到子命令的用法，以及它们的典型应用.

svnrdump

简单点说，*svnrdump* 本质上就是把 *svnadmin dump* 和 *svnadmin load* 与网络有关的部分单独拿出来，形成一个单独的程序.

```
$ svnrdump help
general usage: svnrdump SUBCOMMAND URL [-r LOWER[:UPPER]]
Type 'svnrdump help <subcommand>' for help on a specific subcommand.
Type 'svnrdump --version' to see the program version and RA modules.

Available subcommands:
   dump
   load
   help (?, h)

$
```

我们将在 “迁移仓库数据”一节 介绍 *svnrdump* 和上面提到的 2 个 *svnadmin* 子命令.

svnsync

svnsync 提供了维护 Subversion 仓库只读镜像 所需的全部功能. 该命令只做一件事—把一个仓库的历史传送到另一 个仓库，完成这项工作的方式有很多种，但它最方便的地方在于操作可以 远程执行—“源” 和 “同步” 仓库，以及 *svnsync* 程序都可以在不同的主机上.

svnsync 的语法和本章介绍的其他命令非常 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
   info
   help (?, h)
$
```

关于如何使用 *svnsync* 复制仓库的更多内容，我们 将在 “仓库复制”一节 介绍.

fsfs-reshard.py

虽然脚本 *fsfs-reshard.py*（放在 Subversion 源代码目录的 *tools/server-side* 子目录内）不是 Subversion 工具集的正式成员，但是如果仓库使用 FSFS 作为后端存储，那么管理员就可以用 *fsfs-reshard.py* 优化性能. 基于 FSFS 的仓库把版本号的相关信息记录在单独的文件里，有时候这些 文件会全部存放在一个目录里，有时候分散在多个目录里.

FSFS 的早期版本把所有的版本号文件—每一个版本号都有对应的 一个文件—放在一个目录里， 在仓库的整个生命周期内，目录内的文件 数量都在增长. 有些系统对同一目录内的文件数量限制较大，即使是限制较 宽或没有限制的系统，如果目录内的文件数量较多也会产生严重的性能问题.

从 Subversion 1.5 开始，基于 FSFS 的仓库在布局上稍有变化：存放 版本号文件的目录（和其他不断增长的目录）是 *碎化地* (*sharded*)，或 者说版本号文件分散在多个目录内. 这可以大幅减少系统定位文件的时间，从而提高仓库的整体读取性能.

在同一个目录内可以存放的文件数量是可配置的（虽然默认值对于大多 数平台而言都是比较合理的），但是如果在仓库使用一段时间后，再去修改 这个配置，将导致 Subversion 无法定位当前正在搜索的文件. 这时候 *fsfs-reshard.py* 就派上的用场.

fsfs-reshard.py 重新编排仓库的目录结构，以满 足需要被碎化的子目录的个数要求，并更新仓库的配置，以便修改能够持久化地保留下来. *fsfs-reshard.py* 和 *svnadmin upgrade* 配合使用时，对于把 1.5 版之前 的仓库升级成最新的文件系统格式和碎化文件非常有用 (Subversion 不会 自动地对文件进行碎化). *fsfs-reshard.py* 还能 继续优化已经碎化过的仓库.

修正提交日志消息

有时候，用户可能会写错提交日志消息（例如拼写错误，或包含了错误的 信息）. 如果仓库的配置允许提交结束后仍然可以对日志消息进行修改（使用 钩子 pre-revprop-change，见 “实现仓库钩子”一节），用户就可以用命令 *svn propset*（见 svn 参考手册—Subversion 命令行客户端 的 svn propset (pset, ps)）“修正” 包含错误的日志消息. 然而，这种做法可能会导致信息永久丢失，因此 Subversion 仓库的默认配置是禁止修改未版本化的属性— 只有管理员除外.

如果管理员需要修改日志消息，将会用到的命令是 *svnadmin setlog*. 命令从给定的文件中读取新的日志 消息，覆盖掉指定的版本号的日志.

```
$ echo "Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

默认情况下，*svnadmin setlog* 受到的限制与企图 修改未版本化属性的客户端受到的限制是一样的—钩子 pre-revprop-change 和 post-revprop-change 仍然会被触发，因此命令若想 成功执行还要求仓库进行相应的配置. 但管理员可以通过向 *svnadmin setlog* 添加选项 `--bypass-hooks`，绕开这条限制.

记住，如果选择绕开钩子，很可能会导致属性发生变化时，不发送通知 邮件，备份系统不对未版本化的属性进行备份，以及诸如此类的事情．管理 员必须非常清楚自己在做什么．

管理磁盘空间

虽然最近几年，存储的成本已经下降了很多，但是如果是对大量的数据进行 版本控制，那么管理员仍然需要关注磁盘的使用情况．在一个活跃的仓库中，每增加一条版本历史，就需要把历史信息备份到其他地方，有可能还会为了 保险起见而多次备份．为了降低备份的数据量和磁盘消耗，知道仓库的哪些 数据需要保持在线，哪些数据需要备份，哪些数据可以删除—就显得非常 重要．

Subversion 如何节约磁盘空间

为了尽量降低仓库的存储空间消耗，Subversion 使用了 *增量存储技术* (*deltification*)．增量存储技术通过一块数据及 相对于它的一系列差异来表示另一块数据，如果两块数据的差异非常小，增量存储技术就可以仅保存其中一组数据以及两组数据之间的差异，而不 需要同时保存两组数据，从而节省存储空间．采用增量存储技术的结果是 原本体积庞大的文件，其所消耗的存储空间与全文保存相比，只占很小的 一部分．

在一开始设计 Subversion 时，就已经包含了增量存储技术，后来也对其进 行不断地改进．从 1.4 开始，Subversion 将对全文表示的文件内容进行 压缩．从 1.6 开始，新特性 *表示共享* (*representations sharing*) 为 Subversion 节 省了更多的空间．该特性允许内容相同的多个文件或多个版本号引用 到一个单一的共享数据实例上，而不是每个人都存一份自己的副本．

删除僵死的事务

虽然不太常见，但仍然存在提交失败的情况，此时便会在仓库中留下 残骸—一个未提交的事务和与之相关的文件或目录修改．造成提交 失败的原因很多，可能是客户端操作被用户粗暴地终止，也可能是在操作 执行中发生了网络错误．不管是什么原因，僵死事务总是有可能出现，除 了占用存储空间外，它们并不会产生实际的危害，但一个讲究的管理员的眼 睛里是揉不得沙子的．

管理员可以用命令 *svnadmin lstxns* 列出未完成 事务：

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

输出中的每一项都能用作 *svnlook* (添加选项 `--transaction (-r)`) 的参数，用来 判断是谁，在什么时候创建了这个事务，在这个事务中做了哪些类型的修改．这些信息有助于管理员判断该事务是否可以安全地删除．如果确实是要删除 一个事务，就把它的名字作为命令 *svnadmin rmtxns* 的参数．实际上，*svnadmin lstxns* 的输出可以直接 作为 *svnadmin rmtxns* 的输入！

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`
$
```

如果管理员打算以上面这种形式执行这两个命令，就得考虑暂时禁止 客户端对仓库的访问，避免在你清理期间有合法的事务产生．例 5.3 “txn-info.sh (打印未完成的事务)” 展示 了如何编写一个 shell 脚本来打印仓库中未完成的事务．

例 5.3. txn-info.sh (打印未完成的事务)

```
#!/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
```

脚本的输出基本上就是多个 *svnlook info* 输出 内容 (见 “svnlook”一节) 的拼接, 就像下面这样:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
39
Trying to commit over a faulty network.
---[ Transaction a45 ]-------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

一个被废弃了很久的事务通常表示一个失败或中止了的提交. 事务的时间 戳可以提供很有用的信息—比如说, 9 个月前开始的操作有没有可能 还是活跃的?

简言之, 决定清理事务不用太过深思熟虑, 很多信息—包括 Apache 的错误和访问日志, Subversion 的操作日志和版本号历史—都可以作为 决策的参考. 当然, 管理员也可以通过与事务的所有者沟通 (比如说通过 电子邮件) 来判断一个看似僵死的事务, 是否真得处于僵死状态.

FSFS 文件系统压缩

在基于 FSFS 的仓库中, 有些文件描述了在一个单独的版本号做了哪些 修改, 有些文件包含了与一个单独的的版本号相关的版本号属性. Subversion 1.5 之前创建的仓库, 把这两种文件分别放在两个目录里. 当仓库中有新的 版本号生成时,

Subversion 就向这两个目录存放更多的文件，随着时间的推 移，目录内的文件数量将增长到非常大的规模．人们已经发现，这会在某些 基于网络的文件系统上造成严重的性能问题．

第一个问题是操作系统必须在短时间内引用大量不同的文件，这会导致 磁盘缓存被快速消耗，于是操作系统会在读取磁盘上花费更多的时间，表 现在 Subversion 身上，就是在访问数据时性能低下．

第二个问题比较微妙．受到大多数文件系统的磁盘空间分配方式的影响，每一个文件实际消耗的存储空间会多于它要求的存储空间．为了管理一个文 件所消耗的额外存储空间在 2 KB 到 16 KB 之间，具体的大小取决于操作 系统的文件系统的实现．反映到以 FSFS 作为后端存储的仓库身上，就是每 一个版本号都会产生额外的存储空间消耗．对于含有大量小版本号的仓库 来说，最明显的影响就是为了存放版本号文件而消耗的磁盘空间，会迅速超 过数据的实际大小．

为了解决这 2 个问题，Subversion 1.6 增加了命令 *svnadmin pack*，它的作用是把一个完整的碎片内的所 有文件都打包到一个单一的文件内，然后再删除原来的文件，从而降低因文 件过多而导致的空间与性能开销．

只要是 1.6 版及以上的文件系统格式，都可以用 *svnadmin pack* 进行压缩（如果文件系统格式较旧，可以用 *svnadmin upgrade* 对仓库进行升级，见 svnadmin 参考手册—Subversion 仓库管理工具 的 svnadmin upgrade）．为了压缩文件系统，只需要 对仓库执行 *svnadmin pack*：

```
$ 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.
$
```

因为打包过程会事先获取所需要的锁，所以管理员可以在活动仓库上 执行这个操作，甚至可以作为钩子 post-commit 的一部分．压缩已经压缩过 的碎片是合法的操作，但不会对仓库的磁盘使用产生影响．

如果仓库以 BDB 作为后端存储，则 *svnadmin pack* 不会对仓库产生任何效果．

迁移仓库数据

Subversion 文件系统的数据分布在仓库的很多文件中，其存储方式通常 只有 Subversion 开发人员才能理解（或对它们感兴趣）．然而，在某些情况 下文件系统的全部或部分数据需要被复制或移动到其他仓库中．

Subversion 通过 *仓库转储流*（*repository dump streams*）实现这个功能．仓库 转储流（如果作为文件存储到磁盘上，习惯上称为 “转储文件”）是一种可移植的平面文件格式，描述了仓库中的各个版本号—是谁，在 什么时候，做了哪些修改等．这种转储流是仓库之间组织版本历史— 全部的或部分的，未修改或修改过的—的主要机制．Subversion 提供了 创建与加载转储流所必需的工具：命令 *svnadmin dump* 和 *svnadmin load*，以及 *svnrdump*．

虽然 Subversion 仓库的转储文件格式包含了人类可读的部分内容 和让人感到熟悉的结构（它的格式很像 RFC 822 格式，这是大多数邮件 所使用的格式），但它 *不是* 一个纯文本文件，而是二进制文件，即使是非常细微的修改也会非常敏感．比如说，很多 文本编辑器会自动转换行结束符，但这样做会损坏文件．

有很多情况都需要对 Subversion 仓库数据进行转储和加载．在 Subversion 的早期阶段，最常见的原因是 Subversion 的演变．随着 Subversion 的不断成 熟，可能会出现这样一种情况：后端数据库概要的变化会导致旧版本的仓库出现 兼容性问题，于

是管理员必须使用旧版的 Subversion 转储仓库数据，再把转 出的数据加载到新版 Subversion 创建的仓库中. 从 Subversion 1.0 开始，不会再出现这种需要转储和加载仓库数据的概要变化，并且 Subversion 开发 人员承诺在次版本之间升级时（例如从 1.3 到 1.4），不会强迫用户转储和加 载仓库. 但除了升级 Subversion 外，还有其他需要用到转储和加载的场景，例如重新部署 Berkeley DB 仓库到新的操作系统或 CPU 平台上，或者在 Berkeley DB 和 FSFS 两种后端存储之间切换，以及从仓库历史中清除被版本 控制的数据（在本章的 “过滤仓库历史”一节 介绍）.

Subversion 仓库的转储文件只是描述了版本历史，它不会携带任何与 未提交的事务，文件系统路径上的用户锁，仓库或服务器的配置（包括钩子 脚本）等有关的信息.

利用 Subversion 仓库的转储文件，管理员还可以实现在不同的后端存储 方式或版本控制系统之间转换. 因为转储文件的绝大部分内容是人类可读的， 用它来描述一般的修改集—修改集中的每一个修改都应当看作是一个新 的版本号—相对来说比较容易. 实际上，工具 *cvs2svn*（见 “把 CVS 仓库转换成 Subversion 仓库”一节）可以通过转储文件，把 CVS 仓库的内容复制到 Subversion 仓库中.

现在，我们只关心如何在 Subversion 仓库之间迁移数据，具体的内容 将在接下来的一节里进行介绍.

使用 svnadmin 迁移仓库数据

无论迁移仓库历史是出于什么样的原因，*svnadmin dump* 和 *svnadmin load* 的用法都非常 简单直接. *svnadmin dump* 按照 Subversion 的文件 系统转储格式，输出一段范围内的版本号. 转储的结果会被打印到标准输出，而提示性的信息则会打印到标准错误，这就允许管理员把输出重定向到文件 的同时，在终端窗口中查看命令的状态输出，例如：

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
* Dumped revision 25.
* Dumped revision 26.
```

命令执行结束时，你将得到一个文件（在上面的例子里，文件名是 *dumpfile*），这个文件包含了在指定的版本号范围 内，存放在仓库中的所有数据. 因为 *svnadmin dump* 从仓库中读取版本号的过程和其他 “读者”（例如 *svn checkout*）读取仓库的过程是一样的，所以 可以在任意时刻，安全地执行 *svnadmin dump*.

与 *svnadmin dump* 配对的命令 *svnadmin load* 从标准输入读取 Subversion 仓库的转储文件，把文件中 的版本号重放到目标仓库中. 在命令的执行过程中仍然会输出提示性的信息，不过这次是打印到标准输出：

```
$ 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) >>>
```

加载的结果是有新的版本号被添加到仓库中—相当于从 Subversion 客户端向仓库提交. 和普通的提交一样, 可以利用钩子, 在加载产生的提交 之前或之后执行特定的操作, 向 *svnadmin load* 添加 选项 `--use-pre-commit-hook` 和 `--use-post-commit-hook`, 可以分别指示 Subversion 为每一个加载的版本号, 执行钩子 pre-commit 和 post-commit. 比如说 管理员可能会利用钩子, 以确保加载的版本号能够经历与普通提交一样的验证 过程. 当然, 在使用这两个选项时要小心—如果钩子 post-commit 会为 每一个新的版本号发送一封邮件, 你可能不希望在几分钟内收到几百上千封 邮件. 关于钩子的更多内容, 在 “实现仓库钩子”一节 介绍.

因为 *svnadmin* 在转储和加载过程中会用到标准 输出和标准输入, 感兴趣的读者可以试一下像下面这样执行命令 (在管道的两边甚至可以使用不同版本的 *svnadmin*):

```
$ svnadmin create newrepos
$ svnadmin dump oldrepos | svnadmin load newrepos
```

默认情况下, 转储文件会很大—比仓库要大得多, 这是因为每个 文件的每个版本在转储文件中都是全文本表示. 这是最快速和最简单的方式, 如果转储文件还会被其他程序 (例如压缩程序, 过滤程序等) 处理, 处理 起来也会非常方便. 但是如果管理员需要长时间保存转储文件, 向 *svnadmin* 添加选项 `--deltas` 可以 减小转储文件的大小, 从而节省存储空间. 添加选项 `--deltas` 后, 文件的连续版本号将会以压缩的二进制差 异格式输出—和仓库保存文件版本号的方式是一样的. 添加选项后命令 会执行得更慢一些, 但得到的转储文件大小会更接近仓库的大小.

我们前面已经提到 *svnadmin dump* 可以输出一段 范围内的版本号. 使用选项 `--revision (-r)` 指定一个单独的, 或一段范围内的版本号进行转 储. 如果省略该选项, 仓库内所有的版本号都会被转储.

```
$ svnadmin dump myrepos -r 23 > rev-23.dumpfile
$ svnadmin dump myrepos -r 100:200 > revs-100-200.dumpfile
```

Subversion 在转储新的版本号时, 输出的信息仅能满足加载过程根据 前面的版本号来重新创建新的版本号. 换句话说, 对于转储文件中给定的 任意一个版本号, 只有在该版本号中被修改了的项目, 才会出现在转储文件 里, 唯一的例外是被当前 *svnadmin dump* 转储的第一 个版本号.

默认情况下, Subversion 不会把第一个被转储的版本号表示成与前一 个版本号的差异. 第一个原因是对于第一个版本号来说, 它的前一个版本号 在转储文件中不存在, 第二个原因是 Subversion 无法预知加载转储文件的 仓库状态. 为了确保每次

执行 *svnadmin dump* 所产生 的输出是自我完备的，Subversion 在默认情况下将会完整地表示被转储的 第一个版本号，包括它的每一个目录，文件，和版本号的每一个属性.

然而，管理员可以修改这种默认行为. 如果在转储时添加了选项 `--incremental`, *svnadmin* 会把 转储的第一个版本号与前一个版本号作比较，跟转储范围中剩下的其他 版本号那样，只输出在版本号中被修改了的内容. 这样做的好处是管理员可以创建出几个较小的转储文件—而不是一整个大文件—它们被 加载时可以连续进行，就像下面这样：

```
$ svnadmin dump myrepos -r 0:1000 > dumpfile1
$ svnadmin dump myrepos -r 1001:2000 --incremental > dumpfile2
$ svnadmin dump myrepos -r 2001:3000 --incremental > dumpfile3
```

可以用下面的命令序列，把这几个转储文件加载到新的仓库中：

```
$ svnadmin load newrepos < dumpfile1
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

选项 `--incremental` 的另一个奇妙用法是向已有的 转储文件添加新的版本号. 比如说，管理员可能会利用钩子 post-commit, 将每次触发钩子的版本号转储到同一个转储文件中. 又或者是每晚都运行 一个脚本，把上一次脚本运行结束后，仓库中新增的版本号转储到同一个 文件中. 利用这种方式，*svnadmin dump* 就能实现仓库 的备份.

利用转储文件，还能把不同的几个仓库合并成一个仓库. 为 *svnadmin load* 添加选项 `--parent-dir`，就能为加载过程指定一个新的虚拟根目录，这就意味着如果你有三个仓库的转储文件—假设文件名分别是 *calc-dumpfile*, *cal-dumpfile*, *ss-dumpfile*—先创建一个将会用来加载所有 转储文件的仓库：

```
$ svnadmin create /var/svn/projects
$
```

然后，在仓库中为每一个转储文件创建一个对应的目录：

```
$ 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，把转储文件加载到各自对应的目录内：

```
$ 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
…
$
```

使用 svnrdump 迁移仓库数据

Subversion 1.7 引入了一个新工具，*svnrdump*. 它提供了较为特殊的功能，本质上就是 *svnadmin dump* 和 *svnadmin load* (见 “使用 svnadmin 迁移仓库数据”一节) 的跨网络 版本. *svnrdump dump* 从一个远程仓库转储数据，打印 到标准输出；

svnrdump load 从标准输入读取转储数据，加载到一个远程仓库上. *svnrdump* 可以像 *svnadmin dump* 那样生成增量转储，甚至可以只转储 仓库的某个子目录，*svnadmin* 却无法做到这一点.

svnadmin 与 *svnrdump* 之间 最关键的区别在于后者不需要直接访问仓库，*svnrdump* 使用与 Subversion 客户端相同的仓库访问 (Repository Access, 简称 RA) 协议完成操作 的远程执行，因此用户可能需要提供认证证书，除此之外，远程交互可能还会 受到 Subversion 服务器的授权限制.

svnrdump dump 要求远程服务器运行的是 Subversion 1.4 或更新的版本. 它能生成的转储流的唯一类型，就是添加 了选项 `--deltas` 后，*svnadmin dump* 所生成的转储流. 在典型的使用情况下这并没有什么 特殊之处，但是如果你想在生成的转储流上执行一些特定类型的转换，这 些转换可能会受到影响.

因为在提交完新的版本号后，版本号的属性将被修改，所以 *svnrdump load* 要求目标仓库通过钩子 pre-revprop-change，允许修改版本号属性，更多信息见 Subversion 仓库钩子参考手册 的 pre-revprop-change.

管理员可以一起使用 *svnadmin* 和 *svnrdump*，例如用 *svnrdump dump* 从远程仓库获取转储流，然后再把结果通过管道输送给 *svnadmin load*，把远程仓库历史复制到本地仓库，或者反之，把本地仓库 的历史复制到远程仓库.

如果使用 `file://` 形式的 URL，*svnrdump* 也能访问本地仓库，但仍然需要借助 Subversion 的仓库访问 (Repository Access, 简称 RA) 抽象层— 这种情况下使用 *svnadmin* 会比较划算.

过滤仓库历史

因为 Subversion 在存储版本历史时，大量地使用了二进制差异算法和 数据压缩 (在完全封闭的数据库系统中是可选的)，如果管理员觉得不是很 困难而手工地修改历史，这是非常不明智的做法，大家应该极力避免这样操作. 数据一旦存储到仓库中，Subversion 通常不会允许管理员轻易地删除数据. [6] 但总会出现需要修改仓库历史的情况，例如从历史中抹去与某个文件相关的所 有记录，这个文件出现在历史中只是个意外 (或者因为其他一些原因). [7] 又或者 是多个项目本来共享同一个仓库，现在你想把它们分别存放到自己独享的一个 仓库中. 为了完成这些任务，仓库数据需要向管理员提供更加容易管理和延展 的表示形式—Subversion 仓库转储格式.

我们已经在 “迁移仓库数据” 一节 说过，Subversion 的仓库转储格式是用户提交到仓库中的修改的表示形式，它是人类 可读懂的. 使用 *svnadmin dump* 或 *svnrdump dump* 获取转储数据，用 *svnadmin load* 或 *svnrdump load* 把转储数据加载到仓库中. 采用人类 可读的转储格式的最大好处是用户可以手工地查看和修改转储文件. 当然，坏 处是如果转储文件非常庞大 (例如包含了三年历史的转储数据)，手工地查看和 修改转储文件将会非常耗时.

所以我们需要 *svndumpfilter*. 它可以作为仓库 转储流的基于路径的过滤器，用户所要做的就是向它提供希望保留或删除的路 径列表，然后把仓库转储流以管道的方式输送给 *svndumpfilter*，最终得到的转储数据就只会包含用户 (显式地或隐式地) 希望保留的路径.

现在介绍一个使用 *svndumpfilter* 的实际例子. 本章 早些时候 (见 “规划仓库的组织方式” 一节)，我们讨论了如何规划仓库的布局—为每一个项目创建一个单独的仓库，或 者把所有的项目都放在一个仓库中，或其他布局. 但是随着工作的进行，

[6] 这不正是你使用版本控制系统的原因吗?

[7] 小心谨慎地从版本化的数据中删除某些数据实际上是允许的，“清除” 特性是 Subversion 必须提供的功能之一，也是 Subversion 开发人员想尽快实现的功能之一.

用户 可能会觉得仓库的布局需要做一些修改, 比较常见的修改是把原本放在同一个 仓库中的多个项目, 分别放到属于自己的一个单独的仓库中.

假设我们的仓库包含了三个项目: `calc`, `calendar` 和 `spreadsheet`, 它们在 仓库中的位置如下:

```
/
  calc/
    trunk/
    branches/
    tags/
  calendar/
    trunk/
    branches/
    tags/
  spreadsheet/
    trunk/
    branches/
    tags/
```

为了把三个项目分开存放到三个仓库, 首先转储整个仓库:

```
$ svnadmin dump /var/svn/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

然后, 用 *svndumpfilter* 过滤转储文件, 每次过滤 一个项目的顶层目录, 得到三个新的转储文件:

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile > ss-dumpfile
…
$
```

这时候, 你必须做出决定. 每一个转储文件都将创建一个有效的仓库, 但保留的路径与原仓库中的路径一模一样, 也就是说即使你想为项目 `calc` 创建一个单独的仓库, 根据转储文件创建出的 仓库也会有顶层目录 *calc* 存在. 如果想把目录 *trunk*, *tags* 和 *branches* 放在仓库的根目录下, 你可能希望能够 修改转储文件, 调整头部 `Node-path` 和 `Node-copyfrom-path`, 使得它们不再包含路径分量 *calc/*. 并且, 你可能还想删除创建目录 *calc* 的转储数据, 这部分的转储数据看起来就像下面 这样:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```

如果你已经决定手工地修改转储文件，以便删除顶层目录，一定要确保 你所用的编辑器不会自动地把行结束符转换成本地格式（例如把 `\r\n` 转换成 `\n`），以免文件的 内容与元数据不一致．否则的话，转储文件将变成一堆废纸．

剩下的工作就是创建三个新的仓库，然后分别加载对应的转储文件，并忽略 在转储流中发现的 UUID:

```
$ 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.
…
$
```

svndumpfilter 的两个子命令都支持用于决定如何 处理 “空” 版本号的选项．如果一个版本号只包含了被过滤掉的 路径的修改，就可以把这个空的版本号当成不需要的版本号．为了允许用户 决定如何处理这两种版本号，*svndumpfilter* 提供了以下选项：

`--drop-empty-revs`

不要生成空版本号—直接忽略它们．

`--renumber-revs`

如果空版本号被丢弃（通过选项 `--drop-empty-revs`），修改后面所有的版本号的号码，使得版本号号码是连续的．

`--preserve-revprops`

如果空版本号未被丢弃，则保留它们的版本号属性（日志消息，作者，日期，和其他自定义的属性）．如果未指定该选项，则该版本号 将只会包含原始的提交日期和一条生成的日志消息，该消息指出版本号 是被 *svndumpfilter* 清空的．

虽然 *svndumpfilter* 非常实用，而且可以节省 大量的时间，但它仍然有几点需要特别注意的地方．首先，命令对路径语义 非常敏感．要特别注意转储文件中的路径是否以斜杠开始，即头部 `Node-path` 和 `Node-copyfrom-path`．

```
…
Node-path: spreadsheet/Makefile
```

...

如果路径以斜杠开始（也就是说路径含有前导斜杠），那么 *svndumpfilter include* 和 *svndumpfilter exclude* 的路径参数也应该以斜杠 开始（反之亦然）. 更进一步，如果转储文件对前导斜杠的使用不太一致，[8] 那你应该对路径进行规范化处理，使得它们全部 都有（或都没有）前导斜杠.

另外，被复制的路径也可能会带来一些麻烦. Subversion 支持在仓库中 执行复制操作—通过复制已存在的路径来创建新的路径. 在仓库的生命 周期中，有可能出现这种时刻：你从一个被 *svndumpfilter* 排除的位置复制了一个文件或目录，放到另一个被 *svndumpfilter* 包含的位置上. 为了保证转储数据是自我 满足的，*svndumpfilter* 仍然需要显示新路径的添加 —包含了通过复制而创建的文件的所有内容—但并不把新路径的添加 表示成某个源路径的复制， 因为这个源路径在已过滤的转储数据中并不存在. 但是由于 Subversion 的转储格式只会显示每个版本号中发生变化的内容，因此被复制的数据源可能不是现成的. 如果管理员觉得在仓库中存在这种 类型的复制，那就要重新考虑被包含或排除的路径，或许应该包含在复制操作中充当数据源的路径.

最后，*svndumpfilter* 在过滤路径时采取的是非常 字面的理解. 如果你试图复制一个根目录为 *trunk/my-project* 的仓库的历史，到它自己的仓库中，那你就要用 *svndumpfilter include* 保留 *trunk/my-project* 内的所有修改，但是生成的转储 文件对于将来加载它的仓库没有任何假定. 特别地，转储文件可能以添加目录 *trunk/my-project* 的版本号作为开始，但它 不会 包含创建目录 *trunk* 的 版本号（因为 *trunk* 不匹配包含过滤器）. 管理员需要 确保在加载转储文件前，转储文件期望存在的目录在目标仓库中确实存在.

仓库复制

有时候，如果仓库的历史能和另一个仓库保持一模一样，那么在完成很 多事情时将会变得非常方便. 比如最明显的一个就是当主仓库无法访问时（可能是系统硬件故障，网络故障等），把服务切换到备份仓库. 其他场景 还包括利用镜像仓库，把访问负载分散到多台服务器上，实现软升级等.

Subversion 提供了 *svnsync* 实现仓库的复制. *svnsync* 的工作本质上就是要求 Subversion 服务 “重放” 版本号，每次一个，然后利用这个版本号的相关信息，在另一个仓库中模拟一个相同的提交. 仓库所在的主机和执行 *svnsync* 的主机不必是同一台—如果命令的参数 是仓库的 URL，*svnsync* 将通过 Subversion 的仓库访问 (Repository Access, RA) 接口完成工作. *svnsync* 所要求的就是源仓库的读权限和目标仓库的读写权限.

svnsync 要求远程的源仓库的 Subversion 版本 必须至少是 1.4.

使用 svnsync 复制仓库

假设你已经有了一个源仓库，现在想为它创建一个镜像，下一件需要准备 的东西就是充当镜像的目标仓库. 这个目标仓库可以使用与源仓库不同的 后端存储机制（见 文件系统）—Subversion 的抽象层保证了具体的后端存储不会对复制操作产生影响. 但是在默认情况下，目标仓库此时不能包含任何版本历史（我们将在 本节的后面介绍一种例外情况）.

svnsync 用于交流版本号信息的协议对源仓库与 目标仓库不匹配的版本历史非常敏感，因此，虽然 *svnsync* 并没有 要求 目标 仓库是只读的，[9]除了 *svnsync*，如果还允许其他进程或 用户修改目标仓库的历史，常常会导致灾难性的后果.

[8] *svnadmin dump* 对前导斜杠的处理 策略总是一致的（不包含前导斜杠），生成转储数据的其他程序就不一定 了.

[9] 实际上，目标仓库不能是完全只读的，否则 的话，*svnsync* 就不能有效地复制历史.

不要用除了 *svnsync* 之外的其他方法修改镜像 仓库，使得镜像仓库偏离源仓库的历史．发生在镜像仓库的提交和版本号 属性修改只能由 *svnsync* 完成．

对目标仓库的另一项要求是允许 *svnsync* 修改版 本号属性．因为 *svnsync* 仍然受到目标仓库的钩子的 影响，而仓库的默认配置还不全面（见 Subversion 仓库钩子参考手册 的 pre-revprop-change）．管理员需要 显式地实现钩子 pre-revprop-change，在钩子脚本中允许设置和修改版本号 属性．如果这些条件都满足了，那么创建镜像前的工作就已经准备就绪了．

一种很好的做法是实现一种授权方式，使得除了执行复制操作的 进程外，不允许任何其他用户或程序修改镜像仓库．

现在我们将介绍一个使用 *svnsync* 的典型场景，如果读者觉得这里介绍的情况与自己的环境不太符合，大可不必理会我们的例子．

我们将会对存放本书源代码的 Subversion 仓库创建镜像仓库，并把镜 像仓库发布到 Internet 上，镜像仓库将托管在与源仓库不同的主机上．本书的源代码仓库允许匿名只读访问，但修改仓库需要验证身份．（这里先 不介绍如何配置 Subversion 服务器，这是 第 6 章 服务器配置 的主题．）为了使例子更加有趣，我们 将在第三台主机上发起复制操作—例如笔者当前正在使用的主机．

首先，创建用作镜像的仓库．这一步及后面的两个步骤都要求托管镜像 仓库的主机提供 shell 访问权限，一旦仓库配置完成，就不用再直接访问 主机．

```
$ ssh admin@svn.example.com "svnadmin create /var/svn/svn-mirror"
admin@svn.example.com's password: ********
$
```

现在，我们已经有了一个自己的仓库，得益于服务器的配置，我们可以 在 Internet 上访问到仓库．因为我们不希望除了复制之外的其他过程修改 仓库，所以需要把除了复制之外的其他修改操作区分出来．为了实现这个 目标，为复制过程使用一个专用的用户名，只有使用用户名 `syncuser` 提交的修改才会被允许．

我们将使用 Subversion 的钩子系统保证复制过程可以完成它需要 完成的操作，并且只有它能够做这些事情．为此我们需要实现两个钩子 —pre-revprop-change 和 start-commit．我们的 pre-revprop-change 钩子脚本的内容见 例 5.4 “镜像仓库的 pre-revprop-change 钩子脚本”，基本思路是如果试图修改属性的用户是 `syncuser`，则 允许；否则禁止修改属性．

例 5.4. 镜像仓库的 pre-revprop-change 钩子脚本

```
#!/bin/sh

USER="$3"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may change revision properties" >&2
exit 1
```

上面的脚本针对的是版本号属性的修改．现在考虑如何做到只允许用户 `syncuser` 向仓库提交新的版本号，这需要用到 start-commit 钩子脚本，如 例 5.5 “镜像仓库的 start-commit 钩子脚本” 所示．

例 5.5. 镜像仓库的 start-commit 钩子脚本

```
#!/bin/sh

USER="$2"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may commit new revisions" >&2
exit 1
```

安装了我们的钩子脚本后，并确保 Subversion 服务器对脚本具有可 执行权限，镜像仓库这边的设置就算结束了，现在开始真正地创建镜像.

使用 *svnsync* 做的第一件事就是告诉目标仓库，它是源仓库的镜像，这会用到子命令 *svnsync initialize*，目标仓库与源仓库的 URL 将作为命令的参数. Subversion 1.4 要求镜像必须针对整个仓库，从 Subversion 1.5 开始，允许只对仓库的子目录创建镜像.

```
$ 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 \
                     https://svn.code.sf.net/p/svnbook/source \
                     --sync-username syncuser --sync-password syncpass
Copied properties for revision 0 (svn:sync-* properties skipped).
NOTE: Normalized svn:* properties to LF line endings (1 rev-props, 0 node-props).
$
```

目标仓库现在已经记住它是本书源码仓库的镜像. 需要注意的是我们向 *svnsync* 提供了用户名和密码—这是镜像仓库 的 pre-revprop-change 钩子所要求的.

在 Subversion 1.4, *svnsync* 选项 `--username` 和 `--password` 的值 同时用于源仓库和目标仓库的身份验证，如果用户在两个仓库中的证书 不是完全一样，就会造成问题，尤其是命令在非交互模式下运行时（通过添加选项 `--non-interactive`), Subversion 1.5 通过引入两对新的选项解决了这个问题. 选项 `--source-username` 和 `--source-username` 用于指定源仓库的身份验证证书；选项 `--sync-username` 和 `--sync-password` 用于指定目标仓库的身份验证证书.（为了兼容旧版本，仍然保留旧选项 `--username` 和 `--password`, 但我们建议使用新选项.）

现在到了最有趣的部分. 只要一个命令，就能要求 *svnsync* 把源仓库中未复制的版本号，复制到目标 仓库.[10] 子命令 *svnsync synchronize* 查看 存放在目标仓库中的特殊的版本号属性，从而确定源仓库的镜像进度— 在这个例子里，最近一次创建镜像的版本号是 r0. 然后 *svnsync* 查询源仓库，确定源仓库最新的版本号是多少，最后请求源仓库的服务器开始重放从 0 到最新版本号之间的所有版本号. 随着 *svnsync* 不断地从源仓库服务器获取到结果，它开始把版本号作为新提交，转发到目标仓库服务器上.

[10] 虽然阅读这段文字和例子只需要几秒钟的时间，但 *svnsync* 实际消耗的时间比这长得多.

```
$ svnsync help synchronize
synchronize (sync): usage: svnsync synchronize DEST_URL [SOURCE_URL]

Transfer all pending revisions to the destination from the source
with which it was initialized.
…
$ svnsync synchronize http://svn.example.com/svn-mirror \
                      https://svn.code.sf.net/p/svnbook/source
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Transmitting file data .
Committed revision 3.
Copied properties for revision 3.
…
Transmitting file data .
Committed revision 4063.
Copied properties for revision 4063.
Transmitting file data .
Committed revision 4064.
Copied properties for revision 4064.
Transmitting file data ....
Committed revision 4065.
Copied properties for revision 4065.
$
```

其中比较有趣的地方是，对于每一个需要创建镜像的版本号，首先向 目标仓库提交该版本号，然后再修改版本号属性．之所以需要这种两步骤 的复制是因为提交操作由用户 `syncuser` 完成，版本 号的时间戳是创建时的时间，*svnsync* 接下来通过 一系列的属性修改操作，使得源仓库的版本号属性与目标仓库的版本号属性 保持一致，其中就包括修改版本号的作者和时间戳．

还值得注意的一点是 *svnsync* 会记住工作的当 前进度，从而允许操作被中断，并在以后的某个时间人上次中断的地方 重新开始．如果在命令执行的过程中，网络临时断开，只要再次执行 *svnsync synchronize* 即可．实际上，如果有新的 版本号出现在源仓库中，你所需要做的也就是执行 *svnsync synchronize* 而已．

作为记账操作的一部分，*svnsync* 在镜像仓库 记录了源仓库的 URL．正是因为这个原因，在初始化后执行的 *svnsync* 命令不会 要求 在命令行上必须提供源仓库的 URL．但是为了安全起见，我们建议用户继 续在命令行上提供源仓库的 URL 参数．取决定于具体的部署方式，在镜像仓库检索源仓库信息，或推送版本化数据时，任凭 *svnsync* 信任源仓库 URL 可能不太安全．

svnsync 记账

svnsync 必须有能力修改镜像仓库的版本号 属性，因为这些属性是镜像数据的一部分．如果版本号属性在源仓库 被修改了，这些修改也要反应到镜像仓库中．但是 *svnsync* 还会用到一套定制的版本号属性集合— 存放在镜像仓库的版本号 0 上—用作内部的记账操作．这些属性 包含的信息包括源仓库的 URL 和 UUID，还有一些状态跟踪信息．

状态跟踪信息中，有一个标志信息用来指出 “目前有同步操作 正在进行中”，这是为了避免多个 *svnsync* 往同一个目标仓库复制数据时，产生冲突．一般来说，用户不用关心这些 属性（这些属性全都以 `svn:sync-` 开始），但在少数 情况下，如果同步过程意外失败，Subversion 可能来不及删除这个状态 标志，这将导致后面的同步操作失败，因为未删除的标志表明当前同步操作 正在进行中（但实际上并没有）．幸运的是，从这种错误状态中恢复非常 容易，在 Subversion 1.7，你可以通过为 *svnsync* 添加新增的选项 `--steal-lock`；如果 1.7 之前的版本，只需要删除镜像仓库版本号 0 上的属性 `svn:sync-lock`：

```
$ svn propdel --revprop -r0 svn:sync-lock http://svn.example.com/svn-mirror
property 'svn:sync-lock' deleted from repository revision 0
$
```

另外，*svnsync* 还会把源仓库的 URL 存放到 镜像仓库的记账属性中．如果在命令行中省略了源仓库的 URL，后面的 同步操作就使用镜像仓库的特殊属性 `svn:sync-from-url` 的值，作为源仓库的 URL．同步过程按照字面意义理解并使用属性值，所以要小心使用非完全限定的 域名（例如用 `svnbook` 代替 `svnbook.red-bean.com`，这种域名只有在主机直接 与 `svnbook.red-bean.com` 的网络相连时，才能 正常工作）；无法解析的域名；网络环境不同，解析也不同的域名；以及 IP 地址（有可能发生变化）．如果用户需要把一个已存在的镜像仓库 映射到另一个不同的 URL（但仍然是同一个源仓库），可以修改存放该 信息的记账属性．Subversion 1.7 及其新版可以使用 *svnsync init --allow-non-empty* 重新初始化镜像 仓库的源仓库 URL：

```
$ svnsync initialize --allow-non-empty http://svn.example.com/svn-mirror \
                                       NEW-SOURCE-URL
Copied properties for revision 4065.
$
```

如果 Subversion 版本较旧，管理员需要手工调整记账属性 `svn:sync-from-url` 的值：

```
$ 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
$
```

关于这些记账属性的另一件有趣的事情是：如果 *svnsync* 在源仓库也发现了这些记账用的属性，那么 *svnsync* 不会对这些属性进行镜像操作．原因是显然的，归结到底，其实是因为 *svnsync* 无法区分从从源仓库复制来的属性，和为了满足自己记账的需要而 维护的属性．如果用户在维护一个仓库的镜像的镜像，就会出现这种情况，当 *svnsync* 在源仓库版本号 0 上看到它自己的记账 属性时，就会直接忽略它们．

Subversion 1.6 添加了新的子命令 *svnsync info*，用来显示目标仓库的记账属性．

```
$ svnsync help info
info: usage: svnsync info DEST_URL

Print information about the synchronization destination repository
located at DEST_URL.
…
$ svnsync info http://svn.example.com/svn-mirror
Source URL: https://svn.code.sf.net/p/svnbook/source
Source Repository UUID: 931749d0-5854-0410-9456-f14be4d6b398
Last Merged Revision: 4065
$
```

然而，*svnsync* 无法做到面面俱到．因为用户可 以在任意时刻修改版本号属性，而且这些修改不会留下任何历史信息，所以 复制操作要特别注意这种情况．假设你已经复制了仓库的前 15 个版本号，如果有人修改了版本号 12 的版本号属性，则 *svnsync* 不会知道这件事，更不会回过头来重新复制版本号 12．你必须使用 子命令 *svnsync copy-revprops*（或者还需要一些额外 工具的辅助），把特定的或指定范围内的版本号的所有属性都复制过来．

```
$ svnsync help copy-revprops
copy-revprops: usage:

    1. svnsync copy-revprops DEST_URL [SOURCE_URL]
    2. svnsync copy-revprops DEST_URL REV[:REV2]

…
$ svnsync copy-revprops http://svn.example.com/svn-mirror 12
Copied properties for revision 12.
$
```

为了使用 *svnsync* 复制仓库，你可能想设置 一个自动化过程．比如说，我们展示的例子的模式是 “抓取与推送”，你可能觉得更方便的做法是在钩子 post-commit 和 post-revprop-change 完成版本号到镜像仓库的推送，这种 做法有助于镜像仓库时刻保持最新的状态．

使用 svnsync 进行部分复制

svnsync 不仅限于复制仓库的全部内容，它也能 进行部分复制．一个常见的例子是，如果用户只对仓库的部分内容具有读取权限，那么 *svnsync* 也能优雅地对仓库进行镜像，但是它只会复制它能读取到的仓库内容．显然，这种镜像并不能作为有效的仓库备份策略．

从 Subversion 1.5 开始，*svnsync* 支持对仓库 的子目录进行复制．在复制时，除了用把 *svnsync init* 的 URL 参数指定成待复制的仓库子目录的 URL 外，其余步骤和复制整个 仓库是完全一样的．现在同步过程就只会复制源仓库的子目录内的版本号，但有些限制条件需要注意．首先，*svnsync* 不支持把 源仓库内的多个不连贯的子目录复制到一个镜像仓库中—此时正确的 做法应该是复制它们的公共父目录．然后，因为过滤操作是完全基于路径的，所以说如果被复制的子目录曾经被重命名过，则镜像仓库只会包含在指定的 URL 中出现的版本号．类似的，如果源仓库的子目录在将来被重命名了，则同步过程将无法继续，因为你所指定的源仓库的子目录 URL 已经不再有效．

创建镜像的小窍门

前面我们已经介绍了为了给一个已存在的仓库创建镜像需要完成哪些 工作．对于很多人而言，使用 *svnsync* 传送成千— 甚至成百万—的版本号历史所带来的代价，就像是看一场被掌声中断 了很久的表演．幸运的是，Subversion 1.7 提供了一个变通方 法，通过为 *svnsync initialize* 添加新选项 `--allow-non-empty`，该选项允许用户在把仓库初始化成 另一个仓库的镜像时，不去检查将被初始化的镜像仓库是否含有版本历史．通过前面几次使用过程中的警告，读者应该很快就能看出必须小心使用这个 选项．但是，如果用户拥有源仓库的管理员权限，那么这个选项就会非常 方便，因为用户可以直接复制仓库，然后把复制出的仓库初始化成新镜像：

```
$ svnadmin hotcopy /path/to/repos /path/to/mirror-repos
$ ### create /path/to/mirror-repos/hooks/pre-revprop-change
$ svnsync initialize file:///path/to/mirror-repos \
                     file:///path/to/repos
svnsync: E000022: Destination repository already contains revision history; co
```

```
nsider using --allow-non-empty if the repository's revisions are known to mirr
or their respective revisions in the source repository
$ svnsync initialize --allow-non-empty file:///path/to/mirror-repos \
                                       file:///path/to/repos
Copied properties for revision 32042.
$
```

如果管理员使用的 Subversion 版本低于 1.7（即 *svnsync initialize* 不支持选项 `--allow-non-empty`），还可以通过其他手段实现相同的目的，那就是 认真地 修改仓库副本（该副本将作为源仓库的镜像）的版本号 r0 的属性，使得 r0 的属性与 *svnsync* 将会创建的记账属性相同.

复制小结

本节讨论了几种用于复制版本号历史到其他仓库的方法，现在从普通 用户的角度看待这些操作：仓库复制和各种不同情况下的执行方式将会如 何影响客户端?

如果用户需要同时和仓库及其镜像打交道，那么使用一个 单独的 工作副本同时与多个仓库交互是有可能做到 的，但要满足一些条件. 首先，用户要确保主仓库和镜像仓库拥有相同的 UUID（默认情况下并不相同），关于仓库 UUID 的更多内容，见 “管理仓库的 UUID”一节.

一旦两个仓库拥有相同的 UUID，用户就可以用命令 *svn relocate* 把工作副本重定向到任意一个仓库，见 svn 参考手册—Subversion 命令行客户端 的 svn relocate. 但其中有一个潜在的问题：如果 主仓库和镜像仓库不是完全同步，而工作副本当前指向主仓库，并且处于最 新的状态，如果此时把工作副本重定向到过时的镜像仓库，工作副本就会 报错，因为本来应该存在的版本号突然无缘无故的消失了. 如果发生了这种 情况，可以再把工作副本重定向回原来的主仓库，然后等镜像仓库与主仓库 完全同步，或者把工作副本回退到在镜像仓库中存在的版本号，然后再重 定向工作副本.

最后，要注意 *svnsync* 所提供的基于版本号的 复制流程只会复制版本号. 只有仓库转储文件格式所携带的信息类型才能 用于复制，因此，*svnsync*（以及 *svnrdump*，见 “使用 svnrdump 迁移仓库数据”一节）受到的 限制与仓库转储流受到的限制类似，它们不会复制已实现的钩子，仓库或 服务器的配置，未提交的事务，或用户施加在仓库路径上的锁.

仓库备份

自现代计算机诞生以来，不管出现了多么先进的技术，有一点总是挥之不 去—有时候，事情会变得非常糟糕. 即使是最小心翼翼的管理员，也不得 不面对断电，网络故障，内存与硬盘损坏. 因此本节的主题是如何备份仓库数 据.

对于 Subversion 仓库管理员来说，有两种备份策略—全量备份与 增量备份. 全量备份涉及到在一个操作中记录所有的信息，这些信息可以在 灾难发生后，重新构造出原来的仓库. 通常意味着复制整个仓库目录（包括 后端存储 Berkeley DB 或 FSFS 的数据）. 增量备份每次记录的数据量相对 较少：它只备份上一次备份以来，发生变化的仓库数据.

对于全量备份来说，这个笨拙的方法看起来似乎很合理，但是除非管理员 临时禁止对仓库的其他访问，否则的话，如果只是简单地复制目录，得到的 备份也可能是有问题的. Berkeley DB 的文档描述了一种特定的数据库文件 复制顺序，它可以保证得到的备份是有效的，类似的复制顺序同样存在于 FSFS. 管理员不用考虑如何实现它们所要求的复制顺序，因为 Subversion 开发团队 已经帮你实现好了，命令 *svnadmin hotcopy* 会处理好 仓库热拷贝涉及到的各种细枝末节，它们调用方式就像 Unix 的 *cp* 或 Windows 的 *copy* 那样简单：

```
$ svnadmin hotcopy /var/svn/repos /var/svn/repos-backup
```

得到的备份是一个完整的 Subversion 仓库，能够在原仓库出现故障时 顶替上去.

围绕该命令，还有其他一些额外的工具可供使用. Subversion 源代码目录 *tools/backup* 存放了一个脚本：*hot-backup.py*. *hot-backup.py* 在 *svnadmin hotcopy* 之上增加了一些备份管理策略，允许用户只保留最近几次的仓库备份. *hot-backup.py* 自动管理通过备份得到的仓库目录的名字，避免出现名字冲突，而且会 “轮换” 旧备份，只保留最近的几次备份. 即使管理员使用的是 增量备份策略，也有使用 *hot-backup.py* 的需求. 例如 管理员可以在调度程序（比如 Unix 系统中的 *cron*) 中执行 *hot-backup.py*, 从而实现每晚运行一次（或 管理员认为合适的其他时间间隔）脚本.

围绕仓库转储数据的生成与存放，有些管理员会使用不同的备份方法. 在 “迁移仓库数据”一节 我们介绍了如何 使用带有选项 `--incremental` 的 *svnadmin dump*, 在给定的版本号或版本号范围内执行增量备份. 当然，也可以忽略选项 `--incremental`, 实现一个完全备份. 使用仓库转储数据的好处是这种备份信息的格式非常灵活—它与特定 的平台，仓库文件系统类型，Subversion 的版本及其所使用的函数库都是 无关的. 但灵活性也带来了一定的开销，就是需要较长的时间才能完全恢复 数据—每当有一个新的版本号提交时，时间就会变得更长一点. 另外，和其他备份方法相比，如果修改了已经备份过的版本号的版本号属性，这些 修改将不会体现在增量的转储数据中. 由于这些原因，我们建议读者不要单独 依靠基于转储的备份策略.

从 Subversion 1.8 开始，*svnadmin hotcopy* 支持 选项 `--incremental`, 允许对 FSFS 仓库进行增量热拷贝，在增量热拷贝模式下，已经复制到目标仓库的版本号数据不会再复制一次. 如果 *svnadmin hotcopy* 带上选项 `--incremental`, Subversion 将只会复制新的版本号，以及 上一次热拷贝后，大小或时间戳发生变化的版本号. 而且，与 *svnsync* 或 *svnadmin dump --incremental* 有所不同的是 *svnadmin hotcopy --incremental* 仅受限于磁盘的读写性能，在备份大型仓库时，增量热拷贝可以节省大量的时间.

可以看到，不同的备份方式都有各自的优点和缺点，目前最简单的选择就 是全量的热拷贝，它总能得到一份可用的仓库副本，一旦主仓库出现故障，只要简单地递归复制目录，就能从备份仓库中恢复. 不幸的是，如果同时存 在多个仓库副本，这些全量拷贝所消耗的磁盘空间将会是很可观的. 与此相对，生成增量副本更快，消耗的磁盘空间也更少，但是复原过程就比较痛苦了，经常需要应用多个增量备份. 其他几种备份方法也有各自的特点. 管理员需要 在备份和复原的开销之间，找到最适合的平衡点.

svnsync（见 “仓库复制”一节) 提供了非常方便的折衷 方案. 如果管理员周期性地把主仓库同步到只读的镜像仓库，在主仓库发生 故障时，镜像仓库就会是一个很好的替补. 这种方法的主要缺点是只有版本化 的仓库数据才会被同步—仓库的配置文件，用户指定的仓库路径锁，以 及其他存放在仓库目录中的项目，只要它们不在仓库的版本化文件系统中，就不会被 *svnsync* 处理.

在任何一种备份场景下，管理员都要注意版本号属性将如何影响他们的 备份. 版本号属性的修改不会产生新的版本号，不会触发钩子 post-commit, 甚至也不会触发钩子 pre-revprop-change 和 post-revprop-change. [11] 而且用户可以按照任意的时间顺序修改 版本号属性—在任意时刻修改任意一个版本号的属性—在增量备份 的情况下，如果某个版本号已经包含在前一次备份中，它的属性在后来被修改 了，那么修改将不会包含在后面的增量备份中.

一般来说，只有真正偏执的人才想在每次提交后备份整个仓库，然而，如果仓库已经具备了一些相对细致的冗余机制（例如每次提交后的邮件通知 或增量转储），那么仓库管理员可能会把数据库的热拷贝作为每夜例行工作的 一部分去执行. 这是你的数据—你想怎么保护它们都不为过.

很多时候，备份仓库的最佳方式是本节所描述的各种方法的组合. 比如说 Subversion 开发人员备份 Subversion 源代码仓库的方式是每晚使用 *hot-backup.py* 和异地的 *rsync* 完成全量备份；为所有的提交和属性修改通知邮件维护多份归档；由不同的 志愿者使用 *svnsync* 维护多份仓库镜像. 你的最终 方案可能与此类似，但应该根据你的具体需求维持好易用性与安全性

[11] 命令 *svnadmin setlog* 可以完全旁路 掉钩子接口

之间的 平衡. 无论你怎么做, 应该偶尔检查备份是否可用—如果连备胎都有漏洞, 那还怎么用? 虽然这一切都无法避免硬件遭受命运的捶打[12], 但备份确实能够帮助你从灾难中恢复.

管理仓库的 UUID

每个 Subversion 仓库都有一个全局统一标识 (universally unique identifier, 简称 UUID) 与之关联. 当其他手段不够完善时 (例如检查仓库的 URL, 但 URL 可以会变化), 客户端可以使用 UUID 识别仓库. 大多数管理员 极少需要考虑仓库的 UUID, 对他们而言, UUID 只是 Subversion 的一个实现 上的细节而已. 然而, 少数情况下这个细节也需要引起注意.

一般来说, 管理员希望活动仓库的 UUID 是独一无二的, 毕竟这就是 UUID 的主要特点. 但在某些情况下需要两个仓库拥有一模一样的 UUID, 比如 说管理员为仓库制作了一个副本, 并且希望该副本是源仓库的完美镜像, 因为 管理员希望当备份仓库替换掉活动仓库时, 用户不会突然看到一个似乎不同的 仓库. 在转储和加载仓库历史时 (见 “迁移仓库数据”一节), 管理员可以根据实际情况 决定是否向目标仓库应用封装在转储流中的 UUID.

有若干种方式可以用来设置或重置仓库的 UUID, 对于 Subversion 1.5 而言, 用到的命令是 *svnadmin setuuid*. 如果在命令行上显式地提供了 UUID 参数, 命令将验证 UUID 的格式是否正确, 如果正确就 把它设置到仓库上. 如果省略了 UUID 参数, 命令就自动为仓库生成一个全新 的 UUID.

```
$ 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
$
```

如果你用的是 1.5 版之前的 Subversion, 那么设置 UUID 的过程会更复杂 一点. 为了设置 UUID, 管理员可以把带有新 UUID 的桩转储文件, 以管道地方式 传递给 `svnadmin load --force-uuid` *`REPOS-PATH`* , 从而显式设置仓库的 UUID.

```
$ 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
$
```

使用旧版的 Subversion 来生成全新的 UUID 并不是一件非常容易的事情, 管理员最好找到生成 UUID 的其他方法, 然后再通过上面的方式为仓库显式地 设置新 UUID.

[12] 你知道的—这只是对各种 “变化莫测的问题” 的统称.

移动与删除仓库

Subversion 仓库的所有数据都存放在仓库目录中，所以说管理员可以使用 操作系统提供的命令—Unix 系统的 *mv*, *cp -a*, *rm -r*; Windows 系统的 *move*, *copy*, *rmdir /s /q*; 或其他图形化文件浏览器提供的鼠标和菜单操作— 实现仓库的移动，重命名，复制和删除.

当然，在做完上面的操作后还有些清理工作需要完成. 比如说，管理员需要 为移动后的仓库更新 Subversion 服务器的配置，或者删除一些配置（因为相关 的仓库已经被移除了）. 如果管理员设置了与仓库有关的自动化信息发布系统，它 们可能需要更新，钩子脚本可能需要重新配置，还可能需要通知用户. . . 工作列 表可以不停地写下去，至少应该覆盖到与仓库有关的构建过程.

对于通过复制得到的仓库，需要注意 Subversion 使用 UUID 区分仓库. 如 果管理员是用一个典型的递归复制命令来复制仓库，那么和源仓库相比，两者完 全没有区别—甚至连 UUID 也相同. 在某些情况下这是合理的效果，但有 时候却不是，此时管理员需要为其中一个仓库重新生成一个新的 UUID. 关于如何 管理仓库的 UUID，见 “管理仓库的 UUID”一节.

小结

读到这里，读者应该具备了创建，配置与维护 Subversion 仓库的基本知识，本章介绍了多个和这些任务相关的工具，对于常见的管理问题，我们也提供了很 多建议与解决方案.

剩下的就是由你来决定把什么数据放在仓库里，以及如何将仓库发布到网络 上. 下一章讨论的是与网络相关的内容.

第 6 章 服务器配置

一个 Subversion 仓库可以被多个客户端同时访问，这些客户端和仓库都在同 一台主机上，通过 `file://` 形式的 URL 进行访问．不过在 典型的情况下，Subversion 仓库是存放在一台单独的服务器上，办公室— 甚至全世界—的任意一台主机都能访问到这台服务器．

本章介绍如何把 Subversion 仓库暴露给远程的客户端使用．我们将会介绍 Subversion 目前支持的服务器机制和各个配置，读完本章后，读者将有能力判断 什么样的网络配置才是正确的，以及如何在自己的主机上进行配置．

概览

Subversion 具有一个抽象的仓库访问层，这就意味着仓库可以 被任意类型的服务器进程以程序化地方式进行访问，并且客户端 “仓库访问 API” 允许程序员编写协议相关的插件．从理论上讲，Subversion 支持 任意类型的网络实现，但在实际使用中，只有两种 Subversion 服务器用得比较 广泛．

Apache HTTP 服务器（也被称作 *httpd*）是一 个非常流行的网页服务器；利用 *mod_dav_svn* 模块，Apache 就能访问仓库，并通过 WebDAV/DeltaV 协议支持客户端访问，WebDAV/DeltaV 协议是 HTTP 的扩展．因为 Apache 的可扩展性非常强，它提供 了大量可 “免费” 使用的特性，例如加密的 SSL 通信，日志记录，可集成第三方认证系统，受限的仓库网页访问界面．

另一种就是 *svnserve*：一种小型的轻量级服务 器程序，使用定制的协议与客户端通信．因为它所使用的协议是专为 Subversion 而设计的有状态协议（相对于无状态的 HTTP 协议），它提供了更快速的网络操作 —但同时也要付出一些代价．虽然 *svnserve* 可以利用 SASL 提供各种各样的认证和加密选项，但它不支持日志记录和网页浏览．然而，搭建 *svnserve* 服务器非常方便，非常适合刚开始接触 Subversion 的小型团队．

svnserve 所使用的协议还能被 SSH 包裹，这种部署 形式和以传统方式部署的 *svnserve* 相比，在特性上有很 大的不同．此时 SSH 将用于加密所有的连接，只使用 SSH 进行认证，这就要求 用户在服务器主机上必须拥有真实的系统账户（而普通的 *svnserve* 拥有一套自己的用户账户）．最后，这种部署 配置要求每个用户都要派生一个私有的临时 *svnserve* 进程，（从系统权限的角度来看）这就相当于允许一组本地用户使用 `file://` URL 访问仓库，基于路径的访问控制将失去意义，因为用户是在直接访问仓库的数据库文件．

表 6.1 “各种 Subversion 服务器选项的比较” 总结了三种典型 的服务器部署方式．

表 6.1. 各种 Subversion 服务器选项的比较

特性	**Apache + mod_dav_svn**	**svnserve**	**svnserve + SSH**
认证选项	HTTP Basic or Digest 认证，X.509 证书，LDAP，NTLM，或 Apache httpd 支持的其他认证机制	默认是 CRAM-MD5；LDAP，NTLM，或 SASL 支持的其他认证机制	SSH
用户帐户选项	私有的 “用户” 文件，或 Apache httpd 支持的 其他机制 (LDAP，SQL 等)	私有的 “用户” 文件，或 SASL 支持的其他机制 (LDAP，SQL 等)	系统账户

特性	Apache + mod_dav_svn	svnserve	svnserve + SSH
授权选项	可以在整个仓库或特定的路径上授予读/写权限	可以在整个仓库或特定的路径上授予读/写权限	只能在整个仓库上授予读/写权限
加密	通过可选的 SSL (https) 实现	通过可选的 SASL 特性实现	由 SSH 连接实现
日志记录	支持在较高的层次上记录 Subversion 操作，也支持为每一个 HTTP 请求记录日志	只支持从较高的层次上记录 Subversion 操作	只支持从较高的层次上记录 Subversion 操作
互操作性	其他 WebDAV 客户端可访问	只能被 svn 客户端访问	只能被 svn 客户端访问
网页浏览	有限的内建支持，或者通过第三方工具实现，例如 ViewVC	只能通过第三方工具实现，例如 ViewVC	只能通过第三方工具实现，例如 ViewVC
主从服务器复制	从服务器到主服务器可以使用透明的写代理	只能创建只读的从服务器	只能创建只读的从服务器
访问速度	较慢	较快	较快
初始设置	较复杂	非常简单	中等

选择一种服务器配置

那么，我应该使用哪种服务器配置，哪一种是最好的?

显然，这个问题不存在正确的答案. 每一个开发团队都有不同的需求，每一种服务器配置都代表了不同的权衡. Subversion 不会偏爱任何一种，也不会认为某种服务器配置更加 “官方”.

下面列出几点在选择服务器部署配置时应该考虑的因素.

svnserve 服务器

应该用它的理由:

- 设置方便.
- 网络协议是有状态的，并且比 WebDAV 快很多.
- 不需要在服务器上创建系统帐户.
- 密码不会在网络上传输.

不应该用它的理由:

- 默认情况下，只有一种认证方式可用，网络协议是未加密的，而且服务器以明文的形式存放密码.（这些都能通过配置 SASL 加 以修改，但也带来了更多的工作量.）
- 缺乏高级的日志设施.
- 缺乏内建的网页浏览界面.（为了实现网页浏览，管理员必须 安装额外的网页服务器和仓库浏览软件.）

svnserve + SSH

应该用它的理由：

- 网络协议是有状态的，并且比 WebDAV 快很多.
- 管理员可以利用已有的 SSH 帐户和用户基础设施.
- 所有的网络流量都是加密的.

不应该用它的理由：

- 只有一种认证方式可用.
- 缺乏高级的日志设施.
- 要求用户属于相同的系统用户组，或者使用共享的 SSH 密钥.
- 如果使用得不恰当，将产生与文件权限有关的问题.

Apache HTTP 服务器

应该用它的理由：

- 允许 Subversion 去使用已经集成到 Apache 中的多种认证 机制.
- 无需在服务器上创建系统帐户.
- 有完备的 Apache 日志可供使用.
- 网络流量经由 SSL 加密.
- 一般来说，HTTP(S) 可以通过企业防火墙.
- 内建的仓库浏览特性可被网页浏览器使用.
- 仓库可以被当作网络驱动器进行挂载，实现完全透明化的版本控制（见 “自动版本控制”一节).

不应该用它的理由：

- 比 *svnserve* 慢得多，因为 HTTP 是一 种无状态的协议，需要更多的网络周转.
- 初始设置比较复杂.

建议

一般情况下，对于想要快速搭建 Subversion 服务器的小团队而言，本书 作者推荐最普通的 *svnserve*，它的设置最简单，维护成本也很低. 如果有新的需求产生，管理员总是可以切换到更复杂的部署方式.

根据多年的用户支持经验，下面列出几点一般性的建议和技巧：

- 如果管理员想为团队搭建尽可能简单的 Subversion 服务器，那么 最简单的选择就是 *svnserve*. 然而，需要注意的是 仓库的数据将在网络上以明文形式传输，如果服务器完全部署在公司的 LAN 或 VPN 内部，那就不会带来什么问题. 相反，如果仓库可被因特网 访问到，管理员要么确保仓库存放的不是敏感数据（例如只包含了开源 的代码），要么使用 SASL 对网络传输进行加密.
- 如果管理员想把已有的身份系统 (LDAP, Active Directory, NTLM, X.509 等）集成到 Subversion 服务器中，那就必须选择 Apache 服务器，或配有 SASL 的 *svnserve*.
- 如果管理员想使用 Apache 或 *svnserve*, 要在 服务器系统中创建一个新用户 *svn*, 然后以该用户的 身份运行服务器进程. 确保用户 *svn* 完全拥有仓库 目录，从安全的角度来看，这种做法使得仓库的数据能够保持孤立，还能 利用操作系统的文件系统权限，保证只有 Subversion 服务器进程才能修改 仓库目录.
- 如果已有的基础设施严重依赖 SSH 账户， 并且团队成员在服务器上 都有自己的系统账户， 此时比较好的部署方式是 *svnserve* + SSH. 但如果是对外公开的仓库，则我 们不建议这样做，一般而言，相比于真正的系统账户，通过 *svnserve* 或 Apache 管理的（虚假）账户来访问 仓库是一种更安全的做法. 如果管理员对加密通信仍然具有强烈的渴望，我们建议选择配有 SSL 的 Apache, 或配有 SASL 的 *svnserve*.
- 不要 被这种简单的想法引诱：让所有的用户 使用 `file://` URL 直接访问仓库. 即使仓库已经 准备好通过网络共享被所有人访问，但这仍然不是个好主意. 这种做法 移除了用户与仓库之间的所有保护层：用户可以有意（或无意）地破坏仓库数据库，为了检查或升级而对仓库进行下线操作，也会变得非常困难，而且会产生一系列与文件权限有关的问题（见 “支持多种仓库访问方法”一节). 注意，这同时也是使用 `svn+ssh://` URL 访问仓库时需要注意的地方— 从安全的角度来看，它和本地用户使用 `file://` 的 情况是等效的，如果管理员不够认真，将会导致同样的问题.

svnserve, 一个定制化的服务器

svnserve 是一个轻量级的服务器程序，基于 TCP/IP, 使用 一种定制化的，有状态的协议与客户端通信，客户端使用 `svn://` 或 `svn+ssh://` 形式的 URL 访问 *svnserve* 服务器. 本节介绍运行 *svnserve* 的多种方式，服务器如何认证客户端，以及如何为 仓库配置合适的访问权限.

调用服务器

运行程序 *svnserve* 有以下几种方式：

- 作为一个独立的守护进程运行 *svnserve*, 运行 过程中监听请求.
- 如果在特定的端口接收到了一个新请求，就让 Unix 守护进程 *inetd* 临时派生 *svnserve*.
- 使用 SSH, 在加密的通道上调用一个临时的 *svnserve*.
- 作为 Microsoft Windows 服务，运行 *svnserve*.
- 作为一个 launchd 作业，运行 *svnserve*.

下面的几个小节详细介绍这些不同的 *svnserve* 部署方式.

svnserve 作为守护进程

最简单的方式，就是把 *svnserve* 作为一个守护进程 运行，执行时需要添加选项 `-d`:

```
$ svnserve -d
$               # svnserve is now running, listening on port 3690
```

以守护进程模式运行 *svnserve* 时，可以使用 选项 `--listen-port` 和 `--listen-host` 修改进程所 “绑定” 的端口号和主机名.

svnserve 一旦成功启动，服务器上的所有仓库 都能通过网络进行访问. 如果客户端需要访问仓库，必须在仓库的 URL 参数中指定一个 绝对 路径. 比如说某个仓库在 服务器上的位置是 */var/svn/project1*，那么客户 端访问仓库的 URL 参数就可以写成 `svn://host.example.com/var/svn/project1`. 为了增加安全 性，可以为 *svnserve* 添加选项 `-r`，使得只有指定路径下的仓库才会被导出，例如：

```
$ svnserve -d -r /var/svn
…
```

使用选项 `-r` 等价于修改了 *svnserve* 的根目录，客户端访问仓库的所使用的 URL 也能写得更加简短：

```
$ svn checkout svn://host.example.com/project1
…
```

由 inetd 调用 svnserve

如果管理员希望由 *inetd* 启动进程，就给 *svnserve* 添加选项 `-i` (`--inetd`). 在下面的例子里，我们展示了在命令行上 执行 `svnserve -i` 的输出，但要注意的是命令实际 上并没有启动进程；例子后面的内容介绍了如何配置 *inetd*，使得它能够启动 *svnserve*.

```
$ svnserve -i
( success ( 2 2 ( ) ( edit-pipeline svndiff1 absent-entries commit-revprops d\
epth log-revprops atomic-revprops partial-replay ) ) )
```

如果使用选项 `--inetd` 调用 *svnserve*，它会尝试使用定制化的协议，通过 *stdin* 和 *stdout* 与 Subversion 通信，这是由 *inetd* 所启动的程序的标准 行为. IANA 将端口 3690 保留给 Subversion 使用，所以说在一个类 Unix 系统上，管理员可以安全地在 */etc/services* 中添 加以下内容（如果原来没有的话）：

```
svn             3690/tcp    # Subversion
svn             3690/udp    # Subversion
```

如果服务器使用的是经典的类 Unix *inetd* 守护 进程，就在 */etc/inetd.conf* 添加下面这一行：

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

确保用户 “svnowner” 对仓库具有适当的访问权限. 现在，当客户在端口 3690 上连接服务服务器时，*inetd* 就会派生出一个 *svnserve*，负责处理客户端发来的 请求. 当然，管理员也可以在配置文件里为 *svnserve* 添加选项 `-r`，从而限制被导出的仓库.

由 xinetd 调用 svnserve

某些系统提供了 *xinetd* 作为 *inetd* 的替代品，幸运的是，管理员也可以配置 *svnserve* 由 *xinetd* 启动. 为了实现这点，管理员创建一个配置 文件 */etc/xinetd.d/svn*，文件的内容是：

```
# default: on
# description: Subversion server for the svn protocol
service svn
```

```
{
  disabled        = no
  port            = 3690
  socket_type     = stream
  protocol        = tcp
  wait            = no
  user            = subversion
  server          = /usr/local/bin/svnserve
  server_args     = -i -r /path/to/repositories
}
```

要确保 */etc/services* 包含了 `svn` 协议所使用的端口（见 “由 inetd 调用 svnserve”一节），否则的话，守护进程将无法正常启动.

如果是基于 Redhat 的 Linux 发行版，管理员需要使用 *chkconfig --add svn* 激活新的服务，后面就可以用图 形化配置工具禁止或允许服务器程序.

基于隧道的 svnserve

另一种启动方式是通过添加选项 `-t`，以隧道模式启动 *svnserve*. 隧道模式假设有一个远程服务程序（例如 *rsh* 或 *ssh* 已经成功地授权了 一个用户，并且以该 *用户的身份* 启动了一个私有 的 *svnserve* 进程.（用户几乎没有必要在命令行 启动带有选项 `-t` 的 *svnserve*，相反，SSH 守护进程会替用户执行这个操作）程序 *svnserve* 像往常一样运行（通过 *stdin* 和 *stdout* 与其他进程通信），它还假设网络数据可以通过某种隧道，被自动重定向回客户端. 当隧道代理 以这种方式启动 *svnserve* 时，要确保被授权的用户 对仓库数据库文件具有读写权限，在本质上它和本地用户通过 `file://` URL 访问仓库的情况是一样的.

更多的细节见 “SSH 隧道”一节.

svnserve 作为 Windows 服务

如果服务器所用的 Windows 系统是 Windows NT 的后代 (Windows 2000 或 更新的版本)，管理员就能把 *svnserve* 作为一个标准 的 Windows 服务启动，和独立的守护进程启动方式（添加选项 `--daemon` (`-d`)) 相比，这通常能带 来更好的体验. 为了以守护进程的方式启动 *svnserve*，我们需要打开一个控制台，输入命令，然后任由控制台永远地运行下去. 然而，在后台运行的 Windows 服务可以在系统引导时自动启动，可以使用和其他 Windows 服务一样的管理接口来启动或停止服务.

为了定义一个新的 Windows 服务，需要用到命令行工具 *SC.EXE*. 类似于 *inetd* 的配置 文件，管理员必须准确地指定 *svnserve* 的启动方式，以便 Windows 在开机时启动相应的服务：

```
C:\> sc create svn
        binpath= "C:\svn\bin\svnserve.exe --service -r C:\repos"
        displayname= "Subversion Server"
        depend= Tcpip
        start= auto
```

上面的命令行定义了一个新的，名为 `svn` 的 Windows 服务，当服务启动时，它将执行程序 *svnserve.exe*. 在这个例子里有很多需要注意的地方.

首先，启动 *svnserve.exe* 时必须带上参数 `--service`，其他选项必须出现在同一行，不能再添加 会引起冲突的选项，例如 `--daemon` (`-d`), `--tunnel` 或 `--inetd` (`-i`), 但可以添加选项 `-r` 或 `--listen-port`. 第二，注意命令行里的空格：模

式 `key= value` 中，`key` 和 `=` 之间不能有空格，而 `key=` 和 `value` 之间有且仅有 一个空格. 最后，要注意被调用的命令行里的空格. 如果目录名含有空格（或其他需要转义的字符），就把 `binpath` 内的路径 包裹在一对双引号中，但要对双引号进行转义：

```
C:\> sc create svn
        binpath= "\"C:\program files\svn\bin\svnserve.exe\" --service -r C:\repos"
        displayname= "Subversion Server"
        depend= Tcpip
        start= auto
```

还要注意 `binpath` 容易让人产生误解— 它的值是一个 命令行，而不是可执行文件的路径. 因此，如果它的值含有内嵌的空格，就要用双引号包围起来.

服务定义完成后，可以使用标准的 GUI 工具（服务管理控制面板）来 停止，启动或查询服务，也要以使用命令行工具：

```
C:\> net stop svn
C:\> net start svn
```

服务还能被卸载，方法是删除它的定义：`sc delete svn`，但在这之前记得先停止服务！程序 *SC.EXE* 还有很多子命令和选项，执行 `sc /?` 查看完整的命令帮助信息.

svnserve 作为 launchd 作业

Mac OS X (10.4 及更新的版本）使用 *launchd*，在 系统范围和用户范围内管理进程（包括守护进程）. 一个 *launchd* 由 XML 文件内的参数指定，命令 *launchctl* 用于管理作业的生命周期.

如果 *svnserve* 被配置成作为一个 *launchd* 作业，那么当有 `svn://` 网络流量需要处理时，将自动启动 *svnserve*. 这要比 手动地启动 *svnserve* 并把它作为长时间运行的后台 进程要方便得多.

为了把 *svnserve* 配置成一个 *launchd* 作业，首先创建一个名为 */Library/LaunchDaemons/org.apache.subversion.svnserve.plist* 的作业定义文件，例 6.1 “svnserve 的 launchd 作业定义的一个示例” 展示 了该文件的一个例子.

例 6.1. svnserve 的 launchd 作业定义的一个示例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>org.apache.subversion.svnserve</string>
        <key>ServiceDescription</key>
        <string>Host Subversion repositories using svn:// scheme</string>
        <key>ProgramArguments</key>
        <array>
            <string>/usr/bin/svnserve</string>
            <string>--inetd</string>
            <string>--root=/var/svn</string>
```

```
        </array>
        <key>UserName</key>
        <string>svn</string>
        <key>GroupName</key>
        <string>svn</string>
        <key>inetdCompatibility</key>
        <dict>
            <key>Wait</key>
            <false/>
        </dict>
        <key>Sockets</key>
        <dict>
            <key>Listeners</key>
            <array>
                <dict>
                    <key>SockServiceName</key>
                    <string>svn</string>
                    <key>Bonjour</key>
                    <true/>
                </dict>
            </array>
        </dict>
    </dict>
</plist>
```

launchd 系统学习起来有一定的困难，幸运的是 本节所介绍的 *launchd* 命令都有相关的文档可供 参考，例如执行 `man launchd` 查看 *launchd* 的手册页，执行 `man launchd.plist` 查看作业定义文件的格式.

作业定义文件创建完毕后，就可以用 *launchctl load* 激活作业：

```
$ sudo launchctl load \
       -w /Library/LaunchDaemons/org.apache.subversion.svnserve.plist
```

需要澄清的是，上面的命令并没有启动 *svnserve*，它只是告诉 *launchd* 当网络端口 `svn` 有网络数据到达时如何启动 *svnserve*，当网络数据处理完毕后，*svnserve* 进程就会终止.

因为我们希望 *svnserve* 成为整个系统范围内 的守护进程，所以要用 *sudo* 命令，作为系统管理员 去管理作业. 定义文件内的 `UserName` 和 `GroupName` 是可选的—如果忽略它们，作业的 所有者将是加载该作业的用户.

禁用作业的方法也很简单—使用 *launchctl unload*：

```
$ sudo launchctl unload \
       -w /Library/LaunchDaemons/org.apache.subversion.svnserve.plist
```

launchctl 也提供了查询作业状态的命令，如果 作业已加载，那么作业定义文件中，`Label` 所指定的 内容将会出现在命令的输出中：

```
$ sudo launchctl list | grep org.apache.subversion.svnserve
```

```
-       0       org.apache.subversion.svnserve
$
```

内建的认证与授权

当客户端连接到 *svnserve* 进程时，将会发生以下事件：

- 客户端选择一个特定的仓库.
- 服务器读取仓库的 *conf/svnserve.conf*，施加文件所描述的认证与授权策略.
- 取决于具体的策略，可能会发生下面几件事中的一件：
  - 允许客户端以匿名的方式提出请求，不会收到任何认证要求.
  - 客户端可能在任意时刻收到认证要求.
  - 如果操作是在隧道模式下进行，客户端将声明它已经在外部认证 过了（通常是 SSH).

默认情况下，*svnserve* 只知道如何发送一个 CRAM-MD5[1] 授权请求，在本质上，就是服务器向客户端发送了一小段数据. 客户端使用 MD5 散列算法 为数据和密码的混合物创建指纹，然后发送该指纹，作为认证请求的响应. 服 务器对存放在本地的密码进行同样的计算，以验证它们是否相同. *在 任何情况下都不会在网络上传输明文密码.*

如果 *svnserve* 支持 SASL，除了知道如何发送 CRAM-MD5 请求外，*svnserve* 还能使用其他几种认证 机制，本章后面的 “svnserve 使用 SASL” 一节 将会介绍如何配置 SASL 认证和加密.

当然，客户端也可以通过一个隧道代理（例如 *ssh*），实现外部认证. 在这种情况下，服务器只是简单地检查自己的用户身份，然后 使用该用户名作为已认证的用户. 更多的相关内容，见 “SSH 隧道” 一节.

读者可能已经猜到了，仓库里的 *svnserve.conf* 正是控制认证和授权策略的关键. 当它和本节所描述的其他附加文件配合工作 时，*svnserve.conf* 向管理员提供了用于控制用户认证 和授权策略的完整方案. 我们将要讨论的每个文件所使用的格式，与其他配置 文件的格式相同（见 “运行时配置区域” 一节）：节名用一对方括号标记（`[` 和 `]`），注释由井号 (`#`) 开始，每一节都包含了可被赋值的特定变量 (`variable = value`). 下面介绍各个文件并学习如何 使用它们.

创建一个用户文件和认证域

现在，你所需要的所有变量都在 *svnserve.conf* 的 `[general]` 部分. 先从修改这些变量的值开始：为存放用户名和密码的文件选择一个名字，以及选择一个认证域：

```
[general]
password-db = userfile
realm = example realm
```

`realm` 的值是你自己定义的一个名字，它告诉 客户端它们正在连接的是哪一个 “认证空间”；Subversion 客户端在认证的提示信息里显示 `realm` 的值，并 用它（再加上服务器的主机名和端口号）作为缓存在磁盘上的证书的键（见 “缓存证书” 一节). 变量 `password-db` 指向一个单独的文件，它包含了 一连串的用户名和密码，文件的格式和 *svnserve.conf* 是相同的，例如：

[1] 见 RFC 2195.

```
[users]
harry = foopassword
sally = barpassword
```

变量 password-db 的值可以是指向用户文件 的绝对路径或相对路径，管理员很容易就能把用户文件设置到仓库的 *conf/* 目录内，和 *svnserve.conf* 放在一起. 另外，多个仓库还能 共享同一个用户文件，在这种情况下，文件应该放在更加开放的位置. 共享同一用户文件的仓库还要配置相同的认证域，因为用户名列表在本质上 就已经定义了一个认证域. 无论用户文件放在何处，都要设置好它的 读写权限. 如果管理员知道 *svnserve* 将以哪些用户 身份运行，在必要时可限制用户文件的读取权限.

设置访问控制

还有两个变量可以在 *svnserve.conf* 里设置: 它们决定了未验证（匿名）的用户和已验证的用户可以做哪些事情. 变量 anon-access 和 auth-access 可被设置的值有 none, read 和 write. 设置为 none 将 禁止读和写; read 允许以只读方式访问仓库; write 允许对仓库进行完全的读写访问. 例如:

```
[general]
password-db = userfile
realm = example realm

# anonymous users can only read the repository
anon-access = read

# authenticated users can both read and write
auth-access = write
```

例子所展示的其实就是变量的默认值，以免管理员忘记设置它们. 如 果管理员需要更加保守的设置，可以完全禁止匿名访问:

```
[general]
password-db = userfile
realm = example realm

# anonymous users aren't allowed
anon-access = none

# authenticated users can both read and write
auth-access = write
```

服务器进程不仅可以理解施加到仓库上的全局的访问控制，还能理解 施加到文件或目录上的更细粒度的访问控制. 为了利用后者，管理员创建一 个文件，文件包含了更细致的规则，然后让变量 authz-db 指向该文件:

```
[general]
password-db = userfile
realm = example realm

# Specific access rules for specific locations
authz-db = authzfile
```

我们在 “基于路径的授权”一节 详细 介绍 *authzfile* 的语法. 注意, 变量 `authz-db` 与 `anon-access`, `auth-access` 并非互不相容, 如果同时定义了这三个 变量, 则只有在 所有 规则都被满足的情况下, 才能允许访问.

svnserve 使用 SASL

对于许多团队而言, 使用 *svnserve* 内建的 CRAM-MD5 认证就已足够. 然而, 如果服务器和 Subversion 客户端支持 SASL (Cyrus Simple Authentication and Security Layer) 函数库, 那么管理员就有了 大量的认证和加密选项可供选择.

> **什么是 SASL?**
>
> Cyrus Simple Authenticated and Security Layer (简称 SASL) 是卡耐基梅隆 大学开发的一款开源软件, 可以给任意一种网络协议添加通用的认证和 授权功能, 从 Subversion 1.5 开始, *svnserve* 服务器和 *svn* 客户端开始支持 SASL. 不过 SASL 并非总是可用, 如果 Subversion 是你自己编译安装的, 为了利用 SASL, 系统中至少要安装 SASL 2.1 版, 并且在 Subversion 构建过程中能够检测 到 SASL 的存在. 执行 `svn --version` 时, Subversion 客户端命令行工具将报告 Cyrus SASL 是否可用. 如果你安装 的是已经编译好了的二进制包, 那你就要检查 SASL 是否被编译到了安装 包里.
>
> SASL 包含大量可插拔的模块, 这些模块代表了不同的认证机制: Kerberos (GSSAPI), NTLM, One-Time-Passwords (OTP), DIGEST-MD5, LDAP, Secure-Remote-Password (SRP) 等. 某些机制在你的系统中可能 无法使用, 使用前检查系统中的 SASL 提供了哪些模块.
>
> Cyrus SASL 源代码及其文档的下载地址是 http://asg.web.cmu.edu/sasl/sasl-library.html.

正常情况下, 当 Subversion 客户端连接到 *svnserve* 时, 服务器以宣告它所支持的功能作为响应. 如果服务器的配置要求认证, 服务器将向客户端发起认证请求, 并列出它所支持的认证机制, 客户端从中选择 一种认证机制, 通过几个往返消息携带认证信息. 即使 SASL 不可用, 客户端 和服务器也能使用内建的 CRAM-MD5 和 ANONYMOUS 认证机制 (见 “内建的认证与授权”一节). 如果服务器和客户端 支持 SASL, 那么可供选择的认证机制就比较多, 但是管理员必须在服务器 端显式地配置 SASL, 服务端才能向客户端宣告这些认证机制是可用的.

使用 SASL 进行认证

为了在服务器上激活 SASL, 管理员要做两件事. 首先, 在仓库的 *svnserve.conf* 里创建 `[sasl]` 节, 并为变量 `use-sasl` 赋值:

```
[sasl]
use-sasl = true
```

然后, 在 SASL 库函数能够找到的位置创建一个名为 *svn.conf* 的配置文件—最典型的位置就是 SASL 插件所处的位置, 因此管理员需要定位 SASL 插件在系统中的位置, 例如 */usr/lib/sasl2/* 或 */etc/sasl2/*. (注意, 本段所说的配置文件是 *svn.conf*, 不是仓库中的 *svnserve.conf*!).

如果服务器的系统是 Windows, 你需要编辑系统注册表 (使用工具 *regedit*), 以便告诉 SASL 去哪里搜索所需要的 文件. 在系统注册表中添加一个新的注册表项, 表项的名字是 `[HKEY_LOCAL_MACHINE\SOFTWARE\CarnegieMellon\Project Cyrus\SASL Library]`, 并在其中新增两项: 一项是 `SearchPath` (它的值是一个 指向目录的路径, 目录包含了 SASL 动态链接库), 另一项是 `ConfFile` (它的值是一个指向目录的路径, 目录内含有 管理员创建的 *svn.conf* 文件).

因为 SASL 提供了多种不同的认证机制, 描述每一种可能的服务器端 配置是不切实际的 (而且也超出了本书的范围), 所以我们建议读者自己去 阅读 SASL 源代码目录内, *doc* 子目录内的文档, 文档详细介绍了每一种认证机制, 以及如何正确地

配置服务器，以便使用 这些认证机制. 为了方便讨论，我们将介绍一个配置 DIGEST-MD5 的简单 示例. 如果你的 *svn.conf* 含有以下内容：

```
pwcheck_method: auxprop
auxprop_plugin: sasldb
sasldb_path: /etc/my_sasldb
mech_list: DIGEST-MD5
```

上面的配置向客户端宣告了 DIGEST-MD5 认证机制，使用存放在 */etc/my_sasldb* 里的私有密码数据库检查用户 输入的密码的正确性. 管理员可以使用 *saslpasswd2* 向密码数据库添加或修改用户名和密码：

```
$ saslpasswd2 -c -f /etc/my_sasldb -u realm username
```

有些地方需要注意：首先要确保 *saslpasswd2* 的 “认证域” 参数和定义在 *svnserve.conf* 里的认证域是一致的，如果它们不一致，认证将会失败. 另外，受限于 SASL，认证域必须是不带空格的字符串. 最后，如果你决定使用标准的 SASL 密码数据库，需要确保进程 *svnserve* 对数据库文件具有读 权限（某些认证机制—例如 OTP—还会要求写权限）.

这只是一种配置 SASL 的简单方式. 还有其他多种认证机制可供选择，密码也能以其他格式存在，例如 LDAP 或 SQL 数据库，具体的细节请参考 SASL 文档.

注意， 如果管理员将服务器配置成仅允许使用 SASL 认证机制， 这同时 也在要求所有连接到服务器的客户端必须支持 SASL，不支持 SASL 的客户 端（包括 1.5 版之前的所有客户端）将无法完成认证，但是另一个方面，这种配置也正是你所想要的效果（“所有的客户端都必须使用 Kerberos!”). 然而，如果仍然存在不支持 SASL 的客户端需要 连接服务器，就要确保 CRAM-MD5 认证机制是可用的，因为所有的客户端 都支持 CRAM-MD5.

SASL 加密

如果特定的机制支持，那么 SASL 也能实现数据加密. 内建的 CRAM-MD5 不支持加密，但 DIGEST-MD5 支持，有些机制（例如 SRP) 还会用到 OpenSSL 函数库. 为了开启或禁止加密的不同级别，你需要在仓库的 *svnserve.conf* 里定义两个值：

```
[sasl]
use-sasl = true
min-encryption = 128
max-encryption = 256
```

变量 `min-encryption` 和 `max-encryption` 决定加密的级别. 为了完全禁止加密，就把两个变量都设为 0. 为了开启简单的数据检验（即防止数据被篡改，保证数据的完整性，但没有对数据进行加密），把两个变量都设为 1. 如果 管理员希望允许—但并非强制—加密，就把 `min-encryption` 设为 0，把 `max-encryption` 设为稍微大点的值. 为了强制要求 对数据进行加密，把两个变量都设为大于 1 的数. 在上面的例子里，我们 要求客户端的加密至少为 128 位，但不多于 256 位.

SSH 隧道

svnserve 内建的认证机制（和 SASL) 使用起来 非常方便，因为它避免了创建真正的系统账户. 但另一方面，管理员可能 已经建立了一套完善的 SSH 认证框架，项目所有的开发人员都拥有自己的 系统账户，而且能够通过 SSH 登录到服务器.

结合使用 SSH 和 *svnserve* 比较简单，客户端只要 用 `svn+ssh://` 形式的 URL 连接服务器即可：

```
$ whoami
harry
```

```
$ svn list svn+ssh://host.example.com/repos/project
harryssh@host.example.com's password:  *****

foo
bar
baz
…
```

在上面的例子里，Subversion 客户端唤起一个本地的 *ssh* 进程，连接到 `host.example.com`，作为用户 `harryssh`（根据 SSH 用户配置）进行认证，然后在远程的服务器中，以用户 `harryssh` 的身份派生 一个私有的 *svnserve* 进程. 命令 *svnserve* 在隧道模式 (`-t`) 下执行，它的网络协议行走在由 *ssh*—隧道代理—提供 的加密通道中. 如果客户端执行一个提交操作，认证过的用户名 `harryssh` 将作为新版本号的作者.

这里需要强调的一点是 Subversion 客户端 *并没有* 连接到运行着的 *svnserve* 守护进程，这种访问方式不 要求 *svnserve* 守护进程存在，即使存在也不会被注意 到. 它完全依赖 *ssh* 临时派生的 *svnserve* 进程，当网络连接关闭时，*svnserve* 进程就会终止.

当使用 `svn+ssh://` URL 访问仓库时，要记住提出 认证要求的程序是 *ssh*，而 *不是* 客户端程序 *svn*，这就意味着不会出现密码缓存（见 “缓存证书”一节). Subversion 客户端经常向仓库发起多个连接，由于密码缓存，用户通常不会注意到这点，然而，当用户使用 `svn+ssh://` 连接仓库时，客户端每发起 一次连接，*ssh* 都会要求用户输入密码，用户可能会对 此感到恼怒. 解决问题的办法是使用一个单独的 SSH 密码缓存工具，例如 类 Unix 系统中的 *ssh-agent*，或 Windows 系统中的 *pageant*.

在隧道模式下操作时，授权主要由仓库数据文件的操作系统权限控制，这和 Harry 使用 `file://` 直接访问仓库的情形基本一致. 如果有多个系统用户会直接访问仓库，管理员可能想把他们都放到 一个用户组里，同时还要注意文件模式创建屏蔽字（记得阅读本章后面的 “支持多种仓库访问方法”一节). 即使在隧道模式下，你也可以使用 *svnserve.conf* 屏蔽特定的访问方式，只需要设置 `auth-access = read` 或 `auth-access = none`.[2]

读者可能以为关于 SSH 隧道的内容就此结束，然而并没有. Subversion 允许用户在运行时配置文件 *config*（见 “运行时配置区域”一节) 里创建定制化的隧道行为. 例如使用 RSH，而不是 SSH，[3] 具体的配置方式是在 *config* 的 `[tunnels]` 节添加如 下内容：

```
[tunnels]
rsh = rsh --
```

现在，为了使用新的隧道配置，把访问仓库的 URL 模式更改为 `svn+rsh://`，例如 `svn+rsh://host/path`，此时 Subversion 客户端相当于 在执行 `rsh -- host svnserve -t`. 如果用户在 URL 中包含了用户名（例如 `svn+rsh://username@host/path`，则客户端也会在待执行的命令中包含用户名 (`rsh -- username@host svnserve -t`).

注意，在定义基于 RSH 的隧道时，我们在隧道命令行添加了选项结束 参数 `--`，这是为了避免把一个错误的主机名当成隧道 命令的另一个选项，使用其他隧道程序时也要考虑这个问题（例如 SSH).

不过，你也可以定义更加灵活的隧道方案：

```
[tunnels]
```

[2] 注意，*svnserve* 施加的访问限制只有在以下情况中才是 有效的：用户无法旁路掉 *svnserve* 限制的条件，并且没有使用其他工具（例如 *cd* 和 *vi*) 直接访问仓库. 关于如何实现这些访问限制，见 “控制被调用的命令”一节.

[3] 实际上我们不推荐使用 RSH，因为它的安 全性远不如 SSH.

```
joessh = $JOESSH /opt/alternate/ssh -p 29934 --
```

这个例子有两点需要说明，首先它展示了 Subversion 客户端如何启动 一个特定的隧道程序（例子里是 */opt/alternate/ssh*），并带有特定的选项. 此时，访问 URL `svn+joessh` 将启动 一个特定的 SSH 程序，并带有参数 `-p 29934`—如果 你希望隧道程序连接到一个非标准的端口，那么这种方法就比较方便.

第二，例子展示了如何使用环境变量去覆盖隧道程序的名字. 通过设置 环境变量 `SVN_SSH` 来覆盖默认的 SSH 隧道代理是一种 方便的做法，但是如果你希望不同的服务器使用不同的变量覆盖，甚至在连接 每个服务器时，其端口和选项也不尽相同，这时候就要用到本例所介绍的方法. 如果用户设置了环境变量 `JOESSH`，它的值将会覆盖掉 变量 `joessh` 原来的值—Subversion 客户端将会 执行 *$JOESSH*，而不是 `/opt/alternate/ssh -p 29934`.

SSH 配置技巧

除了可以控制客户端执行 *ssh* 的方式外，还能 控制服务器上的 *sshd* 的行为. 本节我们将介绍如何 控制由 *sshd* 启动的 *svnserve*，以及多个用户如何共享一个系统账号.

初始化设置

首先，先确定你将用来启动 *svnserve* 的账户 的家目录. 确保账户已经安装了 SSH 公钥与私钥，并且用户可通过公钥 认证进行登录. 密码认证将无法工作，因此下面将要介绍的 SSH 技巧 全都是在围绕 SSH *authorized_keys* 文件.

如果 *authorized_keys* 事先不存在，直接创建 即可（在 Unix 系统中，它的典型位置是 *~/.ssh/authorized_keys* ）. 文件的每一行都描述了一个允许连接的公钥，行的典型样式为：

```
ssh-dsa AAAABtce9euch… user@example.com
```

第一列描述密钥的类型，第二列是 base64 编码的密钥，第三列是注释. 除了这三列，其实还可以添加一个 `command` 字段：

```
command="program" ssh-dsa AAAABtce9euch… user@example.com
```

如果含有 `command` 字段，SSH 守护进程将会执行 该字段所指定的程序，而不是 Subversion 客户端所请求的以隧道模式 启动的 *svnserve*. 这种行为允许我们实现多种服务器 端技巧，在下面的例子里，我们把 *authorized_keys* 的每一行简写为：

```
command="program" TYPE KEY COMMENT
```

控制被调用的命令

因为我们可以指定在服务器端被执行的命令，所以很容易就能指定 一个特殊的 *svnserve* 程序，并给它传递额外的参数：

```
command="/path/to/svnserve -t -r /virtual/root" TYPE KEY COMMENT
```

在上面的例子里，*/path/to/svnserve* 可能是 一个定制化的 *svnserve* 包装脚本，脚本将会重新 设置文件权限掩码（见 “支持多种仓库访问方法”一节）. 例子还展示了如何修改 *svnserve* 的文件系统根目录，当以守护进程方式运行 *svnserve* 时，修改进程的根 目录是很常见的操作，这么做可以是为了限制用户对系统目录空间的访问，也可以是为了在输入 `svn+ssh://` URL 的路径参数时，减少用户打字的工作量.

多个用户共享同一个账户也是有可能的， 方法是为每一个用户生成一对 公钥与私钥， 然后把每个公钥的内容都写入文件 *authorized_keys* 内，每行一个，并使用选项 `--tunnel-user`：

```
command="svnserve -t --tunnel-user=harry" TYPE1 KEY1 harry@example.com
command="svnserve -t --tunnel-user=sally" TYPE2 KEY2 sally@example.com
```

上面的例子允许 Harry 和 Sally 使用相同的账户，通过各自的公钥 认证来连接服务器. 每一行都指定了一条待执行的命令，选项 `--tunnel-user` 告诉 *svnserve* 它的参数是已认证的用户，如果没有加上 `--tunnel-user`，那么 *svnserve* 会认为所有的提交都来自被共享的 账户.

最后一点需要提醒的是：如果一个用户可通过共享账户的公钥访问 服务器，即使在 *authorized_keys* 里设置了 `command`，也可能仍然允许其他形式的 SSH 访问. 例如 用户仍然能够通过 SSH 获取 shell 访问权限，或者 X11 窗口，或者一般 性的端口转发. 为了使用户的权限尽可能得小，在 `command` 后面添加一些限制选项：

```
  command="svnserve -t --tunnel-user=harry",no-port-forwarding,no-agent-forw
arding,no-X11-forwarding,no-pty TYPE1 KEY1 harry@example.com
```

注意上面的内容必须写在同一行内，因为 *authorized_keys* 不支持通过反斜杠来实现行的 延续，例子里的换行只是为了方便排版.

svnserve 配置参考

上一节提到了 *svnserve.conf* 支持的众多选项，利用这些选项，当用户通过 *svnserve* 服务器访问 Subversion 时，管理员就能够实现对 Subversion 行为的控制. 本节将对 *svnserve* 支持的 *所有* 选项 进行一个总结.

配置文件 *svnserve.conf* 的格式是典型的 INI 风格，选项是一对 名字/值，通过带名字的节进行分组.（这种格式和 Subversion 客户端的运行时配置所使用的格式相同.）这里将会介绍配置 文件里的每一节，及其支持的各个选项.

默认情况下，*svnserve* 会查阅每个仓库的 *conf/svnserve.conf*，为了让一个 *svnserve* 的运行实例访问到的所有仓库都使用同一个 配置文件，就给它添加选项 `--config-file`.

在下面的几节里，我们将使用正式名称 *svnserve.conf* 指代 *svnserve* 配置文件，实际上配置文件还可以取其他名字，但我们相信读者不会感到 迷惑.

通用配置

`[general]` 节包含了最常用到的 *svnserve* 配置选项.

`anon-access`

控制未认证的（匿名的）用户的访问权限，有效值包括 `write`, `read`, 和 `none`, 其中 `read` 是 默认值.

`auth-access`

控制已认证的用户的访问权限，有效值包括 `write`, `read` 和 `none`, 其中 `write` 是 默认值.

`authz-db`

指定仓库访问权限配置文件的路径（见 “基于路径的访问控制”一节). 如果是一个普通的本地路径，除非路径以正斜杠 (`/`) 开始，否则的话路径就看成是相对于包含 了 *svnserve.conf* 的目录的相对路径. 如果没 有指定路径，将禁止基于路径的访问权限控制.

作为一种特殊的情况，可以把 Subversion 仓库内的文件指定为 访问权限配置文件，使用本地 URL（以 `file://` 开始）指定文件的位置. 另外，还可以用相对的仓库 URL（以 `^/` 开始），使得 *svnserve* 根据相对 URL 访问仓库内的访问权限配置文件.

`force-username-case`

在和访问权限配置文件（由选项 `authz-db` 指定）里的规则比较之前，指定用户名的大小写形式，有效值包括 `upper`（用户名的大小写形式），`lower`（用户名的小写形式）和 `none`（不改变用户名的大小写形式）. 默认情况下，*svnserve* 不改变用户名的大小写形式.

`groups-db`

指定组文件的路径. 如果是一个普通的本地路径，除非路径以正 斜杠 (`/`) 开始，否则的话路径就看成是相对于 包含了 *svnserve.conf* 的目录的相对路径.

还可以把 Subversion 仓库内的文件指定为组文件. 使用本地 URL（以 `file://` 开始）指定文件的位置. 另外，还可以用相对的仓库 URL（以 `^/` 开始），使得 *svnserve* 根据相对 URL 访问仓库内的组文件.

`hooks-env`

指定钩子脚本环境配置文件的路径. 该选项覆盖了文件在每个 仓库内的默认位置，如果写成绝对路径，就可以用同一个文件为多个 仓库的钩子脚本环境进行配置. 除非写成绝对路径，否则的话就看成 是相对于包含了 *svnserve.conf* 的目录的 相对路径.

关于钩子环境配置文件的更多信息，见 “钩子脚本环境配置”一节.

`password-db`

指定密码文件的路径. 除非路径以正 斜杠 (`/`) 开始，否则的话路径就看成是相对于 包含了 *svnserve.conf* 的目录的相对路径. 注意，如果使用了 SASL 特性，则该选项将被忽略.

`realm`

指定仓库的认证域. 该选项主要被客户端使用，用来关联缓存的 认证证书和特定的某个或某些仓库，正因为如此，除非多个仓库使用 了相同的密码数据库，否则的话，最好把每个仓库的认证域都设置 成独一无二的值. 仓库认证域的默认值是它的 UUID.

Cyrus SASL 配置

`[sasl]` 节包含了专门针对 *svnserve* 可选特性 SASL (Cyrus Simple Authentication and Security Layer) 的配置，关于 SASL 更详细的信息 以及它的益处，见 “svnserve 使用 SASL”一节.

`max-encryption`

指定安全层加密算法最大的期望长度—整数的二进制位数. `0` 表示 “不加密”，`1` 表示 “只检查完整性”，默认值是 `256` (256 位加密).

`min-encryption`

指定安全层加密算法最小的期望长度—整数的二进制位数. 特殊值 `0` 表示 “不加密”，`1` 表示 “只检查完整性”，默认值是 `0`（不加密）.

`use-sasl`

指定是否开启 Cyrus SASL 特性 (`true` 或 `false`). 注意，只有在编译 *svnserve* 时添加了对 SASL 的支持，才能开启 该特性. 默认值是 `false`.

httpd, Apache HTTP 服务器

Apache HTTP 服务器是 Subversion 可使用的 “重型” 网络 服务器. 借助一个定制的模块, *httpd* 允许客户端通过 WebDAV/Delta 协议 [4] 访问 Subversion 仓库, WebDAV/Delta 协议是 HTTP 1.1 的扩展. WebDAV/Delta 在万维网核心 协议 HTTP 的基础上, 增加了写功能—确切地说, 是版本化的写. 这样做的 结果是得到了一个标准化的, 健壮的软件系统, 可以方便地作为 Apache 2.0 软件的一部分进行打包, 受到多种操作系统和第三方软件的支持, 也不要求网络 管理员开通额外的端口. [5] 因为 Apache-Subversion 服务器比 *svnserve* 拥有更多的特性, 因此设置起来会更加困难—灵活性往往伴随着复杂性.

下面将要介绍的很多内容都包含了关于 Apache 配置指令的引用, 虽然某些 例子用到了 Apache 的配置指令, 但完整地介绍它们已经超出了本章的范畴. Apache 团队维护了非常优秀的文档供用户参考, 可以到它的官网 http://httpd.apache.org 获取, 例如, 关于 Apache 配置指令的 文档在 http://httpd.apache.org/docs/current/mod/directives.html.

另外, 在管理员修改 Apache 设置的过程中, 有可能会有错误发生, 如果你 还不太熟悉 Apache 的日志子系统, 现在应该着手熟悉它. 文件 *httpd.conf* 可以指定 Apache 所生成的访问与错误 日志的存放位置 (配置指令分别是 `CustomLog` 和 `ErrorLog`). Subversion 的 *mod_dav_svn* 也用到了 Apache 的错误日志接口. 管理员 可以通过查看这些日志文件定位问题发生的原因.

先决条件

为了能让用户使用 HTTP 协议访问仓库, 你需要 4 项组件, 包含在 2 个 软件包里. 你需要 Apache *httpd* 2.0 或更新的版本, DAV 模块 *mod_dav* (包含在 *httpd* 软件包里), Subversion, 以 及随 Subversion 软件包一起发布的 *mod_dav_svn* 模块. 这些组件一旦准备完毕, 为仓库添加 HTTP 网络访问能力的步骤就简单 了:

- 为 httpd 加载 *mod_dav*, 并启动 httpd
- 安装 *mod_dav_svn*, 它将使用 Subversion 的 库函数来访问仓库
- 修改 *httpd.conf*, 以便导出 Subversion 仓库

前面两步你可以从源代码编译安装 *httpd* 和 Subversion, 或者安装它们的二进制包来完成. 关于如何编译 Subversion, 以便支持 Apache HTTP Server, 以及如何配置 Apache, 见 Subversion 源代码顶层目录下的 *INSTALL* 文件.

Apache 基本配置

所有组件安装完毕后, 剩下的工作就是通过 *httpd.conf* 配置 Apache. 为了让 Apache 加载 *mod_dav_svn*, 要用到配置指令 `LoadModule`, 这条配置指令必须出现在任何与 Subversion 有关的配置项之前. 如果你是按照默认的布局来安装 Apache, 则

[4] 见 http://www.webdav.org/.

[5] 他们真得很讨厌这样做.

mod_dav_svn 会被安装到 Apache 安装目录 (通常是 */usr/lib64/httpd/*) 的 *modules* 子目录内. 配置指令 `LoadModule` 的语法非常简单, 包含模块名及其共享库 文件的路径:

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Apache 将 `LoadModule` 共享库文件的路径解释成 相对于服务器进程根目录的路径. 对于上面的例子而言, Apache 将会在它的 *moduels/* 子目录内搜索 Subversion DAV 模块的 共享库文件. 取决于 Subversion 在系统中的安装方式, 你可能需要指定 不同的路径, 甚至像下面这样的绝对路径:

```
LoadModule dav_svn_module     C:/Subversion/libexec/mod_dav_svn.so
```

如果 *mod_dav* 被编译成一个共享库文件 (而不是 被直接编译进 *httpd* 二进制文件里), 那么它也需要 一个类似的 `LoadModule` 指令, 注意, 它要出现在 *mod_dav_svn* 加载指令的前面:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

在配置文件的后面, 你需要把 Subversion 仓库的位置告诉给 Apache. 指令 `Location` 具有与 XML 类似的格式, 它以开标签 开始, 以闭标签结束, 在开标签和闭标签之间可以包含多个配置指令. `Location` 的目的是在处理指定的 URL 及其子路径上 的请求时, 做一些特殊的操作. 对于 Subversion 而言, 就是希望 Apache 将指向 Subversion 仓库的请求交由 DAV 层进行处理. 下面的例子告诉 Apache, 如果 URL 的路径部分 (URL 中, 跟在服务器名和端口号后面的部分) 以 */repos/* 开始, 就把请求交由 DAV 处理:

```
<Location /repos>
  DAV svn
  SVNPath /var/svn/repository
</Location>
```

如果你计划支持多个位于同一父目录下的 Subversion 仓库, 可以 使用指令 `SVNParentPath` 指明公共的父目录. 例如, 如果你将会在 */var/svn* 目录下创建多个 Subversion 仓库, 访问这些仓库的 URL 是 `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2` 等, 那你就可以在 *httpd.conf* 里这样写:

```
<Location /svn>
  DAV svn

  # Automatically map any "/svn/foo" URL to repository /var/svn/foo
  SVNParentPath /var/svn
</Location>
```

利用这种语法, Apache 将会把路径部分以 */svn/* 开始的 URL 的处理代理给 Subversion DAV, 它将假设由 `SVNParentPath` 所指定的目录内的所有子目录都是 Subversion 仓库. 相对于 `SVNPath`, 使用 `SVNParentPath` 更加方便, 因为在添加或删除仓库 时不用重启 Apache.

需要注意的是在定义新的 `Location` 时, 不要和 其他已有的 `Location` 重叠. 比如说 `DocumentRoot` 被导出到 */www*, 那就不要再导出 `<Location /www/repos>` 内的 Subversion 仓库, 因为如果 Apache 接到一个访问 */www/repos/foo.c* 的请求, 它就没办法确认这是 `DocumentRoot` 内的 *repos/foo.c*, 还是代理给 *mod_dav_svn*, 并由它返回 Subversion 仓库内的 *foo.c*, 这种错误通常的结果是返回一个 `301 Moved Permanently` 响应.

服务器名和 COPY 请求

Subversion 使用请求类型 `COPY` 来完成服务器 端的文件和目录的复制. 作为 Apache 模块完整性检查的一部分, 复制的源和目标必须在同一台主机上, 为了满足这个要求, 你需要把服务器的主 机名告诉给 *mod_dav*. 在 *httpd.conf* 里, 可以用指令 `ServerName` 完成:

```
ServerName svn.example.com
```

如果你通过指令 `NameVirtualHost` 使用了 Apache 的虚拟托管支持, 那你可能需要使用指令 `ServerAlias` 指定服务器已经知晓的其他名字. 再次 强调, 详细的信息请参阅 Apache 文档.

现在, 你必须认真考虑与权限有关的问题. 如果 Apache 已经作为你的 网页服务器运行了一段时间, 服务器上可能积累了一定量的内容—网页, 脚本等, 这些文件的权限配置允许 Apache 对它们进行访问. 当 Apache 作为 Subversion 服务器时, 也要求 Subversion 仓库的读写权限配置正确.

你需要确定一种权限设置, 以便满足 Subversion 的需求, 而不影响已 有的网页或脚本. 这可能意味着修改 Subversion 仓库的权限, 以便与 Apache 提供的其他服务一致, 或者是使用 *httpd.conf* 的 `User` 与 `Group` 指令, 去指定 Apache 运行时的用户名与用户组, 这些用户名与用户组正是 Subversion 仓库的所有者. 正确设置权限的方法不是唯一的, 每一个管理员都可以选择一 种适合自己的方式, 只是需要注意的是, Subversion 搭配 Apache 的最常见 问题就是与权限有关的问题.

认证选项

到这里为止, 如果你的 *httpd.conf* 包含了类似 下面的内容:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn
</Location>
```

那么你的仓库对于外界而言是 “匿名” 访问的. 除非你为 Subversion 仓库配置了认证与授权策略, 否则的话, 那些通过指令 `Location` 指定的仓库将对所有人开放. 换句话说就是:

- 任何人都可以用 Subversion 客户端, 根据仓库 (或者它的子目录) 的 URL 检出工作副本.
- 任何人都可以用网页浏览器浏览仓库的最新内容.
- 任何人都可以向仓库提交修改.

当然, 你可能已经设置好了一个 pre-commit 钩子, 以便阻止那些不符合 要求的提交 (见 “实现仓库钩子”一节). 但是如果你接 着读下去, 就会发现其实我们还可以使用 Apache 的内建机制, 以一种特定 的方式来限制访问.

认证可以阻止无效的用户直接访问仓库, 但是它无法保护有效用户 的网络流量的隐私. 为服务器配置 SSL 加密, 可以为用户添加额外的一层 保护, 关于如何配置 SSL 加密, 见 “使用 SSL 保护网络流量”一节.

Basic 认证

对客户端进行认证的最简单的方式是使用 HTTP Basic 认证机制，它 仅仅是使用用户名与密码验证用户的身份. Apache 提供了命令行工具 *htpasswd*[6] 来管理包含用户名与密码的文件.

Basic 认证 非常 不安全，因为它以几乎明 文的方式在网络上传输密码，建议使用更加安全的 Digest 认证机制，具体的细节见 “Digest 认证” 一节.

首先创建一个密码文件，并为用户 Harry 和 Sally 授予访问权限.

```
$ ### First time: use -c to create the file
$ ### Use -m to use MD5 encryption of the password, which is more secure
$ htpasswd -c -m /etc/svn-auth.htpasswd harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth.htpasswd sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

然后，确保 Apache 可以访问到提供 Basic 认证和相关功能的模块：*mod_auth_basic*, *mod_authn_file* 和 *mod_authz_user*. 在大部分情况下，这些模块本来 就已经被编译进 *httpd*, 如果没有，你可能需要显 式地使用配置指令 `LoadModule` 加载它们：

```
LoadModule auth_basic_module   modules/mod_auth_basic.so
LoadModule authn_file_module   modules/mod_authn_file.so
LoadModule authz_user_module   moduels/mod_authz_user.so
```

确定 Apache 具备必需的功能后，接下来你就可以在 `<Location>` 块内添加必要的配置指令，告诉 Apache 你想用哪一种认证类型，例如：

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Basic
  AuthName "Subversion repository"
  AuthType Basic
  AuthBasicProvider file
  AuthUserFile /etc/svn-auth.htpasswd
</Location>
```

这些配置指令的意义是：

- `AuthName` 是你为认证域所选定的一个任意的 名字，大多数浏览器会在提示用户输入用户名与密码的对话框上显示 这个名字.

[6] 见 http://httpd.apache.org/docs/current/programs/htpasswd.html.

- `AuthType` 指定认证的类型.
- `AuthBasicProvider` 指定由谁来提供 Basic 认证，我们的例子里写得是一个本地密码文件.
- `AuthUserFile` 指定密码文件的路径.

然而，这个 `<Location>` 并没有做任何有 用的工作，它仅仅是告诉 Apache 如果 授权是必须的，它应该要求 Subversion 客户端提供用户名与密码.（如果有需要，Apache 自己也会要求认证.）然而这里还有不完善的地方，那就是告诉 Apache 哪些客户请求才需要授权，当前的配置是所有的请求都不需要认证，此时最 简单的配置就是通过添加 `Require valid-user` 要求 所有的 客户端请求都需要认证：

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Basic
  AuthName "Subversion repository"
  AuthType Basic
  AuthBasicProvider file
  AuthUserFile /etc/svn-auth.htpasswd

  # Authorization: Authenticated users only
  Require valid-user
</Location>
```

关于配置指令 `Require` 的更多细节，以及设置 授权策略的其他方式，请参考 “授权选项”一节.

`AuthBasicProvider` 的默认值是 `file`，所以我们不会在后面的例子里显式地写出 来. 但有一点需要注意， 如果在更广的上下文内你已经把 `AuthBasicProvider` 设置成了其他值， 那就需要在 Subversion 的 `<Location>` 内再显式地把 `AuthBasicProvider` 设置成 `file`.

Digest 认证

Digest 认证比 Basic 认证更加完善，它允许服务器验证客户客户端 的身份，而不会在网络上以明文的形式传输密码. 服务器和客户端会使用 不可逆的 MD5 算法计算敏感信息的散列值，敏感信息包括用户名，密码，所请求的 URI，由服务器生成的 *nonce*（一次性数字，每次认证时都会变化）. 客户端把散列值发给服务器，服务器 验证散列值是否正确.

为 Apache 配置 Digest 认证非常简单，首先你要确保模块 *mod_auth_digest*（而不是 *mod_auth_basic*）是可用的，然后再在上面例子的基础 上做一些小改动：

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthDigestProvider file
  AuthUserFile /etc/svn-auth.htdigest
```

```
  # Authorization: Authenticated users only
  Require valid-user
</Location>
```

注意到 `AuthType` 现在被设置成了 `Digest`, 而且为 `AuthUserFile` 指定了一个不同的路径. Digest 认证所使用的文件格式与 Basic 认证的 不同, 这种格式的文件使用 Apache 的命令行工具 *htdigest* [7] 创建与管理, 而不是 *htpasswd*. Digest 认证也有 “认证域” 的概念, 它由配置指令 `AuthName` 指定.

Digest 认证指定认证信息来源的配置指令是 `AuthDigestProvider`, `AuthDigestProvider` 的默认值是 `file`, 所以例子里的 `AuthDigestProvider file` 并不是必需的, 除非你要覆盖从更广的配置上下文继承而来的, 与 `file` 不同的值.

可以像下面这样创建密码文件:

```
$ ### First time: use -c to create the file
$ htdigest -c /etc/svn-auth.htdigest "Subversion repository" harry
Adding password for harry in realm Subversion repository.
New password: *****
Re-type new password: *****
$ htdigest /etc/svn-auth.htdigest "Subversion repository" sally
Adding user sally in realm Subversion repository
New password: *******
Re-type new password: *******
$
```

授权选项

到这里为止, 你已经知道了如何配置认证, 但还没有提到授权. Apache 可 以要求客户端认证他们的身份, 但 Apache 还不知道如何允许或限制客户端 的访问权限, 本节将介绍两种控制客户端对仓库的访问权限的策略.

完全访问控制

访问控制最简单的形式是只授权特定的用户可以对仓库进行读取或读写.

你可以通过在 `<Location>` 添加 `Require valid-user`, 从而允许用户对仓库执行所有 可能的操作. 下面的例子只允许成功认证的客户端对 Subversion 仓库执行任意的操作:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest
```

[7] 见 http://httpd.apache.org/docs/current/programs/htdigest.html.

```
  # Authorization: Authenticated users only
  Require valid-user
</Location>
```

有时候，你并不需要这么严格的设置. 比如说托管 Subversion 源代码 的服务器 (https://svn.apache.org/repos/asf/subversion/) 允许所有人 对仓库执行只读操作（例如检出工作副本，浏览仓库等），但只允许认证 用户执行写操作. 配置指令 `Limit` 和 `LimitExcept` 可以实现这种有选择的访问限制，和 配置指令 `Location` 一样，前面两个配置指令也有开 标签和闭标签，管理员需要把它们放在 `<Location>` 内部.

`Limit` 和 `LimitExcept` 内的参数是 HTTP 请求类型，这些请求类型将会受到这两个配置指令的 影响. 比如说为了允许匿名的只读访问，管理员需要使用配置指令 `LimitExcept`（为指令添加请求类型参数 `GET`, `PROPFIND`, `OPTIONS` 和 `REPORT`), 还要把 前面提到的配置指令 `Require valid-user` 写到 `<LimitExcept>` 内.

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Authenticated users only for non-read-only
  #                (write) operations; allow anonymous reads
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

前面展示的只是一些很简单的例子，如果想知道关于 Apache 访问控制和 配置指令 `Require` 的更多细节，可以参考 Apache 文档的教程集合 (http://httpd.apache.org/docs-2.0/misc/tutorials.html) 中的 `Security` 部分.

每个目录的访问控制

还可以使用 Apache 模块 *mod_authz_svn* 进行 更细致的权限设置，该模块截取从客户端发往服务器的 URL，然后请求 模块 *mod_dav_svn* 对 URL 进行解码，根据定义在 配置文件里的访问策略，可能会禁止客户端的请求.

如果你是自己从源代码编译安装的 Subversion，那么默认情况下，模块 *mod_authz_svn* 将和 *mod_dav_svn* 一起被编译安装，许多二进制包也会自动安装这两个模块. 为了确认模块已 被正确地安装，在 *httpd.conf* 里将 *mod_authz_svn* 的加载放到 *mod_dav_svn* 之后：

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
LoadModule authz_svn_module   modules/mod_authz_svn.so
```

为了激活 *mod_authz_svn*，你需要在 `<Location>` 里，用配置指令 `AuthzSVNAccessFile` 指定一个文件，这个文件包含了仓库内各个文件路径的权限策略. 从 Subversion 1.7 开始，还可以用 配置指令 `AuthzSVNReposRelativeAccessFile` 指定 每个仓库各自的访问权限配置文件.（过一会儿，我们就会讨论该文件的 格式.）

Apache 非常灵活，所以你可以从 3 种通用模式中选择一种进行配置．首先，先选择一种基本配置模式．（下面介绍的例子非常简单，关于 Apache 认证与授权选择的更多细节，请参考 Apache 的文档．）

最开放的做法是允许所有人访问，这意味着 Apache 从不会要求客户 端进行认证，把所有的用户都当成 “匿名用户”．（见 例 6.2 “匿名访问的配置示例”．）

例 6.2. 匿名访问的配置示例

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: None

  # Authorization: Path-based access control
  AuthzSVNAccessFile /path/to/access/file
</Location>
```

相反，你可以要求 Apache 对所有的客户端进行认证，下面的配置 使用配置指令 `Require valid-user` 无条件地要求 认证，而且还指定了用户的认证方式．（见 例 6.3 “认证访问的配置示例”．）

例 6.3. 认证访问的配置示例

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Path-based access control; authenticated users only
  AuthzSVNAccessFile /path/to/access/file
  Require valid-user
</Location>
```

第三种常见的配置模式是同时允许认证与匿名访问．比如说很多管理 员通常会允许匿名用户读取特定的仓库目录，但较为敏感的区域仅允许 被认证用户访问．在这种配置下，所有用户首先以匿名身份访问仓库，在 任意时刻，如果你的访问控制策略要求使用真正的用户名，Apache 将向 客户端发起认证要求．为了实现这种配置，使用配置指令 `Satisfy Any` 和 `Require valid-user`．（见 例 6.4 “匿名/认证混合访问的配置示例”．）

例 6.4. 匿名/认证混合访问的配置示例

```
<Location /repos>
  DAV svn
```

```
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Path-based access control; try anonymous access
  #                first, but authenticate if necessary
  AuthzSVNAccessFile /path/to/access/file
  Satisfy Any
  Require valid-user
</Location>
```

下一步就是创建授权文件，文件内包含了访问仓库中特定路径的规则，我们将在 “基于路径的授权”一节 介绍 如何编写授权文件.

禁止基于路径的检查

模块 *mod_dav_svn* 会做大量的工作，以便确保 被管理员标记为 “不可读” 的数据不会被意外地泄漏，这意味着它需要仔细地监控由客户端命令（例如 *svn checkout* 和 *svn update*) 返回的路径和文件内容. 如果客户端命令遇到了一个它不可读的路径，该 路径就会被忽略. 对于历史或重命名追溯—例如对一个早就被重命名 过的文件执行 `svn cat -r OLD foo.c`—如果其中一个对象以前的名字被禁止读取，那么重命名追溯就会被终止.

有时候这些路径检查的代价将会非常高昂，特别是对 *svn log* 而言. 当检索一个版本号列表时，服务器查看每个版本号 内被修改的路径， 检查路径的可读性， 如果碰到一个不可读的路径， 它将不 会出现在版本号的修改路径列表里 （选项 `--verbose (-v)` 能够显示被版本号修改的路径)，整个日志消息也不 会显示出来. 不管怎么说，如果被版本号影响的文件数量非常大，那么这种 检查将会非常耗时. 这就是安全的代价：即使你根本就没有配置像 *mod_authz_svn* 这样的模块，模块 *mod_dav_svn* 仍然会要求 Apache *httpd* 对每一个路径执行授权检查. 模块 *mod_dav_svn* 无法知晓系统中已经安装了哪些授权模块，所以它就要求 Apache 调用可能存在的任意模块.

另一方面，管理员还可以牺牲部分安全，以换取速度. 如果你没有实施 任意类型的每目录授权（即没有使用 *mod_authz_svn* 或类似的模块)，就可以禁止这些路径检查，禁止方式是在 *httpd.conf* 里使用配置指令 `SVNPathAuthz`，见 例 6.5 “完全禁止路径检查” 所示.

例 6.5. 完全禁止路径检查

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  SVNPathAuthz off
</Location>
```

配置指令 `SVNPathAuthz` 的默认值是 `on`，当把它设置成 `off` 后，所有基于路径的授权检查都会被禁止，*mod_dav_svn* 也不会再在每个发现的路径上唤起授权检查.

存放在仓库内的访问权限配置文件

从 Subversion 1.8 开始，访问权限配置文件可以存放在仓库内，这个 仓库可以是应用了本文件的同一仓库，或者是另一个完全不同的仓库. 这个 功能为 Subversion 基于路径的授权配置添加了版本控制的特性.

配置指令 `AuthzSVNAccessFile` 和 `AuthzSVNReposRelativeAccessFile` 都可以用来指定 仓库内的访问权限配置文件的位置. 配置指令的参数既可以是表示绝对路径 的 `file://`，也可以是表示相对路径的 URL（以 `^/` 开始）.

比如说，可以像 例 6.6 “为多个仓库指定同一个位于仓库内的访问配置文件” 那样，为仓库内的访问权限配置文件指定一个绝对路径.

例 6.6. 为多个仓库指定同一个位于仓库内的访问配置文件

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn
  AuthzSVNAccessFile file:///var/svn/authzrepo/authz
</Location>
```

还可以像 例 6.7 “为每个仓库都指定一个仓库内的访问配置文件” 那样 为仓库内的访问权限配置文件指定一个相对路径.

例 6.7. 为每个仓库都指定一个仓库内的访问配置文件

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn
  AuthzSVNReposRelativeAccessFile ^/authz
</Location>
```

使用 SSL 保护网络流量

通过 `http://` 连接仓库意味着 Subversion 所 有的活动都会在网络上暴露无遗，也就是说像检出，提交和更新这些操作 都有可能被未授权的网络嗅探工具所拦截. 使用 SSL 加密网络流量是保护 敏感数据不在网络上泄露的常用方法.

如果 Subversion 客户端工具在编译时开启了 OpenSSL，它就可以使用 `https://` 形式的 URL 连接 Apache 服务器，于是所有 的网络流量都会使用每连接会话密钥进行加密. Subversion 客户端所使用的 WebDAV 函数库不仅可以验证服务器的证书，当服务器提出要求时，它也可 以为客户端提供证书.

Subversion 服务器 SSL 证书配置

如何为客户端和服务器生成 SSL 证书，以及如何配置 Apache 以便 使用这些证书，已经超出了本书的范畴，读者可参考 Apache 的文档 (http://httpd.apache.org/docs/current/ssl/).

[8]但是，自签署的证书仍然无法抵御 “中间人 攻击”（在客户端首次见到证书之前），和嗅探敏感数据相比，这种攻击更难防范.

来自知名组织的 SSL 证书通常需要花钱购买，但如果只需要满足最低 限度的要求，你可以让 Apache 使用自签署的证书，这种证书由 OpenSSL 生成. [8]

Subversion 客户端 SSL 证书管理

当使用 `https://` 形式的 URL 连接 Apache 时，Subversion 客户端将会收到两种类型的响应：

- 服务器证书
- 针对客户端证书的请求

服务器证书

当客户端收到服务器证书时，它需要验证服务器身份的真实性，OpenSSL 完成验证的方法是检查服务器证书的签发人，也就是 证书颁发机构 (*certificate authority*, 简称 CA). 如果 OpenSSL 无法自动信任 CA, 或者是发生的错误 (例如认证超时或主机名不匹配)，那么 Subversion 客户端工具将询问用户是否要信任服务器的证书：

```
$ svn list https://host.example.com/repos/project

Error validating server certificate for 'https://host.example.com:443':
 - The certificate is not issued by a trusted authority.  Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: host.example.com
 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56 2006 GMT
 - Issuer: CA, example.com, Sometown, California, US
 - Fingerprint: 7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b

(R)eject, accept (t)emporarily or accept (p)ermanently?
```

用户可能会在网页浏览器看到相同的对话框 (浏览器只是一个 HTTP 客户端)，如果选择 p, Subversion 将把 服务器证书缓存在本地的 *auth/* 目录内，你的用户名和密码也缓存在这里 (见 “缓存证书”一节), 今后再次 连接服务器时，将会自动信任证书.

运行时配置文件 *servers* 允许 Subversion 客户端自动信任特定的 CA, 信任既可以是全局的，也可以是基于每个主 机的，方法是用变量 `ssl-authority-files` 指定 PEM 编码的 CA 证书，证书之间用分号分开：

```
[global]
ssl-authority-files = /path/to/CAcert1.pem;/path/to/CAcert2.pem
```

很多 OpenSSL 安装包预定义了一套 “默认的” CA, 这些 CA 得到了非常普遍的信任. 为了让 Subversion 客户端自动 信任 OpenSSL 的证书，把变量 `ssl-trust-default-ca` 设置成 `true`.

客户端证书盘问

如果客户端收到一个证书请求，那便是服务器要求客户端提供它的 身份，客户端必须提供由 CA 签名过的证书，而该 CA 是服务器所信任 的，除了证书，还要发送一个 回应 (*challenge response*), 这个回应证明了客户 端拥有与证书关联的私钥. 私

钥和证书通常被加密后存放在本地磁盘上，被一个密码保护．当 Subversion 客户端收到证书的盘问时，它将询问 用户密钥与证书的存放路径，以及对应的密码：

```
$ svn list https://host.example.com/repos/project

Authentication realm: https://host.example.com:443
Client certificate filename: /path/to/my/cert.p12
Passphrase for '/path/to/my/cert.p12':  ********
```

在上面的例子里，客户端证书存放在一个 `.p12` 文件里．为了让 Subversion 使用证书，证书的格式必须是 PKCS#12, 这是一种可移植的标准格式，大多数网页浏览器支持导入或导出这种 格式的证书，除了浏览器，还可以用 OpenSSL 命令行工具把已有的 证书转换成 PKCS#12 格式．

运行时配置文件 *servers* 允许用户基于 每个主机，自动完成证书请求的响应．如果用户设置了变量 `ssl-client-cert-file` 和 `ssl-client-cert-password`, Subversion 将自动 响应证书请求，而不会提示用户：

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = /path/to/my/cert.p12
ssl-client-cert-password = somepassword
```

更注重安全的用户可能并不想设置变量 `ssl-client-cert-password`.

优化性能

Apache HTTP 服务器非常注重性能，不过你仍然可以通过修改配置为 Subversion 服务争取更高的性能．本节将介绍几种比较重要的配置修改，但是需要注意的某些 *httpd.conf* 配置选项将会影响 Apache 服务器的整体表现，而不仅仅是 Subversion 服务，因此管理员在 修改配置时，需要考虑对其他服务的影响．

KeepAlive

默认情况下，Apache HTTP 服务器允许为多个请求复用同一个连接，这对于 Subversion 而言非常有好处，因为 Subversion 在一个单独的 操作中，很可能会快速产生成百上千个请求，而重新打开一个服务器的 连接是一件颇费周张的事．在连接被服务器关闭之前，Subversion 会在同 一个连接内发送尽可能多的请求．配置指令 `KeepAlive` 用于开启或禁止连接重用功能，它的默认值是 `On`.

但是还有另一个配置指令用于限制客户端在一个单独的连接内，可以提交 的请求数量：`MaxKeepAliveRequests`, 它的默认值是 `100`. 对于版本较旧的 Subversion 而言，它的默认值 已经足够了，但是 Subversion 1.8 使用了不同的 HTTP 通信函数库（称为 Serf), 为了获取特定的零碎信息，Serf 更倾向于发送若干个小请求，而不是 请求服务器在一个单独的响应中，传回一大块数据．因此，我们建议把 至少把 `MaxKeepAliveRequests` 设置为 `1000`.

```
#
# KeepAlive: Whether or not to allow persistent connections (more than
# one request per connection). Set to "Off" to deactivate.
#
KeepAlive On
```

```
#
# MaxKeepAliveRequests: The maximum number of requests to allow
# during a persistent connection. Set to 0 to allow an unlimited amount.
# We recommend you leave this number high, for maximum performance.
#
MaxKeepAliveRequests 1000
```

批量更新

Subversion 1.8 客户端和它旧版之间最大的不同点在于更新操作 (*svn checkout*, *svn update*, *svn switch* 等) 的处理过程. 老版客户端使用 Neon HTTP 函数库实现通信, Neon 函数库更喜欢在一个单独的请求中, 向 服务器索要全部的信息. 管理员可能在他们的服务器日志里见到过这种日志: 先是一些握手操作, 然后是一个带有大量数据的 `REPORT` 请求, 这些数据就是整个的检出/更新数据集!

使用 Serf 函数库的 Subversion 客户端—版本大于或等于 Subversion 1.8—仍然会发送 `REPORT` 请求, 但在请求内会设置一些稍微不同的标志, 这些标志告诉服务器不用发送全 部的数据, 而是发送一个清单, 客户端随后根据清单向服务器请求更明确 的数据, 从而完成整个操作. 在服务器的 *access_log* 日志里, 这种 `REPORT` 请求后面会出现很多小块的 `GETS` 请求 (如果是旧版 Subversion, 则是 `PROPFIND` 请求).

上面的做法有好有坏. 批量更新的做法虽然在服务器上产生的日志更少, 但在操作运行的过程中, 被占用的 Apache HTTP 服务器子进程将无暇顾及 其他工作, 而当前操作可能还需要很长时间才能完成. 非批量更新提供了 设置内容缓存的机会 (缓存可用于提高性能), 但生成的服务器日志比批量 更新多得多. 无论是由于哪种原因, 管理员可能想对客户端施加更多的限制. Subversion 1.6 为 *mod_dav_svn* 添加了一个新的二 元配置指令 `SVNAllowBulkUpdates`, 用于配置服务器是否允许批量更新. 在 Subversion 1.8, `SVNAllowBulkUpdates` 的值除了 `On` 和 `Off` 外, 还新增了 `Prefer`, 如果 `SVNAllowBulkUpdates` 被设置为 `Prefer`, 受支持的客户端 (1.8 或更新的版本) 将尝试 使用批量更新, 除非另有指定.

其他好处

关于 Apache 和 *mod_dav_svn* 的认证与授权, 我 们已经介绍了大部分, 不过 Apache 还提供了一些非常有效的特性.

仓库浏览

Apache/WebDAV 最实用的功能之一是允许用户直接在网页浏览器上浏览 仓库内的文件与目录. 因为 Subversion 使用 URL 标识仓库内的文件, 基于 HTTP 的 URL 可以直接输入到网页浏览器的地址栏上, 然后浏览器 向服务器发送 HTTP `GET` 请求, 根据 URL 所指向的 资源是文件还是目录, *mod_dav_svn* 将返回目录内的 文件列表, 或文件的内容.

URL 语法

如果 URL 没有指定所请求的资源的版本, *mod_dav_svn* 将返回最新的版本, 这种做法最大的 好处是你可以把 Subversion URL (例如某篇文档的 URL) 发给其他同事, 而这些 URL 将始终指向文档的最新版. 当然, 你也可以把这些 URL 作 为超链接放到网站上.

从 Subversion 1.6 开始, *mod_dav_svn* 支持 一种用于查看旧版本文件与目录的 URL 语法. 这种语法使用 URL 的查询 字符串部分指定限定版本号和 (或) 实施版本号, Subversion 将会把这些 版本号对应的文件与目录显示到网页浏览器上. 为了指

定限定版本号，在 URL 的查询字符串部分添加 `p=PEGREV` 形式的 名字/值 对（其中，*PEGREV* 是一个版本号）；为了指定实施版本号，在 URL 的查询字符串部分添加 `r=REV` 形式的 名字/值 对（其中，*REV* 是一个版本号）.

比如说，你想查看 */trunk* 里的最新版的 *README.txt*，就在网页浏览器的地址栏里输入类似 于下面的 URL:

```
http://host.example.com/repos/project/trunk/README.txt
```

如果你想查看该文件的旧版，在 URL 的查询字符部分添加实施版本号：

```
http://host.example.com/repos/project/trunk/README.txt?r=1234
```

如果你想查看的文件在最新版中已经被删除了，那又该怎么办? 这 时候就要用到限定版本号：

```
http://host.example.com/repos/project/trunk/deleted-thing.txt?p=321
```

当然，你还可以结合使用限定版本号和实施版本号，更精细地指定待 查看的项目：

```
http://host.example.com/repos/project/trunk/renamed-thing.txt?p=123&r=21
```

上面的 URL 将显示对象在版本号 21 时的内容，这个对象在版本 号为 123 时，位于 */trunk/renamed-thing.txt*. 关于 “限定版本号” 和 “实施版本号” 的详细 介绍，见 “限定版本号与实施版本号” 一节，理解它们可能会 让你感到有点头晕.

从 Subversion 1.8 开始，*mod_dav_svn* 具有了 替换关键字的能力. 如果 *mod_dav_svn* 在文件的 URL 里发现了查询参数 `kw=1`，它将在递送文件内容 时，替换掉其中的关键字. 如果省略参数 `kw`，或是 赋予了除 `1` 之外的其他值，则 Subversion 将保持 默认行为，即在递送文件内容时，不替换其中的关键字.

通常情况下，关键字替换是作为工作副本管理工作的一部分，在客户 端执行，所以说在不使用工作副本的情况下，让服务器递送一份关键字被 替换后的文件是一件非常方便的事情.

比如说，你想查看位于项目目录 */trunk* 内，最新版本的 *README.txt*，而且还要求替换文件 内的关键字，则在 URL 的后面添加查询参数 `kw=1`:

```
http://host.example.com/repos/project/trunk/README.txt?kw=1
```

和客户端的关键字展开一样，只有在文件上设置属性 `svn:keywords` 后，被指定的关键字才会被替换，关于关键字替换的更多内容，见 “关键字替换” 一节.

作为提醒，*mod_dav_svn* 所能提供的仓库浏览 体验比较有限，你只能看到目录列表和文件内容，但无法看到版本号属性（例如日志消息） 或文件/目录属性. 如果用户需要更强大的仓库浏览功能，可以借助第三方软件，例如 ViewVC (http://viewvc.org), Trac (http://trac.edgewall.org) 和 WebSVN (http://websvnphp.github.io). 这些第三方软件不会影响 *mod_dav_svn* 的内建 “可浏览性”，而且还提供了更强大的功能，包括显示前面提到的属性，显示文件的两个 版本号之间的差异等.

合适的 MIME 类型

浏览一个 Subversion 仓库时，网页浏览器通过查看 `Content-Type` 来判断如何提供文件的内容. `Content-Type` 是 Apache 答复 HTTP `GET` 请求的消息中的一段头部信息，这段头部信息存 放 MIME 类型. 默认情况下，Apache 告诉浏览器所有的仓库文件的 MIME 类型都是默认值，通常是 `text/plain`. 然而，用户 有时候可能对这种处理不太满意，因为他希望仓库里的文件能够以更有意义 的方式呈现—比如说把 *foo.html* 按照 HTML 的格式进行显示.

为了让文件在浏览器内以更加合适的方式呈现，需要为文件设置 `svn:mime-type` 属性，详细的内容在 “文件内容类型” 一节，另外，还可以配置客户端，使得在第一次把文件存放到仓库是时，自动地为文件 设置合适的 `svn:mime-type` 属性，见 “自动属性设置” 一节.

继续我们的例子，如果 *foo.html* 的 `svn:mime-type` 属性已经被设置为 `text/html`, Apache 将告诉浏览器按照 HTML 格式 来显示 *foo.html*. 用户还可以为图片文件的 `svn:mime-type` 属性设置合适的 `image/*`, 最终得到一个可以直接从仓库中浏览的 网站！只要网站不包含任何动态生成的内容，这是完全可以做到的.

定制外观

通常来说，用户更经常使用普通文件的 URL—毕竟文件的内容 才是人们感兴趣的东西. 不过，在少数情况下用户仍然需要浏览 Subversion 的目录列表，此时用户可能会觉得目录列表的外观过于简单，缺乏美感（或者说无法引起人们的兴趣）. 为了允许对目录列表的外观进行修改，Subversion 提供了一个 XML 索引特性. 在仓库的 *httpd.conf* `Location` 配置块里，如果增加一个配置指令 `SVNIndexXSLT`, *mod_dav_svn* 在显示目录列表时，将生成一个 XML 输出，并引用到用户所选择的 XSLT 样式表：

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn
  SVNIndexXSLT "/svnindex.xsl"
  …
</Location>
```

利用配置指令 `SVNIndexXSLT` 和一个优秀的 XSLT 样式表，就可以让目录列表的显示风格与网站上的其他部分保持 一致. 如果用户愿意，还可以使用 Subversion 提供的样式表示例，样式表示例放在源码包的 *tools/xslt/* 目录内. 需要注意的是提供给配置指令 `SVNIndexXSLT` 的路径 实际上是一个 URL 路径—浏览器必须能够读取到样式表，这样才能 使用它们！

罗列仓库

如果你通过配置指令 `SVNParentPath`, 在一个 单独的 URL 上服务多个仓库，那么 Apache 就有可能在浏览器上列出所有的仓库，你所需要做的就是激活配置指令 `SVNListParentPath`:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn
  SVNListParentPath on
  …
</Location>
```

如果某个用户在浏览器上打开了 URL `http://host.example.com/svn/`，他将会看所有的 位于 */var/svn* 目录内的 Subversion 仓库. 这样 做显然不太安全，所以 `SVNListParentPath` 的默认 值是 `off`.

Apache 日志

Apache 本质上是一个 HTTP 服务器，它支持异常灵活的日志特性. 讨论 日志的所有配置方式已经超出了本书的范畴，但我们必须指出的是即使是最 普通的 *httpd.conf* 也会让 Apache 生成两种日志：*error_log* 和 *access_log*. 这些日志可能会出现在不同的地方，但通常都在 Apache 安装路径的日志 目录内.（如果是 Unix 系统，它们通常在 */usr/local/apache2/logs/*.）

文件 *error_log* 记录了 Apache 遇到的所有 内部错误，文件 *access_log* 则记录了 Apache 收 到的每一个 HTTP 请求. 利用这些日志，管理员就能很方便地看出 Subversion 客户端来自哪个 IP, 特定客户端访问服务器的频率高低，哪些用户认证成功，以及哪些请求成功或失败.

不幸的是，由于 HTTP 是一种无状态的协议，即使是最简单的 Subversion 客户端操作也会产生多个网络请求. 要想从 *access_log* 推论出客户端在做什么操作是一件非常困 难的工作—大多数操作看起来就像是一系列神秘的 PROPPATCH, GET, PUT 和 REPORT 请求. 更为严重 的是，很多客户端操作发送的是几乎相同的请求序列，这就使得分辨这些 请求变得更加困难.

还好，*mod_dav_svn* 可以帮到你. 为了激活特性 “操作日志” (operational logging), 管理员可以要求 *mod_dav_svn* 创建一个单独的日志文件，文件从较 高的层次描述了客户端所执行的操作.

为了让 *mod_dav_svn* 从较高的层次描述客户端所 执行的操作，管理员需要使用 Apache 的配置指令 CustomLog (关于该配置指令的详细信息可以参考 Apache 的文档). 注意，该配置指令必须出现在 Subversion 配置块 Location 的 外面：

```
<Location /svn>
  DAV svn
  …
</Location>

CustomLog logs/svn_logfile "%t %u %{SVN-ACTION}e" env=SVN-ACTION
```

在上面的例子里，我们请求 Apache 在标准的 *log* 目录下创建一个特殊的日志文件: *svn_logfile*. 变 量 %t 和 %u 分别被替换为请求 接到的时间和用户名，但其中最需要关注的是出现两次的 SVN-ACTION.当 Apache 看见 SVN-ACTION 时，它把该字符串替换为环境变量 SVN-ACTION 的值，每当 *mod_dav_svn* 检测到一个客户端操作时，它就自动设置 环境变量 SVN-ACTION 的值.

所以说，我们不用面对像下面这样复杂的 *access_log*:

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/vcc/default HTTP/1.1" 207 398
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/bln/59 HTTP/1.1" 207 449
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc HTTP/1.1" 207 647
[26/Jan/2007:22:25:29 -0600] "REPORT /svn/calc/!svn/vcc/default HTTP/1.1" 200 607
[26/Jan/2007:22:25:31 -0600] "OPTIONS /svn/calc HTTP/1.1" 200 188
[26/Jan/2007:22:25:31 -0600] "MKACTIVITY /svn/calc/!svn/act/e6035ef7-5df0-4ac0-b811-4be7c823f998
 HTTP/1.1" 201 227
…
```

你完全可以追求更容易理解的 *svn_logfile*:

```
[26/Jan/2007:22:24:20 -0600] - get-dir /tags r1729 props
[26/Jan/2007:22:24:27 -0600] - update /trunk r1729 depth=infinity
[26/Jan/2007:22:25:29 -0600] - status /trunk/foo r1729 depth=infinity
[26/Jan/2007:22:25:31 -0600] sally commit r1730
```

除了环境变量 SVN-ACTION, *mod_dav_svn* 还会设置 SVN-REPOS 和 SVN-REPOS-NAME, 这两个环境变量分别存放仓库的文件系统路径和路径的最后一个分量. 如果你想把多个仓库的日志写到同一 个日志文件里，那么你可能需要把这两个环境变量包含到 CustomLog 格式字符串里. 可被日志记录的操作的详尽 列表，见 “高层日志记录”一节.

显示，Apache 为 Subversion 所记录的日志越多，服务器上的磁盘消耗 也就越多，对于非常活跃的 Subversion 服务器而言，每天产生数兆大小的 日志并不少见. 只有当日志能被有效地处理时，它才是有价值的，而庞大的 日志文件很快就会让分析难以为继. 关于如何管理 Apache HTTP 服务器日志，有很多标准的做法可供参考，介绍它们已经超出了本书的范畴，管理员应该 选择一种最适合他们的日志轮换与归档方式.

如果 Subversion 只是产生了太多没什么用的日志，那又该怎么办？比如说在 “批量更新”一节 我们 说过 Subversion 客户端在执行检出和其他更新操作时，所采取的特定 方式将会导致服务器快速产生大量日志，因为请求每一段数据的日志都是单独记录 (Subversion 以前的版本可能不会这样做)．在这种情况下，管理 员可能需要配置 Apache，以便静默某些日志．

Apache HTTP 模块 *mod_setenvif* 提供了配置 指令 `SetEnvIf`，可根据条件设置环境变量．利用 `SetEnvIf`，就可以让 `CustomLog` 根据环境变量的状态，有条件地为请求记录日志．下面的配置示例告诉 服务器不要为指向私有 Subversion URL 的 `GET` 和 `PROPFIND` 请求记录日志．

```
# Matches everything, just to initialize the "in_repos" variable.
SetEnvIf Request_URI "^" in_repos=0

# Set "in_repos" if this is a request for a private Subversion URL.
SetEnvIf Request_URI "/!svn/" in_repos=1

# Set "do_not_log" for non-public request types we don't care to log.
SetEnvIf Request_Method "GET" do_not_log
SetEnvIf Request_Method "PROPFIND" do_not_log

# Unset "do_not_log" for URLs that aren't private Subversion URLs.
SetEnvIf in_repos 0 !do_not_log

# Log requests, but only if "do_not_log" isn't set.
CustomLog logs/access_log env=!do_not_log
```

利用上面的配置，*httpd* 仍然会为指向公开的 Subversion URL 的 `GET` 请求记录日志，这些请求 有可能是用户在网页上浏览仓库时产生的．对于指向私有 的 Subversion URL 的 `GET` 和 `PROPFIND` 请求—这些请求可能是检出操作所 产生的—将不会记录日志．

直写代理

使用 Apache 作为 Subversion 服务器的一大好处是它可以用来实现简单的 副本备份．比如说，你的团队分布在全球的四个办公室内，而 Subversion 仓库只能放在其中一个办公室中，这就意味着其他三个办公室将享受不到好 的访问体验—当他们更新和提交代码时，看到的很可能是缓慢的响应 时间．对于这种问题，最有效的解决方案是搭建一套系统，该系统由一个 Apache 主 (*master*) 服务器与若干个 从 (*slave*) 服务器组成．如果在每一个办公室都放置 一个从服务器，当用户检出工作副本时，将从最近的服务器上检出—所 有的读操作都在本地的从服务器上完成，写操作将被自动路由到主服务器．当提交完成时，主服务器自动地使用备份工具 *svnsync*，把新版本号 “推送” 给其他所有的从服务器．

上面的方案可以极大地提高用户的访问速度，因为 Subversion 客户端 所产生的网络流量中，有 80–90% 都是读请求，如果这些请求都由 本地的 服务器进行处理，将是一个很大的性能提升．

本节，我们将介绍如何搭建一个标准的一主多从服务器系统，注意，Apache 的版本至少是 2.2.0（加载了模块 *mod_proxy*），Subversion (*mod_dav_svn*) 至少是 1.5．

我们这里所介绍的方案，只是配置 Subversion 直写代理的众多方案 中的一种，还有其他方案可供选择，例如从服务器可以定期地从主服务器 上拉取修改，而不是主服务器主动向从服务器推送修改．又或者是主服务 器先将修改推送到某个从服务器，这个从服务器再将相同的修改推送给下 一个从服务器，依次类推．管理员可以先通过本节理解当部署 Subversion WebDAV 代理时，发生了哪些事情，然后再实现一种最适合自己的方案．

配置服务器

首先，按照通常的方式修改主服务器的 *httpd.conf*，使得仓库能在特定的 URI 位置被 访问到，按照你自己的需求，配置认证与授权．主服务器配置完成后，按照相同的步骤配置从服务器，不过从服务器要额外配置一个 `SVNMasterURI` 配置指令：

```
<Location /svn>
  DAV svn
  SVNPath /var/svn/repos
  SVNMasterURI http://master.example.com/svn
  …
</Location>
```

配置指令 `SVNMasterURI` 告诉从服务器把所有 的写请求都重定向到主服务器（写请求重定向由 Apache 模块 *mod_proxy* 自动完成）．然而，普通的读请求仍然由 从服务器处理．一定要确保主服务器和从服务器都配置了相同的认证与 授权，否则的话会非常让人头疼．

接下来，我们需要解决无限递归的问题．根据目前的配置，想像一下 当客户端向主服务器提交一个修改时，将会发生什么现象．当提交完成后，主服务器使用 *svnsync* 将新的版本号复制给每一个 从服务器，但是在从服务器看来，*svnsync* 只不过是 提交修改的另一个客户端而已，于是从服务器会马上把这个写请求重定向 给主服务器，这就导致了无限递归．

解决办法是让主服务器把版本号复制到从服务器上与主服务器不同的 `<Location>`，这个位置 不 设置写请求代理，但接受且只接受从主服务器 IP 地址发过来的提交：

```
<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>
```

设置备份

主服务器和从服务器上的 `Location` 配置完毕后，接下来需要配置主服务器将修改复制给所有的从服务器，其中需要用到 *svnsync*，关于该命令的详细介绍在 “使用 svnsync 复制仓库”一节．

首先要确保的是从服务器上的每一个仓库都含有钩子脚本 pre-revprop-change，该脚本用于开启版本号属性修改（修改版本号属性 是接收端接收到 *svnsync* 结束时的标准步骤）．然后 登录到主服务器，将每一个从服务器的仓库配置为从主服务器本地磁盘上 的仓库接收数据：

```
$ 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 \
               file:///var/svn/repos
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 \
               file:///var/svn/repos
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 \
               file:///var/svn/repos
Transmitting file data ....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .......
Committed revision 2.
Copied properties for revision 2.
…
```

上面的命令执行完后，修改主服务器的钩子脚本 post-commit, 使得在提交完成后，调用 *svnsync* 将版本号复制 给每一个从服务器:

```
#!/bin/sh
# Post-commit script to replicate newly committed revision to slaves

svnsync sync http://slave1.example.com/svn-proxy-sync \
             file:///var/svn/repos > /dev/null 2>&1 &
svnsync sync http://slave2.example.com/svn-proxy-sync \
             file:///var/svn/repos > /dev/null 2>&1 &
svnsync sync http://slave3.example.com/svn-proxy-sync \
             file:///var/svn/repos > /dev/null 2>&1 &
```

每一行末尾的 & 并非绝对必需，添加它的 目的是为了让命令在后台执行，于是 Subversion 客户端就不必等待 post-commit 脚本里的命令全部执行完毕. 除了 post-commit 钩子脚本，管理员还要编写 post-revprop-change 钩子脚本，以便在用户修改了版本 号属性（例如日志消息）后，从服务器也能接收到这个修改：

```
#!/bin/sh
# Post-revprop-change script to replicate revprop-changes to slaves

REV=${2}
svnsync copy-revprops http://slave1.example.com/svn-proxy-sync \
                      file:///var/svn/repos \
                      -r ${REV} > /dev/null 2>&1 &
svnsync copy-revprops http://slave2.example.com/svn-proxy-sync \
                      file:///var/svn/repos \
                      -r ${REV} > /dev/null 2>&1 &
svnsync copy-revprops http://slave3.example.com/svn-proxy-sync \
                      file:///var/svn/repos \
                      -r ${REV} > /dev/null 2>&1 &
```

现在还未处理的就是用户级别的锁（也就是与 *svn lock* 相关的锁），在提交操作执行期间，由主 服务器完成锁的施加，但是关于锁的全部信息在读取操作（例如由从服务器 负责处理的 *svn update* 和 *svn status*）执行期间，需要进行分发. 因此，一个功能完善的代理设置需要把主服务器中与锁有关的信息复制到 从服务器上. 不幸的是，该问题的大多数解决方案无论如何都达不到要求 [9]. 很多团队根本就不使用 Subversion 的锁，所以这些解决方案存在的问题并不会给你带来麻烦. 不过对于那些确实要用到锁的团队而言，我们目前也无法提供有用的建议.

警告

主从副本系统现在就可以正式工作了，不过有些事情需要提前注意. 这里介绍的副本策略不能抵抗服务器或网络崩溃的情况. 举例来说，如果 其中一个 *svnsync* 由于某种原因失败了，那么从 服务器就会悄无声息的失败，即使有用户声称他们已经提交了版本号 100，但是后面执行 *svn update* 时，本地从服务器将告诉 他们版本号 100 并不存在！当然，如果又有新的提交发生，并且随后的 *svnsync* 都执行成功了，那么问题就会被自动地修复 —*svnsync* 会复制所以未复制的版本号. 不过 管理员仍然想设置某种带外的监控，以便能够侦测到失败的同步，并强制 运行 *svnsync*，修正错误.

直写代理的另一个限制涉及到主服务器与从服务器的 Subversion 版本不匹配. 新发布的 Subversion 很可能为服务器与客户端之间所使用 的网络协议添加了新特性，由于客户端只和从服务器进行特性协商，因此 最终使用的协议版本和特性集合由从服务器的 Subversion 版本决定. 不过，写操作是被逐字逐句地传递给主服务器，因此，如果主服务器的 Subversion 版本较旧，从服务器在与客户端进行特性协商时，可能会返回从服务器 支持，而主服务器不支持的特性，结果是客户端使用了主服务器不理解的 特性，最终导致操作失败.

为了缓和上面的问题，Subversion 1.8 引入了一个新的 Apache 配置 指令—`SVNMasterVersion`. 在每个从服务器上，把 `SVNMasterVersion` 都设置成主服务器的 Subversion 版本号，这样从服务器在与客户端协商特性时，就会考虑到主 服务器的 Subversion 版本.

不幸的是， Subversion 1.7 不支持配置指令 `SVNMasterVersion`， 如果从服务器的版本是 Subversion 1.7， 而主服务器的版本比 Subversion 1.7 旧， 那么管理员 就需要在从服务器的 `<Location>` 配置块里 加上配置指令 `SVNAdvertiseV2Protocol Off`.

[9] http://subversion.tigris.org/issues/show_bug.cgi?id=3457 记录了各种方案存在的问题.

为了尽量减少可能的麻烦，最好在主服务器和从服务器上运行相 同版本的 Subversion.

是否可以用 svnserve 设置副本?

如果管理员使用 *svnserve*—而不是 Apache—作为服务器软件，那么仍然可以使用钩子脚本执行 *svnsync*，完成主服务器到从服务器的版本号复制．但不幸的是，在撰写本书时，还没有办法让 *svnserve* 自动地把写请求重定向至主服务器，这就意味着用户只能从从服务器上 检出只读的工作副本，管理员必须把从服务器配置成禁止写访问．如果 想创建某个开源项目的只读 “镜像”，这倒是一种不错的 办法，但这就不是一个透明的代理系统了．

Apache 的其他特性

作为一个优秀的网页服务器，Apache 提供的诸多特性同样也可以用于 提高 Subversion 的功能性与安全性．如果 Subversion 客户端在编译时 打开了对 SSL (Secure Socket Layer，安全套接字层，前面我们已经介绍 过了）的支持，客户端就可以使用 `https://` 形式 的 URL 访问 Apache 服务器，享受高质量的加密网络服务．

同样有用的特性还包括支持端口指定（而不是使用默认的 HTTP 80 端口），或者是为 Subversion 仓库指定一个虚拟域名，或者是支持通过 HTTP 代理 访问仓库．

最后，由于 *mod_dav_svn* 使用的是 WebDAV/DeltaV 协议的一个子集，所以说我们还可以通过第三方的 DAV 客户访问仓库．大多 数现代操作系统 (Win32, OS X 和 Linux) 都支持把 DAV 服务器作为一个 标准的网络 “共享目录” 挂载到本地．这是一个比较复杂 的主题，但是一旦实现就会觉得非常奇妙，更多的细节见 附录 C, *WebDAV 与自动版本控制*．

注意，除了我们这里介绍的之外，*mod_dav_svn* 还 有很多不常用的配置，对于感兴趣的读者，我们在 “mod_dav_svn 配置指令” 一节 列出了可用 在 *httpd.conf* 中的所有 *mod_dav_svn* 配置指令．

Subversion Apache HTTP 服务器配置参考

在上面一节，我们介绍了很多配置指令，通过在 *httpd.conf* 中使用这些配置指令，管理员就可以 配置 Subversion 服务器的具体行为，以及服务器所能提供的功能．本节将 快速总结 Subversion 提供的 Apache HTTP 服务器模块的 所有 配置指令．

mod_dav_svn 配置指令

下面是 Subversion 提供的 Apache HTTP 服务器模块 *mod_dav_svn* 支持的所有配置指令．

`DAV svn`

必须被包含在 Subversion 仓库的 `Directory` 或 `Location` 配置块内．`DAV svn` 告诉 *httpd* 使用 Subversion 提供的 *mod_dav* 后端驱动 处理所有的请求．

`SVNActivitiesDB` *`directory-path`*

指定活动数据库的存放目录．默认情况下，*mod_dav_svn* 使用仓库中的 *dav/activities.d* 作为活动数据库的存放 目录（若没有则创建该目录）．该选项所指定的目录路径必须是绝对 路径．

如果定义了 SVNParentPath, *mod_dav_svn* 就会把仓库路径的最后一个分量 附加到 SVNActivitiesDB 所指定的路径上，例如：

```
<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>
```

`SVNAdvertiseV2Protocol On|Off`

在 Subversion 1.7 引入，该配置指令决定 *mod_dav_svn* 是否应该支持 1.7 引入的新版 HTTP 协议. 大多数管理员都会选择禁止 SVNAdvertiseV2Protocol (而默认值是 On). 如果选择打开 SVNAdvertiseV2Protocol, 就能享受到新版 协议带来的性能提升. 然而，如果服务器被设置为另一个服务器的 直写代理，而另一个服务器并不支持新版协议，那就要设置成 Off.

`SVNAllowBulkUpdates On|Off|Prefer`

该配置指令决定， 对于更新类型的请求， 是否支持全包含 (all-inclusive) 的响应. Subversion 客户端通过发送请求 REPORT 向服务器索取关于目录检出和更新 的信息，客户端可以请求服务器以两种方式之一返回信息：在一个 单独的响应中携带全部的信息，或者是只返回一段概略性的信息，然后 Subversion 客户端再根据这段信息，向服务器请求 额外的 数据. 如果 SVNAllowBulkUpdates 被设置成 Off, *mod_dav_svn* 将 按照第二种方式响应 REPORT 请求，无论客户 端所请求的是哪一种响应类型.

SVNAllowBulkUpdates 的默认值是 On, 也就是说服务器会根据客户端所请求的 响应类型来回复请求. 从 Subversion 1.8 开始, SVNAllowBulkUpdates 也可以被设置成 Prefer, 它和 On 类似，但是服务器将向客户端宣告它 更喜欢 按照第一种方式处理更新请求.

大多数人根本就不会用到这个配置指令， 它的存在主要是为了满足 管理员的这种需要—为了安全或审计需要， 强迫 Subversion 客户端在更新或检出时，单独地抓取文件与目录，从而在 Apache 的 日志里留下请求 GET 和 PROPFIND 的审计记录.

`SVNAutoversioning On|Off`

如果配置指令的值是 On, 来自 WebDAV 的写请求将自动生成提交，版本号的日志消息也是一条自动生成的 消息. 如果 SVNAutoversioning 的值是 On, 你可能还需要设置上 ModMimeUsePathInfo On, 于是 *mod_mime* 就能自动地为文件设置正确的 svn:mime-type 属性 (当然，*mod_mime* 也只能尽最大努力做到正确). 更多的 细节见 附录 C, *WebDAV 与自动版本控制*. SVNAutoversioning 的默认值是 Off.

`SVNCacheFullTexts On|Off`

如果被设置成 On, 并且内存缓存足够，Subversion 将全文缓存文件的内容，这可以极大地提升服务器的性能. (另请参阅 SVNInMemoryCacheSize) 默认值是 Off.

`SVNCacheTextDeltas On|Off`

如果被设置成 `On` 并且内存缓存足够，Subversion 将缓存文件内容的差异部分，这可以极大地提升服务器 的性能.（另请参阅 `SVNInMemoryCacheSize`) 默认值是 `Off`.

`SVNCompressionLevel` *`level`*

指定在网络上传输文件内容时所用的压缩级别. 把 `SVNCompressionLevel` 设置成 `0` 将禁止压缩，最大值是 `9`, 默认值是 `5`.

`SVNHooksEnv` *`file-path`*

指定 Subversion 仓库钩子脚本环境配置文件的路径，这个配置文件 描述了钩子脚本运行时的初始环境，更多的细节见“钩子脚本环境配置”一节.

`SVNIndexXSLT` *`directory-path`*

为目录索引指定一个 XML 变换规则的 URI, 该配置指令是可选的.

`SVNInMemoryCacheSize` *`size`*

指定 Subversion 每个进程的内存对象缓存的最大大小（单位 KB). 默认值是 `16384`, 把 `SVNInMemoryCacheSize` 设置为 `0` 表示禁止缓存.

`SVNListParentPath On|Off`

如果设置为 `On`, 将允许客户端向 `SVNParentPath` 发送 `GET` 请求，请求的结果是返回目录 `SVNParentPath` 下的所有仓库组成的列表. 默认值是 `Off`.

`SVNMasterURI` *`url`*

指定 Subversion 主仓库的 URI（用于设置直写代理）.

`SVNMasterVersion` *`X.Y`*

指定主仓库的 Subversion 服务器的版本（用于设置直写代理）.

`SVNParentPath` *`directory-path`*

指定一个目录路径， 而该目录的子目录都是 Subversion 仓库. 在一个 Subversion 仓库的配置块内， 要么出现 `SVNParentPath`, 要么出现 `SVNPath`, 但这两个配置指令不能同时出现或都 不出现.

`SVNPath` *`directory-path`*

指定一个 Subversion 仓库的路径. 在一个 Subversion 仓库的 配置块内， 要么出现 `SVNPath`, 要么出现 `SVNParentPath`, 但这两个配置指令不能同时出 现或都不出现.

`SVNPathAuthz On|Off|short_circuit`

控制基于路径的授权检查，控制方式包括允许子请求 (`On`), 禁止子请求 (`Off`, 见 “禁止基于路径的检查”一节), 或者直接询问 *mod_authz_svn* (`short_circuit`). 默认值是 `On`.

`SVNReposName` *`name`*

指定 Subversion 仓库在 `HTTP GET` 响应 中所使用的名字. 这个值将作为所有目录列表的标题（使用网页浏览 器浏览 Subversion 仓库时将会看到该标题）. 该配置指令是可选的.

当 Subversion 尝试匹配访问控制文件里的规则时, 不会用到 `SVNReposName` 所指定的名字. 用于匹配访问 控制文件的仓库名总是来自仓库的 URL, 更多的细节见 “基于路径的访问控制”一节.

`SVNSpecialURI` *`component`*

为特殊的 Subversion 资源指定 URI 分量（名字空间）. 默认 值是 `!svn`, 绝大多数管理员从来不会用到这个 配置指令, 除非仓库中存在名为 *!svn* 的 文件. 如果你在服务器投入使用后再修改此值, 那么所有已存在的 工作副本都将无法正确工作, 用户也会把怒火发泄到你身上.

`SVNUseUTF8 On|Off`

如果设置为 `On`, *mod_dav_svn* 将使用 UTF-8 编码的仓库根目录 路径与钩子脚本通信, 并且期望脚本的输出（例如错误消息）也是使 用 UTF-8 编码. 默认值是 `Off`, 这意味着 *mod_dav_svn* 将使用 ASCII 编码的仓库根目录 路径与钩子脚本交互. 该配置指令在 Subversion 1.8 引进.

管理员应该确保钩子脚本的字符集和编码方案与所有可能的调用 方式相匹配. 比如说, 某个仓库同时使用 *httpd* 和 *svnserve* 暴露给用户, 如果该配置指令 被设置为 `On`, 那么 *svnserve* 也要配置成使用 UTF-8 编码（通过 在它的环境变量中设置合适的地区选项）. 而且, 如果钩子脚本所 访问到的文件系统路径（例如仓库根目录的路径）包含了非 ASCII 码字符, 那么这些路径的编码也必须和脚本的编码方案相匹配.

mod_authz_svn 配置指令

以下的配置指令由 *mod_authz_svn* 提供, 它是 Subversion 基于路径授权的 Apache HTTP 服务器模块. 关于如何在 Subversion 中使用基于路径的授权, 见 “基于路径的授权”一节.

`AuthzForceUsernameCase Upper|Lower`

在检查授权之前, 把已认证的用户名转换成大写或小写形式. 当 用户名与授权规则文件中记录的用户名相比较时, 是区分大小写的, 这个配置指令可以把大小写混合的用户名转换成大小写一致的形式.

`AuthzSVNAccessFile` *`file-path`*

在路径为 *file-path* 的文件中查找访问 权限配置, 访问权限配置描述了 Subversion 仓库中的路径的 访问权限. 在 Subversion 仓库的配置块内, 可以出现这个配置指令或 `AuthzSVNReposRelativeAccessFile`, 但不能同 时出现.

从 Subversion 1.8 开始, `AuthzSVNAccessFile` 接受一个 URL 作为参数, 这个 URL 指向了 Subversion 仓库内的一个文件, 这个仓库可以是 应用规则文件的同一仓库或不同仓库.

`AuthzSVNAnonymous On|Off`

设置成 `Off`, 将禁止模块 *mod_authz_svn* 的两个特例行为: 与配置指令 `Satisfy Any` 的相互影响, 和强制实施授权策略（即使没有出现配置指令 `Require`). 该配置指令的 默认值是 `On`.

`AuthzSVNAuthoritative On|Off`

设置成 `Off` 将允许把访问控制传递到下层 模块. 默认值是 `On`.

`AuthzSVNNoAuthWhenAnonymousAllowed On|Off`

设置成 `On` 将禁止匿名用户发来的请求的 认证与授权检查, 默认值是 `On`.

`AuthzSVNReposRelativeAccessFile` *`file-path`*

在路径为 *file-path* 的文件中查找访问 权限配置, 访问权限配置描述了 Subversion 仓库中的路径的 访问权限. 和 `AuthzSVNAccessFile` 不同的是, `AuthzSVNReposRelativeAccessFile` 定义的是相 对于仓库中的 *conf/* 目录的路径. 换句话说, 由 *file-path* 所指定的文件, 必须能让仓库 通过相对路径访问到. 在仓库的配置块内, 可以出现配置指令 `AuthzSVNReposRelativeAccessFile` 或 `AuthzSVNAccessFile`, 但两者不能同时出现. 该配置指令在 Subversion 1.7 引进.

从 Subversion 1.8 开始, `AuthzSVNReposRelativeAccessFile` 接受一个 URL 作为参数, 这个 URL 指向了 Subversion 仓库内的一个文件, 这个仓库可以是应用规则文件的同一仓库或不同仓库.

基于路径的授权

Apache 和 *svnserve* 都可以向用户授予或剥夺权限. 权限通常是针对整个仓库: 允许 (或禁止) 用户读取仓库, 以及允许 (或禁止) 用户写仓库.

实际上, 管理员还可以设置更加精细的访问权限. 例如允许一部分的用户 写 仓库中的某个特定目录, 但禁止其他用户; 或者是禁止大部分用户读取某个特定的 目录, 但允许少数用户读取. 访问权限的精细程度甚至可以细致到单个文件.

Apache 和 *svnserve* 使用相同的文件格式描述基于 路径的访问权限. 本节我们将介绍文件的格式, 以及如何配置 Subversion 服务 器, 以便利用规则文件管理基于路径的授权.

[10] 这是本书的一个重要主题

你是否真得需要基于路径的访问控制?

很多管理员在第一次配置 Subversion 时，就急不可待地开始设置基于 路径的授权. 管理员通常知道团队成员与项目的对应关系，所以很容易就能 决定应该把特定目录的权限授予给哪些人，不授予给哪些人. 乍看起来非常 自然，还可以缓解管理员过度控制仓库的欲望.

但是，这样做通常会产生一些可见（和不可见）的代价. 可见的代价是 服务器需要做大量的配置工作，以便确保用户对每个特定的目录都有正确的 读写权限，在某些情况下，这会产生很明显的性能损失. 对于不可见的代价，考虑一下你所创建的文化. 在大多数情况下，特定的用户 *不应该* 向特定的仓库目录提交修改，这种约定没必要 通过技术手段加以约束. 团队成员有时候会不由自主地与他们协作，例如通过 向不属于他们的仓库目录提供修改，帮助别人完成工作. 如果在服务器配置上 禁止了这种形式的合作，无形中就树起了一道协作屏障. 随着项目的发展与 新成员的加入，管理员需要不停地创建新规则，这会带来大量额外的维护工作.

记住，这是一个版本控制系统！即使有人不小心向错误的目录提交了修改，这些修改也可以轻易地撤消. 如果用户故意向错误地目录提交修改，这只能算 作人际问题，要在 Subversion 外部解决.

所以，在管理员限制用户的访问权限之前，要问一下自己这是否真的有必要，是否只是为了 “听起来不错”，是否值得牺牲服务器的性能. 放任 用户的访问权限并不会带来非常大的风险，而且依赖技术手段来解决社交问题是 一种不好的做法![10]

考虑一个例子，假设 Subversion 项目对哪些人应该向哪些目录提交 有一套自己约定俗成的规则，但这些规则总是通过社交来实施. 这是一种 良好地社区信任模型，特别是对于开源项目而言. 当然，有时候确实需要 基于路径的访问控制，比如说在公司内部，某些数据非常敏感，只能把访问 权授予给少数人.

基于路径的访问控制

Subversion 通过模块 *mod_authz_svn* 向 Apache 提供基于路径的访问控制，模块 *mod_authz_svn* 必须 在配置文件 *httpd.conf* 内，使用配置指令 `LoadModule` 进行加载，加载方式与模块 *mod_dav_svn* 相同. 为了让仓库能够使用这个模块，在配置文件 *httpd.conf* 内用配置指令 `AuthzSVNAccessFile` 或 `AuthzSVNReposRelativeAccessFile` 来指定访问权限配置 文件.（详细的说明见 “每个目录的访问控制”一节.）

为了给 *svnserve* 配置基于路径的访问控制，只需要把 *svnserve.conf* 内的配置变量 `authz-db` 的值指定成你的访问权限配置文件.

一旦服务器知道了去哪儿查找你的访问权限配置文件，接下来需要做的就是 定义访问权限.

Subversion 访问权限配置文件的语法与 *svnserve.conf* 和运行时配置文件的语法相同. 忽略以 `#` 开始的行，在最简单的形式中，文件的每一节的名字都指定了一个被版本控制的路径，可能还指定了包含该路径的仓库. 换句话说，除了少数几个被保留的节外，节名只有这两种形式：如果使用了配置指令 `AuthzSVNAccessFile`，则要么是 `[repos-name:path]`，要么是 `[path]`. 如果使用了配置指令 `AuthzSVNReposRelativeAccessFile` 指定了每个仓库的访问权限配置文件，则只能使用 `[path]` 这种形式. 已认证的用户名是每一节的选项的名字，选项的值描述了用户对该 路径的访问权限：只读 (`r`) 或读写 (`rw`). 如果用户名未出现在节中，则不具有该路径的 任何访问权限.

出现在访问权限配置文件中的路径必须使用 Subversion 的 “内部风格” 编写，在大部分情况下，这意味着路径使用 UTF-8 编码，使用斜杠 (`/`) 作为路径中各个分量 的分隔符（即使是 Windows 系统，也要使用斜

杠). 还要注意这些路径 中的字符不会进行任何形式地转义—比如说路径中的空格必须与 文件中的节名 (例如 `[repos-name:path with spaces]`) 完全 一致.

下面是访问权限配置文件的一个例子, 文件把仓库 `calc` 的路径 */branches/calc/bug-142* (及其子目录) 的 读权限授予 Sally, 把读写权限授予 Harry:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

Subversion 1.7 之前的版本会把仓库名和路径都转化成小写形式后, 再与访问权限配置文件进行匹配, 因此是不区分大小写的. 但从 Subversion 1.7 开始不再如此. 因此, 如果你需要从较旧的版本升级到 1.7, 你应该重新检查访问权限配置文件中的大小写是否正确.

被授权模块检查的仓库名直接来自仓库的路径, 具体的表现受到两个服务 器选项的影响. *mod_dav_svn* 只使用仓库根目录 URL 的 最后一个分量 [11], 而 *svnserve* 则使用完整的, 相对 于服务器根目录 (由命令行选项 `--root(-r)` 指定) 的仓库路径.

如果同时通过 Apache 服务器与 *svnserve* 为 同一个仓库服务, 那么 *mod_dav_svn* 与 *svnserve* 决定仓库名的不同之外将会带来问题. 通常情况下, 管理员更喜欢为两种服务器配置同一个访问权限配置文件, 然而, 为了能让访问权限配置文件正常工作, 管理员必须确保访问权限配置文件里 的仓库名对于两种服务器而言都是兼容的—例如把 *svnserve* 的根目录配置成和 *mod_dav_svn* 的配置指令 `SVNParentPath` 相同的值, 或者为每个仓库指定一个 单独的访问权限配置文件, 这样就不用在文件里提到仓库名.

如果你使用了配置指令 `SVNParentPath`, 在访问权限 配置文件的节名中指定仓库名就非常重要, 因为如果省略了仓库名, 例如把节名写 成 `[/home/dir]`, 那么该节将匹配 *每*个 仓库中的 */some/dir*. 然而, 如果使用了配置指令 `SVNPath`, 那么在节名中 只提供路径也是可以的—毕竟这时候只存在一个仓库.

路径的权限继承自父目录, 这意味着我们可以为 Sally 指定一个访问 策略不同的子目录. 继续我们前面的例子, 现在我们要为 Sally 授予分支中 某个子目录的写权限, 而 Sally 原来只对分支拥有只读权限.

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

# give sally write access only to the 'testing' subdir
[calc:/branches/calc/bug-142/testing]
sally = rw
```

现在 Sally 对分支的子目录 *testing* 拥有写权 限, 但对于分支的其他部分仍然只具有只读权限. 而 Harry 对整个分支拥有 读写权限.

我们还可以通过规则的继承, 显式地阻止用户的权限, 方法是把用户名 设置成空:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

[11] 通过配置指令 `SVNReposName`, 在 *httpd.conf* 里配置的任意一个人类可读懂的仓库名, 都将被授权模块忽略. 前面已经 说过, 访问权限配置文件里的节名必须引用到对服务器敏感的仓库路径.

```
[calc:/branches/calc/bug-142/secret]
harry =
```

在这个例子里，Harry 对整个 *bug-142* 目录具有 读写权限，但却无法访问其中的子目录 *secret*.

需要记住的是最明确的路径总是最先被匹配. 服务器总 是试图匹配路径本身，然后是路径的父路径，然后再是父路径的父路径，以此类推. 这样做的影响是如果我们在访问权限配置文件里添加一个特定的 路径，那么它的权限配置就会覆盖从父目录继承而来的权限配置.

类似的， 指定了仓库名的节的权限配置将覆盖那些没有指定仓库名的节， 比如说访问权限配置文件里同时出现了 `[calc:/some/path]` 和 `[/some/path]`，那么对于仓库 `calc`，将会使用第一种配置，而忽略第二种.

在默认情况下，任何用户对任意一个仓库都不具备访问权限，这意味着 如果从头开始写访问权限配置文件，你可能希望所有用户至少对仓库的根目录具有 只读权限. 可以通过把用户名设置成通配符 (`*`) 实现这 种配置，此时通配符 `*` 表示 “所有用户”：

```
[/]
* = r
```

这是一种很常见的配置，注意在节的名字中没有指定仓库的名字. 上面的 配置将把所有仓库的读权限授予给所有用户. 一旦用户对仓库具有了读权限，接下来就可以根据具体的需要，把特定仓库的特定目录的读写权限 (`rw`) 授予特定的用户.

虽然前面的例子都是针对目录的权限配置，因为这是最常见的情况，实际上管理员完全可以针对文件设置访问权限.

```
[calendar:/projects/calendar/manager.ics]
harry = rw
sally = r
```

用户组

访问权限配置文件还允许管理员定义用户组，就像 Unix 里的 */etc/group*. 为了定义用户组，在访问权限配置文件里 创建一个名为 `groups` 的节，然后在节内描述每一个 用户组：变量名定义了用户组的名字，而变量的值则是逗号分隔的，属于该用户组的用户名.

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = harry, sally, joe, frank, jane
```

用户组的权限授予和用户名相同，为了与用户名相区别，在用户组的名字前 要加一个 `@` 符号：

```
[calc:/projects/calc]
@calc-developers = rw

[paint:/projects/paint]
jane = r
@paint-developers = rw
```

需要特别注意的是用户组权限并不会被用户权限所覆盖， 而是会进行 *叠加*. 在上面的例子里， Jane 是用户组 `paint-developers` 的成员，因此她对仓库 `paint` 具有读写权限，再叠加上 `jane = r`，Jane 最终的权限仍然是可读写. 如果用户 已经是某个用户组的成员，那就不可能再把用户的权限限制得比用户组的权限 还小.

用户组还可以包含其他的用户组:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = @calc-developers, @paint-developers
```

用户别名

某些认证系统仅支持相对较短的用户名, 例如 `harry`, `sally`, `joe` 这样简短的名字, 而其他一些认证系统—例如那些使用了 LDAP 和 SSL 的认证系统— 却支持更复杂的用户名, 例如在一个使用了 LDAP 的认证系统中, Harry 的 用户名可以是 `CN=Harold Hacker,OU=Engineers,DC=red-bean,DC=com`. 如果在访问权限配置文件里出现这些用户名, 文件将变得非常臃肿, 这些又长又 晦涩的用户名还很容易写错.

幸运的是, Subversion 1.5 为访问权限配置文件添加了对用户别名的支持. 有了用户别名, 对于复杂的用户名, 管理员只需要在赋予别名的地方写一次 就够了.

用户别名定义在访问权限配置文件的一个特殊的节— `aliases`, 节内的每一个变量名都定义了一个别名, 而 变量值则是真实的用户名.

```
[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
…
```

用户别名定义完成后, 在访问权限配置文件内, 只要是能出现真实用户名的 地方都能用别名替代, 唯一的区别是要在别名前添加符号 `&`, 以便与真实的用户名进行区分.

```
[groups]
calc-developers = &harry, &sally, &joe
paint-developers = &frank, &sally, &jane
everyone = @calc-developers, @paint-developers
```

如果用户名经常发生变化, 那么使用别名也能带来方便. 利用别名, 当 用户名发生变化时, 只需要在定义别名的地方更新一次即可, 而不用在整个 文件内搜索并替换.

访问权限控制的高级特性

从 Subversion 1.5 开始, 访问权限配置文件还支持一些 “魔力” 符号, 这些符号可以帮助管理员基于用户的认证类别来 制定访问权限. 其中一个符号是 `$authenticated`, 用于将权限授予给所有已认证的用户. 类似的还有 `$anonymous`, 它表示所有 *未认证* 的用户.

```
[calendar:/projects/calendar]
$anonymous = r
$authenticated = rw
```

访问权限配置文件语法的另一个很有用的魔力符号是波浪号 (`~`), 用于排除某些用户. 在访问权限配置文件中, 如果 在用户名, 用户别名, 用户组或认证类别前加上波浪号, 就表示将访问权限 授予给与规则 *不匹配* 的用户. 虽然下面的配置容易 让人产生不必要的困惑, 但它和上面的例子是等效的:

```
[calendar:/projects/calendar]
~$authenticated = r
~$anonymous = rw
```

下面是一个更恰当的，使用 ~ 的例子：

```
[groups]
calc-developers = &harry, &sally, &joe
calc-owners = &hewlett, &packard
calc = @calc-developers, @calc-owners

# Any calc participant has read-write access...
[calc:/projects/calc]
@calc = rw

# ...but only allow the owners to make and modify release tags.
[calc:/projects/calc/tags]
~@calc-owners = r
```

访问权限控制的一些陷阱

如果你使用 Apache 作为 Subversion 服务器，而且还设置了仓库的 某些子目录对某些用户是不可读的，那么在执行 *svn checkout* 时，你需要意识到可能会出现的低效 行为.

取决于 Subversion 客户端所使用的 HTTP 函数库，它可能会要求检出或 更新的全部载荷在一个单独的响应中递送，当这种情况发生时，一开始的认证 请求就是 Apache 向用户要求认证的 唯一 机会，这会产生一些很奇怪的副作用. 比如说，如果仓库的某个子目录只有 Sally 具有读权限，而 Harry 检出了该目录的父目录，他的客户端将以 Harry 的身 份完成最初的认证. 随着服务器向客户端发送大块的响应数据，当发送 Harry 不具有读权限的那个子目录时，服务器再也没有办法再次发送认证请求，于是 该子目录被忽略，而用户却无法通过重新以 Sally 的身份进行认证来避免这 种情况.

类似的道理，如果仓库的根目录对于匿名用户是可读的，那么检出整个仓库 时就不会向用户要求认证—同样，服务器将直接忽略不可读的目录，而不 会在检出到一半时向用户要求认证. [12]

高层日志记录

Apache 的 *httpd* 和 Subversion 的 *svnserve* 都支持在较高的层次为 Subversion 操作记录 日志. 虽然这两种服务器配置高层日志的方式不太一样，但它们输出的日志都 遵循相同的语法.

为了让 *svnserve* 开启高层日志，只需在启动 *svnserve* 时带上选项 `--log-file`, 选项的值是日志文件路径.

```
$ svnserve -d -r /path/to/repositories --log-file /var/log/svn.log
```

在 Apache 中启用高层日志要稍微复杂一些，但本质上就是 Apache 日志 输出机制的扩展，配置方式见 “Apache 日志” 一节.

下面列出了在启用高层日志后，服务器将为 Subversion 记录的日志消息列表，还给出了具体的日志消息示例.

[12] 关于这个问题的更多信息，见 http://blogs.collab.net/subversion/2007/03/authz_and_anon_/ 的博文 *Authz and Anon Authn Agony*.

检出或导出

```
checkout-or-export /path r62 depth=infinity
```

提交

```
commit harry r100
```

差异比较

```
diff /path r15:20 depth=infinity ignore-ancestry
diff /path1@15 /path2@20 depth=infinity ignore-ancestry
```

抓取一个目录

```
get-dir /trunk r17 text
```

抓取一个文件

```
get-file /path r20 props
```

抓取一个文件版本号

```
get-file-revs /path r12:15 include-merged-revisions
```

抓取合并信息

```
get-mergeinfo (/path1 /path2)
```

加锁

```
lock /path steal
```

日志

```
log (/path1,/path2,/path3) r20:90 discover-changed-paths revprops=()
```

重放版本号 (*svnsync*)

```
replay /path r19
```

修改版本号属性

```
change-rev-prop r50 propertyname
```

版本号属性列表

```
rev-proplist r34
```

状态

```
status /path r62 depth=infinity
```

切换

```
switch /pathA /pathB@50 depth=infinity
```

解锁

```
unlock /path break
```

更新

```
update /path r17 send-copyfrom-args
```

为了方便管理员对 Subversion 高层日志输出进行后期处理，Subversion 源代码包提供了一个 Python 模块，（位于 *tools/server-side/svn_server_log_parse.py*），可用于解析 Subversion 的日志.

服务器优化

在向用户提供 Subversion 服务时，一个尽职的管理员还应该考虑容量规划 与性能调优. Subversion 对于服务器资源（例如 CPU 和内存）并不是非常贪婪，但是任意一个服务都能从优化中获益，特别是当服务的使用量像火箭一样急剧 上升时 [13]. 本节 我们将介绍如何调整服务器配置，以便 Subversion 提供更好的性能与可扩展性.

数据缓存

一般来说，Subversion 服务器代价最高的工作就是从仓库中读取数据，Subversion 1.6 试图通过在内存中缓存特定种类的数据来减小这种代价. Subversion 1.7 在这方面走得更远，它不仅缓存了代价较高的操作的执行 结果，还向服务器提供了调整缓存大小与行为的方法.

对于 *svnserve*，管理员可以通过命令行选项 `--memory-cache-size (-M)` 指定缓存 的大小，还可以通过布尔选项 `--cache-fulltexts` 和 `--cache-txdeltas`，分别指定是否需要缓存文件内容的 全文本与差异文本.

```
$ svnserve -d -r /path/to/repositories \
           --memory-cache-size 1024 \
           --cache-txdeltas yes \
           --cache-fulltexts yes
…
$
```

mod_dav_svn 向 *httpd.conf* 提供了等价的缓存配置， 与上面三个 *svnserve* 命令行 选项对应的配置指令分别是 `SVNInMemoryCacheSize`, `SVNCacheFullTexts` 和 `SVNCacheTextDeltas`:

```
<IfModule dav_svn_module>
  # Enable a 1 Gb Subversion data cache for both fulltext and deltas.
  SVNInMemoryCacheSize 1048576
  SVNCacheTextDeltas On
  SVNCacheFullTexts On
</IfModule>
```

那么，应该把这些选项设置成什么值呢? 当然，你需要考虑服务器的哪些 资源是可用的. 为了尽量从缓存中获益，缓存的大小应至少能够存放仓库中最 常被访问的全部文件（例如项目的主干分支 *trunk*）.

把缓存大小设置成 `0` 将使得 Subversion 转而 使用 1.6 引入的旧的缓存机制.

[13] 对于 Subversion 而言，如果它的使用量上升，正好 说明了它是多么的受欢迎，可靠和易于使用.

到目前为止，只有使用了 FSFS 作为后端存储的仓库才能利用本节 提到的缓存功能.

网络数据压缩

对数据进行压缩后再传输，可以大大减少网络流量，但这也会造成服务 器端（和客户端）消耗更多的 CPU 资源. 根据服务器的 CPU 能力，客户端 访问服务器的典型模式，以及客户端与服务器之间的网络带宽，管理员可能会 想调整服务器应该尽多大的努力来压缩数据. 为了帮助管理员进行这种调整，Subversion 1.7 为 *svnserve* 提供了选项 `--compression (-c)`，为 *mod_dav_svn* 提供了配置指令 `SVNCompressionLevel`，它们都接受一个 0 到 9（含 9）之间的整数，9 表示尽最大努力压缩数据，0 则禁止压缩.

比如说，在一个 1 G 带宽的本地局域网 (LAN) 内，服务器就没必要对 待传输的数据进行压缩（如果服务器对数据进行了压缩，客户端在接收到数据 后还要解压），因为网速已经足够快了，即使对待传输的数据进行压缩，用户 也不会感觉到明显的性能提升. 另一方面，如果访问服务器的大多数客户端，其网络带宽都比较低，那么对待传输的数据进行压缩可以大大改善用户检验.

支持多种仓库访问方法

读者已经见到了访问仓库的多种方式，但是否有可能同时以多种方式（安全地）访问仓库? 如果你有一点先见之明，那么答案是肯定的.

在任意时刻，下面这些进程都可能需要仓库的读写权限：

- 普通的系统用户使用 Subversion 客户端（以用户的身份运行），通过 URL `file://` 直接访问仓库
- 普通的系统用户连接到 SSH 派生的私有 *svnserve* 进程（以用户的身份运行），通过该进程修改仓库
- 以特定身份运行的 *svnserve* 进程— 或者是一个守护进程，或者由 *inetd* 启动
- 一个 Apache *httpd* 进程，以一个特定的身份 运行

如果仓库要同时支持多种访问方式，那么管理员最常遇到的问题通常与仓库 的所有权和权限有关. 前面列表中提到的进程（或用户）是否应该具有读写仓库 底层数据文件的权限? 假设你有一个类 Unix 系统，一种比较简单直接的做法是 把所有可能使用仓库的用户都加入到名为 `svn` 的用户组，而用户组 `svn` 对仓库拥有完整的所有权. 但这样做还不足够，因为进程可能使用了不太友好的文件模式创建屏蔽字—该进程创建的文件 完全禁止了其他进程的访问.

因此，为仓库用户创建了公共的用户组之后，下一步是强制每一个访问仓库 的进程使用一个合理的文件模式创建屏蔽字. 对于直接访问仓库的用户，你可以 把程序 *svn* 封装成一个脚本，脚本首先执行 `umask 002`，然后再运行真实的 *svn* 程序. 你也可以为 *svnserve* 程序写一个类似的封装脚本，以及在 Apache 的启动脚本 *apachectl* 内添加一行 `umask 002`. 例如：

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

在类 Unix 系统上还会遇到另一个常见的问题：比如说，如果你的仓库是使用 Berkeley DB 作为后端存储，那么它很少会去创建新的日志文件来记录自己的 操作. 即使整个仓库都被用户组 `svn` 所拥有，这些新创建 的日志文件也不一定属于用户组 `svn`，这会导致更多的权限 问题. 一种比较好的解决办法是为仓库的 *db* 子目录设置 SUID，这将使得所有新创建的日志文件都具有和父目录相同的用户组.

一旦你克服了这些困难，此时仓库对于所有必要的进程来说，应该都是可访问 的了. 虽然看起来可能有点凌乱和复杂，但是对于如何处理多个用户共享相同文件 的写权限这个问题，一直都没有一个优雅的解决办法.

幸运的是，大多数仓库管理员 *不必* 进行这么复杂 的配置. 如果用户想访问托管在本机上的仓库，除了 `file://`，其实还可以使用 `http://` 或 `svn://`，在填写主机名时需要写成 `localhost`. 为仓库支持多种访问方法可能比看上去更麻烦，我们建议你选择一种满足自己需要的访问方式，然后坚持用它！

svn+ssh:// 服务器检查列表

对于已经有了 SSH 账户的用户，为了让他们共享仓库而不会产生权限上 的问题，其中的过程可能会比较麻烦. 如果你（作为一个管理员）对需要在类 Unix 系统上完成的工作不太清楚，这里列出了一个检查列表，总结了本节 讨论的几个主题：

- 所有的 SSH 用户都需要仓库的读写权限，因此要把他们都放到一个 用户组内.
- 仓库完全被用户组所拥有.
- 把用户组的权限设置成可读写.
- 在访问仓库时，用户需要使用合理的文件模式创建屏蔽字，因此把 可执行文件 *svnserve* 包装成一个脚本，脚本先执行 `umask 002`，然后再运行真实的 *svnserve* 程序.
- 采取类似的方式处理 *svnlook* 和 *svnadmin*，例如先设置好文件模式创建屏蔽字再执行 命令，或者使用和上面一样的包装脚本.

第 7 章 定制自己的 Subversion 体验

版本控制是一个很复杂的主题，它提供了完成任务的无数种方法．本书已经 介绍了 Subversion 命令行客户端的各个子命令，以及它们各自的选项，本章将 介绍更多的定制 Subversion 的方法，包括 Subversion 运行时配置，外部程序，Subversion 与操作系统本地化的交互等．

运行时配置区域

Subversion 为用户提供了许多可配置的行为，有很多选项，用户希望它们 能被应用到所有的 Subversion 操作中．为了方便用户，Subversion 提供了配置 文件，这些配置文件被单独地放在了 Subversion 配置区域，这就避免了用户每 次都要在命令行上指定参数．

Subversion 运行时配置区域 (*runtime configuration area*) 是一个由选项名和 选项值组成的双层体系，第一层是一个目录，目录里包含了许多配置文件，每 个配置文件是 INI 格式的文本文件，文件里的 “节” 是双层 体系的第二层．用户可以使用任意一个文本编辑器（例如 Emacs 或 vi) 编辑 这些配置文件，文件里的配置指令将被客户端命令行工具读取，用来指定客户端命令行工具的行为．

配置区域的布局

客户端命令行工具 *svn* 首次运行时，它将创建一个 每用户配置区域．在类 Unix 系统上，每用户配置区域是一个在用户家目录下 的 *.subversion* 目录．在 Win32 系统上，配置区域是 一个名为 *Subversion* 的文件夹，通常放在用户配置 文件目录（通常是一个隐藏目录）的 *Application Data* 区域内，而 *Application Data* 的确切位置在每 个 Win32 系统上可能都不太一样，它的确切位置由系统注册表 (Windows Registry) 决定[1]．在后面的内容里，我们将用 *.subversion* 指代 Subversion 的每用户 配置区域．

除了每用户配置区域，Subversion 还会考虑全局配置区域，全局配置区域 允许管理员为所有用户设定一个默认配置．注意，每用户配置区域的设置会 覆盖全局配置区域的设置，而 *svn* 的命令行参数则会 覆盖每用户配置区域的设置．对于类 Unix 系统，全局配置区域在 */etc/subversion*; 对于 Windows 系统，全局配置区域 在 *Application Data* （同样，它的确切位置由系统注 册表决定）的 *Subversion* 目录内．与每用户配置区域不 同的是 *svn* 不会在启动时尝试创建全局配置区域．

每用户配置区域包含了三个文件—两个配置文件 (*config* 和 *servers*) 和 一个 *README.txt*, *README.txt* 介绍了 INI 格式的语法．每用户配置区域被创建时，配置文件包含了 Subversion 所支持的选项的默认值，大多数内容都被分组注释，注释描述了选项值将会如何影响 Subversion 的行为．为了修改配置，用户只需要用文本 编辑器打开配置文件，然后修改选项的值．如果你想恢复每用户配置区域的 默认配置，只需要删除（或移动）每用户配置区域的目录，然后随便执行一 个无害的 *svn* 命令，例如 `svn --version`, 此时就会重新生成包含了默认配置 的每用户配置区域．

Subversion 还允许用户通过命令行选项 `--config-option` 修改个别的配置选项，如果用户想要临时修改配置，那么这个选项就会很有用．关于 `--config-option` 的详细用法，见 svn 选项.

每用户配置区域还包含了认证数据的缓存．目录 *auth* 内含有多个子目录，这些子目录包含了 Subversion 支持的不同认证方式所需的缓存信息．目录 *auth* 的权限配置是仅允许用户进行访问．

[1] 环境变量 APPDATA 的 值指出了 *Application Data* 区域的路径，所以说 用户总是可以用 *%APPDATA%\Subversion* 引用 Subversion 每用户配置区域的路径．

配置与 Windows 系统注册表

除了通常的基于 INI 格式的配置文件，在 Windows 平台上，Subversion 还 可以使用系统注册表存放配置选项. 选项的名字和值与 INI 文件相同，“文件/节” 的层次结构仍然保留了下来，不过是以另一种形式 呈现—文件与节变成了注册表键树的层次.

Subversion 在注册表的 `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` 里搜索全局配置， 例如选项 `global-ignores` 在文件 *config* 里的位置是在 `[miscellany]` 节内， 对应的注册表位置是 `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`. 每用户的配置在 `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

基于注册表的配置在基于文件的配置 之前 被读取，因此配置文件将会覆盖注册表的配置. 换句话说，在 Windows 系统中，Subversion 会在下面这些位置搜索配置信息，编号小的位置的配置将会覆盖 编号大的配置：

1. 命令行选项
2. 每用户的 INI 文件
3. 每用户的系统注册表
4. 全局的 INI 文件
5. 全局的系统注册表

Windows 系统注册表实际上不支持用于注释的记号，但 Subversion 自己会忽略所有名字以井号 (#) 开始的键，在效果上 等价于把选项注释掉，这就避免了用户仅仅为了删除选项而把对应的键从注册 表中删除，也简化了恢复选项的操作.

客户端命令行工具 *svn* 从来不会修改 Windows 系统注册表，也不会在首次运行时在注册表内创建默认配置，用户需要自己使用 Windows 工具 *REGEDIT* 创建所需的键. 另外，用户 也可以自己编写一个 *.reg* 文件（例如 例 7.1 “系统注册表项文件 (.reg) 的一个示例” 所展示的 那样），然后在文件浏览器内双击该文件的图标，这将导致文件内的数据被合并 到系统注册表内.

例 7.1. 系统注册表项文件 (.reg) 的一个示例

```
REGEDIT4

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\groups]

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-auth-types"="basic;digest;negotiate"
"#http-compression"="yes"
"#http-library"=""
"#http-proxy-exceptions"=""
"#http-proxy-host"=""
"#http-proxy-password"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
```

```
"#http-timeout"="0"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""
"#ssl-pkcs11-provider"=""
"#ssl-trust-default-ca"=""
"#store-auth-creds"="yes"
"#store-passwords"="yes"
"#store-plaintext-passwords"="ask"
"#store-ssl-client-cert-pp"="yes"
"#store-ssl-client-cert-pp-plaintext"="ask"
"#username"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#password-stores"="windows-cryptoapi"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#diff-cmd"=""
"#diff-extensions"="-u"
"#diff3-cmd"=""
"#diff3-has-program-arg"=""
"#editor-cmd"="notepad"
"#merge-tool-cmd"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#enable-auto-props"="no"
"#global-ignores"="*.o *.lo *.la *.al .libs *.so *.so.[0-9]* *.a *.pyc *.pyo *.rej *~
 #*# .#* .*.swp .DS_Store"
"#interactive-conflicts"="yes"
"#log-encoding"=""
"#mime-types-file"=""
"#no-unlock"="no"
"#preserved-conflict-file-exts"="doc ppt xls od?"
"#use-commit-times"="no"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auto-props]
```

例 7.1 “系统注册表项文件 (.reg) 的一个示例” 展示了一个 *.reg* 文件，文件内包含了最常用到的配置 选项以及它们的默认值. 注意，文件同时包含了全局配置（与网络代理相关的 选项）与每用户配置（例如编辑器和密码缓存等），而且所有的选项都被注释了，为了修改配置，你只需要删除选项名前的井号 (#)，然后 再修改选项值.

运行时配置选项

本节我们将介绍 Subversion 目前支持的运行时配置选项.

通用配置选项

文件 *config* 包含了与网络无关的运行时配置 选项，到目前为止用到的配置选项并不多，为了方便今后扩展，这些选项 被分组到了不同的节内.

`[auth]` 节包含了和仓库认证与授权有关的配置，它包含的配置选项有：

`password-stores`

选项值是由逗号分隔的列表，列表指出了当 Subversion 存放 和检索缓存的认证证书时，应该使用操作系统提供的哪些方法，以及 这些方法的优先级. 默认值是 `gnome-keyring, kwallet, keychain, gpg-agent, windows-crypto-api`, 分别表示 GNOME Keyring, KDE Wallet, Mac OS Keychain, GnuPG Agent 和 Microsoft Windows 密码 API. 如果列表中出现了操作系统不支持 的方法，该方法将被忽略.

`store-passwords`

该选项已经被 *config* 弃用，它现在 在文件 *servers* 内，作为一个每服务器配置项，见 “每服务器配置”一节.

`store-auth-creds`

该选项已经被 *config* 弃用，它现在 在文件 *servers* 内，作为一个每服务器配置项，见 “每服务器配置”一节.

`[helpers]` 节控制 Subversion 所使用的外部 程序，它包含的配置选项有：

`diff-cmd`

指定差异比较工具的绝对路径，Subversion 在生成 “差异” 输出时（例如执行命令 *svn diff*）将会用到该工具. 默认情况下，Subversion 使用的是自己内部的差异比较库函数来生成差异输出—设置该 选项将导致 Subversion 转而使用外部差异比较工具. 关于如何使用 外部差异比较工具的更多信息，见 “使用外部差异比较与合并工具”一节.

`diff-extensions`

就像命令行选项 `--extensions (-x)`, 该选项向差异比较引擎（内部的或外部的）传递额外的参数. 选项的有效值与 Subversion 所使用的差异比较 引擎有关，更多的细节见 `svn help diff` 的输出. 该选项的默认值是 `-u`.

`diff3-cmd`

该选项指定三路差异比较工具的绝对路径. 当工作副本从服务器 接收更新时， Subversion 将使用该工具把服务器上的更新合并到本地. 默认情况下 Subversion 使用的是自己内部的差异比较库函数来合并 更新—设置该选项将导致 Subversion 转而使用外部工具来完成 合并. 关于如何使用三路差异比较工具的更多信息，见 “使用外部差异比较与合并工具”一节.

`diff3-has-program-arg`

如果由选项 `diff3-cmd` 指定的程序支持 命令行参数 `--diff-program`, 那么就应该把 `diff3-has-program-arg` 设置成 `true`.

`editor-cmd`

指定文本编辑器程序. 当 Subversion 向用户请求输入文本，或者交互式地解决冲突时将会用到该程序. 关于 Subversion 使用 外部编辑器的更多信息，见 “使用外部编辑器”一节.

`merge-tool-cmd`

指定一个合并工具，用于 Subversion 执行三路合并操作. 关于这种工具的更多信息，见 “使用外部差异比较与合并工具”一节.

`[tunnels]` 节允许用户定义新的隧道方案，用 于 *svnserve* 和 `svn://` 的 客户端连接，更多的细节见“SSH 隧道”一节.

`[miscellany]` 节则包含了不属于其他地方的 所有选项，在 `[miscellany]` 里，你可以找到：

`enable-auto-props`

指示 Subversion 为新增和导入的文件自动设置属性. 选项的默认 值是 `no`，将其设置成 `yes` 将开启自动属性设置. 配置文件内的 `[auto-props]` 节用于指定哪些文件应该设置什么 样的属性.

`global-ignores`

执行命令 *svn status* 时，除了被版本控制 的文件外，Subversion 同时还会列出未被版本控制的文件，但是这些 文件的名字前会出现一个 ? 字符（见 “查看修改的整体概述”一节). 列出这些未被版本 控制的项目有时候会让人感到讨厌—例如编译过程中生成的目标 文件就没必要列出来. 选项 `global-ignores` 包含 了由空白字符分隔的文本表达式列表，如果某个文件的名字与这些表达式 中的一个或多个相匹配，*svn status* 就不会显示 这个文件名，除非它是被版本控制的文件. 选项的默认值是 `*.o *.lo *.la *.al .libs *.so *.so.[0-9]* *.a *.pyc *.pyo *.rej *~ #*# .#* .*.swp .DS_Store`.

和 *svn status* 一样，命令 *svn add* 和 *svn import* 也会忽略与 `global-ignores` 匹配的文件. 为了显式地表示 不想忽略某个文件，可以在执行这三个命令时，显式地指定文件名，或使用命令行参数 `--no-ignore`.

关于如何更加精细地忽略文件，见 “忽略未被版本控制的项”一节.

`interactive-conflicts`

指示 Subversion 是否应该尝试交互地解决冲突. 如果它的值是 `yes`（默认值），在冲突发生时，Subversion 将按照 “解决冲突”一节 介绍的方式，询问用户如何处理冲突；否则的话，Subversion 只会给文件设置冲突 标记，然后继续往下操作，从而推迟解决冲突.

`log-encoding`

该选项用于设置日志消息的默认字符编码，它是命令行参数 `--encoding`（见 svn 选项) 的等效选项. Subversion 仓库使用 UTF-8 编码存放日志消息，并且假设用户使用的是操作系统 的默认编码来编写日志消息，如果你不是用的操作系统的默认编码 来编写日志消息的话，那你就应该把字符编码的名字写到 `log-encoding` 里.

`mime-types-file`

该选项在 Subversion 1.5 引入，用于指定一个 MIME 类型映射 文件，例如由 Apache HTTP 服务器提供的 *mime.types*. Subversion 使用 MIME 类型映射 文件为新增或导入的文件设置 MIME 类型. 关于 Subversion 如何 检测和使用文件内容类型的更多信息，见 “自动属性设置”一节 和 “文件内容类型”一节.

`no-unlock`

这个选项和 *svn commit* 命令行选项 `--no-unlock` 相对应，命令行选项 `--no-unlock` 告诉 Subversion 在提交修改后不 要释放文件上的锁. 如果把 `no-unlock` 设置 成 `yes`，Subversion 就不会自动释放锁，为 了解锁，用户必须显式地执行 *svn unlock*. 默认值是 `no`.

`preserved-conflict-file-exts`

选项的值是由空格分隔的文件扩展名列表，当 Subversion 生成冲突文件名时必须保留这些扩展名. 该选项在 Subversion 1.5 引入，默认值为空.

当 Subversion 检测到有冲突发生时，会把冲突的解决交由用户 处理. 为了帮助解决冲突，Subversion 把互相冲突的两个版本的文件 保存到工作副本里，默认情况下，这些文件的名字是原来的文件名 再加上一个特定的扩展名，例如 *.mine* 或 *.REV*(*REV* 表示一个版本号). 这种命名方式 可能产生的一个问题是在某些操作系统上，文件扩展名决定了用于打开它的默认应用程序，为文件名添加一个特定的扩展名可能会导致无法 使用正确的应用程序轻易地打开文件. 比如说文件 *ReleaseNotes.pdf* 有冲突发生，冲突文件可 能被命名成 *ReleaseNotes.pdf.mine* 和 *ReleaseNotes.pdf.r4231*，操作系统使用 Adobe Acrobat Reader 打开扩展名为 *.pdf* 的文件，但是很可能没有默认程序能够打开扩展名为 *.r4231* 的文件.

配置选项 `preserved-conflict-file-exts` 可以帮助用户解决这一问题. 对于扩展名和选项相匹配的文件，Subversion 像往常一样在冲突文件的名字末尾加上特定的扩展名后，会再次添加文件原来的扩展名. 对于前面的例子，假设 `preserved-conflict-file-exts` 的值含有 `pdf`，那么为 *ReleaseNotes.pdf* 所生成的冲突文件将 被命名成 *ReleaseNotes.pdf.mine.pdf* 和 *ReleaseNotes.pdf.r4231.pdf*. 因为冲突文件 的扩展名仍然是 *.pdf*，因此在打开它们时 会用到正确的默认应用程序.

`use-commit-times`

通常情况下，工作副本里的文件包含了能够反映文件最后一次是 在什么时候被进程—例如文本编辑器或 *svn* 命令—修改了的时间戳. 这个时间戳对于软件开发非常有用，因 为软件构建系统通过查看时间戳来决定是否需要重新编译某个文件.

然而在其他场景下，如果这个时间戳反映的是工作副本里的文件 最后一次在仓库里被修改的时间—这在某些情况下会非常方便. 命令 *svn export* 总是把它所生成的目录树的时间 戳设置成“最后一次提交的时间”. 如果把 `use-commit-times` 设置成 `yes`，那么命令 *svn checkout*, *svn update*, *svn switch* 和 *svn revert* 也会把它们所修改的文件的时间戳设置成最后一次提交的时间.

`[auto-props]` 节控制着 Subversion 客户端如何 为新增和导入的文件自动设置属性. 它包含了任意数量的键值对，键值对的 格式是 *PATTERN = PROPNAME=VALUE*`[;`*PROPNAME=VALUE* `...]`，其中 *PATTERN* 是一个文件 名模式，用于匹配一个或多个文件名，剩下的部分是分号分隔的属性赋值语句.（如果在属性名或属性值中需要用到分号，就连着写两次分号来转义.）

```
$ cat ~/.subversion/config
…
[auto-props]
*.c = svn:eol-style=native
*.html = svn:eol-style=native;svn:mime-type=text/html;; charset=UTF8
*.sh = svn:eol-style=native;svn:executable
…
$ cd projects/myproject
$ svn status
?       www/index.html
$ svn add www/index.html
A         www/index.html
$ svn diff www/index.html
…
```

```
Property changes on: www/index.html
___________________________________________________________________
Added: svn:mime-type
## -0,0 +1 ##
+text/html; charset=UTF8
Added: svn:eol-style
## -0,0 +1 ##
+native
$
```

如果一个文件匹配了多个模式，它就会被设置上多个属性，但 Subversion 无法保证属性会按照出现在配置文件里的顺序设置到文件上，所以说用户不能 期望用一个属性设置去覆盖另一个属性设置. 用户可以在文件 *config* 里看到自动属性设置的几个示例. 如果你希望 开启自动属性设置，别忘了把 `[miscellany]` 节的 `enable-auto-props` 选项设置成 `yes`.

Subversion 1.8 增加了一个新的 `[working-copy]` 节，用于配置工作副本，它包含的选项有：

`exclusive-locking-clients`

开启工作副本的 SQLite 互斥锁， 它可用于提升位于网络磁盘上 的工作副本的性能. 如果把该选项设置成 `svn`, Subversion 命令行客户端将使用互斥锁，这会降低加锁的开销，但 并不是说同一时刻只能允许一个客户端访问工作副本，如果工作副本 已经被锁定了，那么试图访问工作副本的其他客户端会阻塞 10 秒钟，然后返回一个错误. 在大多数情况下，人们更喜欢用共享锁，但是如果 工作副本是在网络磁盘（而不是本地磁盘）上，那么锁的开销就更值得 关注. 对于放在网络磁盘上的大型工作副本，使用互斥锁会得到很大的 性能提升，在某些情况下甚至能达到两三倍的提升. 该选项在 Subversion 1.8 引入.

`exclusive-locking`

如果把该选项设置成 `true`, 将为所有 Subversion 1.8 客户端开启工作副本的互斥锁，而某些客户端可能 会无法正常工作. 选项的默认值是 `false`, 在 Subversion 1.8 引入.

每服务器配置

文件 *server* 包含了与网络层有关的配置选项. 文件内有两个特殊的节—`[groups]` 和 `[global]`. `[groups]` 节本质上 是一个交叉引用表. 节内选项的名字是文件内其他节的名字，选项的值是 文本表达式 (*globs*) —可能包含通配符的文本. 向服务器发送请求时，Subversion 将会把 服务器的主机名和这些文本表达式进行匹配.

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
…

[collabnet]
…
```

如果 Subversion 需要访问网络，它将尝试把服务器的主机名和 `[groups]` 节中的选项进行匹配. 如果匹配成功，Subversion 就会在 *servers* 里查找以选项名命名的 节，然后从该节读取与这个服务器有关的配置.

如果服务器的主机名不与 [groups] 内的任意一 个文本表达式相匹配，Subversion 就从 [global] 读取 服务器配置. [global] 所包含的选项与其他节完全 相同（当然，除了 [groups] 节），这些选项有：

http-auth-types

选项包含了客户端支持的 HTTP 认证类型， 类型之间用分号分开. 有效的认证类型包括 basic，digest 和 negotiate，默认行为是允许所有的认证类型. 举个例子，如果客户端要求不能以 明文方式传输认证证书，可以把该选项设置成 digest;negotiate—忽略 basic 类型.（注意，这种设置只对基于 Neon 的 HTTP 模块才有效.）

http-compression

指定是否对发往 DAV 已就绪的服务器的网络数据进行压缩，默认值是 yes（但只有在网络层支持压缩的 情况下才会真正对数据进行压缩）. 把选项设置成 no 将禁止压缩，例如当我们需要对网络传输 进行调试时.

http-library

该选项允许用户指定他们更愿意使用哪种 WebDAV 访问模块（通常是每服务器配置）. 在 1.8 版之前，Subversion 提供了两种 WebDAV 访问模块：较老的 libsvn_ra_neon（对应的选项值是 neon）和较新的 libsvn_ra_serf（对应的选项值是 serf）. 从 1.8 开始，虽然 Subversion 只支持 libsvn_ra_serf，但这个配置选项还是保留了 下来，因为运行时配置区域无法知晓 Subversion 的版本. 如果用户 的系统中安装了多种 Subversion 版本，在使用旧版客户端访问服务 器时，他们可能仍然希望为 http-library 设置上 libsvn_ra_neon.

http-proxy-exceptions

选项值是由逗号分隔的列表，列表的元素是仓库服务器的主机 名模式，符合模式的服务器将不会使用代理，而是直接访问. 模式 的写法和 Unix Shell 的文件名模式写法相同.

http-proxy-host

指定基于 HTTP 的 Subversion 请求必须经过的代理服务器的 主机名. 默认值是空，这意味着 Subversion 不会尝试把 HTTP 请 求路由给代理服务器，而是直接发给目标服务器.

http-proxy-password

指定提供给代理服务器的密码，默认值是空.

http-proxy-port

指定代理服务器的端口号，默认值是空.

http-proxy-username

指定提供给代理服务器的用户名，默认值是空.

http-timeout

指定服务器响应的等待时间， 以秒为单位. 如果用户发现低速 的网络常常导致 Subversion 命令超时， 那你应该把 http-timeout 调大. Subversion 1.8（或更 老的，基于 Serf 的版本）用 0 表示不设置 超时.

`neon-debug-mask`

选项值是一个整数掩码，Neon HTTP 库函数将根据这个掩码来 决定打印哪些调试信息，默认值是 `0`，这会 禁止打印所有的调试信息. 在 1.8 之前的版本，大多数 Subversion 客户端都使用 Neon（借助 `libsvn_ra_neon` 仓库访问模块）完成客户端和服务器之间的 WebDAV/HTTP 通信. 但从 Subversion 1.8 开始不再支持 `libsvn_ra_neon`，所以这个选项在新版 Subversion 里被淘汰了.

`ssl-authority-files`

选项值是分号分隔的路径列表，其中每个路径都指向一个文件，该文件包含了认证机构的证书，当 Subversion 客户端使用 HTTPS 访问仓库时将信任这些证书.

`ssl-client-cert-file`

如果一个主机（或多个主机）要求提供一个 SSL 客户端证书，用户通常会收到一个要求输入证书路径的提示信息. 把该选项设置 成证书的路径，Subversion 就可以自动搜索到证书，而不会再提示 用户输入证书路径. 并不存在一个存在证书的标准位置，Subversion 会从用户指定的任意位置读取证书.

`ssl-client-cert-password`

如果用户的 SSL 客户端证书被一个密码加密，每当 Subversion 使用证书都会提示用户输入密码. 如果你觉得这有点烦人，而且你也 不介意把密码写到 *servers* 文件里，那就 把该选项设置成证书的密码，这样你就不会再收到要求输入证书密码的 提示.

`ssl-pkcs11-provider`

选项值是 PKCS#11 提供商的名字，它刻画了 SSL 客户端的证书（如果服务器要求提供证书的话）. 这个设置只对 Subversion 基于 Neon 的 HTTP 模块才有效，而这种模块在 Subversion 1.8 中已经被移除了.

`ssl-trust-default-ca`

如果用户希望 Subversion 自动信任 OpenSSL 附带的默认证书 的话，就把它设置成 `yes`.

`store-auth-creds`

该选项基于上和 `store-passwords` 相同，唯一的不同点是它用于开启或禁止 所有的 认证信息缓存，包括用户名，密码，服务器证书，以及其他能被缓存 的凭证.

`store-passwords`

指定 Subversion 是否应该缓存用户密码，这个密码是服务器 要求认证时用户输入的. 默认值是 `yes`，如果 设置成 `no` 将禁止缓存密码. 用户可以使用 *svn* 的命令行选项 `--no-auth-cache` 覆盖掉 `store-passwords` 的设置（当然，这种覆盖只 对支持 `--no-auth-cache` 的子命令才有意义）. 关于密码缓存的更多信息，见 “缓存证书”一节. 注意，无论是否设置 `store-passwords`，Subversion 都不会以明文方式存放密码，除非 `store-plaintext-passwords` 被设置成 `yes`.

`store-plaintext-passwords`

该选项只在类 Unix 系统上才需要注意. 如果密码只能以明文方式 缓存在磁盘上（具体的位置是 *~/.subversion/auth/*），此时 Subversion 将 根据该选项决定接下来该怎么做. 如果把它设置成 `yes` 或 `no`，将分别允许 或禁止以明文方式缓存密

码. 默认值是 ask, 于是 Subversion 每当遇到一个 新的 密码时, 都会询问用户是否以明文方式把它缓存到 *~/.subversion/auth/*.

`store-ssl-client-cert-pp`

该选项用于设置 Subversion 是否应该缓存用户输入的 SSL 客户端证书密码, 默认值是 yes, 设置成 no 将禁止缓存.

`store-ssl-client-cert-pp-plaintext`

当 Subversion 尝试缓存 SSL 客户端证书密码时, 该选项用于 控制是否允许以明文方式存放到磁盘上. 默认值是 ask, 使得 Subversion 每次遇到一个 新的 SSL 客户端证书密码时都会向用户询问 是否以明文方式缓存到 *~/.subversion/auth/*, 把选项设置成 yes 或 no 将允许或禁止明文缓存, Subversion 也不会再询问用户.

本地化

本地化 (*localization*) 指 的是让程序按照某个地区特有的方式进行运转的行为. 如果程序按照你所习惯的 方式来格式化数字或日期, 或者使用你的母语来输入（或输出）消息, 那我们就 说该程序是 本地化的 (*localized*). 本节介绍 Subversion 在本地化上所 做的工作.

理解地区

大多数现代操作系统都有一个 “当前地区” 的概念— 即遵守本地化惯例的地区或国家. 这些惯例—在典型情况下由系统的运行 时配置机制来选取—会影响程序向用户呈现数据以及向用户接收数据的方 式.

在大多数类 Unix 系统上, 你可以用命令 *locale* 查看和地区有关的运行时配置选项的值:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
$
```

命令的输出是与地区有关的环境变量, 以及它们的当前值. 在上面的示例中, 变量的值都是默认的 C, 但用户可以把它们再设置成其他 国家/语言的编码组合. 例如, 如果把 LC_TIME 设置成 fr_CA, 程序就会知道要按照说法语的加拿大人的文化习惯 来呈现时间和日期; 如果把 LC_MESSAGES 设置成 zh_TW, 程序就会知道要使用繁体中文来显示消息. 设置 LC_ALL 相当于把每一个变量都设置成同一个值. 如果某个 变量没有设置, 它就会使用 LANG 的值作为默认值. 为了 得到 Unix 系统上所有可用的地区, 执行 `locale -a`.

Windows 配置地区的地方在控制面板的 “地区和语言选项”，在这里你可以查看和选择各种地区设置, 甚至可以非常细致地修改显示方式的 既有格式.

Subversion 对本地化的支持

Subversion 客户端命令行工具 *svn* 使用两种方式 支持本地化配置. 第一种方式是根据环境变量 LC_MESSAGES 的值来决定按照哪一种语言打印所有的消息, 例如:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

无论是 Unix, 还是 Windows, 这种行为的表现都是一样的. 但是, 你的 操作系统可能只支持特定的地区, 导致 Subversion 无法使用某些语言打印 消息. 为了输出本地化的消息, 需要有志愿者将每一条消息都翻译成本地语言. 译文使用 GNU 的 gettext 软件包进行编写, 得到的译文文件以 *.mo* 作为扩展名, 例如德文的译文文件是 *de.mo*. 这些译文文件安装在系统中的某个位置, 在 Unix 系统上, 它们的典型位置是 */usr/share/locale/*, 而 Windows 的典型位置是 Subversion 安装目录的 *share\locale* 子目录内. 译文文件一旦安装完成, 它将按照它所服务的程序来命名, 例如 *de.mo* 安装后, 最终的名字可能是 */usr/share/locale/de/LC_MESSAGES/subversion.mo*. 通过浏览已安装的 *.mo* 文件, 你就可以知道你的 Subversion 客户端可以说哪些语言.

Subversion 支持本地化的第二种方式涉及到 *svn* 如何 解释用户的输入. 仓库使用 UTF-8 编码记录所有的路径, 文件名和日志消息. 在这种情况下, 我们可以说仓库是 国际化的 (*internationalized*)—也就是说仓库已经准备 好接受任意一种人类语言. Subversion 客户端必须负责只向仓库发送 UTF-8 编码的文件名和日志消息, 所以它必须将本地字符编码的数据转化成 UTF-8 编码.

例如, 你创建了一个文件 *caffè.txt*, 然后在提交 时把日志消息写成了 “Adesso il caffè è più forte.” 文件名和 日志消息都包含了非 ASCII 字符, 由于你把本地语言设置成了 it_IT, 所以 Subversion 客户端知道它们是意大利文, 然后它使用意大利字符集把文件名和日志消息转换成 UTF-8 编码, 再发送给 仓库.

注意, 虽然仓库要求文件名和日志消息必须是 UTF-8 编码, 但它并不在乎 文件的内容是什么编码, Subversion 把文件的内容看成是不透明的字节串, 服务器端和客户端都不会试图去理解文件内容的字符集或编码方案.

字符集转换错误

在使用 Subversion 时, 你可能会遇到下面这种错误:

```
svn: E000022: Can't convert string from native encoding to 'UTF-8':
…
svn: E000022: Can't convert string from 'UTF-8' to native encoding:
…
```

发生这种错误的典型情况是 Subversion 客户端接收到了仓库发来的 UTF-8 字符串, 但并非字符串中的所有字符都能用本地字符集进行表示. 比如说你的 本地语言是 en_US, 但你的同事却提交了一个用日文命名 的文件, 当你执行命令 *svn update* 时可能就会遇到这种 错误.

解决办法之一是使用一种 可以 表示所有 UTF-8 字符的本地语言. 或者是修改仓库中的文件名和日志消息 (别忘了和你的日本 同事打声招呼—应该提前决定好项目所使用的语言, 这样的话所有的 参与者都能使用相同的本地语言).

使用外部编辑器

为了向 Subversion 添加数据，最明显的方式就是添加新文件，或者向已有 的文件提交修改，但是除文件内容之外的其他信息则只存在于仓库中，其中一些 信息—例如日志消息，与锁有关的注释，以及某些属性值—是纯文本 的，而且由用户显式提供. 这些信息中的大部分都可以用选项 `--message (-m)` 和 `--file (-F)` 传递给 Subversion 客户端.

这两个选项各有利弊. 例如，当你执行提交操作时，如果你已经事先把日志 消息写到了一个文本文件里，那么选项 `--file (-F)` 就可以工作得很好. 如果你还没有把日志消息写到一个 文件里，那你可以通过选项 `--message (-m)` 把日志消息写到命令行上，不幸的是，如果你想在命令行上输入多于一行的日志 消息就会比较困难. 用户希望更灵活的输入形式—多行文本和不受约束的 日志消息编辑.

Subversion 允许用户指定一个外部文本编辑器，必要时，Subversion 会 启动该编辑器，从而允许用户更加灵活地输入文本信息（例如日志消息和属性 值）. 有多种方式用于指定外部编辑器，当 Subversion 想要启动外部编辑器时，将按照如下顺序查看应该启动哪个编辑器：

1. 命令行选项 `--editor-cmd`

2. 环境变量 `SVN_EDITOR`

3. 运行时配置选项 `editor-cmd`

4. 环境变量 `VISUAL`

5. 环境变量 `EDITOR`

6. 可能是被编译到 Subversion 库中的备用编辑器（在官方构建的版本中 不存在）

上面任意一个选项或变量的值都是一个将被 shell 执行的命令行的开头，Subversion 会在命令行的后面加上一个空格，然后再加上一个临时文件的路径. 所以说，为了能被 Subversion 使用，所指定的编辑器必须支持它的最后一个命令 行参数是待编辑的文件，而且在保存文件时不能更改路径，编辑器成功退出时 必须返回 0.

外部编辑器可以为提交类的子命令（例如 *svn commit*, *svn import*, *svn mkdir* 或 *svn delete*）提供日志消息，如果没有指定选项 `--message (-m)` 和 `--file (-F)`，那么 Subversion 就会尝试 启动外部编辑器，命令 *svn propedit* 几乎就是在围绕着 外部编辑器. 从 Subversion 1.5 开始，Subversion 还可以在交互式地解决冲突 时，按照用户的要求来启动外部编辑器. 奇怪的是，在编写锁注释时无法使用 外部编辑器.

使用外部差异比较与合并工具

Subversion 与二路以及三路差异比较工具之间的接口可以一直追溯到它的 起源时期，那时候 Subversion 的差异比较功能直接来自 GNU 工具的调用，尤其是 *diff* 和 *diff3*. Subversion 为了得到期望的效果，在调用这些工具时会加上一些参数或选项，其中的大部分 选项都是工具所特有的. 随着 Subversion 的不断开发，它渐渐地具有了自己的 差异比较库函数，但同时也为客户端工具增加了选项 `--diff-cmd` 和 `--diff3-cmd`，于是用户 可以告诉客户端使用 GNU 差异比较工具 *diff* 和 *diff3*，而非内建的差异比较库函数. 如果使用了选项 `--diff-cmd` 或 `--diff3-cmd`，Subversion 就会忽略内建的差异比较库函数，转而调用外部的差异比较程序，在调用时传递 必要的选项.

读者应该很容易想到，既然 Subversion 可以使用 GNU 的 *diff* 和 *diff3*，那当然也可以使用 其他的差异比较工具，毕竟 Subversion 并不知道它所调用的工具到底是不是 GNU 的差异比较工具. 不过，在使用这些外部工具时，唯一可配置的地方就是它 们在系统中的路径—而非选项或参数的顺序等. Subversion 仍然会像往常 一样把 GNU 工具的选项传递给你所指定的外部差异比较工具，无论它们理不理解 这些选项，大多数用户的困惑即来自于此.

什么时候决定调用差异比较工具由 Subversion 内部进行决定，而且还 受到文件是否是文本文件的影响，后者由属性 `svn:mime-type` 决定. 比如说，即使说你有一个漂亮的，可以对 Microsoft Word 文档进行差异比较和合并的工具，但是如果 Word 文档的 MIME 类型是二进制文件（例如 `application/msword`），那么 Subversion 也不会调用该 工具. 幸运的是，你可以为 *svn diff* 增加选项 `--force`，强制 Subversion 对文件进行差异比较. 关于 MIME 类型设置的更多内容，见 “文件内容类型”一节

Subversion 1.5 引入了交互式的冲突解决（见 “解决冲突”一节），这项特性提供的选项之一是允许用户 交互式地调用一个第三方合并工具. 如果 Subversion 需要调用第三方合并工具，它就会检查用户是否已经指定了工具，它首先检查环境变量 `SVN_MERGE`，如果该变量没有被设置，它就继续检查运行时 配置选项 `merge-tool-cmd`. 一旦找到外部合并工具，它就 调用该工具.

由于三路差异比较工具和合并工具的目标是基本相同的 （把各自的， 但 是互相重叠的修改和谐地合并到一起），Subversion 会根据不同的情景使用 它们. 在与用户交互时调用 Subversion 内建的三路差异比较引擎和它的外部 替代品其实是有风险的，因为使用这些工具而耽搁的时间可能会导致某些对 时间比较敏感的操作失败，这时候 Subversion 更愿意交互式地调用外部合并 工具.

虽然 Subversion 与外部合并工具之间的接口，比 Subversion 与 *diff*，*diff3* 之间的接口要简单得多，但是能够找到完全符合 Subversion 要求的外部合并工具还是没那么简单的. 为 Subversion 指定外部差异比较和合并工具的关键是使用包装脚本，包装脚本 的功能是把 Subversion 传过来的参数转换成外部工具能够理解的参数，然后再 把外部工具的输出转换成 Subversion 支持的格式. 下面几节进行了具体的介绍.

外部差异比较工具

Subversion 按照 GNU *diff* 命令的要求向外部差异 比较工具传递参数，同时期望外部差异比较工具按照 GNU *diff* 的要求返回正确的退出值. 对于大多数差异比较 工具，它们通常只对第 6 和第 7 个参数—分别表示差异的左边内容与右边 内容的文件路径—感兴趣. 注意，Subversion 为第一个被修改的文件 运行一次差异比较工具，所以说如果你的差异比较工具是异步运行的（或者说 在后台运行），那么多个运行实例可能会同时运行. 最后，如果差异比较工具 检测到了有差异，则 Subversion 希望工具返回退出值 1，如果没有检测到差异 则返回退出值 0，若返回其他退出值则认为工具遇到了错误.[2]

例 7.2 “diffwrap.py” 和 例 7.3 “diffwrap.bat” 分别展示了如何使用 Python 和 Windows 批处理脚本编写外部差异比较工具 的包装脚本.

例 7.2. diffwrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite diff program here.
```

[2] GNU *diff* 的手册是这么说的：“退出值 0 表示没有检测到差异，1 表示检测到差异，2 表示遇到了错误.”

```
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.
```

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.
```

外部三路差异比较工具

当 Subversion 要执行非交互式的合并时就会调用三路差异比较程序，如果被调用的是外部工具，那么 Subversion 就会按照 GNU *diff3* 的要求向外部工具传递参数，并且期望外部工具 以表示成功的值退出，而合并完成后的全部文件内容是打印到标准输出（这样的话 Subversion 就能把它们重定向到任意一个文件中）. 对于大多数可 供选择的合并工具来说，它们只对第 9, 第 10 和第 11 个参数感兴趣，这 3 个参数分别表示 “自己的”，“较老的” 和 “你的” 文件路径. 注意，由于 Subversion 依赖合并工具所 产生的输出，因此你的脚本必须等到工具的输出全部都传递给 Subversion 后才能退出. 当脚本最终退出时，如果合并成功，那它应该以 0 作为退出值，如果还有未解决的冲突，那就以 1 作为退出值，除了 0 和 1 之外的其他值 都被视为发生了严重的错误.

例 7.4 “diff3wrap.py” 和 例 7.5 “diff3wrap.bat” 是 外部三路差异比较工具的包装脚本模板，分别用 Python 和 Windows 批处理脚本编写.

例 7.4. diff3wrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite three-way diff program here.
DIFF3 = "/usr/local/bin/my-diff3-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 three-way diff command (change the following line to make
# sense for your three-way diff 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 three-way diff program here.
SET DIFF3="C:\Program Files\Funky Stuff\My Diff3 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 three-way diff command (change the following line to make
REM sense for your three-way diff 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.
```

外部合并工具

在交互式地解决冲突时，Subversion 可以选择调用一个外部合并工具. Subversion 提供给外部合并工具的参数有：未修改的基础文件路径，“他们的” 文件路径（含有上游修改），“我们的” 文件路径（含有本地修改），用于存放合并工具合并后的文件路径，以及发生 冲突的文件的工作副本路径（相对于合并操作的原始目标）. 如果执行成功，合并工具应该返回 0，如果发生错误则返回 1.

例 7.6 “mergewrap.py” 和 例 7.7 “mergewrap.bat” 是 外部合并工具的包装脚本模板，分别用 Python 和 Windows 批处理脚本编写.

例 7.6. mergewrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite merge program here.
MERGE = "/usr/local/bin/my-merge-tool"

# Get the paths provided by Subversion.
BASE   = sys.argv[1]
THEIRS = sys.argv[2]
MINE   = sys.argv[3]
MERGED = sys.argv[4]
WCPATH = sys.argv[5]

# Call the merge command (change the following line to make sense for
# your merge program).
cmd = [MERGE, '--base', BASE, '--mine', MINE, '--theirs', THEIRS,
              '--outfile', MERGED]
os.execv(cmd[0], cmd)

# Return an errorcode of 0 if the conflict was resolved; 1 otherwise.
# Any other errorcode will be treated as fatal.
```

例 7.7. mergewrap.bat

```
@ECHO OFF

REM Configure your favorite merge program here.
SET MERGE="C:\Program Files\Funky Stuff\My Merge Tool.exe"
```

```
REM Get the paths provided by Subversion.
SET BASE=%1
SET THEIRS=%2
SET MINE=%3
SET MERGED=%4
SET WCPATH=%5

REM Call the merge command (change the following line to make sense for
REM your merge program).
%MERGE% --base %BASE% --mine %MINE% --theirs %THEIRS% --outfile %MERGED%

REM Return an errorcode of 0 if the conflict was resolved; 1 otherwise.
REM Any other errorcode will be treated as fatal.
```

小结

有时候，只能用一种正确的方式来完成任务，有时候却存在许多种正确方式．开发人员明白虽然 Subversion 的大多数行为对于大多数用户来说都是可接受的，但总会存在无法令人满意的角落．在这种情况下，Subversion 允许用户决定如何 处理．

本章我们介绍了 Subversion 的运行时配置系统，以及其他可用来配置 运行行为的机制．如果你是一个开发人员，下一章将从开发人员的角度介绍如何 利用 Subversion 提供的库函数，进一步定制 Subversion．

第 8 章 嵌入 Subversion

Subversion 具有模块化的设计：它由众多由 C 编写而成的库函数实现．每一个 库函数都有一个定义良好的目标和应用程序编程接口 (Application Programming Interface, 简称 API), 这种接口不仅可以被 Subversion 使用，还能被任意一个希 望通过编程接口控制 Subversion 的软件使用．另外，Subversion 的 API 不仅能被 C 程序使用，也能被其他高级语言编写的程序使用，例如 Python, Perl, Java 和 Ruby.

本章的目标读者是那些希望通过 Subversion API 或它的各种语言绑定来控制 Subversion 的人．如果你希望围绕 Subversion 编写健壮的包装脚本来简化 你的工作， 或者正在开发 Subversion 与其他软件之间更加复杂的集成， 或者仅仅 是对 Subversion 的库函数感到好奇，那你应该认真阅读本章．但是如果你觉得 自己没必要从开发的层次上使用 Subversion, 那你完全可以跳过本章，这并不会 影响到你作为一个 Subversion 普通用户的体验.

层次化的函数库设计

Subversion 的每一个核心函数库都隶属于三个层次之一──仓库层，仓库访问 (Repository Access, 简称 RA) 层和客户端层（见 图 1 “Subversion 的架构”). 我们将会简单地介绍这些层次，但是在这之前，先简单地总结一下 Subversion 的各个函数库．为了保持一致，我们把函数库的名字写成删除了扩展名后，在 Unix 中的库文件名（例如 *libsvn_fs*, *libsvn_wc*, *mod_dav_svn* 等）.

libsvn_client

客户端程序的主要接口

libsvn_delta

目录树和字节流差异比较例程

libsvn_diff

文件内容差异比较和合并例程

libsvn_fs

文件系统公共函数和模块加载例程

libsvn_fs_base

Berkeley DB 文件系统后端

libsvn_fs_fs

原生文件系统 (FSFS) 后端

libsvn_ra

仓库访问公共例程和模块加载例程

libsvn_ra_local

本地仓库访问模块

libsvn_ra_serf

另一个 WebDAV 仓库访问模块（试验性的）

libsvn_ra_svn

使用定制协议的仓库访问模块

libsvn_repos

仓库接口

libsvn_subr

各种辅助例程

libsvn_wc

工作副本管理函数库

mod_authz_svn

Apache 授权模块，用于借助 WebDAV 的 Subversion 仓库访问

mod_dav_svn

Apache 模块，用于将 WebDAV 操作映射到 Subversion 的 对应操作

在上面的介绍中，“各种” 这个词只出现了一次，这是一个 好现象，因为 Subversion 开发团队总是尽量将功能放到正确的层次和函数库 中实现. 站在开发人员的角度来看，模块化设计最大的好处可能是降低了复杂 度，于是你就可以快速地勾勒出 “整体面貌”，更加容易地决定 功能所属的位置.

模块化设计的另一个好处是允许我们重新实现给定的模块， 只要保持 API 兼容 性， 就不会影响其他模块. 在某种意义上， Subversion 已经在这样做了. 函数库 *libsvn_ra_local*, *libsvn_ra_serf* 和 *libsvn_ra_svn* 各自都实现了一套相同的接口，它们都 是作为 *libsvn_ra* 的插件与仓库访问层通信— *libsvn_ra_local* 与仓库直接通信， 另外两个通过网络与 仓库通信. *libsvn_fs_base* 和 *libsvn_fs_fs* 是另一对用不同方式实现了相同接口的函数 库—它们是作为 *libsvn_fs* 的插件.

客户端本身也突出了 Subversion 模块化设计的优越性. 函数库 *libsvn_client* 是开发 Subversion 客户端的一站式商店（见 “客户端层”一节). 所以说虽然 Subversion 发行版提供了命令行客户端工具 *svn*, 但还有 一些第三方程序提供了图形化的客户端工具，这些图形化工具使用了和命令行工具 相同的 API. 模块化设计在 Subversion 的推广中起到了非常重要的作用.

仓库层

当说到 Subversion 仓库层时，我们通常谈论的是两个基本概念—版本化 文件系统的实现（通过 *libsvn_fs* 函数库访问，它依赖 *libsvn_fs_base* 和 *libsvn_fs_fs* 这两个函数库），以及围绕它的仓库逻辑（在 *libsvn_repos* 里实现）. 这些函数库为版本化数

据的各 个版本提供了存储和报告机制. 仓库层通过仓库访问层连接到客户端层, 从 Subversion 用户的角度来看, 仓库层是“线段的另一端”

Subversion 文件系统并不是在操作系统内核态实现的文件系统 (在内核态实现 的文件系统有 Linux ext2 或 NTFS 等), 它是一个虚拟文件系统, “文件” 和 “目录” 不是以真实的文件和目录的 形式 (真实的文件和目录就是你在 shell 中能够看到的那些文件和目录) 存放 到磁盘上, 而是使用了两种抽象存储后端—Berkeley DB 或一个平坦文件 系统 (关于这两种存储后端的更多信息, 见 文件系统). Subversion 开发团队甚至 在考虑为 Subversion 支持更多类型的后端数据库系统, 他们或许会通过 ODBC (Open Database Connectivity, 开放数据库连接) 实现这一特性. 实际上, Google 代码托管 (Google Code Project Hosting) 服务已经做过类似的工作: Google 在 2006 年中期宣称他们的开源团队已经开发了一个私有的 Subversion 文件系统插件, 该插件允许 Subversion 使用 Google Bigtable 数据库作为 存储后端.

其他文件系统 API 所能提供的功能, *libsvn_fs* 的 API 也能提供—你可以创建或删除文件和目录, 复制或移动, 修改文件内容等. 除此之外, *libsvn_fs* 还提供了不太常见的功能, 例如 在文件和目录上添加, 修改和删除元数据 (“属性”). 更重要的是, Subversion 的文件系统是一个版本化的文件系统, 这意味着在你修改目录树时, Subversion 记住了目录树被修改前的样子, 以及上次修改前的样子, 上上次 修改前的样子, 可以一直追溯到文件系统被创建的时候.

针对目录树的所有修改都是在一个 Subversion 提交事务的上下文中完成, 下面是修改文件系统的简化过程:

1. 开始一个 Subversion 提交事务.
2. 执行修改 (添加, 删除, 修改属性等).
3. 提交事务.

事务一旦提交, 文件系统的修改就已经作为历史财产持久化地保存下来. 每次轮回都会产生一个新的版本号, 每个版本号都是一个永远可访问的只读 快照.

两种事务

用户很容易把 Subversion 的事务与后端数据库所提供的事务支持弄混, 特别是考虑到前者在 *libsvn_fs_base* 里的代码和 Berkeley DB 数据库的代码非常接近. 这两种事务都用于提供原子性和隔离性, 换句话说, 事务允许用户以这样一 种方式执行一个操作集合—要么集合中的所有操作都执行成功, 要么一 个都不执行—同时不会干扰到操作数据的其他进程.

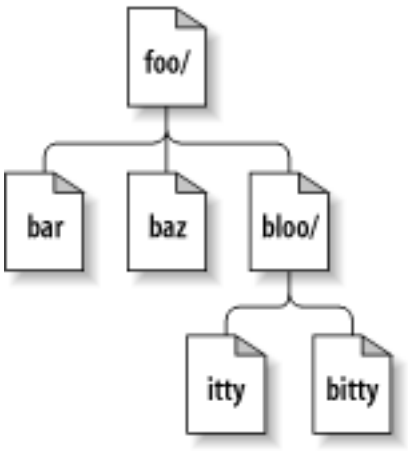
数据库事务所围绕的操作通常与修改数据库的数据有关 (例如修改表的一 行), 而 Subversion 事务所涵盖的范围更大, 层次也更高, 例如针对文件或 目录集合的修改, 这些修改将作为一个新的版本号存放到文件系统中. 如果 读者的思路还跟得上, 再考虑这样一个事实—Subversion 是在一个数据 库事务中完成 Subversion 事务的创建 (如果 Subversion 事务创建失败, 那么数据库看起来就像是从来没有创建过 Subversion 事务).

幸运的是对于文件系统 API 用户而言, 由数据库所提供的事务支持几乎 是不可见的 (对于模块化的函数库而言, 这应该是人们所期待的效果), 只 有当人们想知道文件系统的实现细节时, 数据库事务才变成可见的.

文件系统接口提供的大多数功能都是针对文件系统中的路径进行操作, 也 就是说从文件系统外部看来, 描述与访问文件版本号的主要机制都要通过路径 字符串 (例如 */foo/bar*) 实施, 类似于在 shell 程序 中处理文件与目录. 你可以用路径参数调用正确的函数来创建新文件与目录, 也可以用其他函数查询文件信息.

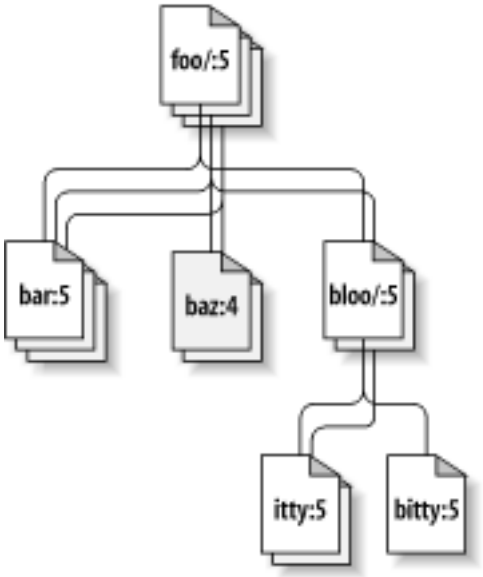
与大多数文件系统不同的是, 仅仅依靠路径无法在 Subversion 中唯一 地识别一个文件或目录. 把目录树看成是一个二维系统, 结点的兄弟代表了 一种横向移动, 而进入结点的子目录则是一种纵向移动. 图 8.1 “二维坐标系下的文件与目录” 展示了一个目录树的 典型表示.

图 8.1. 二维坐标系下的文件与目录



Subversion 文件系统的不同点在于它还有第三个维度—时间 [1] Subversion 文件系统提供的函数如果需要一 个 *path* 参数，那么很可能也会需要一个 *root* 参数，这个 `svn_fs_root_t` 类型的参数或者描述了一个版本号，或者描述了一个 Subversion 事务 (完成中 的版本号)，它所提供的第三个维度可用于区分不同版本号下的 */foo/bar*. 图 8.2 “Subversion 目录树的三维表示” 展示了增加了第三个 维度后，目录树的三维表示.

图 8.2. Subversion 目录树的三维表示



我们前面已经说过，*libsvn_fs* API 类似于其他 文件系统，不同点是它提供了版本控制的功能，不仅限于 Subversion, 任何 对版本化文件系统感兴趣的程序都能使用它的接口. 虽然文件系统 API 对于 基本的文件与目录的版本控制已经提供了足够的支持，但 Subversion 的要求 不仅于此，所以就有了 *libsvn_repos*.

Subversion 仓库函数库 *libsvn_repos* 从逻辑上 讲处于 *libsvn_fs* 之上，在下层版本化文件系统的 基础上提供了更多的功能. *libsvn_repos* 只封装了 文件系统的某些接口，包括 Subversion 事务的创建与提交，版本号属性的修改，之所以要封装这些接

[1] 某些科幻迷可能会感到震惊，因为他们一直以来都认为时间 是第 四 个维度，对此造成的情感创伤我们表示 道歉.

口是因为仓库层为它们关联了钩子. 仓库的钩子系统 并不要求一定要关联到一个版本化文件系统, 因此它们属于仓库函数库.

钩子机制是把仓库函数库与文件系统函数库分开的原因之一. *libsvn_repos* 还提供了以下功能:

- 创建, 打开, 销毁和恢复 Subversion 仓库 (包括仓库里的文件系统).
- 描述两个文件系统树之间的差异.
- 查询全部 (或部分) 版本号的提交日志消息, 在每个版本号中 都有一些文件系统中的文件与目录被修改.
- 生成一个人类可读懂的文件系统 “转储” 文件— 它是文件系统中版本号的完整表示.
- 解析转储文件, 把转储后的版本号加载到另一个 Subversion 仓库里.

随着 Subversion 的不断演变, 仓库函数库将与文件系统函数库共同成长, 提供越来越丰富的功能.

仓库访问层

如果说 Subversion 仓库层是 “线段的另一端”, 那么仓库 访问层 (Repository Access, 简称 RA) 就是线段本身. 仓库访问层充满了 客户端函数库与仓库函数库之间互相传递的数据. 仓库访问层包含的 函数库有模块加载库 *libsvn_ra*, 仓库访问模块本身 (目前的访问模块有 *libsvn_ra_local*, *libsvn_ra_serf* 和 *libsvn_ra_svn*), 以及仓库访问模块所需的其他 函数库 (例如 *mod_dav_svn* 或 *svnserve*).

Subversion 使用 URL 标识仓库资源, URL 的协议部分 (通常是 `file://`, `http://`, `https://`, `svn://` 或 `svn+ssh://`) 决定了使用哪种仓库访问模块处理请求. 每个仓库访问模块都注册了它支持的协议, 于是 RA 加载函数就能在运行时决定使用哪个模块. 用户可以执行 `svn --version` 查看 可用的 RA 模块及其所支持的协议:

```
$ svn --version
svn, version 1.8.0-dev (under development)
   compiled Jan  8 2013, 11:45:25 on i686-pc-linux-gnu

Copyright (C) 2013 The Apache Software Foundation.
This software consists of contributions made by many people;
see the NOTICE file for more information.
Subversion is open source software, see http://subversion.apache.org/

The following repository access (RA) modules are available:

* ra_svn : Module for accessing a repository using the svn network protocol.
  - with Cyrus SASL authentication
  - 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
```

$

RA 层导出的 API 包含了用于发送和接收版本化数据的必要功能，而且每 一个 RA 插件都可以使用一种特定的协议完成这些任务，例如 *libsvn_ra_serf* 使用 HTTP/WebDAV（还可以选择使用 SSL 加密）与运行着 *mod_dav_svn* 模块的 Apache HTTP 服务器通信；*libsvn_ra_svn* 使用一种 Subversion 特有的协议与 *svnserve* 服务器通信.

RA 层使用了模块化的设计，因为 Subversion 开发人员考虑到人们可能 还想使用其他协议访问 Subversion 仓库，这就使得新协议的开发更加方便. 开发人员仅仅需要写一个实现了 RA 接口的函数库，新的函数库可以使用已 有的网络协议或你自己发明的新协议，你甚至可以使用进程间通信 (IPC) 或 基于电子邮件的协议. Subversion 提供了 API，而你则提供创造性.

客户端层

在客户端，工作副本是所有操作发生的地方，客户端实现的所有功能都是为 了更好地管理工作副本—包含了众多文件与子目录的目录，作为一个 或多个仓库在本地的，可编辑的 “映射”—并且向仓库 访问层发送或接收修改.

Subversion 工作副本函数库 *libsvn_wc* 负责管理工作 副本的数据，为了完成这个任务，函数库把工作副本有关的管理信息都存放在 一个特殊的子目录内，这个子目录的名字是 *.svn*，每 个工作副本都有这个目录，目录内包含了用于记录工作副本状态的各种文件与 目录，为管理性的操作提供了一个私有工作空间. 如果读者熟悉 CVS，就会发 现 *.svn* 的功能与 CVS 工作副本里的 *CVS* 目录非常类似.

Subversion 客户端函数库 *libsvn_client* 所负责的工作是最广泛的，它负责混合工作副本函数库与仓库访问层函数 库的功能，进而向应用程序提供最高层次的 API，允许应用程序执行最一 般的版本控制操作. 例如函数 `svn_client_checkout` 接收一个 URL 作为参数，它将 URL 传递给 RA 层，并打开一个关联到特定 仓库的已认证会话，然后函数向仓库请求一个特定的目录树，并将此目录树 发送给工作副本函数库，最终在磁盘上得到一个完整的工作副本（包括目录 *.svn*）.

客户端函数库被设计成可被任意的应用程序使用， Subversion 源代码 包已经包含了一个命令行客户端， 不过基于客户端函数库写出一个 GUI 客户 端并没有多大的难度. 新的客户端没必要封装已有的命令行客户端— 它们完全可以通过 *libsvn_client* API 获得相同的 功能，数据和回调机制. 实际上，Subversion 源代码包包含了一个最小化的 客户端实现（代码在 *tools/examples/minimal_client.c*），展示了如何使 用 Subversion API 实现一个简单的客户端程序.

直接绑定—关于正确性的一些话

为什么你的 GUI 程序应该使用 *libsvn_client* 开发，而不是直接封装一个命令行程序? 前者除了效率更高之外，也更 加正确. 基于客户端函数库开发的命令行程序（例如 Subversion 所提供的）需要把 C 类型的反馈或请求数据高效地翻译成人类可读懂的格式，这种 翻译是有损的，也就是说程序可能无法呈现从 API 获取到的所有信息，或者为了紧凑显示而与其他信息进行组合.

如果你的程序建立在命令行程序的封装之上，那么程序只能访问到已 被翻译过的信息（上面我们刚说过，信息可能是不完整的），而这些信息 将被 *再次* 翻译成 *程序自己的* 表示格式. 每一次封装，原始数据 的完整性被破坏的就越多，类似于拷贝音频或视频磁带的拷贝（的拷贝 …）.

基于 API 进行开发，而不是封装其他程序的另一大原因是 Subversion 保证了 API 的兼容性. 在次版本号不同的 API 之间（例如 1.3 和 1.4），其函数原型不会发生变化，也就是说在升级 Subversion 时，不 必升级你自己的程序. 特定的一些函数可能不再赞成使用，但它们仍然可 以正常工作，这就给了你一定的缓冲时间升级到最新的 API. 然而 Subversion 命令行程序的输出无法保证这种兼容性.

使用 API

基于 Subversion 函数库 API 开发应用程序是一件相对比较直截了当 的事. Subversion 主要由 C 函数库组成, 它们的头文件 (*.h*) 在源代码包的 *subversion/include* 目录内. 如果你从源代码编译 安装了 Subversion, 这些头文件就会被复制到你的系统目录中 (例如 */usr/local/include*). 这些头文件代表了能够被 用户访问到的 Subversion 函数库的全部函数与类型. Subversion 开发 社区非常注重 API 的文档—头文件里已经包含了关于如何使用 API 的 完整文档.

浏览头文件时, 你注意到的第一件事可能是 Subversion 的数据类型 和函数是被名字空间保护起来的, 详细地说, 每一个公开的 Subversion 符号 名都以 `svn_` 开始, 然后是表示该符号定义所在的 函数库的编码 (例如 `wc`, `client`, `fs` 等), 然后是一个下划线 (_), 然后是符号名剩下的部分. 半公开的函数 (只被单个函数库的源文件们使用, 在该函数库外无法使用, 而且这些半公开函数的定义位于能使用它们的库函数 目录内) 使用不同的命名模式, 在函数库编码后面是两个连续的下划线 (__), 而非一个下划线. 源文件内的私有函数 没有特定的前缀, 它们都被声明为 `static`. 当然, 这些 命名规范对编译器没什么意义, 但却有助于开发人员理解函数和数据类型的 作用域.

学习如何使用 Subversion API 进行开发的另一个信息来源是官网的 编程文档 (http://subversion.apache.org/docs/community-guide/). 这篇文档主要针对 Subversion 开发人员, 但对于把 Subversion 用作第 三方函数库的开发人员来说同样有用. [2]

Apache 可移植运行库

除了 Subversion 自己的数据类型, 你还会看到很多以 `apr_` 开始的数据类型—这些类型来自 Apache 可移植运行库 (Apache Portable Runtime, 简称 APR). APR 是 Apache 开发的可移植库, 最初是为了将服务器代码中与操作系统 相关的代码和不相关的代码分离开, 最终产生了一个提供通用 API 的 函数库, 这些 API 隐藏了操作系统之间的差异. 虽然 Apache HTTP 服务 器是 APR 的第一个用户, 但 Subversion 开发团队很快就意识到了 APR 的价值. 使用 APR 库意味着在 Subversion 代码中不存在依赖操作系统版 本的代码, 同时还意味着只要操作系统能编译和运行 Apache HTTP 服务器, 那它就能够编译和运行 Subversion 客户端程序, 目前 APR 支持的操作系统 包括所有的 Unix 系统, Win32, BeOS, OS/2 和 Mac OS X.

除了为不同的操作系统提供一致的系统调用实现外, [3] APR 还提供了许多定制化的数据类型, 例如动态数 组和哈希表, 这些数据类型在 Subversion 中用得非常广泛, 其中出现 得最多的类型是 `apr_pool_t`—APR 内存池— 它几乎出现在每一个 Subversion API 函数原型中. Subversion 使用 内存池完成所有内部的内存分配, (除非外部的函数库为它的参数指定了 一个不同的内存管理机制 [4]) 不过并不要求 Subversion API 的用户 也要用 APR 内存池完成内存管理, 他只需要向 Subversion API 提供一 个内存池参数即可. 这就要求 Subversion API 用户必须在编译时链接 APR, 必须调用 `apr_initialize()` 初始化 APR 子 系统, 然后调用 `svn_pool_create()`, `svn_pool_clear()` 和 `svn_pool_destroy()` 完成内存池的创建和管理.

[2] 毕竟 Subversion 也使用了 Subversion 的 API.
[3] Subversion 尽可能使用 ANSI 规定的系统调用和数据类型.
[4] Berkeley DB 就是这样一种 函数库.

面向内存池编程

几乎每一个用过 C 语言做过开发的程序员都会对内存管理感到畏惧. 分配足够多的内存, 跟踪内存的分配, 不再使用时释放内存— 这些工作可以变得非常复杂. 如果处理不当, 很可能会导致程序— 甚至整个操作系统—崩溃.

另一方面, 高级程序设计语言要么把内存管理的工作从程序员的 手中完成解放出来, 要么只有在非常有必要的情况下 (例如极致的内存 优化) 才由程序员来管理内存. 例如 Java 和 Python 使用了 垃圾收集 (*garbage collection*), 当需要时为对象分配内存, 以及当对象 不再被使用时释放内存.

APR 提供了一种介于两者之间的方法, 称为 基于池的内存管理 (*pool-based memory management*), 内存池允许程序员使用一种比较 粗糙的粒度控制内存的使用—关注内存的每一大块 (或 “池”), 而非每一个分配的对象. 不是使用 `malloc()` 及其亲友来分配内存, 而是调用 APR 库函数从内存池中分配内存. 如果从内存池中分配而来的对象都 已使用完毕, 你就可以销毁整个内存池, 同时也销毁了从该内存池中 分配的 *所有* 对象. 于是, 你的程序不用再跟踪 将被释放的单个对象, 只需要考虑这些对象普遍的生命周期, 然后从 生命周期 (内存池被创建和销毁的时间) 匹配的内存池中分配这些对象.

函数与不透明数据

为了充分利用异步化的行为, 以及向 Subversion API 用户提供钩子函数, 以便按照定制化的方式处理数据, 许多函数都接受这样一对参数: 一个 指向回调函数的指针以及一个指向不透明数据 (称为 *baton*) 的指针, baton 携带了回调函数所需的各种上下文信息. Baton 通常就是一个 C 语言结构体, 它带有回调函数所需 的额外信息, 而这些信息对于调用回调函数的代码来说是不透明的.

URL 和路径要求

由于远程版本控制操作是 Subversion 存在的最重要理由, 因此我们 需要注意对国际化 (i18n) 的支持. 毕竟 “远程” 不仅意味 着 “跨越办公室”, 它还可能意味着 “跨越国界”. 为了支持国际化, Subversion 所有接受路径参数的公共接口都要求这些路径 是规范化的—可通过调用函数 `svn_dirent_canonicalize()` 和 `svn_uri_canonicalize()` 分别得到规范化的本地 文件系统路径和 URL—而且是 UTF-8 编码. 举个例子, 任意一个使用 *libsvn_client* 的客户端程序在把路径传递给 Subversion 函数库之前, 都要先把本地编码的路径转换成 UTF-8 编码. 在得到 Subversion 产生的路径之后, 要先把这些路径转换成本地编码, 然后再交给非 Subversion 函数进行处理. 幸运的是, Subversion 提供了 一套函数 (见 *subversion/include/svn_utf.h*) 用于完成这些 编码转换.

另外, Subversion API 要求所有的 URL 参数必须符合 URI 编码 规则. 比如说你不能把文件 *My File.txt* 的 URL 写成 `file:///home/username/My File.txt`, 而应该写成 `file:///home/username/My%20File.txt`. 同样, Subversion 提供了函数 `svn_path_uri_encode()` 和 `svn_path_uri_decode()` 分别用于 URI 的编码 和解码.

使用除了 C 和 C++ 之外的语言

如果你希望使用除了 C 之外的程序—例如 Python 或 Perl 脚本—调用 Subversion 函数库, 对此, Subversion 通过 SWIG (Simplified Wrapper and Interface Generator) 提供了一些支持. Subversion 的 SWIG 绑定位于 *subversion/bindings/swig*, 虽然它们还在不断 成熟中, 但是现在已经是可用的了. 这些绑定通过封装脚本, 把脚本语言 的数据类型翻译成 Subversion C 函数库所需的数据类型, 从而允许你 间接调用 Subversion API.

Subversion 开发团队已经花了很多精力为 Python, Perl 和 Ruby 开发功能齐全的, 由 SWIG 生成的绑定. 在一定程度上, 为这些脚本语言 准备 SWIG 接口所做的准备工作可以重用到 SWIG 支持的其他语言上 (包括 C#, Guile, Java, MzScheme, OCaml,

PHP, Tcl 等). 然而, 如果接口过于复杂, SWIG 在不同语言之间翻译还需要帮助时, 那么开发人员还需要付出额外的开发工作. 关于 SWIG 的详细信息, 见官网 http://www.swig.org/.

Subversion 还拥有针对 Java 的语言绑定. Javahl 绑定 (位于 *subversion/bindings/java*) 不是基于 SWIG, 而是 Java 和手工编写的 JNI 的混合物. Javahl 涵盖了 Subversion 客户端的大部分 API, 它主要面对基于 Java 的 Subversion 客户端实现 和 IDE 集成.

虽然语言绑定从开发人员那儿受到的关注度比不上 Subversion 的核心 模块, 但通常而言这些绑定已经是生产就绪的了. 有大量的脚本 和应用程序, Subversion GUI 客户端和其他第三方工具都已经成功地把 Subversion 语言绑定应用到它们的 Subversion 集成中.

为了使用其他语言与 Subversion 交互, 如果我们还能有其他一些选择, 那将会是一件非常有价值的事情, 例如不是由 Subversion 开发社区提供的 语言绑定, 其中有两个值得我们关注, 第一个是 Barry Scott 开发的 PySVN 绑定 (https://pysvn.sourceforge.io/), 一种很流行的 Python 绑定, 与 Subversion 所提供的 Python 相比, PySVN 呈现的接口 更具有 Python 风格; 其二, 如果你正在寻找一种纯 Java 实现的 Subversion, 可以试试 SVNKit (http://svnkit.com/).

SVNKit 与 javahl

2005 年, 一个叫做 TMate 的小公司宣布发行 JavaSVN 1.0.0 —它是 Subversion 的纯 Java 语言实现. 从那时起, 该项目被重 命名为 SVNKit (官网是 http://svnkit.com/), SVNKit 获得了巨大的成功, 它为不同的 Subversion 客户端, IDE 集成 和第三方工具提供 Subversion 功能.

与 javahl 的不同点在于 SVNKit 不是 Subversion 核心函数库的封装, 事实上 SVNKit 与 Subversion 没有共享一行代码, 但是人们还是很容易 混淆 SVNKit 和 javahl, 甚至搞不清楚自己正在用的是哪个函数库. 读者 必须清醒地认识到 SVNKit 和 javahl 在某些方面非常不同, 首先, SVNKit 和 Subversion 一样都是开源软件, 但 SVNKit 的授权更加严格. [5] 然后, SVNKit 的目标是完全用 Java 实现, 因此在复制 Subversion 的功能时, 既要跟上官方 Subversion 的脚步, 也要考虑用 Java 实现相同功能的可 行性. 例如 SVNKit 无法通过 `file://` 协议访问 使用 Berkeley DB 作为后端存储的仓库, 因为不存在纯 Java 实现的 Berkeley DB, 其文件格式可以完全兼容 Berkeley DB 的原生语言实现.

SVNKit 具有完善的可靠性记录跟踪机制. 作为一个纯 Java 实现的 软件, 在面对编程错误时将更加健壮—SVNKit 的错误会产生一个 可捕获的 Java 异常, 而如果通过 javahl 使用 Subversion 核心函数库 时发生了错误, 可能会导致整个 JRE (Java Runtime Environment) 崩溃. 所以说在选择基于 Java 的 Subversion 实现时, 应该仔细衡量代价.

代码示例

例 8.1 “使用仓库层” 展示了一 个用 C 语言编写的代码示例, 说明了我们已经介绍过的几个概念. 示例 同时使用了仓库和文件系统接口 (可以从函数名的 `svn_repos_` 和 `svn_fs_` 前缀 看出) 来创建一个新的版本号, 该版本号添加了一个新目录. 你可以从示例 里看到 APR 内存池的用法—它们只是作为参数被传递给 Subversion 库函数. 示例还展示了 Subversion 较为晦涩的错误处理—必须显式 处理所有的 Subversion 错误, 以避免出现内存泄漏 (或程序失败).

例 8.1. 使用仓库层

[5] 如果软件用到了 SVNKit, 或者用到了使用了 SVNKit 的软件, 那么 任意形式的二次发布必须携带关于如何获取软件完整源代码的信息. 详细 的授权见 http://svnkit.com/license.html.

```
/* Convert a Subversion error into a simple boolean error code.
 *
 * NOTE:  Subversion errors must be cleared (using svn_error_clear())
 *        because they are allocated from the global pool, else memory
 *        leaking occurs.
 */
#define INT_ERR(expr)                           \
  do {                                          \
    svn_error_t *__temperr = (expr);            \
    if (__temperr)                              \
      {                                         \
        svn_error_clear(__temperr);             \
        return 1;                               \
      }                                         \
    return 0;                                   \
  } while (0)

/* Create a new directory at the path NEW_DIRECTORY in the Subversion
 * repository located at REPOS_PATH.  Perform all memory allocation in
 * POOL.  This function will create a new revision for the addition of
 * NEW_DIRECTORY.  Return zero if the operation completes
 * successfully, nonzero otherwise.
 */
static int
make_new_directory(const char *repos_path,
                   const char *new_directory,
                   apr_pool_t *pool)
{
  svn_error_t *err;
  svn_repos_t *repos;
  svn_fs_t *fs;
  svn_revnum_t youngest_rev;
  svn_fs_txn_t *txn;
  svn_fs_root_t *txn_root;
  const char *conflict_str;

  /* Open the repository located at REPOS_PATH.
   */
  INT_ERR(svn_repos_open(&repos, repos_path, pool));

  /* Get a pointer to the filesystem object that is stored in REPOS.
   */
  fs = svn_repos_fs(repos);

  /* Ask the filesystem to tell us the youngest revision that
   * currently exists.
   */
  INT_ERR(svn_fs_youngest_rev(&youngest_rev, fs, pool));
```

```
/* Begin a new transaction that is based on YOUNGEST_REV.  We are
 * less likely to have our later commit rejected as conflicting if we
 * always try to make our changes against a copy of the latest snapshot
 * of the filesystem tree.
 */
INT_ERR(svn_repos_fs_begin_txn_for_commit2(&txn, repos, youngest_rev,
                                           apr_hash_make(pool), pool));

/* Now that we have started a new Subversion transaction, get a root
 * object that represents that transaction.
 */
INT_ERR(svn_fs_txn_root(&txn_root, txn, pool));

/* Create our new directory under the transaction root, at the path
 * NEW_DIRECTORY.
 */
INT_ERR(svn_fs_make_dir(txn_root, new_directory, pool));

/* Commit the transaction, creating a new revision of the filesystem
 * which includes our added directory path.
 */
err = svn_repos_fs_commit_txn(&conflict_str, repos,
                              &youngest_rev, txn, pool);
if (! err)
  {
    /* No error?  Excellent!  Print a brief report of our success.
     */
    printf("Directory '%s' was successfully added as new revision "
           "'%ld'.\n", new_directory, youngest_rev);
  }
else if (err->apr_err == SVN_ERR_FS_CONFLICT)
  {
    /* Uh-oh.  Our commit failed as the result of a conflict
     * (someone else seems to have made changes to the same area
     * of the filesystem that we tried to modify).  Print an error
     * message.
     */
    printf("A conflict occurred at path '%s' while attempting "
           "to add directory '%s' to the repository at '%s'.\n",
           conflict_str, new_directory, repos_path);
  }
else
  {
    /* Some other error has occurred.  Print an error message.
     */
    printf("An error occurred while attempting to add directory '%s' "
           "to the repository at '%s'.\n",
```

```
            new_directory, repos_path);
    }

  INT_ERR(err);
}
```

注意到在 例 8.1 “使用仓库层” 里，代码本可以简单 地调用 `svn_fs_commit_txn()` 来提交事务，但文件 系统 API 对于仓库函数库的钩子机制一无所知. 如果你希望每次提交完一个 事务后，Subversion 仓库都会自动执行一些非 Subversion 任务（例如发送 一封描述了事务所做的修改的邮件到邮件列表），你需要使用 *libsvn_repos* 包装后的函数—在上面的例子里就是 `svn_repos_fs_commit_txn()`—这些函数添 加了钩子触发功能.（关于 Subversion 钩子机制的更多内容，见 “实现仓库钩子”一节.）

现在换另一种语言. 例 8.2 “使用 Python 访问仓库层” 使用了 Subversion 的 SWIG Python 绑定来递归地搜索仓库最新的版本号，并打印 出在搜索过程中达到的不同路径.

例 8.2. 使用 Python 访问仓库层

```
#!/usr/bin/python

"""Crawl a repository, printing versioned object path names."""

import sys
import os.path
import svn.fs, svn.core, svn.repos

def crawl_filesystem_dir(root, directory):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
    a list of all the paths at or below DIRECTORY."""

    # Print the name of this path.
    print directory + "/"

    # Get the directory entries for DIRECTORY.
    entries = svn.fs.svn_fs_dir_entries(root, directory)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Calculate the entry's full path.
        full_path = directory + '/' + name

        # If the entry is a directory, recurse.  The recursion will return
        # a list with the entry and all its children, which we will add to
        # our running list of paths.
        if svn.fs.svn_fs_is_dir(root, full_path):
            crawl_filesystem_dir(root, full_path)
        else:
```

```
            # Else it's a file, so print its path here.
            print full_path

def crawl_youngest(repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj)

    # Open a root object representing the youngest (HEAD) revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj, youngest_rev)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "")

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
                         % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize the repository path.
    repos_path = svn.core.svn_dirent_canonicalize(sys.argv[1])

    # Do the real work.
    crawl_youngest(repos_path)
```

同样的程序如果用 C 语言实现，那就需要考虑 APR 的内存池子系统，但是 Python 自动处理内存的分配与释放. 在 C 语言里，你需要考虑各种 定制化数据类型（例如 APR 函数库提供的数据类型），这些数据类型用于 表示哈希项和路径列表，但是 Python 内建了用于表示哈希 (Python 将其 称为 “字典”）和列表的数据类型，而且提供了丰富的函数 用来管理这些数据结构. 于是 SWIG（在 Subversion 语言绑定层的某些 定制化修改的帮助下）负责把这些定制化的数据类型映射到目标语言的本 地类型，用户就能更加直观地使用目标语言.

Subversion 的 Python 绑定也能运用到工作副本的操作中. 在本章的 前一节里，我们提到了 *libsvn_client* 接口，以及 它存在的唯一目标就是简化 Subversion 客户端程序的开发. 例 8.3 “用 Python 实现 svn status” 展示了如何使用 SWIG Python 绑定访问 *libsvn_client* 函数库，来 实现一个简化版的 *svn status* 命令.

例 8.3. 用 Python 实现 svn status

```
#!/usr/bin/env python
```

```
"""Crawl a working copy directory, printing status information."""

import sys
import os.path
import getopt
import svn.core, svn.client, svn.wc

def generate_status_code(status):
    """Translate a status value into a single-character status code,
    using the same logic as the Subversion command-line client."""
    code_map = { svn.wc.svn_wc_status_none        : ' ',
                 svn.wc.svn_wc_status_normal      : ' ',
                 svn.wc.svn_wc_status_added       : 'A',
                 svn.wc.svn_wc_status_missing     : '!',
                 svn.wc.svn_wc_status_incomplete  : '!',
                 svn.wc.svn_wc_status_deleted     : 'D',
                 svn.wc.svn_wc_status_replaced    : 'R',
                 svn.wc.svn_wc_status_modified    : 'M',
                 svn.wc.svn_wc_status_conflicted  : 'C',
                 svn.wc.svn_wc_status_obstructed  : '~',
                 svn.wc.svn_wc_status_ignored     : 'I',
                 svn.wc.svn_wc_status_external    : 'X',
                 svn.wc.svn_wc_status_unversioned : '?',
               }
    return code_map.get(status, '?')

def do_status(wc_path, verbose, prefix):
    # Build a client context baton.
    ctx = svn.client.svn_client_create_context()

    def _status_callback(path, status):
        """A callback function for svn_client_status."""

        # Print the path, minus the bit that overlaps with the root of
        # the status crawl
        text_status = generate_status_code(status.text_status)
        prop_status = generate_status_code(status.prop_status)
        prefix_text = ''
        if prefix is not None:
            prefix_text = prefix + " "
        print '%s%s%s  %s' % (prefix_text, 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

  Print working copy status, optionally with a bit of prefix text.

Options:
  --help, -h    : Show this usage message
  --prefix ARG  : Print ARG, followed by a space, before each line of output
  --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", "prefix=", "verbose"])
    except getopt.GetoptError:
        usage_and_exit(1)
    verbose = 0
    prefix = None
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage_and_exit(0)
        if opt in ("--prefix"):
            prefix = arg
        if opt in ("-v", "--verbose"):
            verbose = 1
    if len(args) != 1:
        usage_and_exit(2)

    # Canonicalize the working copy path.
    wc_path = svn.core.svn_dirent_canonicalize(args[0])

    # Do the real work.
    try:
        do_status(wc_path, verbose, prefix)
    except svn.core.SubversionException, e:
        sys.stderr.write("Error (%d): %s\n" % (e.apr_err, e.message))
        sys.exit(1)
```

和 例 8.2 “使用 Python 访问仓库层” 一样，上面的程序不使用内存池，大部分情况下都是用的 Python 内建的 数据类型.

先把用户提供的路径参数转化成规范化形式（调用 `svn_dirent_canonicalize()` 或 `svn_uri_canonicalize()`），然后再传递给其他 API，否则的话可能会导致 Subversion C 库函数断言失败，引起程序异常退出.

使用 Python 调用 Subversion API 的用户可能对回调函数在 Python 中的实现比较感兴趣. 前面已经说过，Subversion C API 对编程范式—回调函数/baton—的使用非常广泛，C 函数如果接受一个回调函数和 baton，那么在 Python 中将只接受一个回调函数，那么主调函数如何向回调函数传递任意的上下文信息呢？在 Python 里，这是通过作用域规则和参数的默认值来实现的. 你可以从 例 8.3 “用 Python 实现 svn status” 看到具体的例子，函数 `svn_client_status2()` 得到了一个回调函数（`_status_callback()`），但却没有 baton—函数 `_status_callback()` 能够访问到用户提供的前缀字符串是因为变量 `prefix` 自动落到了函数的作用域内.

小结

Subversion 最伟大的特性之一并不是从它的命令行客户端或其他工具中得到，而是 Subversion 以模块化的方式进行设计，提供了稳定而公开的 API，于是其他人—例如你—就可以自己开发驱动 Subversion 的软件.

本章，我们从更底层地角度介绍了 Subversion 的架构和逻辑层，并描述了它的公共 API，以及类似的用于各层之间通信的 API. 许多开发人员都发现了 Subversion API 的有趣用法，从简单的仓库钩子脚本，到 Subversion 与其他应用程序的集成，再到完全不同的版本控制系统. 你还能想到更奇妙的用法吗？

部分 II. Subversion 命令行参考手册

目录

svn 参考手册—Subversion 命令行客户端

svn 是 Subversion 官方的命令行客户端，通过一 系列特定于任务的子命令向用户提供功能，大多数子命令都接受很多参数，用 于精细地控制程序的行为.

使用 *svn* 程序时，子命令和非选项参数必须按照特定 的顺序出现在命令行上，但是选项却可能以任意地顺序出现（当然，选项必须 出现在程序名之后），通常情况下选项的顺序是无关紧要的. 例如，下面的命令 都是 *svn status* 的有效使用形式，而且都是按照相同的 方式进行解释：

```
$ svn -vq status myfile
$ svn status -v -q myfile
$ svn -q status -v myfile
$ svn status -vq myfile
$ svn status myfile -qv
```

下面几节介绍了各个子命令以及 *svn* 的命令行选项，同时还展示了每个子命令的典型用法.

虽然各个子命令的选项不完全相同，但所有的选项都在同一个名字空间内 —无论对于哪个子命令，选项的意义是完全相同的. 例如无论用户执行 的是哪个子命令，选项 `--verbose(-v)` 的意义总是 “详细输出”.

如果用户向 *svn* 传递了子命令不支持的选项，那么 程序通常会马上报错退出，但是从 Subversion 1.5 开始，有几个选项可被所有（或者说几乎所有）的子命令支持，因此它们就被当成了可被所有的子命 令接受，即使该选项对子命令不会产生任何效果.（这种行为主要是为了方便从 定制化的封装脚本中调用 *svn*.) 这些选项作为全局选项 出现在命令行客户端的帮助信息中，可以用下面的命令看到它们：

```
$ svn help upgrade
upgrade: Upgrade the metadata storage format for a working copy.
usage: upgrade [WCPATH...]

  Local modifications are preserved.

Valid options:
  -q [--quiet]             : print nothing, or only summary information

Global options:
  --username ARG           : specify a username ARG
  --password ARG           : specify a password ARG
  --no-auth-cache          : do not cache authentication tokens
  --non-interactive        : do no interactive prompting (default is to prompt
                             only if standard input is a terminal device)
  --force-interactive      : do interactive prompting even if standard input
                             is not a terminal device
  --trust-server-cert      : accept SSL server certificates from unknown
                             certificate authorities without prompting (but only
                             with '--non-interactive')
  --config-dir ARG         : read user configuration files from directory ARG
  --config-option ARG      : set user configuration option in the format:
                                 FILE:SECTION:OPTION=[VALUE]
```

```
                        For example:
                            servers:global:http-library=serf
$
```

svn 的子命令支持以下选项:

svn 选项

`--accept` *`ACTION`*

为自动的冲突解决指定处理措施，当 Subversion 注意到冲突发生时不会再询问用户如何处理. Subversion 支持的 *`ACTION`* 如下所示，但不同的子命令，其可用 的 *`ACTION`* 是不同的:

`postpone` (`p`)

对冲突不采取任何措施，将冲突记录起来，以便稍后再作处理.

`edit` (`e`)

在文本编辑器中打开每一个冲突文件，由用户手工处理冲突.

`launch` (`l`)

为每一个冲突文件调用交互式的冲突合并工具.

`base`

在把服务器上的修改合并到工作副本之前，把文件指定成版本 号为 BASE，且未修改的版本.

`working`

将文件在工作副本中的当前版本作为冲突解决后的版本，在这 之前用户应该手工处理了文件里的冲突.

`mine-full` (`mf`)

如果文件发生了冲突，则保留文件里的所有本地修改，丢弃 该文件从服务器接收到的所有修改.

`theirs-full` (`tf`)

如果文件发生了冲突，则丢弃文件里的所有本地修改，保留 从服务器接收到的所有修改.

`mine-conflict` (`mc`)

如果文件的某一区域发生了冲突，则保留该区域的本地修改，丢弃从服务器接收到的同一区域上的修改.

`theirs-conflict` (`tc`)

如果文件的某一区域发生了冲突，则丢弃该区域的本地修改，保留从服务器接收到的同一区域上的修改.

为了查看特定的子命令支持哪些 *`ACTION`*，阅读 *svn help `SUBCOMMAND`* 的输出.

`--allow-mixed-revisions`

在合并时，禁止去核实目标及其子文件的版本号是否是相同的. 从 Subversion 1.7 开始，默认都会去核实版本号是否相同. 虽然推荐的 做法是让待合并的工作副本处于一个相同的版本号下，但必要时还是可以 用这个选项允许向版本号混合的工作副本执行合并.

`--auto-props`

允许自动属性设置，它将覆盖运行时配置指令 `enable-auto-props` 的设置.

`--change` (`-c`) *ARG*

使用一个特定的 “修改” 执行所请求的操作. 通常来说，这个选项是 `-r` *ARG-1*`:`*ARG* 的同义语. 某些子命令允许选项的参数是逗号分隔的版本号列表（即 `-c` *ARG1*`,`*ARG2*`,`*ARG3*），用户还可以用连字符分隔的两个版本号指定一个版本号范围（即 `-c` *ARG1*`-`*ARG2*），范围包括起始与结束版本号. 最后，如果版本号是一个负数，则表示相反 的版本号范围，例如 `-c -45` 等价于 `-r 45:44`.

`--changelist` (`--cl`) *ARG*

告诉 Subversion 只去操作属于变更列表 *ARG* 的文件，用户可以多次使用这个选项，从而指定多个变更列表.

`--config-dir` *DIR*

告诉 Subversion 从指定的目录（而不是默认的 *.subversion*）中读取配置信息.

所有的 *svn* 子命令都支持该选项.

`--config-option` *CONFSPEC*

在命令执行期间设置运行时配置选项. *CONFSPEC* 是一个字符串，指定了运行时配置 选项的名字空间，选项名和选项值，形式是 *FILE*:*SECTION*:*OPTION*=[*VALUE*]，其中，*FILE* 和 *SECTION* 分别是选项所在的运行时配置文件（*config* 或 *server*）和配置 文件里的节. *OPTION* 就是选项本身，而 *VALUE*（如果有的话）就是选项的值. 比如 说用户想要临时禁止 HTTP 压缩，那就可以把 `--config-option` 写成 `--config-option=servers:global:http-compression=no`. 该选项可以在命令行上出现多次.

所有的 *svn* 子命令都支持该选项.

`--depth` *ARG*

告诉 Subversion 把命令的操作范围限制在一个指定的目录深度内. *ARG* 可以是 `empty`（目标本身），或 `files`（目标和目标的直接子文件，不包括直接子目录），`immediates`（目标和目标的直接 子文件，包括直接子目录），或 `infinity`（目标和 目标的所有子孙—即完全递归）.

`--diff`

告诉 *svn log* 输出版本号所包含的差异（按照 *svn diff* 的方式进行输出）.

`--diff-cmd` *CMD*

指定一个外部差异比较工具. 如果在执行 *svn diff* 时没有指定该选项, Subversion 将会使用它自己的差异比较引擎, 默认 按照标准差异格式进行输出. 如果用户希望使用一个外部的差异比较工具, 就可以用该选项实现, 然后还可以用选项 `--extensions (-x)` 为外部差异比较工具传递选项.

`--diff3-cmd` *CMD*

指定一个外部的三路差异比较工具 (用于合并文件的修改).

`--dry-run`

执行命令的所有过程, 除了做出实际的修改—无论是修改工作 副本还是仓库.

`--editor-cmd` *CMD*

指定一个外部文本编辑器, 用于编辑日志消息或属性值. 关于如何 指定一个默认文本编辑器, 见 “通用配置选项”一节 的 `editor-cmd` 节.

`--encoding` *ENC*

告诉 Subversion 你的提交消息是用该选项所指定的字符集编码编写的. 默认情况下, Subversion 根据操作系统的本地语言环境判断提交消息的 字符集编码, 如果你用了其他字符集编码, 就要用该选项显式地告诉 Subversion 你所用的字符集编码.

`--extensions (-x)` *ARG*

为 Subversion 的差异比较引擎指定扩展选项, 有效的扩展选项有:

`--ignore-space-change (-b)`

忽略空白字符在数量方面的变化.

`--ignore-all-space (-w)`

忽略所有的空白字符.

`--ignore-eol-style`

忽略 EOL (end-of-line, 行结束标记) 的变化.

`--show-c-function (-p)`

在差异比较输出中显示 C 程序的函数名.

`--unified (-u)`

显示宽度为 3 行的标准差异上下文.

ARG 的默认值是 `-u`, 如果你希望指定多个扩展选项, 就把它们放在一 对双引号中.

需要注意的是，如果差异比较引擎是一个外部的差异比较工具，那么 选项 `--extensions(-x)` 的值 不仅限于以上提到的这些，而是可以设置成 任意 值.

`--file(-F)` *`FILENAME`*

将文件的内容传递给子命令，不同的子命令将文件内容用于不同的目的，例如 *svn commit* 把文件内容作为提交日志消息，而 *svn propset* 把文件内容作为属性值.

`--force`

强迫命令或操作往下执行. 在正常使用时， Subversion 会阻止用户 执行某些操作， 但是用户可以用这个选项告诉 Subversion: “我 知道我正在做什么，也知道这样做的可能后果，所以请继续往下执行. ”用这个选项就好像一个电工在未断电的情况下工作—如果 你不知道自己正在做什么，你可能会对命令的执行结果感到震惊.

`--force-log`

迫使 Subversion 接受传递给 `--message(-m)` 或 `--file(-F)` 的可疑参数，默认情况下，如果传递给这两个 选项的参数看起来好像是子命令的目标参数，那么 Subversion 将会报 错退出. 比如说用户向选项 `--file(-F)` 传递了一个处于版本控制下的文件路径，Subversion 将会认为用户犯了一个错误: 这个文件路径应该作为子命令 的目标参数，选项 `--file(-F)` 的参数应该是一个未被版本控制的文件路径. 为了表明自己的意图，并且 阻止 Subversion 进行这种检查，用户就可以用选项 `--force-log` 迫使 Subversion 无条件地接受日志消息.

`--force-interactive`

如果标准输入不是一个终端设备，强迫 *svn* 以 交互模式运行.

所有的 *svn* 子命令都支持该选项.

`--git`

按照分布式版本控制系统 Git 的格式，打印 *svn diff* 的标准差异输出.

`--help(-h, -?)`

如果在执行时带了一个或多个子命令，则显示各个子命令的内建帮助 文档. 如果只是单独使用，则显示客户端命令行工具的总体帮助文档.

`--ignore-ancestry`

告诉 Subversion 在计算差异时忽略祖先（仅依赖路径上的内容），对 *svn merge* 而言，该选项还会禁止 合并跟踪.

`--ignore-externals`

忽略外部定义和外部工作副本.

`--ignore-keywords`

禁止关键字替换.

`--ignore-properties`

告诉 *svn diff* 忽略属性的变化.

`--ignore-whitespace`

告诉 *svn patch* 在识别补丁上下文时忽略 空白字符.

`--incremental`

按照一种增量的格式打印输出，该格式允许将本次输出级连到之前 的相同类型的输出.

`--internal-diff`

告诉 Subversion 始终使用内建的差异比较引擎，无论用户是否在 运行时配置中指定了外部差异比较工具.

`--keep-changelists`

告诉 Subversion 在提交成功后不要把文件从变更列表中移除.

`--keep-local`

执行完 *svn delete* 后，在工作副本中保留 文件.

`--limit` (`-l`) *NUM*

显示日志消息的前 *NUM* 项.

`--message` (`-m`) *MESSAGE*

表示用户将在命令行上编写日志消息或锁的注释，选项的后面即是 用户写的内容，例如:

```
$ svn commit -m "They don't make Sunday."
```

`--native-eol` *ARG*

告诉 *svn export* 使用指定的行结束标记作为 系统的本地标记，这将会影响那些 `svn:eol-style` 属性值为 `native` 的文件. *ARG* 的有效值包括 `CR`, `LF` 和 `CRLF`.

`--new` *ARG*

使用 *ARG* 作为较新的目标（与 *svn diff* 配合使用）.

`--no-auth-cache`

禁止在 Subversion 运行时配置目录里缓存认证信息（例如用户名 和密码）.

所有的 *svn* 子命令都支持该选项.

`--no-auto-props`

禁止自动属性设置，该选项会覆盖运行时配置选项 `enable-auto-props`.

`--no-diff-added`

禁止为新增的文件输出差异. 默认情况下, 新文件的差异输出就像是 往一个已有的空文件内写入了全部内容后的差异输出效果.

`--no-diff-deleted`

禁止为删除了的文件输出差异. 默认情况下, 删除了的文件的差异 输出就像是删除了文件的所有内容 (但不删除文件) 后的差异输出效果.

`--no-ignore`

在显示工作副本状态, 添加文件或导入文件时不要忽略任何文件 (即使文件名与运行时配置选项 `global-ignores`, 属性 `svn:ignore` 或属性 `svn:global-ignores` 里的模式相匹配), 更多的 信息见 “通用配置选项”一节 和 “忽略未被版本控制的项”一节.

`--no-unlock`

告诉 Subversion 不要自动释放锁. (默认的提交行为会释放所有已 提交的文件上的锁.) 更多的信息见 “锁”一节.

`--non-interactive`

禁止所有的交互式提示. 交互式提示的例子包括向用户请求认证证书 和冲突解决. 如果你是在一个自动化的脚本中运行 Subversion, 那么比 起向用户发出交互式请求, 更方便的做法是报错退出.

从 Subversion 1.8 开始, 如果标准输入不是一个终端设备, 那么 *svn* 默认以非交互式模式运行. 为了强制 *svn* 以交互式模式运行, 可使用选项 `--force-interactive`.

所有的 *svn* 子命令都支持该选项.

`--non-recursive (-N)`

不再推荐使用该选项. 禁止子命令递归执行到 子目录内. 大多数子命令默认都会递归执行到子目录内, 但有些不会. 用户 应该不再使用该选项, 而使用更精确的 `--depth`, 对于 大多数子命令而言, 选项 `--non-interactive` 等价于 `--depth=files`, 但是对于 *svn status* 来说, 与 `--non-recursive` 等价 的选项是 `--depth=immediates`, 对于 *svn revert*, *svn add* 和 *svn commit* 来说, 等价的选项则是 `--depth=empty`.

`--notice-ancestry`

在计算差异时要考虑祖先.

`--old ARG`

使用 *ARG* 作为较旧的目标 (与 *svn diff* 配合使用).

`--parents`

作为操作的一部分, 在工作副本或仓库中自动创建不存在的父目录. 这个选项对于自动创建多级子目录非常方便, 如果目标是一个 URL, 那么 所有的目录都会在同一个提交中创建完成.

`--password` *PASSWD*

指定用于 Subversion 服务器认证的密码. 如果没有在命令行上指定 密码, 或者密码有误, 在需要时 Subversion 将提示用户输入密码.

所有的 *svn* 子命令都支持该选项.

`--patch-compatible`

告诉 *svn diff* 的输出要和通用的第三方补丁 工具保持兼容. 该选项等价于 `--show-copies-as-adds --ignore-properties`.

`--properties-only`

告诉 *svn diff* 只输出属性上的变化.

`--quiet (-q)`

告诉 *svn* 在执行时只打印必要的信息.

`--record-only`

告诉 *svn merge* 只合并合并信息, 不合并文件 上的修改.

`--recursive (-R)`

告诉子命令要递归地执行到子目录内 (大多数子命令都会默认递归 执行).

`--reintegrate`

不再推荐使用该选项. 该选项用于 *svn merge* 把特性分支上的修改合并到特性分支的祖先分支上. 从 Subversion 1.8 开始, *svn merge* 能够自动检测这种合并场景并执行 恰当的合并操作, 更多的细节见 “重新整合分支”一节.

`--relocate`

不再推荐使用该选项. 该选项用于 *svn switch* 改变工作副本所指向的仓库位置, 从 Subversion 1.7 开始, 更好的做法 是用命令 *svn relocate*, 更多的细节和示例见 svn relocate.

`--remove`

用于 *svn changelist* 解除—而不是建立 (默认行为)—文件与变更列表之间的关联.

`--reverse-diff`

告诉 *svn patch* 反向应用补丁—把新增 的行看成是被删除的行, 把删除的行看成是新增的行.

`--revision (-r)` *REV*

指定待操作的版本号 (或版本号范围). 该选项接受的参数有整数, 关键字或日期 (日期被花括号包围). 如果你希望指定一个版本号范围, 就在起始版本号与终止版本号之间加个冒号, 例如:

```
$ 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}
```

更多的信息见 “版本号关键字”一节.

`--revprop`

针对版本号属性（而不是文件或目录上的属性）进行操作，该选项 还要求你同时用选项 `--revision(-r)` 指定了一个版本号.

`--search` *ARG*

从日志消息中过滤出和模式 *ARG* 匹配 的那些消息. 如果日志消息的作者，日期，消息的文本内容（除非指定了 选项 `--quiet`）或被修改的路径中的任意一条与模式 匹配，该日志消息就被认为是匹配的. 如果选项 `--search` 出现了多次，只要日志消息和其中的任意一 个 `--search` 匹配，该日志消息就被认为是匹配的. 如果同时还指定了选项 `--limit`，那么 `--limit` 限制的是被搜索的日志消息数量，而不是匹配 的日志消息数量.

搜索模式（也被叫作文件名模式或 Shell 通配符模式）可以包含普通 字符和下面这些通配符：

`?`

匹配任意一个字符.

`*`

匹配任意一个字符串，字符串的长度可以为零.

`[ABC]`

匹配方括号内的任意一个字符.

`--search-and` *ARG*

该选项的参数和前面的 `--search` 或 `--search-and` 的参数联合起来，只有匹配 *所有* 模式的日志消息才被认为是匹配的.

`--set-depth` *ARG*

把目录的粘着 (sticky) 深度设置成 `exclude`, `empty`, `files`, `immediates` 或 `infinity`，关于这些参数的详细信息以及如何使用它们，见 “稀疏目录”一节.

`--show-copies-as-adds`

告诉 *svn diff* 在显示通过复制得到的文件 的差异时，不要显示它们与被复制的源文件之间的差异，而是把它们当作 全新的文件（就好像往一个空文件中写了内容后的差异输出）.

`--show-inherited-props`

告诉 *svn propget* 和 *svn proplist* 显示继承到的版本化属性.

`--show-revs` *`ARG`*

告诉 *svn mergeinfo* 显示特定种类的合并跟踪 信息. *`ARG`* 可以是 `merged` 或 `eligible`, 分别表示 已经合并的版本号和未来有资格被合并的版本号.

`--show-updates` (`-u`)

要求客户端显示关于文件是否过时的信息，它不会更新工作副本中的 文件—它只是会告诉你在下一次执行 *svn update* 时，哪些文件将被更新.

`--stop-on-copy`

该选项将导致 Subversion 子命令在遍历历史时，如果遇到了复制 —即历史中某一位置上的资源是通过复制仓库中其他位置来得到的 —则不再往前遍历历史.

`--strict`

告诉 Subversion 使用更严格的语义，在谈到 “更严格的语义” 时，必须关联上特定的子命令（即 *svn propget*) 才能解释清楚.

`--strip` *`NUM`*

用于 *svn patch* 忽略补丁文件中的前 *`NUM`* 个路径分量.

`--summarize`

告诉子命令只显示总结性的信息.

`--targets` *`FILENAME`*

告诉子命令从文件 *FILENAME* 读取额外的目标 路径参数. *FILENAME* 的每一行都是一个路径，路径 的编码和格式应该和把它们直接写在命令行上时相同.

`--trust-server-cert`

和选项 `--non-interactive` 一起使用时，指示 Subversion 接受由未知的证书机构颁发的 SSL 服务器证书，而不必提 示用户. 为了安全起见，只有当远程服务器和网络路径的完整性很可靠 时才能使用该选项.

所有的 *svn* 子命令都支持该选项.

`--use-merge-history` (`-g`)

使用或显示来自合并历史的额外信息.

`--username` *NAME*

指定用于向 Subversion 服务器认证的用户名，如果没有指定用户名 或者指定的用户名不正确，Subversion 将会再次提示用户输入用户名.

所有的 *svn* 子命令都支持该选项.

`--verbose` (`-v`)

要求子命令输出更详细的信息，这可能会导致客户端输出额外的字段，关于每个文件的详细信息，或与操作有关的额外信息.

`--version`

打印客户端的版本信息. 版本信息不仅包括客户端的版本号 (这里的 版本号指的是软件的版本, 注意不要和 Subversion 的版本号 (revision) 混淆)，还有客户端支持的所有仓库访问模块. 如果加上了选项 `--quiet` (`-q`), 则只打印版本号.

`--with-all-revprops`

和选项 `--xml` 一起使用，指示 Subversion 检索并 输出所有的版本号属性—包括 Subversion 保留给自己内部使用的属性 和用户的自定义属性—到日志中.

`--with-no-revprops`

和选项 `--xml` 一起用在命令 *svn log* 时，指示 Subversion 在日志输出中忽略所有的版本号 属性—包括标准的日志消息, 作者和版本号时间戳属性.

`--with-revprop` *ARG*

如何该选项和其他需要向仓库写数据的命令一起使用时，它可用于设置 版本号属性，格式是 *NAME=VALUE*, 意思是把属性 *NAME* 的值设置成 *VALUE*; 如果该选项和 `--xml` 一起用在命令 *svn log* 里，那么 *ARG* 的值将会显示在日志输出中.

`--xml`

按照 XML 格式打印输出. 输出所使用的 XML 模式（使用 RELAX NG 格式）的相关文件位于 Subversion 源代码树的 *subversion/svn/schema/* 目录内.

目录

名称

svn add — 添加文件，目录或符号链接.

大纲

`svn add PATH...`

描述

计划将文件，目录或符号链接添加到仓库中，在下次提交时，它们就 会被正式地上传并添加到仓库中. 如果用户已经添加了某些文件，但后面又不 想要这些文件了，可以使用 *svn rever* 取消未提交的 新增.

选项

`--auto-props`
`--depth ARG`
`--force`
`--no-auto-props`
`--no-ignore`
`--parents`
`--quiet (-q)`
`--targets FILENAME`

示例

往工作副本中添加一个新文件:

```
$ svn add foo.c
A         foo.c
```

添加目录时，*svn add* 的默认行为是递归的:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

用户可以只添加目录，而不添加目录里的子文件:

```
$ svn add --depth=empty otherdir
A         otherdir
```

如果用户试图添加一个已经被版本控制了的文件，那么 *svn add* 将会报错. 这种行为会影响用户执行这种操作: 递归已经 被版本控制了的目录，添加目录中所有未被版本控制的子文件. 为了迫使 Subversion 递归已被版本控制的目录，需要加上选项 `--force`:

```
$ svn add versioned-dir
svn: warning: W150002: '/home/cmpilato/projects/subversion/site' is already un\
der version control
$ svn add versioned-dir --force
A         versioned-dir/foo.c
A         versioned-dir/somedir/bar.c
A  (bin)  versioned-dir/otherdir/docs/baz.doc
…
```

名称

svn blame (praise, annotate, ann) — 显示文件的每一行最近一次是被谁，在什么时候被修改的.

大纲

`svn blame TARGET[@REV]...`

描述

显示文件的每一行最近一次被修改的作者与版本号信息，Subversion 在 每一行的开头都加上了最后最近一次修改该行的作者的用户名和版本号.

选项

`--extensions (-x) ARG`
`--force`
`--incremental`
`--revision (-r) REV`
`--use-merge-history (-g)`
`--verbose (-v)`
`--xml`

示例

如果用户想查看 *readme.txt* 的每一行的最近一次 修改都是谁，在什么时候做的，就执行：

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry Don't bother reading it.  The boss is a knucklehead.
     3      sally
…
```

从 *svn blame* 的输出可以看到，*readme.txt* 最近一次是 Harry 在版本号 5 修改的，用户还要理解 *svn blame* 对于修改的成立条件是很挑剔 的. 老板在责骂 Harry 之前，应首先考虑 Harry 是不是只修改了这一行 里的某个字符，甚至他只是删除了这一行里的一个多余的空格，整句话一开始 可能并不是他写的. 为了不冤枉 Harry，你需要认真地查看版本号 5 的日志和 Harry 所做的修改：

```
$ svn log -c 5 http://svn.red-bean.com/repos/test/readme.txt
------------------------------------------------------------------------
r5 | harry | 2008-05-29 07:26:12 -0600 (Thu, 29 May 2008) | 1 line

Commit the results of 'double-space-after-period.sh'.

------------------------------------------------------------------------
$ svn diff -c 5 http://svn.red-bean.com/repos/test/readme.txt
Index: http://svn.red-bean.com/repos/test/readme.txt
```

```
===================================================================
--- http://svn.red-bean.com/repos/test/readme.txt (revision 4)
+++ http://svn.red-bean.com/repos/test/readme.txt (revision 5)
@@ -1,5 +1,5 @@
 This is a README file.
-Don't bother reading it. The boss is a knucklehead.
+Don't bother reading it.  The boss is a knucklehead.

 INSTRUCTIONS
 ============
$
```

从版本号 5 的修改来看，结果已经很清晰了，Harry 只是去掉了一个 多余的空格. 幸运的是，选项 `--extensions (-x)` 可以帮助用户找出最近一次针对该行的 有意义的 修改. 例如下面的例子在显示最近一 次修改时忽略了空白字符的变化：

```
$ svn blame -x -b http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     4       jess Don't bother reading it.  The boss is a knucklehead.
     3      sally
…
```

如果带上了选项 `--xml`，用户就可以得到 XML 格式 的输出，但不会出现行本身的内容：

```
$ svn blame --xml http://svn.red-bean.com/repos/test/readme.txt
<?xml version="1.0"?>
<blame>
<target
   path="readme.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>2008-05-29T13:26:12.293121Z</date>
</commit>
</entry>
<entry
   line-number="3">
…
</entry>
</target>
```

```
</blame>
$
```

名称

svn cat — 输出指定的文件的内容.

大纲

```
svn cat TARGET[@REV]...
```

描述

输出指定的文件的内容. 如果只是想看目录包含了哪些文件, 见本章后面的 *svn list*.

选项

```
--revision (-r) REV
```

示例

如果你想查看仓库里的 *readme.txt* 文件, 但又不 想把它检出, 可以这样做:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
Don't bother reading it.  The boss is a knucklehead.

INSTRUCTIONS
============

Step 1:  Do this.

Step 2:  Do that.
$
```

还可以查看文件在特定版本号下的内容:

```
$ svn cat -r 3 http://svn.red-bean.com/repos/test/readme.txt
This is a README file.

INSTRUCTIONS
============

Step 1:  Do this.

Step 2:  Do that.
$
```

你可能很自然地想到用 *svn cat* 查看工作副本中的 文件, 但是要注意的是 *svn cat* 用在工作副本文件上的 默认限定版本号是 BASE, 它是文件未修改时的基础版本 号, 所以如果在 *svn cat* 的输出中看不到本地修改时请 不会感到惊讶.

如果工作副本已经过期了（或者含有本地修改），而你想查看文件在 版本号 `HEAD` 的内容，就加上选项 `--revision(-r)`: `svn cat -r HEAD` *`FILENAME`*

名称

svn changelist (cl) — 为工作副本里的路径关联（或解除关联）一个变更列表.

大纲

```
svn changelist CLNAME TARGET...

svn changelist --remove TARGET...
```

描述

将工作副本里的文件分组到一个变更列表（命名的逻辑分组），以便于用户 在一个工作副本中同时应付多个文件集合.

选项

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--remove
--targets FILENAME
```

示例

编辑三个文件，然后把它们关联到一个变更列表，最后提交变更列表里的 文件：

```
$ svn changelist issue1729 foo.c bar.c baz.c
A [issue1729] foo.c
A [issue1729] bar.c
A [issue1729] baz.c
$ 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
$
```

注意在上面的例子里，只有变更列表 `issue1729` 里的三个文件的修改被提交了.

名称

svn checkout (co) — 从仓库中检出一个工作副本到本地.

大纲

`svn checkout URL[@REV]... [PATH]`

描述

从仓库中检出一个工作副本到本地. 如果忽略 *PATH*, 就使用 URL 的最后一个分量作为 *PATH*. 如果出现了多个 URL, 那么每个 URL 都会被检出到 *PATH* 的一个子目录内, 子目录的 名字是 URL 的最后一个分量.

选项

`--depth ARG`
`--force`
`--ignore-externals`
`--quiet (-q)`
`--revision (-r) REV`

示例

将工作副本检出到名为 *mine* 的目录内:

```
$ 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
$
```

把两个不同的目录检出到两个工作副本中:

```
$ 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
$
```

把两个不同的目录检出到两个工作副本中，不过这两个工作副本都在 *working-copies* 目录里：

```
$ 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
```

如果检出被中断（例如被用户主动中断，或者网络连接断开），为了从 中断的地方继续往下执行，你可以再次执行相同的检出命令，或者在不完整的 工作副本里执行更新命令：

```
$ svn checkout file:///var/svn/repos/test mine
A    mine/a
A    mine/b
^C
svn: E200015: Caught signal
$ svn checkout file:///var/svn/repos/test mine
A    mine/c
^C
svn: E200015: Caught signal
$ svn update mine
Updating 'mine':
A    mine/d
Updated to revision 20.
$
```

如果你希望检出特定版本号的工作副本，就为命令 *svn checkout* 加上选项 `--revision`(`-r`)：

```
$ svn checkout -r 2 file:///var/svn/repos/test mine
A    mine/a
Checked out revision 2.
$
```

在 Subversion 1.7 之前，如果检出命令将要创建的文件已经在目录中存 在了，那么默认情况下 Subversion 将会发出抱怨. Subversion 1.7 对这种 情况采取了不同的处理方式，它将继续执行检出操作，同时把所有妨碍命令的 对象都标记成目录冲突，除非带上了选项 `--force`. 如果 检出命令带上了选项 `--force`, 那么工作副本中未被版本 控制的文件如果和检出命令将要创建的文件重名，那么这些文件仍然会变成 被版本控制了的文件，但是 Subversion 不会去修改文件内容，所以说如果这 些已有的文件和仓库里的文件内容不一样，文件将会包含本地修改.

```
$ mkdir project
$ mkdir project/lib
$ touch project/lib/file.c
$ svn checkout file:///var/svn/repos/project/trunk project --force
E    project/lib
A    project/lib/subdir
E    project/lib/file.c
A    project/lib/anotherfile.c
A    project/include/header.h
Checked out revision 21.
$ svn status wc
M       project/lib/file.c
$ svn diff wc
Index: project/lib/file.c
===================================================================
--- project/lib/file.c (revision 1)
+++ project/lib/file.c (working copy)
@@ -3 +0,0 @@
-/* file.c: Code for acting file-ishly. */
-#include <stdio.h>
-/* Not feeling particularly creative today. */

$
```

现在, 你的选择有: 撤消全部或部分的本地 “修改”，提交这些修改, 或继续修改你的工作副本.

这个特性很适合用来在位地导入未被版本控制的目录树. 首先把目录树 导入仓库中, 然后带上选项 `--force` 把工作副本检出到 一个未被版本控制的目录树中, 在效果上等价于把未被版本控制的目录树转换 成一个工作副本.

```
$ svn mkdir -m "Create newproject project root." \
      file:///var/svn/repos/newproject
$ svn import -m "Import initial newproject codebase." newproject \
      file:///var/svn/repos/newproject/trunk
Adding         newproject/include
Adding         newproject/include/newproject.h
Adding         newproject/lib
Adding         newproject/lib/helpers.c
Adding         newproject/lib/base.c
Adding         newproject/notes
Adding         newproject/notes/README

Committed revision 22.
$ svn checkout file://`pwd`/repos-1.6/newproject/trunk newproject --force
E    newproject/include
E    newproject/include/newproject.h
E    newproject/lib
E    newproject/lib/helpers.c
E    newproject/lib/base.c
```

```
E    newproject/notes
E    newproject/notes/README
Checked out revision 2.
$ svn status newproject
$
```

名称

svn cleanup — 递归地清理工作副本

大纲

```
svn cleanup [PATH...]
```

描述

递归地清理工作副本，删除所有的工作副本锁，并恢复未完成的操作．如果 用户看到了一个 `working copy locked` 错误，就执行 *svn cleanup* 删除所有过期的锁，然后工作副本就能恢复 到一个可用的状态．

假设由于某种原因（例如用户输入或网络连接断开），*svn update* 在执行一个外部差异比较工具时报错退出，那么我们可 以在执行 *svn cleanup* 时带上选项 `--diff3-cmd`，此时 Subversion 在清理工作副本时就 会使用外部差异比较工具继续完成未完成的合并．用户还可以用选项 `--config-dir` 为 *svn cleanup* 指定 任意的配置文件目录，不过选项 `--config-dir` 和 `--diff3-cmd` 的使用频率应该会非常低．

选项

`--diff3-cmd` *CMD*

示例

这里的例子不是很多，因为 *svn cleanup* 不会产生 什么输出．如果没有指定参数 *PATH*，命令将会 使用当前目录（“.”）：

```
$ svn cleanup
$ svn cleanup /var/svn/working-copy
```

名称

svn commit (ci) — 把工作副本里的修改发送到仓库中.

大纲

```
svn commit [PATH...]
```

描述

把工作副本里的修改发送到仓库中. 如果用户没有用选项 `--file` (`-F`) 或 `--message` (`-m`) 提供日志消息, *svn* 将打开一个文本编辑器供用户编写提交日志, 见 “通用配置选项”一节.

如果没有带上选项 `--no-unlock`, *svn commit* 会把所有已提交的 *PATH* 上的锁令牌（如果有的话）发送给仓库, 并在提交完成后解锁.

如果用户已经开始提交并且 Subversion 已经启动了一个文本编辑器 等待用户输入提交日志, 此时仍然可以中止提交. 如果用户希望中止提交, 只需要不保存提交日志并退出编辑器, 此时 Subversion 将会询问用户是 想中止提交, 还是不输入日志直接提交, 还是重新打开编辑器编写日志.

选项

```
--changelist (--cl) ARG
--depth ARG
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force-log
--keep-changelists
--message (-m) MESSAGE
--no-unlock
--quiet (-q)
--targets FILENAME
--with-revprop ARG
```

示例

把提交日志直接写在命令行上, 如果没有指定目标文件, 则默认提交的是 当前目录 (“.”) 下的所有修改:

```
$ svn commit -m "added howto section."
Sending        a
Transmitting file data .
Committed revision 3.
```

提交 *foo.c* 的修改, 并从文件 *msg* 读取提交日志:

```
$ svn commit -F msg foo.c
```

```
Sending        foo.c
Transmitting file data .
Committed revision 5.
```

如果选项 `--file (-F)` 所指定的 文件处于版本控制中，而你的确想从该文件中读取提交日志，就要额外加上 选项 `--force-log`:

```
$ svn commit -F file_under_vc.txt foo.c
svn: E205004: Log message file is a versioned file; use '--force-log' to override

$ svn commit --force-log -F file_under_vc.txt foo.c
Sending        foo.c
Transmitting file data .
Committed revision 6.
```

提交一个将被删除的文件:

```
$ svn commit -m "removed file 'c'."
Deleting       c

Committed revision 7.
```

名称

svn copy (cp) — 在工作副本或仓库中复制一个文件或目录.

大纲

`svn copy SRC[@REV]... DST`

描述

在工作副本或仓库中复制一个或多个文件. *SRC* 和 *DST* 可以是一个工作副本 (WC) 路径或 URL. 当复制多个源文件时, 每个源文件 都将是 *DST* (此时 *DST* 必须是一个已存在的目录) 的直接子文件.

WC → WC

复制并添加一个工作副本路径 (包含历史).

WC → URL

立刻把 WC 的副本提交到 URL.

URL → WC

把 URL 检出到 WC, 并添加 WC.

URL → URL

执行一个服务器端的复制. 这通常用于创建分支和标签.

如果没有提供限定版本号 (即 *@REV*), 那么在复制工作 副本路径时, 默认使用 BASE 版本号, 在复制 URL 时默认使用 HEAD 版本号.

你只能在同一个仓库内复制文件, Subversion 不支持在不同的仓库之间 进行复制.

选项

`--editor-cmd CMD`
`--encoding ENC`
`--file (-F) FILENAME`
`--force-log`
`--ignore-externals`
`--message (-m) MESSAGE`
`--parents`
`--quiet (-q)`
`--revision (-r) REV`

`--with-revprop ARG`

示例

复制工作副本里的一个文件，并把它添加到需要被版本控制的列表里（在提交之前，这个复制操作丝毫不会影响到仓库）：

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +    bar.txt
```

把工作副本里的多个文件复制到一个目录内：

```
$ svn copy bat.c baz.c qux.c src
A         src/bat.c
A         src/baz.c
A         src/qux.c
```

复制工作副本文件 *bat.c* 的版本号 8，并重新 命名复制后的文件.

```
$ svn copy -r 8 bat.c ya-old-bat.c
A         ya-old-bat.c
```

把工作副本里的文件复制到仓库中（这个复制会马上提交，所以用户 需要提供提交日志消息）：

```
$ svn copy near.txt file:///var/svn/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

从仓库复制一个文件到工作副本中（在提交之前，这个复制操作丝毫不会 影响到仓库）：

```
$ svn copy file:///var/svn/repos/test/far-away -r 6 near-here
A         near-here
```

这是找回仓库中已被删除文件的推荐方式！

最后是从一个 URL 复制到另一个 URL:

```
$ 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.
```

这是打标签最简单的方法—只需要用 *svn copy* 把版本号（通常是 HEAD) 复制到 *tags* 目录中.

即使忘记打标签也不需要担心—在任何时候，你总是可以为一个较 老的版本号创建标签：

```
$ 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 (del, remove, rm) — 从工作副本或仓库中删除文件.

大纲

```
svn delete PATH...

svn delete URL...
```

描述

由 *PATH* 指定的文件将在提交时从仓库中 删除. 执行完命令后，文件和未被提交的目录会马上从工作副本中删除，除非 指定了选项 `--keep-local`. 命令不会删除未被版本控制或 含有本地修改的文件，除非指定了选项 `--force`. 如果目录 内含有未被版本控制的文件，或者文件含有本地修改，这个目录就不会被删除，除非指定了选项 `--force`.

由 URL 指定的文件会马上触发一个提交操作，然后直接从仓库中删除. 如果指定了多个 URL，它们将在同一个提交中完成删除.

选项

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--keep-local
--message (-m) MESSAGE
--quiet (-q)
--targets FILENAME
--with-revprop ARG
```

示例

svn delete 会删除工作副本里的文件，但要等到 提交后才会从仓库中删除.

```
$ svn delete myfile
D         myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

删除由 URL 指定的文件会马上触发一个提交操作，所以用户需要提供 提交日志:

```
$ svn delete -m "Deleting file 'yourfile'" \
```

```
            file:///var/svn/repos/test/yourfile

Committed revision 15.
```

下面的例子展示了如何强制删除含有本地修改的文件:

```
$ svn delete over-there
svn: E195006: Use --force to override this restriction (local modifications m\
ay be lost)
svn: E195006: '/home/sally/project/over-there' has local modifications -- com\
mit or revert them first
$ svn delete --force over-there
D         over-there
$
```

加上选项 `--keep-local` 后, 被 *svn delete* 删除的文件将保留在本地. 这个选项在处理下面这种 情况时很有用: 用户不小心把一个本不应该提交的文件提交到了仓库中, 现在他想从仓库中删除该文件, 但又想把它留在工作副本里.

```
$ svn delete --keep-local conf/program.conf
D         conf/program.conf

$ svn commit -m "Remove accidentally-added configuration file."
Deleting       conf/program.conf
Transmitting file data .
Committed revision 21.
$ svn status
?       conf/program.conf
$
```

选项 `--keep-local` 的效果不会扩展到其他工作 副本, 也就是说如果用户提交了删除操作, 这些文件虽然会保留在你的 工作副本里, 但是在更新其他工作副本里, 其他工作副本里的文件仍然会 被删除.

名称

svn diff (di) — 显示两个版本号或路径之间的差异.

大纲

```
svn diff [-c M | -r N[:M]] [TARGET[@REV]...]

svn diff [-r N[:M]] --old=OLD-TGT[@OLDREV] [--new=NEW-TGT[@NEWREV]] [PATH...]

svn diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

描述

显示两个路径之间的差异, 用户可以按照如下方式使用 *svn diff*:

- 直接执行 `svn diff` 来查看工作副本的 本地修改.
- 显示限定版本号为 *REV* 时, *TARGET* 在两个版本号之间的变化. *TARGET* 可以全是工作副本路径或 *URL*. 如果 *TARGET* 是工作副本路径, 则 *N* 默认是 BASE, *M* 默认是工作副本. 如果 *TARGET* 是 *URL*, 则用户必须指定 *N*, 而 *M* 默认 是 HEAD. 选项 `-c M` 等价于 `-r N:M`, 其中 `N = M-1`; 而 `-c -M` 正好相反, 它等价于 `-r M:N`, 其中 `N = M-1`.
- 显示限定版本号为 *OLDREV* 的 *OLD-TGT* 和限定版本号 *NEWREV* 的 *NEW-TGT* 之间的差异. 如果指定了相对于 *OLD-TGT* 和 *NEW-TGT* 的 *PATH*, 那么输出的就是这些路径的差异. *OLD-TGT* 和 *NEW-TGT* 可以是工作副本路径或 *URL[@REV]*. 如果没有指定 *NEW-TGT*, 那么它默认就是 *OLD-TGT*. `-r N` 使得 *OLDREV* 默认是 *N*; `-r N:M` 使得 *OLDREV* 默认是 *N*, *NEWREV* 默认是 *M*.

`svn diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]` 是 `svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV]` 的缩写形式.

`svn diff -r N:M URL` 是 `svn diff -r N:M --old=URL --new=URL` 的缩写形式.

`svn diff [-r N[:M]] URL1[@N] URL2[@M]` 是 `svn diff [-r N[:M]] --old=URL1 --new=URL2` 的缩写形式.

如果 *TARGET* 是一个 URL, 那么 *N* 和 *M* 可以 用选项 `--revision (-r)` 指定, 或者用以前介绍过的 “@” 记号.

如果 *TARGET* 是一个工作副本路径, 那么 命令的默认行为 (没有指定选项 `--revision (-r)`) 就是显示 *TARGET* 的 BASE 与工作副本之间的差异. 如果指定了选项 `--revision (-r)`, 这意味着:

`--revision N:M`

服务器将比较 *TARGET@N* 和 *TARGET@M*.

`--revision N`

客户端将比较 *TARGET@N* 和工作 副本.

如果使用了替代语法, 服务器将比较分别处于版本号 *N* 和 *M* 下 的 *URL1* 和 *URL2*. 如果没有指定 *N* 或 *M*, 将默认使用 HEAD.

默认情况下，*svn diff* 会忽略文件的祖先，而只 比较文件的内容．如果使用了选项 `--notice-ancestry`，那么在比较版本号时就是考虑相关路径的祖先（也就是说，如果你用 *svn diff* 比较了两个内容相同，但祖先不同的文件，你 将会看到文件的整个内容曾经被删除，然后又被添加）．

选项

`--change (-c) ARG`
`--changelist (--cl) ARG`
`--depth ARG`
`--diff-cmd CMD`
`--extensions (-x) ARG`
`--force`
`--git`
`--ignore-properties`
`--internal-diff`
`--new ARG`
`--no-diff-added`
`--no-diff-deleted`
`--notice-ancestry`
`--old ARG`
`--patch-compatible`
`--properties-only`
`--revision (-r) REV`
`--show-copies-as-adds`
`--summarize`
`--xml`

示例

比较 BASE 与工作副本 (*svn diff* 最常见的用途之一)：

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
…
```

查看 *COMMITTERS* 在版本号 9115 发生了哪些 变化：

```
$ svn diff -c 9115 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
…
```

比较工作副本和更老的版本号：

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
…
```

使用 “@” 语法比较版本号 3000 和 3500:

```
$ 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)
…
```

使用范围记号比较版本号 3000 与 3500 (这时候只需要一个 URL 参数) :

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

使用范围记号比较 *trunk* 内的所有文件在 版本号 3000 到 3500 之间的变化:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

使用范围记号比较 *trunk* 内的三个文件在版本 号 3000 到 3500 之间的变化:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk \
       COMMITTERS README HACKING
```

如果你已经有了一个工作副本, 就不用再输入冗长的 URL:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

使用选项 `--diff-cmd` *`CMD`* `--extensions` (`-x`) 向外部差异比较 工具传递参数:

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
$
```

最后，可以同时使用选项 `--xml` 和 `--summarize` 查看修改的 XML 描述，修改的具体内容不 会显示出来：

```
$ 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.

大纲

```
svn export [-r REV] URL[@PEGREV] [PATH]

svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

描述

命令的第一种形式从 *URL* 所指定的仓库 导出一个干净的目录树到 *PATH*. 如果指定了 *REV*, 将导出仓库在版本号 *REV* 时的目录树, 否则的话版本号默认就是 HEAD. 如果省略 *PATH*, 那么 *URL* 的最后一个分量将作为导出后的 目录名.

命令的第二种形式从 *PATH1* 指定的工作 副本导出一个干净的目录树到 *PATH2*. 所有的 本地修改都会保留在导出的目录中, 但不包括不被版本控制的文件.

选项

```
--depth ARG
--force
--ignore-externals
--ignore-keywords
--native-eol ARG
--quiet (-q)
--revision (-r) REV
```

示例

从工作副本导出 (被导出的文件或目录的名字不会打印出来):

```
$ svn export a-wc my-export
Export complete.
```

从仓库导出 (被导出的每一个文件或目录的名字都会打印出来):

```
$ svn export file:///var/svn/repos my-export
A    my-export/test
A    my-export/quiz
…
Exported revision 15.
```

在制作特定于操作系统的发行包时, 如果能为导出的目录树指定特定的 EOL, 那将会非常方便, 选项 `--native-eol` 就是用于这 个目的, 但是它只会影响设置了属性 `svn:eol-style=native` 的文件. 例如, 为 Windows 导出以 CRLF 作为 EOL 的目录树:

```
$ svn export file:///var/svn/repos my-export --native-eol CRLF
```

```
A    my-export/test
A    my-export/quiz
…
Exported revision 15.
```

选项 `--native-eol` 接受的参数有 `LR`, `CR` 和 `CRLF`.

名称

svn help (h, ?) — 帮助!

大纲

`svn help [SUBCOMMAND...]`

描述

如果你无法获取这本书, 那么这个命令就是你使用 Subversion 的最好帮手!

选项

没有选项

名称

svn import — 把未被版本控制的文件或目录树提交到仓库中.

大纲

`svn import [PATH] URL`

描述

递归地把 *PATH* 提交到 *URL*, 如果省略了 *PATH*, 将使用 ".". 在必要时会在仓库中创建父目录. 即使添加了选项 `--force`, 无法被版本控制的文件 (例如 设备文件和命名管道) 也会被忽略.

选项

`--auto-props`
`--depth ARG`
`--editor-cmd CMD`
`--encoding ENC`
`--file (-F) FILENAME`
`--force`
`--force-log`
`--message (-m) MESSAGE`
`--no-auto-props`
`--no-ignore`
`--quiet (-q)`
`--with-revprop ARG`

示例

下面的命令把本地目录 *myproj* 导入到仓库的 *trunk/misc* 目录中. 在导入前不用事先创建 *trunk/misc*—*svn import* 会自动创建所需的目录.

```
$ svn import -m "New import" myproj \
             http://svn.red-bean.com/repos/trunk/misc
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

需要注意的是上面的命令 不会 在仓库中创建目录 *myproj*, 但是如果你希望创建该目录, 就把 *myproj* 作为 URL 的最后一个分量:

```
$ 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.
```

导入完成后，被导入的本地目录 不会 变成工作 副本，用户还是需要使用 *svn checkout* 检出工作副本.

名称

svn info — 显示文件的相关信息，该文件可以在本地工作副本或远程服务器 上.

大纲

```
svn info [TARGET[@REV]...]
```

描述

打印工作副本路径或 URL 的相关信息，相关信息可能包括：

- 该对象所在的仓库信息
- 该对象的最近一次提交
- 该对象上的用户级别的锁
- 本地的调度信息（添加，删除，复制等）
- 本地的冲突信息

选项

```
--changelist (--cl) ARG
--depth ARG
--incremental
--recursive (-R)
--revision (-r) REV
--targets FILENAME
--xml
```

示例

svn info 会显示工作副本路径的所有相关信息，例如显示文件的信息：

```
$ svn info foo.c
Path: foo.c
Name: foo.c
Working Copy Root Path: /home/sally/projects/test
URL: http://svn.red-bean.com/repos/test/foo.c
Relative URL: ^/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
```

显示目录的信息:

```
$ svn info vendors
Path: vendors
Working Copy Root Path: /home/sally/projects/test
URL: http://svn.red-bean.com/repos/test/vendors
Relative URL: ^/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 也能针对 URL 进行操作 (注意在下面 例子里, 文件 *readme.doc* 是被加锁的, 所以输出中 包含了关于锁的信息):

```
$ 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
Relative URL: ^/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
```

最后, 如果添加了选项 `--xml`, *svn info* 还能以 XML 格式打印输出:

```
$ 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>
<relative-url>^/</relative-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 (ls) — 列出仓库中的目录项.

大纲

`svn list [TARGET[@REV]...]`

描述

列出仓库中的每一个 *TARGET* 文件和 *TARGET* 目录的内容. 如果 *TARGET* 是一个工作副本路径, 命令将自动使用 工作路径的 URL.

TARGET 的默认值是 ".", 即当前工作副本目录在仓库中的 URL.

如果带上了选项 `--verbose` (`-v`), *svn list* 将额外显示项目的如下字段:

- 最近一次提交的版本号
- 最近一次提交的作者
- 如果文件被锁定了, 则显示字母 "O" (更多的细节 见前面的 svn info
- 大小 (以字节为单位)
- 最近一次提交的日期和时间

如果添加了选项 `--xml`, 输出将会是 XML 格式 (带有头部信息和封闭的文件元素, 除非还添加了选项 `--incremental`), 所有的信息都会被呈现出来, 但 此时不能再使用选项 `--verbose` (`-v`).

选项

`--depth ARG`
`--incremental`
`--recursive (-R)`
`--revision (-r) REV`
`--verbose (-v)`
`--xml`

示例

如果你想查看仓库里有哪些文件, 但又不想检出工作副本, 那么 *svn list* 就显得很方便了:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
```

…

如果你想查看更多的信息，就添加选项 `--verbose`，就像 Unix 命令 *ls -l*：

```
$ svn list -v 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/
```

添加选项 `--xml` 后，*svn list* 将以 XML 格式显示输出：

```
$ 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>
```

更多的信息见 “列出被版本控制的文件”一节.

名称

svn lock — 根据工作副本路径或 URL，为仓库里的文件加锁，从而阻止其他 用户提交这些文件的修改.

大纲

```
svn lock TARGET...
```

描述

锁定每一个 *TARGET*. 如果存在 *TARGET* 已经被其他用户锁定了， Subversion 将打印一个警告， 然后继续锁定剩下的 *TARGET*. 如果想要从其他用户或工作副本那儿窃取锁，就加上选项 `--force`.

选项

```
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--message (-m) MESSAGE
--targets FILENAME
```

示例

锁定工作副本中的两个文件:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

锁定一个已经被其他用户锁定的文件:

```
$ svn lock tree.jpg
svn: warning: W160035: Path '/tree.jpg is already locked by user 'sally' in fi
lesystem '/var/svn/repos/db'
$ svn lock --force tree.jpg
'tree.jpg' locked by user 'harry'.
```

在没有工作副本的情况下锁定一个文件:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'harry'.
```

更多的细节，见 “锁”一节.

名称

svn log — 显示提交日志消息.

大纲

```
svn log [PATH]

svn log URL[@REV] [PATH...]
```

描述

显示仓库的日志消息. 如果没有指定目标参数, *svn log* 将显示当前工作副本目录的日志消息. 为了提炼输出结果, 你可以指定一个路径, 一个或多个版本号, 或路径与版本号的组合. Subversion 为一个本地路径打印的默认版本号范围是 `BASE:1`.

如果用户只指定了一个 URL, 命令将会为 URL 内的所有文件打印日志 消息, 如果你在 URL 中添加了路径, 命令将只打印该路径的日志消息. Subversion 为一个 URL 打印的默认版本号范围是 `BASE:1`.

如果添加了选项 `--verbose` (`-v`), *svn log* 还会额外打印在本次修改中受到影响的路径. 如果添加了选项 `--quiet` (`-q`), *svn log* 将不会打印日志消息的文本内容, 这个选项和 选项 `--verbose` (`-v`) 是兼容的.

即使在一次提交中受到影响的路径在命令行中出现了多次, 对应的日志 消息也只显示一次. 日志默认会跟随复制历史, 但可以用选项 `--stop-on-copy` 禁止跟随, 这可以用来判断创建分支的 时间点.

选项

```
--change (-c) ARG
--depth ARG
--diff
--diff-cmd CMD
--extensions (-x) ARG
--incremental
--internal-diff
--limit (-l) NUM
--quiet (-q)
--revision (-r) REV
--search ARG
--search-and ARG
--stop-on-copy
--targets FILENAME
--use-merge-history (-g)
--verbose (-v)
--with-all-revprops
--with-no-revprops
--with-revprop ARG
--xml
```

示例

如果直接运行 `svn log`, 则查看的是当前工作 副本目录中, 所有路径的修改历史:

```
$ 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
…
```

查看工作副本中, 某个特定文件的修改历史:

```
$ 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
…
```

如果手上没有工作副本, 还可以根据 URL 查询历史:

```
$ 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
…
```

如果你想查看同一个 URL 下的多个路径, 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
…
```

如果添加了选项 `--verbose` (`-v`), *svn log* 还会打印在版本号中被修改的路径，每个路径 都占据单独的一行，并在行的开头添加能够表示修改类型的操作码.

```
$ 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 只使用了几种操作码，这些操作码和命令 *svn update* 使用的操作码非常类似:

`A`

该项是新增的.

`D`

该项被删除了.

`M`

该项的属性或内容被修改了.

`R`

该项被其他项目替换了.

除了在路径前加上操作码，添加了选项 `--verbose` (`-v`) 的 *svn log* 还会注释路径 是否是通过复制得到的，注释的形式是在路径后加上 (`from` *`COPY-FROM-PATH`*`:`*`COPY-FROM-REV`*).

如果你想串联多个 *svn log* 的执行结果，那你可能 需要选项 `--incremental`. *svn log* 通常会在开始打印日志消息之前，在打印完一条日志消息之后，以及打印完最 后一条日志消息之后打印一行连字符. 如果你用 *svn log* 查询一段版本号范围内的日志，你将会得到:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | …

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

```
r15 | …

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

然而，如果你想把多个不连续的日志消息写到一个文件中，你可能会这 样做：

```
$ svn log -r 14 > mylog
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | …

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

为了避免出现凌乱的双条连字符行，可以为 *svn log* 加上选项 `--incremental`：

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
r27 | …
```

如果已经添加了选项 `--xml`，再加上选项 `--incremental` 后能提供类似的输出控制：

```
$ 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>
```

有时候，如果你用 *svn log* 查询某个路径在 特定版本号上的历史，你可能看不到任何日志被打印出来，例如：

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
```

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

这仅仅意味着该路径在该版本号上没有发生任何修改. 为了得到该版本 号的日志消息, 要么把仓库的根目录 URL 作为目标, 要么你很清楚该版本 号修改了哪个路径, 并把它作为目标:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

从 Subversion 1.7 开始, *svn log* 支持一种特殊 的输出模式, 该模式和 *svn diff* 一样能够把修改以 标准差异格式输出, 方法就是为 *svn log* 加上选项 `--diff`:

```
$ svn log -r 20 touched.txt --diff
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.

Index: touched.txt
===================================================================
--- touched.txt (revision 19)
+++ touched.txt (revision 20)
@@ -1 +1,2 @@
 This is the file 'touched.txt'.
+We add such exciting text to files around here!
------------------------------------------------------------------------
$
```

和 *svn diff* 一样, 用户可以利用选项 `--depth`, `--diff-cmd` 和 `--extensions (-x)` 控制差异比较的 具体行为.

从 Subversion 1.8 开始, 用户可以用选项 `--search` 和 `--search-and` 过滤 *svn log* 的 输出. 如果在命令中用到了这两个选项, 那么仅当版本号的作者, 提交日期, 日志消息内容或被修改的路径和模式匹配时, 日志消息才会被打印出来. 在 匹配被修改的路径时, 还要求加上选项 `--verbose`, 否则 的话 *svn log* 不会去匹配被修改的路径.

搜索模式 (也被称作文本表达式或 Shell 通配符模式) 可以包含普通 字符和下面的通配符:

`?`

匹配任意一个字符.

`*`

匹配任意一个字符序列.

`[ABC]`

匹配中括号内的任意一个字符.

如果选项 `--search` 出现了多次，那么只要任意一个 模式得到匹配，日志消息就会被打印，例如：

```
$ svn log --search sally --search harry https://svn.red-bean.com/repos/test
------------------------------------------------------------------------
r1701 | sally | 2011-10-12 22:35:30 -0600 (Wed, 12 Oct 2011) | 1 line

Add a reminder.
------------------------------------------------------------------------
r1564 | harry | 2011-10-09 22:35:30 -0600 (Sun, 09 Oct 2011) | 1 line

Merge r1560 to the 1.0.x branch.
------------------------------------------------------------------------
$
```

如果选项 `--search` 和 `--search-and` 一起使用，则只有同时匹配所有模式的 日志消息才会被打印，例如：

```
$ svn log --verbose --search sally --search-and /foo/bar https://svn.red-bean.com/repos/test
------------------------------------------------------------------------
r1555 | sally | 2011-07-15 22:33:14 -0600 (Fri, 15 Jul 2011) | 1 line
Changed paths:
M /foo/bar/src.c

Typofix.
------------------------------------------------------------------------
r1530 | sally | 2011-07-13 07:24:11 -0600 (Wed, 13 Jul 2011) | 1 line
Changed paths:
M /foo/bar
M /foo/build

Fix up some svn:ignore properties.
------------------------------------------------------------------------
$
```

选项 `--search` 和 `--search-and` 实际上并没有执行搜索，它们仅仅是过滤 *svn log* 的 输出. 因此，如果还使用了选项 `--limit`, 它将会限制 过滤前的日志消息数量，而不是限制已过滤的日志消息数量.

名称

svn merge — 把两个源的差异应用到工作副本上.

大纲

```
svn merge SOURCE[@REV] [TARGET_WCPATH]

svn merge [-c M[,N...] | -r N:M ...] SOURCE[@REV] [TARGET_WCPATH]

svn merge SOURCE1[@N] SOURCE2[@M] [TARGET_WCPATH]
```

描述

在上面的三种形式中，*TARGET_WCPATH* 都是一个接收差异的工作副本路径. 如果省略了 *TARGET_WCPATH*, 接收差异的将是当前工作目录，如果源的最后一个分量和当前工作目录内的某个文件名相同，则接收差异的将 会是这个文件.

在前两种形式中，*SOURCE* 是一个 URL 或 工作副本路径（此时 Subversion 将自动使用工作副本路径对应的 URL). 如果没有指定限定版本号 *REV*, 就默认使用 HEAD. 对于第三种形式，相同的规则同样适用于 *SOURCE1*, *SOURCE2*, *M* 和 *N*, 唯一的不同点是如果 *SOURCE1* 或 *SOURCE2* 是一个工作副本路径，则必须 显式地 指定限定版本号.

- 自动合并

  第一种形式称为 “自动合并”，它用于执行 “同步” 合并和 “再整合” 合并. “同步” 合并把祖先分支 (*SOURCE*) 中可以合并的修改合并到 目标分支 (*TARGET_WCPATH*) 上. “可以合并的” 修改指的是那些还没有从 *SOURCE* 合并到 *TARGET_WCPATH* 的修改，见 “保持分支同步”一节. “再整合” 合并是把特性分支的修改 (*SOURCE*) 合并到祖先分支 (*TARGET_WCPATH*) 上，见 “重新整合分支”一节 和 “特性分支”一节.

- 精选合并

  第二种形式称为 “精选” 合并，用于合并特定的修改，修改来自 *SOURCE* (限定版本号为 *REV*) 的版本号 *N* 和 *M* 的 比较. 更多的信息见 “精选”一节.

  `-c` 或 `-r` 可以出现多次，也可以混合正向和反向的版本号范围—在合并开始前，Subversion 自动地把版本号范围压缩成最简的表示形式 (这可能会导致空合并或 冲突，从而导致合并中止).

- 二路 URL 合并

  第三种形式称为 “二路 URL 合并”，限定版本号为 *N* 的 *SOURCE1* 和限定版本号为 *M* 的 *SOURCE2* 之间的差异将被应用到 *TARGET_WCPATH* 上. 如果省略了限定版本号，则使用 HEAD.

当两个合并的源路径具有祖先关系时—第一个源路径是另一个源路径 的祖先，或反之 (自动合并总是符合这一条件—如果合并跟踪 是可用的，则 Subversion 会在内部更新合并操作的元数据. 在判断需要合并哪些版本号 时，Subversion 还会把工作副本目标路径上已存在的合并元数据考虑进来，从而避免重复的合并和不必要的冲突.

和 *svn diff* 不一样的是，*svn merge* 在执行合并时还会考虑到文件的祖先，这在处理下面这种 情况时非常有用：把一个分支上的修改合并到另一个分支中，而在其中一个分 支中文件被重命名了.

选项 `--ignore-ancestry` 将禁止 合并跟踪，使得 *svn merge* 在合并时不考虑祖先关系，就像 *svn diff* 那样工作.

选项

`--accept ACTION`
`--allow-mixed-revisions`
`--change (-c) ARG`
`--depth ARG`
`--diff3-cmd CMD`
`--dry-run`
`--extensions (-x) ARG`
`--force`
`--ignore-ancestry`
`--quiet (-q)`
`--record-only`
`--reintegrate`
`--revision (-r) REV`
`--verbose (-v)`

示例

把分支重新整合到主干上—假设你已经有了一个处于最新状态 下的主干工作副本（选项 `--verbose` 使得 *svn merge* 在应用差异前打印额外的信息，这些信息指出了现在 正在执行什么合并. 如果命令的运行时间比较长，那么打印一些信息就会很有帮助）：

```
$ svn merge ^/branches/feature-branch-calc-enhancements trunk --verbose
checking branch relationship...
calculating automatic merge...
merging...
--- Merging r12 through r37 into 'trunk':
U    trunk/calc/brush.c
--- Recording mergeinfo for merge of r12 through r37 into 'trunk':
 U   trunk

$ # build, test, verify, ...

$ svn commit trunk -m "Reintegrate the calc enhancements back to trunk!"
Sending        trunk
Sending        trunk/calc/brush.c
Transmitting file data .
Committed revision 38.
```

为文件合并一个单独的修改（精选合并）：

```
$ svn merge ^/trunk/calc/brush.c branches/1.x/calc/brush.c -c38
--- Merging r38 into 'branches/1.x/calc/brush.c':
```

```
U    branches/1.x/calc/brush.c
--- Recording mergeinfo for merge of r38 into 'branches/1.x/calc/brush.c':
 G   branches/1.x/calc/brush.c
```

把两个不相关的分支间的差异合并到第三个分支上:

```
$ svn merge ^/vendor-drop/vendor-1.0 ^/vendor-drop/vendor-1.1 \
            trunk --ignore-ancestry
--- Merging differences between repository URLs into 'trunk':
U    trunk/draw/draw.py
```

名称

svn mergeinfo — 查询合并信息，见 “合并信息和预览”一节.

大纲

`svn mergeinfo SOURCE_URL[@REV] [TARGET[@REV]]`

描述

查询 *SOURCE-URL* 和 *TARGET* 之间与合并（或潜在的合并）相关的 信息. 如果没有添加选项 `--show-revs`, 命令会把已合并 的版本号以图形化的方式打印出来；否则的话，命令将会列出已合并或可以 合并（但还未合并）的版本号.

选项

`--depth ARG`
`--recursive (-R)`
`--revision (-r) REV`
`--show-revs ARG`

示例

以图形化的方式总结从一个分支合并到另一个分支上的修改：

```
$ svn mergeinfo ^/trunk feature-branch
    youngest  last               repos.
    common    full     tip of    path of
    ancestor  merge    branch    branch

    11        16       33
    |         |        |
  -------| |------------         trunk
      \         \
       \         \
        --| |------------         feature-branch
                    |
                    33
```

列出某个分支上已经合并到另一个分支的所有版本号：

```
$ svn mergeinfo ^/trunk feature-branch --show-revs merged
r15
r16
```

列出某个分支上可以合并（但还未合并）到另一个分支的版本号：

```
$ svn mergeinfo ^/trunk feature-branch --show-revs eligible
r28
```

```
r30
```

名称

svn mkdir — 创建一个新目录.

大纲

```
svn mkdir PATH...

svn mkdir URL...
```

描述

创建一个新目录，新目录的名字是 *PATH* 或 *URL* 的最后一个分量. 如果参数是工作副本 路径 *PATH*，那么目录在创建后被自动置于版本 控制之下；如果参数是 *URL*，那么新目录会被 直接提交到仓库中，如果出现了多个 *URL*，则 它们都是在同一个版本号中提交. 除非添加了选项 `--parents`，否则的话中间目录必须事先存在.

选项

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force-log
--message (-m) MESSAGE
--parents
--quiet (-q)
--with-revprop ARG
```

示例

在工作副本中创建新目录：

```
$ svn mkdir newdir
A         newdir
```

在仓库中创建目录（这会马上产生一个提交操作，所以必须提供日志消息）：

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir

Committed revision 26.
```

名称

svn move (mv) — 移动一个文件或目录.

大纲

`svn move SRC... DST`

描述

该命令用于在工作副本或仓库中移动文件或目录.

这个命令等价于先执行 *svn copy*, 再执行 *svn delete*.

如果 *SRC* 出现了多次, 它们将被移动 到 *DST* 内, 这就意味着 *DST* 必须是一个目录.

Subversion 不支持在工作副本和 URL 之间移动, 而且只能在同一仓库 内移动—即 Subversion 不支持在不同的仓库之间移动. 在同一仓库 内, Subversion 支持以下类型的移动:

WC → WC

在工作副本内移动文件, 但还未提交到仓库中.

URL → URL

在仓库内完成移动, 会马上触发一个提交操作.

如果被移动的文件较多, 那么用户应该使用更轻量的 URL → URL, 在工作副本内移动文件不仅仅是修改目录列表 (还要复制文件, 管理临时文件 和扩展关键字), 可能会耗费较多的时间.

还要注意的是 WC → WC 移动版本号混合的工作副本可能会产生无法 预知的后果 (见 “版本号混合的工作副本”一节).

选项

`--editor-cmd CMD`
`--encoding ENC`
`--file (-F) FILENAME`
`--force`
`--force-log`
`--message (-m) MESSAGE`
`--parents`
`--quiet (-q)`
`--revision (-r) REV`
`--with-revprop ARG`

示例

移动工作副本里的一个文件:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

把工作副本里的几个文件移动到一个目录内:

```
$ 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
```

在仓库内移动一个文件 (这会产生一个提交操作, 所以需要提供提交日志 消息):

```
$ 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 patch — 把一个标准差异格式的补丁应用到工作副本中.

大纲

`svn patch` *`PATCHFILE`* [*`WCPATH`*]

描述

该命令把标准差异格式的补丁 *`PATCHFILE`* 应用到工作副本路径 *`WCPATH`* 上. 和大多数 操作工作副本的子命令一样, 如果省略了 *`WCPATH`*, 应用补丁的将会是当前工作目录. 有很多工具都能生成标准差异格式的补丁, 例如 *svn diff* 和 Unix 命令 *diff*. *svn patch* 会忽略补丁中不是标准差异的内容.

补丁中的修改要么被接受, 要么被拒绝. 如果没办法在准确的位置上 匹配一个修改, 那么 Subversion 将会在更前或更后的位置进行匹配. 还 可以 *模糊地* (*fuzz*) 应用修改—也就是在尝试匹配应用修改的位置时, 忽略上下文中的一 行或更多的行. 如果无法为一个修改找到匹配的位置, 该修改将被认为发生 了冲突, 并被写到一个扩展名为 *.svnpatch.rej* 的 文件中.

svn patch 为每一个打了补丁的文件或目录打印一 行状态, 状态用一个字母表示, 和 *svn update* 非常类似. 用来表示状态的字母有:

`A`

新增 (Added)

`D`

删除 (Deleted)

`C`

冲突 (Conflicted)

`G`

合并 (Merged)

`U`

更新 (Updated)

如果修改是在调整了偏移后应用的, 或者是被模糊应用的, 那么 Subversion 将报告这一情况, 相关的信息在输出时以字符 `>` 开始, 用户应该认真审查这些修改.

如果补丁把文件的所有内容都删除了, 该文件将被自动删除. 类似的, 如果补丁创建了新文件, 那么该文件将被自动添加. 命令 *svn revert* 可以撤消所有不需要的删除或添加.

选项

```
--dry-run
--ignore-whitespace
--quiet (-q)
--reverse-diff
--strip NUM
```

示例

应用一个由 *svn diff* 生成的补丁，该补丁删除了 一个文件，创建了另一个新文件，并修改了第三个文件的内容和属性. 下面是 补丁的内容（假设该补丁的文件名就是 *PATCH*):

```
Index: deleted-file
===================================================================
--- deleted-file (revision 3)
+++ deleted-file (working copy)
@@ -1 +0,0 @@
-This file will be deleted.
Index: changed-file
===================================================================
--- changed-file (revision 4)
+++ changed-file (working copy)
@@ -1,6 +1,6 @@
 The letters in a line of text
 Could make your day much better.
 But expanded into paragraphs,
-I'd tell of kangaroos and calves
+I'd tell of monkeys and giraffes
 Until you were all smiles and laughs
 From my letter made of letters.

Property changes on: changed-file
___________________________________________________________________
Added: propname
## -0,0 +1 ##
+propvalue
Index: added-file
===================================================================
--- added-file (revision 0)
+++ added-file (working copy)
@@ -0,0 +1 @@
+This is an added file.
```

下面我们将在工作副本中应用该补丁，然后再用 *svn diff* 检查 *svn patch* 的执行结果是否正确:

```
$ cd /some/other/workingcopy
$ svn patch /path/to/PATCH
D         deleted-file
UU        changed-file
A         added-file
```

```
$ svn diff
Index: deleted-file
===================================================================
--- deleted-file (revision 3)
+++ deleted-file (working copy)
@@ -1 +0,0 @@
-This file will be deleted.
Index: changed-file
===================================================================
--- changed-file (revision 4)
+++ changed-file (working copy)
@@ -1,6 +1,6 @@
 The letters in a line of text
 Could make your day much better.
 But expanded into paragraphs,
-I'd tell of kangaroos and calves
+I'd tell of monkeys and giraffes
 Until you were all smiles and laughs
 From my letter made of letters.

Property changes on: changed-file
___________________________________________________________________
Added: propname
## -0,0 +1 ##
+propvalue
Index: added-file
===================================================================
--- added-file (revision 0)
+++ added-file (working copy)
@@ -0,0 +1 @@
+This is an added file.
$
```

有时候, 用户可能想“逆向”应用补丁—删除新增的行, 添加被删除的行. 选项 `--reverse-diff` 就是用来完成这一操作. 在下面的例子中, 我们先把工作副本的修改存放到一个补丁文件中, 然后再通过逆向应用补丁来撤消工作副本的修改.

```
$ svn status
M       foo.c
$ svn diff > PATCH
$ cat PATCH
Index: foo.c
===================================================================
--- foo.c (revision 128)
+++ foo.c (working copy)
@@ -1003,7 +1003,7 @@
     return ERROR_ON_THE_G_STRING;

   /* Do something in a loop. */
```

```
-  for (i = 0; i < txns->nelts; i++)
+  for (i = 0; i < txns->nelts; i--)
     {
       status = do_something(i);
       if (status)
$ svn patch --reverse-diff PATCH
U         foo.c
$ svn status
$
```

名称

svn propdel (pdel, pd) — 删除一个属性.

大纲

```
svn propdel PROPNAME [PATH...]
```

```
svn propdel PROPNAME --revprop -r REV [TARGET]
```

描述

该命令将从文件，目录或版本号上删除一个属性. 命令的第一种形式在 工作副本中删除一个版本化的属性；第二种形式在仓库的版本号上删除一个 非版本化的属性 (可选参数 *TARGET* 指定仓库 的 URL).

选项

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
```

示例

删除文件上的一个属性，该文件是在工作副本中.

```
$ svn propdel svn:mime-type some-script
property 'svn:mime-type' deleted from 'some-script'.
```

删除一个版本号上的属性:

```
$ svn propdel --revprop -r 26 release-date
property 'release-date' deleted from repository revision '26'
```

名称

svn propedit (pedit, pe) — 编辑一个或多个项目的版本化属性，参考后面的 svn propset (pset, ps).

大纲

```
svn propedit PROPNAME TARGET...

svn propedit PROPNAME --revprop -r REV [TARGET]
```

描述

在编辑器中编辑一个或多个项目属性. 命令的第一种形式在工作副本中编辑 版本化的属性；第二种形式在仓库的版本号上编辑非版本化的属性（可选参数 *TARGET* 指定仓库的 URL).

选项

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--message (-m) MESSAGE
--revision (-r) REV
--revprop
--with-revprop ARG
```

示例

svn propedit 非常适合编辑属性值比较复杂的属性.

```
$ svn propedit svn:keywords foo.c

    # svn will open in your favorite text editor a temporary file
    # 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.  When you save the temporary file and exit,
    # Subversion will re-read the temporary file and use its updated
    # contents as the new value of the property.

Set new value for property 'svn:keywords' on 'foo.c'
$
```

名称

svn propget (pget, pg) — 打印属性的值.

大纲

```
svn propget PROPNAME [TARGET[@REV]...]

svn propget PROPNAME --revprop -r REV [URL]
```

描述

打印文件，目录或版本号上的属性值. 命令的第一种形式打印工作副本 里的一个或多个项目的属性值；第二种形式打印版本号上的属性值. 关于属性 的更多信息，见 “属性”一节.

选项

```
--changelist (--cl) ARG
--depth ARG
--recursive (-R)
--revision (-r) REV
--revprop
--show-inherited-props
--strict
--verbose (-v)
--xml
```

示例

查看工作副本中某个文件的属性值：

```
$ svn propget svn:keywords foo.c
Author
Date
Rev
```

查看一个版本号的属性值：

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

为了获得更加结构化的输出，可以加上选项 `--verbose` (`-v`)：

```
$ svn propget svn:keywords foo.c --verbose
Properties on 'foo.c':
  svn:keywords
    Author
    Date
```

```
    Rev
```

为了查看继承而来的属性，可以加上选项 `--show-inherited-props`:

```
$ svn pg svn:global-ignores --verbose --show-inherited-props ^/branches/1.x
Inherited properties on 'http://svn.example.com/repos/branches/1.x',
from 'http://svn.example.com/repos':
  svn:global-ignores
    *.diff
    *.patch
```

默认情况下，*svn propget* 在打印完属性值后会额外 打印一个换行符，在大多数情况下这是一种比较合理的行为，但有时候用户 可能想精确地打印属性值，比如属性值根本就不是纯文本，例如 JPEG 缩略 图. 为了禁止打印额外的换行符，可以加上选项 `--strict`.

最后，为了让 *svn propget* 以 XML 格式显示输出，可以加上选项 `--xml`:

```
$ svn propget --xml svn:ignore .
<?xml version="1.0"?>
<properties>
<target
   path="">
<property
   name="svn:ignore">*.o
</property>
</target>
</properties>
```

名称

svn proplist (plist, pl) — 列出所有的属性.

大纲

```
svn proplist [TARGET[@REV]...]

svn proplist --revprop -r REV [TARGET]
```

描述

列出文件，目录或版本号上的所有属性. 命令的第一种形式为工作副本里 的项目列出属性；第二种形式列出版本号上的属性（可选参数 *TARGET* 指定仓库的 URL).

选项

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--show-inherited-props
--verbose (-v)
--xml
```

示例

显示工作副本中中的某个文件上已经设置了哪些属性：

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

加上选项 `--verbose (-v)` 后，*svn proplist* 不仅会打印属性名，还会打印属性值：

```
$ svn proplist -v foo.c
Properties on 'foo.c':
  svn:mime-type
    text/plain
  svn:keywords
    Author Date Rev
  owner
    sally
```

为了列出继承而来的属性，可以加上选项 `--show-inherited-props`:

```
$ svn proplist --show-inherited-props foo.c
Inherited properties on 'foo.c',
from 'http://svn.example.com/repos':
  svn:auto-props
  svn:global-ignores
Inherited properties on 'foo.c',
from '/home/theob/svn/working-copies/baz-wc':
  svn:auto-props
```

最后，为了让 *svn proplist* 以 XML 格式显示输出，可以加上选项 `--xml`:

```
$ svn proplist --xml
<?xml version="1.0"?>
<properties>
<target
   path=".">
<property
   name="svn:ignore"/>
</target>
</properties>
```

名称

svn propset (pset, ps) — 把文件，目录或版本号上的属性 *PROPNAME* 的值设置成 *PROPVAL*.

大纲

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...
```

```
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [TARGET]
```

描述

把文件，目录或版本号上的属性 *PROPNAME* 的值设置成 *PROPVAL*. 命令的第一种形式在工作 副本的项目上设置或修改一个属性；第二种形式为版本号设置或修改一个属性（可选参数 *TARGET* 指定仓库的 URL).

Subversion 预定义了很多 “特殊” 属性，见 “Subversion 的保留属性” 一节.

选项

```
--changelist (--cl) ARG
--depth ARG
--encoding ENC
--file (-F) FILENAME
--force
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--targets FILENAME
```

示例

为一个文件设置 MIME 类型：

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

在 Unix 系统中，如果你希望文件具有可执行权限，可以这样做：

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

为了方便协作，你可能需要设置自定义属性：

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

如果你不小心把一个版本号的日志写错了，现在想把它改回来，可以这样做：

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

如果你没有工作副本，可以使用仓库的 URL：

```
$ svn propset --revprop -r 26 svn:log "Document nap." \
              http://svn.red-bean.com/repos
property 'svn:log' set on repository revision '25'
```

最后，还可以让 *svn propset* 从一个文件中获取属性 值，可以利用这个特性把属性设置成二进制内容：

```
$ svn propset owner-pic -F sally.jpg moo.c
property 'owner-pic' set on 'moo.c'
```

默认情况下，Subversion 禁止用户修改版本号的属性，为了允许修改，仓库管理员必须通过创建钩子脚本 pre-revprop-change 来显式地允许 用户修改版本号的属性. 关于钩子脚本的更多信息，见 “实现仓库钩子”一节.

名称

svn relocate — 修改工作副本所映射的仓库的根 URL.

大纲

```
svn relocate FROM-PREFIX TO-PREFIX [PATH...]

svn relocate TO-URL [PATH]
```

描述

有时候，管理员可能会改变仓库的存放位置，虽然仓库的内容并没有发生 变化，但是它的根 URL 变了．例如由于更换了托管服务器，所以仓库的主机 名发生了变化，又或者说修改了仓库的访问协议，由原来的普通 HTTP (`http://` 改成了 SSL (`https://`). 导致仓库根 URL 发生变化的原因还有很多，但是仓库位置的变化不应该导致 现有的工作副本失效，幸运的是 Subversion 早就考虑到了这一问题．当仓库 的根 URL 发生变化时，用户不必再重新检出一份工作副本，而是可以用命令 *svn relocate* “重写” 工作副本的元数据，使其指向新的仓库根 URL.

命令的第一种形式本质上是在给工作副本中记录的仓库根 URL 做一次 “搜索并替换” 操作．Subversion 把 URL 中的子字符串 *FROM-PREFIX* 替换成 *TO-PREFIX*. 这些 URL 子字符串可以任意地 长或任意地短，只要能够把它们区分出来即可．显然，为了使用这种语法，用户必须同时知道工作副本当前的仓库根 URL 和新的仓库根 URL. (可以使用 命令 *svn info* 得到工作副本当前的仓库根 URL.)

命令的第二种形式只要求用户知道新的仓库根 URL, 但一次只能更新一个 工作副本.

用户常常搞不清楚 *svn switch* 和 *svn relocate* 之间的区别，这里有 2 条经验规则:

- 如果工作副本需要映射到 仓库内 另一个目录 上，就用 *svn switch*.
- 如果工作副本仍然需要映射到相同的仓库目录，但仓库的位置发生了 变化，就用 *svn relocate*.

选项

`--ignore-externals`

示例

假设开始时工作副本是映射到一个本地仓库的 URL:

```
$ svn info | grep ^URL:
URL: file:///var/svn/repos/trunk
$
```

突然有一天，管理员决定重命名本地的仓库目录，但是他忘记把这件事 通知给大家，于是我们在下一次更新工作副本时出错了.

```
$ svn up
```

```
Updating '.':
svn: E180001: Unable to connect to a repository at URL 'file:///var/svn/repos/trunk'
```

向管理员咨询后得知仓库已经更换了位置，并得到了仓库新的 URL. 我们不 想重新检出工作副本，而是让 Subversion 更新现有工作副本的元数据，以便 映射到新的仓库根 URL.

```
$ svn relocate file:///var/svn/new-repos/trunk
$
```

在命令执行期间 Subversion 不会输出什么信息，但我们要的不就是这 种不会出错的命令吗? 现在我们的工作副本已经准备好再次执行 *svn up*.

```
$ svn up
Updating '.':
A    lib/new.c
M    src/code.h
M    src/headers.h
…
```

再次强调，*svn relocate* 只会影响工作副本的 元数据，不会修改任何被版本控制或不被版本控制的文件，更不会执行 版本控制操作（例如提交或更新）.

几个月后，我们又被告知公司将为代码托管设立单独的服务器，开发人员 必须使用 HTTP 访问仓库，于是我们又要更新工作副本的仓库根 URL.

```
$ svn relocate http://svn.company.com/repos/trunk
$
```

我们每次执行 *svn relocate* 时，Subversion 都 会使用新的 URL 与仓库通信，以便确认以下几点：

首先，Subversion 将仓库的 UUID 和存放在工作副本中的仓库 UUID 进行比较，如果这两个 UUID 不相同，Subversion 将禁止修改工作副本的 仓库根 URL. 这样做的考虑是避免工作副本映射到一个不同的仓库上.

第二，Subversion 需要确认元数据更新后的工作副本仍然映射到仓库 内的相同目录上. 这样做的考虑是避免用户不小心把工作副本映射到一个不 同的目录上（这本是 *svn switch* 的工作，见 svn switch (sw)).

另外，Subversion 不允许对工作副本的子目录执行 *svn relocate*，如果你想修改工作副本的仓库根目录，就必须整个 工作副本一起修改. 这是为了保护工作副本元数据和行为的一致性.（而且 在实际工作中，你很难找到一个信服的理由来为工作副本的子目录执行 *svn relocate*）

最后再介绍一个使用 *svn relocate* 的场景. 在使用 了 HTTP 一段时间后，公司决定切换到 SSL. 而仓库 URL 唯一变化的部分是从 `http://` 变成了 `https://`，更新工 作副本的仓库根 URL 的做法有两种，第一种和之前的做法相同，就直接让工作 副本指向新的仓库根 URL:

```
$ svn relocate https://svn.company.com/repos/trunk
$
```

第二种做法是中替换新旧 URL 中不同的部分：

```
$ svn relocate http https
$
```

上面两种做法中的任意一种都能让工作副本指向新的仓库:

默认情况下, *svn relocate* 将会遍历所有的外部 工作副本并更新它们的仓库根 URL, 选项 `--ignore-externals` 将禁止这一行为.

名称

svn resolve — 解决工作副本里的冲突.

大纲

```
svn resolve [PATH...]
```

描述

解决工作副本里的 “冲突” 状态. *svn resolve* 不会从语义上解决冲突标记，而是把发生冲突的项目替换 成指定的版本（交互式地或通过选项 `--accept` 指定），然后再删除与冲突有关的辅助文件. *svn resolve* 将允许 *PATH* 被提交，也就是告诉 Subversion *PATH* 上的冲突已经被 “解决” 了.

关于冲突解决的更多信息，见 “解决冲突”一节.

选项

`--accept ACTION`
`--depth ARG`
`--quiet (-q)`
`--recursive (-R)`
`--targets FILENAME`

示例

下面的例子推迟了冲突的解决，然后 *svn resolve* 用用户的内容替换掉所有的冲突：

```
$ svn update
Updating '.':
Conflict discovered in 'foo.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: p
C    foo.c
Updated to revision 5.
Summary of conflicts:
  Text conflicts: 1
$ svn resolve --accept mine-full foo.c
Resolved conflicted state of 'foo.c'
$
```

名称

svn resolved — 不再推荐使用. 删除工作副本项目上的 “冲突” 状态.

大纲

```
svn resolved PATH...
```

描述

已不再推荐使用该命令, 而应该使用 `svn resolve --accept working PATH`. 关于 *svn resolve*, 见前面的 svn resolve.

svn resolved 删除工作副本项目上的 “冲突” 状态. 命令不会从语义上解决冲突标记, 它仅仅是删除 与冲突有关的辅助文件, 并允许 *PATH* 被提交 到仓库中, 也就是告诉 Subversion 冲突已经被 “解决了”. 关于冲突解决的更多信息, 见 “解决冲突” 一节.

选项

```
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

示例

如果你在更新 *foo.c* 时发生了冲突, 工作副本里 将会出现 3 个新文件:

```
$ svn update
Updating '.':
C    foo.c
Updated to revision 31.
Summary of conflicts:
  Text conflicts: 1
$ ls foo.c*
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
$
```

一旦你已经解决了冲突, 文件 *foo.c* 就已经 准备好提交, 执行 *svn resolved* 告诉 Subversion 你 已经处理好了所有问题.

用户 可以 直接删除与冲突有关的辅助文件, 然后提交, 但是 *svn resolved* 还会在工作副本管理区 内更新一些薄记数据, 所以我们建议用户使用命令, 而不是手工操作.

名称

svn revert — 撤消所有的本地修改.

大纲

```
svn revert PATH...
```

描述

撤消文件或目录上的本地修改，并解决可能存在的冲突状态. *svn revert* 不仅撤消内容上的修改，还会撤消属性上的修改. 最后，*svn revert* 还能撤消被添加，删除或移动（但还未提交）的项目.

选项

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

示例

撤消文件上的所有修改:

```
$ svn revert foo.c
Reverted foo.c
```

如果你希望递归地撤消一个目录内的所有修改，就加上选项 `--depth=infinity`:

```
$ svn revert --depth=infinity .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

最后，你还可以撤消已添加的项目:

```
$ 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
```

使用 *svn revert* 具有一定的危险性，因为它会 丢弃所有的本地修改，也就是用户未提交的修改. 这些本地修改一旦被撤消 了，Subversion 没有任何办法 再把它们恢复回来.

如果用户没有为 *svn revert* 指定任何目标路径，它将什么也不会做，这是为了避免用户不小心丢失自己的本地修改，因此 *svn revert* 要求用户必须显式地指定目标路径.

名称

svn status (stat, st) — 打印工作副本中文件或目录的状态.

大纲

```
svn status [PATH...]
```

描述

打印工作副本中文件或目录的状态. 如果没有指定任何选项, *svn status* 将只打印本地已修改的项目（不会访问仓库）. 如果添加了 选项 `--show-updates` (`-u`), 命令还会 打印与服务器上的最新数据相比, 工作副本中已过时的项目及其版本号. 如果添加了选项 `--verbose` (`-v`), 它将打印每一项的 完整的版本号信息. 如果添加了选项 `--quiet` (`-q`), 它就只打印与本地已修改项目有关的总结信息.

输出中每一行的前 7 列包含了描述项目状态的字符, 每一列都从不同 的角度描述项目的状态.

第一列指出项目是新增的, 被删除的, 还是已修改的:

`' '`

没有任何变化.

`'A'`

该项是新增的.

`'D'`

该项已被删除.

`'M'`

该项已被修改.

`'R'`

该项被工作副本中的其他项目替换了. 这意味着原来的项目已被删除, 然后在原来的位置上添加了一个新项目.

`'C'`

项目在工作副本中的内容（不包含属性）和从服务器上接收到的 更新有冲突.

`'X'`

项目是通过外部定义创建的.

`'I'`

项目是被忽略的（例如由于属性 `svn:ignore`).

'?'

项目不被版本控制.

'!'

项目失踪了（例如用户没有使用 *svn* 提供的 命令来移动或删除项目）. 这同时也意味着目录是不完整的（例如检出或 更新操作被中断了）.

'~'

项目是作为一种文件类型（例如普通文件，目录文件或符号链接）存放到仓库中，但是现在工作副本中的项目已经被另一种文件类型所 取代.

第二列指出文件或目录的属性的状态：

' '

没有变化.

'M'

属性被修改了.

'C'

属性的修改含有冲突.

第三列指出工作副本目录是否被锁定了（见 “有时候你需要的只是清理一下”一节）：

' '

项目未被锁定.

'L'

项目已被锁定.

第四列指出被添加的项目是否含有历史：

' '

被添加的项目不含有历史.

'+'

被添加的项目含有历史.

第五列指出项目是否相对于父目录进行了切换（见 “遍历分支”一节）：

' '

项目是父目录的子文件.

'S'

项目被切换过了.

第六列指出了关于锁的信息:

' '

如果指定了选项 --show-updates (-u), 则说明文件未被锁定. 如果没有指定选项 --show-updates (-u), 则说明文件在工作副本中未被锁定.

'K'

文件在工作副本中被锁定.

'O'

文件被另一个用户或者在另一个工作副本中被锁定, 仅当指定选项 --show-updates (-u) 时才会出 现该状态.

'T'

文件在工作副本中被锁定, 但是锁已经被 “窃取”, 并 且不再有效. 文件当前在仓库中是处于被锁定的状态. 只有指定选项 --show-updates (-u) 时才会显示 该状态.

'B'

文件在工作副本中被锁定, 但是锁已经被 “破坏” 了, 并且不再有效. 文件当前在仓库中未被锁定. 只有指定选项 --show-updates (-u) 时才会显示 该状态.

第七列指出项目是否是目录冲突的受害者:

' '

项目不是目录冲突的受害者.

'C'

项目是目录冲突的受害者.

第八列总是一个空格.

第九列指出仓库中是否有更新的版本 (只有指定选项 --show-updates (-u) 时才会显示该 状态):

' '

仓库中没有更新的版本.

'*'

仓库中有更新的版本.

剩下的字段长度不一，字段之间用空格分开. 如果指定了选项 `--show-updates (-u)` 或 `--verbose (-v)`, 则下一个字段是 工作版本号.

如果指定了选项 `--verbose (-v)`, 则下一个字段是最后一次提交的版本号和作者.

因为项目的工作副本路径总是最后一个字段，因此路径内可以包含空格.

选项

`--changelist (--cl) ARG`
`--depth ARG`
`--ignore-externals`
`--incremental`
`--no-ignore`
`--quiet (-q)`
`--show-updates (-u)`
`--verbose (-v)`
`--xml`

示例

如果你想检查工作副本包含了哪些修改，最方便的做法是:

```
$ svn status wc
 M      wc/bar.c
A  +    wc/qax.c
```

如果你想查看工作副本中的哪些文件已经过时了，就加上选项 `--show-updates (-u)` (这个选项 不会 对工作副本作出任何修改). 下面的例子说明 仓库中的 *wc/foo.c* 比工作副本中的更新:

```
$ svn status -u wc
 M            965    wc/bar.c
        *     965    wc/foo.c
A  +          965    wc/qax.c
Status against revision:    981
```

选项 `--show-updates (-u)` 只会 在项目旁边打印一个星号 (意思是说仓库中 有更新的版本). 选项 `--show-updates (-u)` 不会 打印该项目在仓库中 的版本号 (但你仍然可以用选项 `--verbose (-v)` 查看项目在仓库中的版本号).

在最详细的情况下，*svn status* 打印的信息如下:

```
$ svn status -u -v 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
Status against revision:   981
```

最后，如果指定了选项 `--xml`, 则 *svn status* 将以 XML 格式打印输出:

```
$ 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，见 “查看修改的整体概述”一节.

名称

svn switch (sw) — 将工作副本更新到另一个 URL.

大纲

```
svn switch URL[@PEGREV] [PATH]

svn switch --relocate FROM TO [PATH...]
```

描述

命令的第一种形式（不带有选项 `--relocate`）把工作 副本更新到一个新的 URL. 这是 Subversion 提供的，用于让工作副本跟踪一 个新分支的方式. 如果指定了 *PEGREV*，则 *PEGREV* 决定了 Subversion 在哪个版本号内 查找目标路径. 关于分支切换的更多信息，见 “遍历分支”一节.

从 Subversion 1.7 开始，*svn switch* 要求新 的 URL 必须和工作副本当前的 URL 具有相同的祖先. 为了忽略这一要求，可以指定选项 `--ignore-ancestry`.

如果指定了选项 `--force`，那么 *svn switch* 在添加一个新路径时，如果该路径在切换前是一个未 被版本控制的路径，将不会产生一个错误. 如果未被版本控制的路径和 切换后的新路径类型相同（文件或目录），则路径将被纳入版本控制，但原来的内容保持不变，这也意味着目录的子文件也可能被纳入版本控制. 对于文件 来说，不同的部分将被当作本地修改. 来自仓库的所有属性都会被应用到路径 上.

和大多数子命令一样，用户可以通过选项 `--depth` 限制 *svn switch* 的作用深度. 相应地，用户还可以用 选项 `--set-depth` 为目标路径设置新的 “粘着” 深度.

从 Subversion 1.7 开始，选项 `--relocate` 已不再 推荐使用，而应直接使用 *svn relocate*（见 svn relocate）完成工作副本仓库根 URL 的 更新.

选项

```
--accept ACTION
--depth ARG
--diff3-cmd CMD
--force
--ignore-ancestry
--ignore-externals
--quiet (-q)
--relocate
--revision (-r) REV
--set-depth ARG
```

示例

如果你现在正在目录 *vendors* 内，而现在你想 让该目录指向分支 *vendors-with-fix*:

```
$ 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.
```

工作完成后，可以再切换回分支 *vendors*:

```
$ 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.
```

用户 可以 只切换工作副本的一部分到一个 新的分支，但这不是推荐的做法. 因为很容易忘记工作副本只是部分切换 的，然后用户意外地同时修改并提交到已切换和未切换的目录树中.

名称

svn unlock — 解锁工作副本路径或 URL.

大纲

`svn unlock` *`TARGET`*`...`

描述

解锁每一个 *`TARGET`*. 如果有任意一个 *`TARGET`* 被其他用户加锁了，或者在工作副本 中不存在有效的锁令牌，Subversion 就会打印一个警告，然后继续解锁剩下 的 *`TARGET`*. 为了破坏属于其他用户或工作副本 的锁，可以加上选项 `--force`.

选项

`--force`
`--targets` *`FILENAME`*

示例

解锁工作副本中的两个文件：

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

解锁工作副本里的一个文件，但是该文件已经被其他用户加锁了：

```
$ svn unlock tree.jpg
svn: E195013: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

在没有工作副本的情况下解锁一个文件：

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

更多的内容，见 “锁”一节.

名称

svn update (up) — 更新工作副本.

大纲

```
svn update [PATH...]
```

描述

svn update 把仓库里的更新应用到工作副本里. 如果没有指定版本号, 则默认是 HEAD, 否则的话, *svn update* 把工作副本更新到由选项 `--revision (-r)` 所指定的版本号. 作为更新操作的一部分, *svn update* 还会删除工作副本 中已过时的锁（见“有时候你需要的只是清理一下”一节).

对每一个被更新的项目, Subversion 都会打印一行信息, 信息的开头是 一个字符, 用于表示对项目所采取的动作, 这些字符有:

A

新增

B

被破坏的锁（只出现在第 3 列）

D

被删除

U

被更新

C

发生冲突

G

被合并

E

已存在

出现在第一列的状态字符描述的是文件内容的更新, 出现在第二列的 状态字符描述的是文件属性的更新, 出现在第三列的状态字符则描述的是 关于锁的信息.

和大多数子命令一样, 你可以通过选项 `--depth`, 把更新操作的作用范围限制在一定的范围内. 同样地, 还可以用选项 `--set-depth` 为目标路径设置新的 “粘着” 深度.

选项

`--accept ACTION`
`--changelist (--cl) ARG`
`--depth ARG`
`--diff3-cmd CMD`
`--editor-cmd CMD`
`--force`
`--ignore-externals`
`--parents`
`--quiet (-q)`
`--revision (-r) REV`
`--set-depth ARG`

示例

把仓库中最新的内容更新到本地:

```
$ svn update
Updating '.':
A    newdir/toggle.c
A    newdir/disclose.c
A    newdir/launch.c
D    newdir/README
Updated to revision 32.
```

还可以把工作副本 “更新” 一个更旧的版本 (与 CVS 相比, Subversion 没有 “粘着” 文件的概念, 见 附录 B, *针对 CVS 用户的 Subversion 介绍*):

```
$ svn update -r30
Updating '.':
A    newdir/README
D    newdir/toggle.c
D    newdir/disclose.c
D    newdir/launch.c
U    foo.c
Updated to revision 30.
```

如果你只是想看看某个文件在老版本时的内容, 更方便的做法是使用 命令 *svn cat*—它不会修改工作副本.

svn update 还是配置稀疏工作副本的主要命令. 如果指定了选项 `--set-depth`, 命令将会把工作副本路径 的周围深度修改成用户指定的深度, 从而忽略或延伸工作副本路径 (必要时 将从仓库中抓取数据). 关于稀疏目录的更多信息, 见 “稀疏目录” 一节.

用户可以只调用一次 *svn update* 就能更新多个 目录, Subversion 不仅会忽略不被版本控制的目标, 如果是 Subversion 1.7, 它还会在最后打印一段总结信息:

```
$ cd my-projects
$ svn update *
Updating 'calc':
U    button.c
U    integer.c
Updated to revision 394.
Skipped 'tempfile.tmp'
Updating 'paint':
A    palettes.c
U    brushes.c
Updated to revision 60.
Updating 'ziptastic':
At revision 43.
Summary of updates:
  Updated 'calc' to r394.
  Updated 'paint' to r60.
  Updated 'ziptastic' to r43.
Summary of conflicts:
  Skipped paths: 1
$
```

名称

svn upgrade — 更新工作副本的元数据格式.

大纲

```
svn upgrade [PATH...]
```

描述

随着新版 Subversion 的发布，为了适应新版所添加的特性或为了修复 问题，工作副本的元数据格式可能会发生变化. 在 Subversion 1.7 之前，新版 Subversion 在首次使用旧版 Subversion 所创建的工作副本时，将会 自动升级工作副本的元数据格式. 但是从 Subversion 1.7 开始，用户必须 通过执行命令 *svn upgrade* 来显式地请求 Subversion 去更新工作副本的元数据格式.

选项

`--quiet (-q)`

示例

如果你试图在一个版本较旧的工作副本中使用 Subversion 1.7, Subversion 将会报错:

```
$ svn status
svn: E155036: Please see the 'svn upgrade' command
svn: E155036: Working copy '/home/sally/project' is too old (format 10, create
d by Subversion 1.6)
$
```

这时候就需要执行 *svn upgrade* 把工作副本的元数据 格式升级到当前 Subversion 所支持的格式:

```
$ svn upgrade
Upgraded '.'
Upgraded 'A'
Upgraded 'A/B'
Upgraded 'A/B/E'
Upgraded 'A/B/F'
Upgraded 'A/C'
Upgraded 'A/D'
Upgraded 'A/D/G'
Upgraded 'A/D/H'
$ svn status
D       A/B/E/alpha
M       A/D/gamma
A       A/newfile
$
```

注意，*svn upgrade* 会保留工作副本的本地修改，即使这些修改是由旧版 Subversion 引入的（例如 *svn mv*）.

和过去自动升级工作副本元数据的行为相比，显式地升级工作副本 元数据后，旧的 Subversion 将不再支持升级后的工作副本.

svnadmin 参考手册—Subversion 仓库管理工具

svnadmin 是监控和修复 Subversion 仓库的管理工具. 关 于仓库管理的更多细节, 见 “svnadmin”一节.

由于 *svnadmin* 只有直接访问仓库才能工作 (因 此它只能用在存放了仓库的主机中), 所以说在指定仓库时只能使用文件系统路径, 而不是 URL.

svnadmin 的选项都是全局的, 就像 *svn* 的全局选项:

svnadmin 选项

`--bdb-log-keep`

(特定于 Berkeley DB 的选项.) 禁止自动删除数据库的日志文件. 保留这些日志文件有助于从灾难性的仓库失败中恢复数据.

`--bdb-txn-nosync`

(特定于 Berkeley DB 的选项.) 在提交数据库事务时禁止调用 `fsync`. 该选项可以让 *svnadmin create* 创建一个开启了 `DB_TXN_NOSYNC`, 以 Berkeley DB 作为后端存储的仓库 (该选项可以提升性能, 但会带 来一定的风险).

`--bypass-hooks`

旁路掉仓库的钩子脚本.

`--bypass-prop-validation`

加载一个转储文件时, 禁止对属性值进行检查.

`--clean-logs`

删除无用的 Berkeley DB 日志.

`--compatible-version` *`ARG`*

使用与版本为 *`ARG`* 的 Subversion 兼容的仓库格式.

`--config-dir` *`DIR`*

告诉 Subversion 从指定的目录内读取配置信息, 而不是从默认的目录 (用户家目录中的 *.subversion*) 中读取.

`--deltas`

在创建仓库的转储文件时, 把属性和文件内容上的修改指定成相对于 前一状态的修改.

`--file` (`-F`) *`FILENAME`*

为指定的子命令使用文件中的内容.

`--fs-type` *ARG*

创建一个仓库时，使用 *ARG* 指定的文件 系统类型，*ARG* 可以是 `bdb` 或 `fsfs`.

`--force-uuid`

默认情况下，为一个已经包含了版本号的仓库加载转储数据时，*svnadmin* 会忽略转储数据中的 UUID. 该选项使得 仓库的 UUID 被设置成转储数据中的 UUID.

`--ignore-uuid`

默认情况下，往一个空仓库加载转储数据时，*svnadmin* 会把仓库的 UUID 设置成转储数据中的 UUID. 该选项使得 *svnadmin* 忽略转储数据中的 UUID.

`--incremental`

按照增量格式转储版本号.

`--memory-cache-size` (`-M`) *ARG*

配置额外的内存缓存大小（以 MB 为单位），这种缓存可以用来减少 重复操作，但只能用于以 FSFS 作为后端存储的仓库. 默认值是 16.

`--parent-dir` *DIR*

加载一个转储文件时，把根目录设置成 *DIR*，而不是默认的 /.

`--pre-1.4-compatible`

不再推荐使用. 见选项 `--compatible-version`. 当创建一个新的仓库时，仓库的格式要和 Subversion 1.4 之前的版本保持兼容.

`--pre-1.5-compatible`

不再推荐使用. 见选项 `--compatible-version`. 当创建一个新的仓库时，仓库的格式要和 Subversion 1.5 之前的版本保持兼容.

`--pre-1.6-compatible`

不再推荐使用. 见选项 `--compatible-version`. 当创建一个新的仓库时，仓库的格式要和 Subversion 1.6 之前的版本保持兼容.

`--revision` (`-r`) *ARG*

为后面的操作指定一个版本号.

`--quiet` (`-q`)

不要显示正常的输出—只显示与错误有关的输出.

`--use-post-commit-hook`

加载一个转储文件时，每完成一个版本号的提交，就执行一次仓库 的 post-commit 钩子脚本.

`--use-post-revprop-change-hook`

修改一个版本号属性时，在修改完后执行仓库的 post-revprop-change 钩子脚本.

`--use-pre-commit-hook`

加载一个转储文件时，在提交版本号之前，执行仓库的 pre-commit 钩子脚本，如果脚本返回错误，则中止提交，并结束整个加载 过程.

`--use-pre-revprop-change-hook`

在修改一个版本号属性之前，执行仓库的 pre-revprop-change 钩子脚本，如果脚本返回错误，就中止修改操作并结束.

`--wait`

对于那些需要互斥访问仓库的操作来说，如果仓库锁已经被他人获取，则等待别人放锁，而不是马上报错退出.

目录

名称

svnadmin crashtest — 模拟一个崩溃的进程.

大纲

`svnadmin crashtest` *`REPOS_PATH`*

描述

打开路径为 *`REPOS_PATH`* 的仓库，然后模拟进程在持有一个打开的仓库句柄时崩溃. 该命令用于测试仓库的自动 复原 (Berkeley DB 4.4 引入的新特性). 用户应该不会用到该命令.

选项

无

示例

```
$ svnadmin crashtest /var/svn/repos
Aborted
```

很刺激，是不是?

名称

svnadmin create — 创建一个新的空仓库.

大纲

```
svnadmin create REPOS_PATH
```

描述

在指定的路径下创建一个新的空仓库，如果目录不存在，*svnadmin* 就会创建该目录. [1] 从 Subversion 1.2 开始，*svnadmin* 默认使用 FSFS 作为新仓库的后端存储.

虽然 *svnadmin create* 会创建新仓库所在的目录，但它并不会创建中间目录. 例如，假设你有一个空目录 */var/svn*，那么创建仓库 */var/svn/repos* 不会有任何问题，但是试图创建 */var/svn/subdirectory/repos* 则会报错. 另外，取决于仓库的位置，你可能需要切换到具有特权的用户（例如超级用户 `root`）才能执行 *svnadmin create*.

选项

```
--bdb-log-keep
--bdb-txn-nosync
--compatible-version ARG
--config-dir DIR
--fs-type ARG
--pre-1.4-compatible
--pre-1.5-compatible
--pre-1.6-compatible
```

示例

在 */var/svn* 目录下创建一个名为 *repos* 的新仓库：

```
$ cd /var/svn
$ svnadmin create repos
$
```

Subversion 1.0 总是使用 Berkeley DB 作为仓库的后端存储系统. Subversion 1.1 把 Berkeley DB 作为默认的仓库后端存储，但可以用选项 `--fs-type` 把后端存储系统改成 FSFS:

```
$ cd /var/svn
$ svnadmin create repos --fs-type fsfs
$
```

[1] 记住，*svnadmin* 只接受本地路径，不支持 *URL*.

名称

svnadmin deltify — 对一个版本号范围内的，被修改过的路径执行 deltify [1].

大纲

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

描述

svnadmin deltify 还留在 Subversion 里只是出 于历史原因，该命令已经不再推荐使用，也不会再有地方会用到它.

svnadmin deltify 的历史可以追溯到 Subversion 为管理员提供仓库压缩策略的时期，命令的执行过程非常复杂，但得到的效果却 很有限，如今这个命令已不再推荐使用.

选项

```
--memory-cache-size (-M) ARG
--quiet (-q)
--revision (-r) ARG
```


[1]还没想到合适的中文翻译 -- 译者

名称

svnadmin dump — 把文件系统的内容转储到 *stdout*.

大纲

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental] [--deltas]
```

描述

按照可移植的 “转储文件” 格式，把文件系统的内容转储 到 *stdout*，把错误信息打印到 *stderr*. 按照从版本号 *LOWER* 到 *UPPER* 的顺序进行转储. 如果没有指定版本号，*svnadmin dump* 将会转储所有的版本号，如果只指定了版本号 *LOWER*，则只转储单个版本号. 关于 *svnadmin dump* 的实际用途，见 “迁移仓库数据”一节.

默认情况下，转储流的第一个版本号（即指定版本号范围的起始版本号）相当于把该版本号下的所有文件和目录一次性添加到仓库中，后面的版本号（起始版本号之后的版本号）只包含了在版本号内被修改的文件和目录. 对于 被修改的文件，转储流保存了该文件的全文本内容和属性；对于被修改的目录，转储流保存了该目录的所有属性.

有 2 个选项可以修改 *svnadmin* 在生成转储文件时 的行为. 选项 `--incremental` 使得转储流的第一个版本号 只包含在该版本号内被修改的文件和目录，而不是该版本号下的整个目录树，联系上一段可以看到，这和后面的版本号在转储流中的表示格式是完全相同的. 如果目标仓库已经包含了源仓库的文件与目录，那么该选项就能生成相对更小 的转储文件.

第二个选项是 `--deltas`，它使得 *svnadmin dump* 在转储每个版本号时，只输出该版本号相对于前一个版本号 的差异部分，而不是全文本的文件和属性. 这种做法可以减小（在某些情况下 可以极大地减小）转储文件的大小. 然而，选项 `--deltas` 也有不好的地方—它会耗费更多的 CPU 资源，而且使用第三方工具（例如 *gzip* 和 *bzip2*）对转储文件进行 压缩时，和不加选项 `--deltas` 生成的转储文件相比，其 压缩效果也不如它们.

从 Subversion 1.8 开始，*svndumpfilter* 开始支持通过选项 `--deltas` 创建的转储流.

选项

```
--deltas
--incremental
--memory-cache-size (-M) ARG
--quiet (-q)
--revision (-r) ARG
```

示例

转储整个仓库：

```
$ svnadmin dump /var/svn/repos > full.dump
* Dumped revision 0.
* Dumped revision 1.
```

```
* Dumped revision 2.
…
```

按照增量的格式转储一个单独的版本号:

```
$ svnadmin dump /var/svn/repos -r 21 --incremental > incr.dump
* Dumped revision 21.
```

名称

svnadmin freeze — 在执行某个程序期间，禁止向仓库提交.

大纲

```
svnadmin freeze REPOS_PATH PROGRAM [ARG...]
```

```
svnadmin freeze --file FILENAME PROGRAM [ARG...]
```

描述

svnadmin freeze 禁止在程序 *PROGRAM* (带有参数 *ARG*) 运行期间，向仓库 *REPOS_PATH* 提交修改 (即冻结仓库). 如果客户端在仓库冻结期间向仓库提交修改，提交将会阻塞，直到仓库解除冻结. *svnadmin freeze* 的目的是让第三方备份工具 (例如 *rsync*) 可以安全地对在线仓库进行备份.

如果添加了选项 `--file`, 文件 *FILENAME* 中列出的所有仓库都会被冻结. 文件 的格式是每行一个 *REPOS_PATH*, 仓库被冻结的 顺序和它们在文件中的顺序相同.

选项

`--file (-F) FILENAME`

示例

冻结仓库，然后执行 *rsync* 对仓库进行备份:

```
$ svnadmin freeze /var/svn/repos -- rsync -av /var/svn/repos /backup/repos
```

名称

svnadmin help (h, ?) — 帮助!

大纲

```
svnadmin help [SUBCOMMAND...]
```

描述

如果既无法上网, 手上也没有这本书, 那你就只能依靠这个子命令了.

名称

svnadmin hotcopy — 对仓库进行在线备份.

大纲

`svnadmin hotcopy` *`REPOS_PATH NEW_REPOS_PATH`*

描述

svnadmin hotcopy 为仓库生成一份 “在线” 副本，包括所有的钩子，配置文件和数据库文件. 你可以 在任何时候执行这个命令，而不用担心是否还有其他进程在访问仓库.

在 Subversion 1.8 之前，*svnadmin hotcopy* 总是 生成一份完整的仓库在线副本. 从 Subversion 1.8 开始，*svnadmin hotcopy* 开始支持增量备份. 添加选项 `--incremental` 后，Subversion 将只复制新的版本号和 大小或时间戳发生变化的版本号. 目标仓库的 UUID 和源仓库的 UUID 必须 完全相同. 增量的在线备份只支持使用 FSFS 作为后端存储的仓库.

如果添加了选项 `--clean-logs`, *svnadmin hotcopy* 在执行完在线备份后，将会删除源仓库中不再使用的 Berkeley DB 日志.

选项

`--clean-logs`
`--incremental`

在 “体系结构上的限制”一节 提到，在线备份的，以 Berkeley DB 作为后端存储的仓库在不同的操作系统之间 是不可移植的，在大小端不同的系统之间也不可移植.

名称

svnadmin list-dblogs — 列出所有的 Berkeley DB 日志文件.

大纲

`svnadmin list-dblogs` *`REPOS_PATH`*

描述

Berkeley DB 为仓库的所有修改创建日志，用于灾备恢复. 除非开启了 `DB_LOG_AUTOREMOVE`，否则的话日志文件会不断累积，即使它们中的大部分都不会再被用到，把它们删除有助于节省硬盘空间. 更 多的信息见 “管理磁盘空间”一节.

名称

svnadmin list-unused-dblogs — 询问 Berkeley DB 哪些日志文件可以被安全地删除（该子命令只 适用于以 bdb 作为后端存储的仓库）.

大纲

```
svnadmin list-unused-dblogs REPOS_PATH
```

描述

Berkeley DB 为仓库的所有修改创建日志，用于灾备恢复. 除非开启了 DB_LOG_AUTOREMOVE，否则的话日志文件会不断累积，即使它们中的大部分都不会再被用到，把它们删除有助于节省硬盘空间. 更 多的信息见 “管理磁盘空间”一节.

示例

删除仓库中不会再被用到的日志文件：

```
$ 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 — 从 *stdin* 读取仓库的转储流.

大纲

`svnadmin load` *`REPOS_PATH`* `[-r` *`LOWER`*`[:`*`UPPER`*`]]`

描述

从 *stdin* 读取仓库的转储流，把新的版本号提交 到仓库的文件系统中. 进度信息被打印到 *stdout*. 如果没有指定版本号，*svnadmin load* 将读取并提交所有 的版本号；如果添加了选项 `--revision` (`-r`), *svnadmin load* 将只读取并 提交从 *`LOWER`* 到 *`UPPER`* 的版本号；如果只指定了 *`LOWER`*, 则只加载这一个版本号.

在 Subversion 1.8 之前，*svnadmin load* 只能加 载转储流中包含的 全部 版本号，但是现在可以通过 选项 `--revision` (`-r`), 从转储流中 加载指定的版本号. 这就允许管理员从一个单一的转储流中增量地加载一段 版本号范围，从而让仓库的维护任务变得更加轻松.

选项

`--bypass-prop-validation`
`--force-uuid`
`--ignore-uuid`
`--memory-cache-size (-M)` *`ARG`*
`--parent-dir` *`DIR`*
`--quiet (-q)`
`--revision (-r)` *`ARG`*
`--use-post-commit-hook`
`--use-pre-commit-hook`

示例

下面的例子展示了把转储文件加载到仓库中的输出信息的开始部分（当然，转储文件是通过命令 *svnadmin dump* 创建的）：

```
$ svnadmin load /var/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

还可以把转储流加载到一个子目录内：

```
$ 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.
```

...

新版 Subversion 对内建属性的值会提出越来越严格的格式要求，当然，由旧版 Subversion 创建的属性无法从这种严格的要求中得到任何好处，却 有可能导致属性值被不正确地格式化. 转储流原模原样地保存属性值，因此 Subversion 1.8 在加载属性值格式不正确的转储流时，默认情况下会触发一 个验证性错误. 有几种办法可以避免出现这种问题，第一种是在源仓库中手动 地修正格式不正确的属性值，然后重新创建转储流；或者是手动修改转储流文件 中的属性值；最后，如果你想在加载后再去修正属性值，可以为 *svnadmin load* 添加选项 `--bypass-prop-validation`.

名称

svnadmin lock — 直接在仓库中为一个路径加锁.

大纲

`svnadmin lock` *`REPOS_PATH PATH-IN-REPOS USERNAME FILE`* [*`TOKEN`*]

描述

加载仓库中的路径 *`PATH-IN-REPOS`*, 把锁的持有者设置为 *`USERNAME`*, 使用文件 *`FILE`* 的内容作为锁的注释. 如果提供了 *`TOKEN`*, 它将作为锁的令牌.

选项

`--bypass-hooks`

名称

svnadmin lslocks — 打印所有锁的描述.

大纲

`svnadmin lslocks` *`REPOS_PATH`* [*`PATH-IN-REPOS`*]

描述

打印仓库 *`REPOS_PATH`* 在路径 *`PATH-IN-REPOS`* 下的所有锁的描述. 如果没有 指定 *`PATH-IN-REPOS`*, 默认是仓库的根目录.

选项

无

示例

列出仓库 */var/svn/repos* 所有的锁 (只有一个):

```
$ 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 — 输出所有未提交的事务的名字.

大纲

`svnadmin lstxns REPOS_PATH`

描述

输出所有未提交的事务的名字. 关于未提交的事务是如何出现的, 以及 如何处理它们, 见 “删除僵死的事务”一节.

示例

列出仓库中所有未完成的事务:

```
$ svnadmin lstxns /var/svn/repos/
1w
1x
```

名称

svnadmin pack — 如果有可能的话，把仓库压缩成一个效率更高的存储模式.

大纲

`svnadmin pack` *`REPOS_PATH`*

描述

见 “FSFS 文件系统压缩” 一节.

选项

无

名称

svnadmin recover — 把仓库数据库恢复到一个一致的状态（该命令只适用于以 bdb 作为后端存储的仓库）. 另外，如果 *repos/conf/passwd* 不存在，命令将自动创建一个默认 的密码文件.

大纲

`svnadmin recover` *`REPOS_PATH`*

描述

如果你得到一条说明仓库需要复原的错误信息，就执行这个命令.

选项

`--wait`

示例

复原一个挂起的仓库：

```
$ 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.
```

复原数据库要求获得仓库的排斥性锁.（这是一个 “数据库锁”，见 “锁” 的多种涵义）如果有其他进程正在访问 仓库，*svnadmin recover* 将会报错：

```
$ svnadmin recover /var/svn/repos
svn: E165000: Failed to get exclusive repository access; perhaps another proce
ss such as httpd, svnserve or svn has it open?
$
```

选项 `--wait` 使得 *svnadmin recover* 会一直等到其他进程不再访问仓库为止：

```
$ 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 — 无条件地从仓库中移除一个或多个锁.

大纲

`svnadmin rmlocks` *`REPOS_PATH LOCKED_PATH...`*

描述

从仓库中删除一个或多个锁.

选项

无

示例

下面的例子把仓库 */var/svn/repos* 的文件 *tree.jpg* 和 *house.jpg* 上 的锁删除:

```
$ svnadmin rmlocks /var/svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

名称

svnadmin rmtxns — 从仓库中删除事务.

大纲

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

Description

从仓库中删除未完成的事务，更多的细节见 “删除僵死的事务”一节.

选项

`--quiet (-q)`

示例

删除指定的事务：

```
$ svnadmin rmtxns /var/svn/repos/ 1w 1x
```

方便的是，我们可以把 *lstxns* 的输出用作 *rmtxns* 的输入：

```
$ svnadmin rmtxns /var/svn/repos/  `svnadmin lstxns /var/svn/repos/`
```

上面的例子删除了仓库中所有未提交的事务.

名称

svnadmin setlog — 设置版本号的日志消息.

大纲

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

描述

把版本号 *REVISION* 的日志消息设置成 文件 *FILE* 的内容.

这个命令类似于用带有选项 `--revprop` 的命令 *svn propset* 设置版本号的 `svn:log` 属性，不同点是在使用 *svnadmin setlog* 可以利用选项 `--bypass-hooks` 旁路掉 pre-commit 或 post-commit 钩子 脚本，这对于没有在钩子 pre-revprop-change 中开启版本号修改的情景非常 方便.

Subversion 不会对版本号属性进行版本控制，所以 *svnadmin setlog* 会永久性地覆盖掉之前的日志消息.

选项

```
--bypass-hooks
--revision (-r) ARG
```

示例

把版本号 19 的日志消息设置成文件 *msg* 的 内容:

```
$ svnadmin setlog /var/svn/repos/ -r 19 msg
```

名称

svnadmin setrevprop — 在版本号上设置一个属性.

大纲

```
svnadmin setrevprop REPOS_PATH -r REVISION NAME FILE
```

描述

把版本号 *REVISION* 的属性 *NAME* 的值设置成文件 *FILE* 的内容. 如果想要触发与版本号属性相关的 钩子脚本 (例如在钩子 post-revprop-change 里发送一封邮件, 通知大家 有属性被修改了).

选项

```
--revision (-r) ARG
--use-post-revprop-change-hook
--use-pre-revprop-change-hook
```

示例

下面的例子把版本号属性 `repository-photo` 的值设置 成文件 *sandwich.png* 的内容:

```
$ svnadmin setrevprop /var/svn/repos -r 0 repository-photo sandwich.png
```

可以看到, *svnadmin setrevprop* 在执行成功时不会 输出任何信息.

名称

svnadmin setuuid — 重置仓库的 UUID.

大纲

`svnadmin setuuid REPOS_PATH [NEW_UUID]`

描述

重置仓库 *REPOS_PATH* 的 UUID. 如果提供了 *NEW_UUID*, 它将会是仓库的新 UUID; 否则的话, Subversion 将自动为仓库生成一个新的 UUID.

选项

无

示例

如果管理员已经用 *svnsyn* 把 */var/svn/repos* 备份到了 */var/svn/repos-new*, 现在想用 *repos-new* 作为新的官方仓库, 为了不麻烦用户再重新 检出新的工作副本, 管理员需要把 *repos-new* 的 UUID 设置成 *repos* 的 UUID:

```
$ svnadmin setuuid /var/svn/repos-new 2109a8dd-854f-0410-ad31-d604008985ab
```

可以看到, *svnadmin setuuid* 在执行成功时不会 输出任何信息.

名称

svnadmin unlock — 直接在仓库中解锁一个路径.

大纲

`svnadmin unlock` *`REPOS_PATH LOCKED_PATH USERNAME TOKEN`*

描述

如果锁令牌与 *`TOKEN`* 相匹配, *svnadmin* 将以用户 *`USERNAME`* 的身份解锁 *`LOCKED_PATH`*.

选项

`--bypass-hooks`

名称

svnadmin upgrade — 把仓库升级到 Subversion 支持的最新格式.

大纲

```
svnadmin upgrade REPOS_PATH
```

描述

把仓库 *REPOS_PATH* 升级到 Subversion 支持的最新格式.

命令的目标是为了让管理员方便地对仓库进行升级，而不用通过麻烦的 转储和加载. 为了完成升级，*svnadmin upgrade* 只会 完成最小量的工作，同时保证仓库的完整性. 虽然转储和随后的加载操作可以 保证得到一个处于最佳状态的仓库，但 *svnadmin upgrade* 却无法保证这点.

管理员在升级仓库前，应该先对它进行备份.

选项

无

示例

升级仓库 */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 — 验证仓库里的数据的完整性.

大纲

```
svnadmin verify REPOS_PATH
```

描述

命令的目标是验证仓库的完整性. 基本上讲, 命令做的工作就是转储 所有的版本号, 并丢弃输出—为了检测潜在的硬盘错误和位衰减 (bitrot), 最好定期执行 *svnadmin verify*. 如果命令 报错, 说明仓库内至少含有一个出错的版本号, 这时候管理员应该从备份 恢复出错的版本号 (你备份过, 对吧?).

选项

```
--memory-cache-size (-M) ARG
--quiet (-q)
--revision (-r) ARG
```

示例

验证一个挂起的仓库:

```
$ svnadmin verify /var/svn/repos/
* Verified revision 1729.
```

svnlook 参考手册—Subversion 仓库检查工具

svnlook 是一个命令行工具，用于检查 Subversion 仓库的方方面面，它不会对仓库做出任何修改—它仅仅是 “看一下” 仓库. *svnlook* 经常被仓库钩子 使用，但管理员也可以用它来诊断仓库.

由于 *svnlook* 只有直接访问仓库才能工作（因 此它只能用在存放了仓库的主机中），所以说在指定仓库时只能使用文件系统路径，而不是 URL.

如果没有指定版本号或事务，*svnlook* 默认使用仓库 最新的版本号.

svnlook 的选项都是全局的，就像 *svn* 和 *svnadmin* 的全局选项，但是大 多数选项只对一个子命令起作用，因为 *svnlook*（有意地）限制了功能的作用范围.

svnlook 选项

`--copy-info`

使得 *svnlook changed* 显示复制源的详细信息.

`--diff-cmd` *`CMD`*

指定用于显示文件差异的外部差异比较工具. 如果在执行 *svnlook diff* 时没有指定该选项，它就使用 Subversion 内建的差异比较引擎，默认按照标准差异格式打印输出. 用户 还可以用选项 `--extensions (-x)` 向外部差异比较工具传递额外的选项.

`--diff-copy-from`

显示被复制的项目与复制源之间的差异.

`--extensions (-x)` *`ARG`*

为 Subversion 的差异比较指定扩展选项，有效的扩展选项有：

`--ignore-space-change (-b)`

忽略空白字符在数量方面的变化.

`--ignore-all-space (-w)`

忽略所有的空白字符.

`--ignore-eol-style`

忽略 EOL (end-of-line, 行结束标记）的变化.

`--show-c-function (-p)`

在差异比较输出中显示 C 程序的函数名.

`--unified(-u)`

显示宽度为 3 行的标准差异上下文.

扩展选项的默认值是 `-u`.

需要注意的是, 如果差异比较引擎是一个外部的差异比较工具, 那么 选项 `--extensions(-x)` 的参数 不仅限于以上提到的这些, 而是可以设置成外部差异比较工具能够接受的 任意 参数, 如果你希望传递多个参数, 必须把 它们用双引号包围起来.

`--full-paths`

使得 *svnlook tree* 显示完整的路径, 而不是 层次化的, 逐渐缩进的路径分量.

`--ignore-properties`

禁止 *svnlook diff* 显示属性的变化.

`--limit(-l)` *`ARG`*

把输出的项目个数限制在 *`ARG`* 以内.

`--no-diff-deleted`

禁止 *svnlook diff* 为删除的文件显示差异 输出. 默认情况下, 如果文件在版本号或事务中被删除了, *svnlook diff* 在为该文件显示差异输出时, 其效果 就像是把文件的全部内容删除, 但不删除文件.

`--no-diff-added`

禁止 *svnlook diff* 为新增的文件显示差异输出. 默认情况下, 如果文件是新增的, 那么 *svnlook diff* 在为该文件显示差异输出时, 其效果就像是把所有的内容一下子都添加到一 个已有的空文件里.

`--non-recursive(-N)`

只对一个单独的目录进行操作.

`--properties-only`

使得 *svnlook diff* 只显示属性的变化.

`--revision(-r)` *`REV`*

指定待查看的版本号.

`--revprop`

针对版本号属性进行操作, 而不是文件或目录上的属性, 如果用到 了该选项, 则用户还必须用选项 `--revision(-r)` 指定一个版本号.

`--show-inherited-props`

使得 *svnlook propget* 和 *svnlook proplist* 显示继承而来的属性.

`--transaction(-t)` *`ID`*

指定待查看的事务 ID.

`--show-ids`

为文件系统树中的每一个路径，显示文件系统节点的版本号 ID.

`--verbose(-v)`

显示更加详细的输出. 例如为 *svnlook proplist* 指定该选项时，除了列出属性名，Subversion 还会把属性值打印出来.

`--xml`

按照 XML 格式打印输出.

目录

名称

svnlook author — 打印作者.

大纲

`svnlook author` *`REPOS_PATH`*

描述

打印版本号或事务的作者.

选项

`--revision (-r)` *`REV`*
`--transaction (-t)` *`ID`*

示例

svnlook author 用起来很方便，但打印的信息也很 简单：

```
$ svnlook author -r 40 /var/svn/repos
sally
```

名称

svnlook cat — 输出文件的内容.

大纲

`svnlook cat` *`REPOS_PATH PATH_IN_REPOS`*

描述

输出文件的内容.

选项

`--revision (-r)` *`REV`*
`--transaction (-t)` *`ID`*

示例

下面的命令把文件 */trunk/README* 在事务 `ax8` 时的内容打印出来:

```
$ 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 — 打印发生变化的路径.

大纲

`svnlook changed` *`REPOS_PATH`*

描述

把版本号或事务中发生变化的路径打印出来，同时在每一行的前两列打印 状态字符，这些状态字符和 *svn update* 的意义相同.

`'A '`

项目是新增的

`'D '`

项目被删除了

`'U '`

文件的内容被修改了

`'_U'`

项目的属性被修改了，注意左边有个下划线

`'UU'`

文件的内容和属性都被修改了

区别目录路径与文件路径的方法是看路径末尾有没有字符“/”，末尾带“/”的路径是目录路径.

选项

`--copy-info`
`--revision (-r)` *`REV`*
`--transaction (-t)` *`ID`*

示例

下面的命令把版本号 39 中发生变化的所有文件和路径都打印出来. 输出中 的第一行是一个目录路径，因为该路径以 / 结尾:

```
$ 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
```

在下面的版本号中，有一个文件被重命名了：

```
$ svnlook changed -r 64 /var/svn/repos
A   trunk/vendors/baker/toast.txt
D   trunk/vendors/baker/bread.txt
```

不幸的是，上面的输出并没有阐明被删除的文件和被添加的文件之间的关系，加上选项 `--copy-info` 后就清楚多了：

```
$ 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 — 打印提交日期.

大纲

```
svnlook date REPOS_PATH
```

描述

打印版本号或事务的提交日期.

选项

```
--revision (-r) REV
--transaction (-t) ID
```

示例

下面的命令显示了版本号 40 的提交日期:

```
$ svnlook date -r 40 /var/svn/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

名称

svnlook diff — 打印被修改的文件和属性的差异.

大纲

`svnlook diff *REPOS_PATH*`

描述

按照标准差异格式，打印被修改的文件和属性的差异.

选项

`--diff-cmd *CMD*`
`--diff-copy-from`
`--ignore-properties`
`--no-diff-added`
`--no-diff-deleted`
`--properties-only`
`--revision (-r) *REV*`
`--transaction (-t) *ID*`
`--extensions (-x) *ARG*`

示例

下面的例子展示了新增的（空）文件，被修改的二进制文件和重命名（重命名可以理解成复制和删除）后又被修改的文件的差异比较输出：

```
$ svnlook diff -r 40 /var/svn/repos
Copied: trunk/relish.txt (from rev 39, trunk/vendors/deli/pickle.txt)
===================================================================
--- trunk/relish.txt                            (rev 0)
+++ trunk/relish.txt 2013-01-29 20:39:17 UTC (rev 40)
@@ -0,0 +1 @@
+Pickle relish is mostly made from cucumbers.

Deleted: trunk/vendors/deli/pickle.txt
===================================================================
--- trunk/vendors/deli/pickle.txt                               (rev 39)
+++ trunk/vendors/deli/pickle.txt 2013-01-29 20:39:17 UTC (rev 49)
@@ -1 +0,0 @@
-Pickles are mostly made from cucumbers.

Modified: trunk/vendors/deli/logo.jpg
===================================================================
```

```
(Binary files differ)

Added: trunk/vendors/deli/soda.txt
===================================================================
$
```

默认情况下，*svnlook diff* 把复制后的文件当成 新增的文件对待，仅仅使用不同的标签来区分复制与新增的区别．然而，选项 `--diff-copy-from` 使得 *svnlook diff* 只会输出复制后的文件与源文件之间的差异：

```
$ svnlook diff -r 40 /var/svn/repos --diff-copy-from
Copied: trunk/relish.txt (from rev 39, trunk/vendors/deli/pickle.txt)
===================================================================
--- trunk/vendors/deli/pickle.txt 2013-01-29 20:39:17 UTC (rev 39)
+++ trunk/relish.txt 2013-01-29 20:47:40 UTC (rev 3)
@@ -1 +1 @@
-Pickles are mostly made from cucumbers.
+Pickle relish is mostly made from cucumbers.

Deleted: trunk/vendors/deli/pickle.txt
===================================================================
--- trunk/vendors/deli/pickle.txt (rev 39)
+++ trunk/vendors/deli/pickle.txt 2013-01-29 20:39:17 UTC (rev 40)
@@ -1 +0,0 @@
-Pickles are mostly made from cucumbers.

Modified: trunk/vendors/deli/logo.jpg
===================================================================
(Binary files differ)

Added: trunk/vendors/deli/soda.txt
===================================================================================
$
```

选项 `--no-diff-deleted` 使得 *svnlook diff* 不再输出被删除的文件的差异：

```
$ svnlook diff -r 40 /var/svn/repos --no-diff-deleted
Copied: trunk/relish.txt (from rev 39, trunk/vendors/deli/pickle.txt)
===================================================================
--- trunk/relish.txt                            (rev 0)
+++ trunk/relish.txt 2013-01-29 20:39:17 UTC (rev 40)
@@ -0,0 +1 @@
+Pickle relish is mostly made from cucumbers.

Modified: trunk/vendors/deli/logo.jpg
===================================================================
(Binary files differ)

Added: trunk/vendors/deli/soda.txt
===================================================================================
```

```
$
```

注意，如果文件具有非文本化的 `svn:mime-type` 属性，则 *svnlook diff* 不会输出文件的差异.

名称

svnlook dirs-changed — 输出属性或子文件发生变化的目录.

大纲

```
svnlook dirs-changed REPOS_PATH
```

描述

输出属性或子文件发生变化的目录.

选项

```
--revision (-r) REV
--transaction (-t) ID
```

示例

下面的例子列出了在版本号 40 中发生变化的目录:

```
$ svnlook dirs-changed -r 40 /var/svn/repos
trunk/vendors/deli/
```

名称

svnlook filesize — 输出文件的大小，以字节为单位.

大纲

`svnlook filesize` *`REPOS_PATH PATH_IN_REPOS`*

描述

以字节为单位输出文件的大小，文件位于仓库 *`REPOS_PATH`* 内，且路径为 *`PATH_IN_REPOS`*. 输出的文件大小以 10 为 基数，然后是一个换行符. 如果没有用选项 `--revision` (`-r`) 或 `--transaction` (`-t`) 指定版本号 或事务，则默认使用 `HEAD`.

选项

`--revision (-r)` *`REV`*
`--transaction (-t)` *`ID`*

示例

下面的例子展示了如何查看文件 *trunk/vendors/deli/soda.txt* 在版本号 40 时的大小:

```
$ svnlook filesize -r 40 /var/svn/repos trunk/vendors/deli/soda.txt
23
$
```

名称

svnlook help (h, ?) — 帮助!

大纲

还可以使用 `svnlook -h` 和 `svnlook -?` 这两种调用方式.

描述

打印 *svnlook* 的帮助信息, 就像 *svn help* 能打印 *svn* 的帮助信息那样.

选项

无

名称

svnlook history — 打印仓库中某个路径的历史信息，如果没有指定路径，则默认是 仓库的根目录.

大纲

`svnlook history` *`REPOS_PATH`* [*`PATH_IN_REPOS`*]

描述

打印仓库中某个路径的历史信息，如果没有指定路径，则默认是 仓库的根目录.

选项

`--limit (-l)` *`ARG`*
`--revision (-r)` *`REV`*
`--show-ids`

示例

下面的例子显示了仓库中 */branches/bookstore* 到版本号 13 为止的历史信息：

```
$ 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，日志消息大小和日志消息内容.

大纲

```
svnlook info REPOS_PATH
```

描述

打印作者，时间戳，日志消息大小（以字节为单位）和日志消息内容，每一条信息后面都会跟随一个换行符.

选项

```
--revision (-r) REV
--transaction (-t) ID
```

示例

下面的例子列出了与版本号 40 相关的提交信息：

```
$ 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 — 如果仓库中的某一路径上存在锁，则打印锁的相关信息.

大纲

`svnlook lock` *`REPOS_PATH PATH_IN_REPOS`*

描述

如果仓库中的路径 *`PATH_IN_REPOS`* 上面 存在锁，则打印锁的相关信息；否则的话什么都不打印.

选项

无

示例

下面的命令输出了文件 *tree.jpg* 上的锁的相关 信息：

```
$ 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 — 打印日志消息内容，然后是一个换行符.

大纲

```
svnlook log REPOS_PATH
```

描述

打印日志消息内容.

选项

```
--revision (-r) REV
--transaction (-t) ID
```

示例

下面的命令打印了版本号 40 的日志消息内容：

```
$ svnlook log -r 40 /var/svn/repos/
Rearrange lunch.
```

名称

svnlook propget (pget, pg) — 打印仓库中某一路径上的属性值.

大纲

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

描述

打印仓库中某一路径上的属性值.

选项

```
--revision (-r) REV
--revprop
--show-inherited-props
--transaction (-t) ID
```

示例

下面的命令展示了在仓库 */var/svn/repos* 中，文件 */trunk/sandwich* 在版本号 `HEAD` 时，属性 `seasonings` 的值：

```
$ svnlook pg /var/svn/repos seasonings /trunk/sandwich
mustard
```

下面的命令展示了 */trunk* 在版本号 14 时，属性 `svn:auto-props` 的值：

```
$ svnlook pg repos svn:auto-props trunk --show-inherited-props -v -r14
Inherited properties on '/trunk',
from '/':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native
```

名称

svnlook proplist (plist, pl) — 打印文件和目录上的属性名和属性值.

大纲

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

描述

列出仓库中某一路径上的所有属性. 如果添加了选项 `--verbose` (`-v`), 则还会列出属性值.

选项

```
--revision (-r) REV
--revprop
--show-inherited-props
--transaction (-t) ID
--verbose (-v)
--xml
```

示例

下面的命令列出了文件 */trunk/README* 在版本号 HEAD 时的所有属性:

```
$ svnlook proplist /var/svn/repos /trunk/README
  original-author
  svn:mime-type
```

还是同样的命令, 但是这次添加了选项 `--verbose` (`-v`):

```
$ svnlook -v proplist /var/svn/repos /trunk/README
  original-author : harry
  svn:mime-type : text/plain
```

下面的命令还列出了继承而来的属性:

```
$ svnlook pl /var/svn/repos branches --show-inherited-props -v
Inherited properties on '/branches',
from '/':
  svn:auto-props
    *.py = svn:eol-style=native
    *.c = svn:eol-style=native
    *.h = svn:eol-style=native

  svn:global-ignores
    *.diff
    *.patch
```

```
Properties on '/branches':
```

名称

svnlook tree — 打印目录树.

大纲

`svnlook tree` *`REPOS_PATH`* [*`PATH_IN_REPOS`*]

描述

从 *`PATH_IN_REPOS`* 开始打印目录（如果 没有指定 *`PATH_IN_REPOS`*，则命令仓库的根 目录），还可以选择是否打印节点的版本号 ID.

选项

`--full-paths`
`--non-recursive (-N)`
`--revision (-r)` *`REV`*
`--show-ids`
`--transaction (-t)` *`ID`*

示例

下面的命令输出了在版本号 13 时，仓库的目录树：

```
$ svnlook tree -r 13 /var/svn/repos
/
 trunk/
  button.c
  Makefile
  integer.c
 branches/
  bookstore/
   button.c
   Makefile
   integer.c
…
```

如果添加了选项 `--show-ids`，则还会输出节点的 版本号 ID（版本号 ID 是 Subversion 版本化文件系统内部的标识符，用于唯一地标识文件系统中的一个节点）：

```
$ 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>
…
```

为了能让第三方脚本更方便地解析命令的输出，可以加上选项 `--full-paths`，该选项将使得 *svnlook* 为每一项打印完整的仓库路径，而且不会使用缩进来表示层次关系：

```
$ svnlook tree -r 13 /var/svn/repos --show-ids --full-paths
/ <0.0.r13/811>
trunk/ <1.0.r9/551>
trunk/button.c <2.0.r9/238>
trunk/Makefile <3.0.r7/41>
trunk/integer.c <4.0.r6/98>
branches/ <5.0.r13/593>
branches/bookstore/ <1.1.r13/390>
branches/bookstore/button.c <2.1.r12/85>
branches/bookstore/Makefile <3.0.r7/41>
branches/bookstore/integer.c <4.1.r13/109>
…
```

名称

svnlook uuid — 打印仓库的 UUID.

大纲

`svnlook uuid` *`REPOS_PATH`*

描述

打印仓库的 UUID, UUID 是仓库的通用唯一标识符 (Universal Unique IDentifier), Subversion 使用 UUID 来区分不同的仓库.

选项

无

示例

```
$ svnlook uuid /var/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

名称

svnlook youngest — 打印最年轻的版本号.

大纲

`svnlook youngest` *`REPOS_PATH`*

描述

打印仓库中最年轻的版本号.

选项

无

示例

下面的命令输出仓库中最年轻的版本号:

```
$ svnlook youngest /var/svn/repos/
42
```

svnserve 参考手册—定制化的 Subversion 服务器

目录

名称

svnserve — 通过 Subversion 定制化的网络协议向仓库提供网络访问服务.

大纲

`svnserve [-d | -i | -t | -X] OPTIONS...`

描述

svnserve 允许客户端使用 Subversion 定制化的 网络协议访问仓库.

svnserve 可以作为一个独立进程运行（客户端 将使用 `svn://` 形式的 URL), 也可以在需要时通过 *inetd* 或 *xinetd* 调用 *svnserve*（此时客户端还是使用 `svn://`); 还可以在需要时通过 *sshd* 调用 *svnserve*（此时客户端用的是 `svn+ssh://` 形式的 URL).

除非添加了选项 `--config-file`，否则的话一旦客户 端通过 URL 选择了一个仓库， *svnserve* 将从仓库目录 内的 *conf/svnserve.conf* 文件读取仓库的配置信息，例如认证数据库文件的位置和授权策略. 关于 *svnserve.conf* 的详细信息，见 “svnserve, 一个定制化的服务器”一节.

选项

和前面介绍的命令有所不同, *svnserve* 没有子命令，只有选项.

`--cache-fulltexts` *ARG*

开启或关闭文件内容的全文本缓存（只支持以 FSFS 作为后端存储 的仓库）.

`--cache-txdeltas` *ARG*

开启或关闭文件内容的差异缓存（只支持以 FSFS 作为后端存储 的仓库）.

`--compression` *LEVEL*

指定网络传输时的压缩级别，在 0 到 9 之间. 9 提供最大的压缩比，5 是默认值，0 将禁止压缩.

`--config-file` *FILENAME*

如果添加了该选项， *svnserve* 将在启动时 读取一次 *FILENAME*, 并把配置信息缓存到内存 中. 来自文件的密码和授权配置在 *svnserve* 收到 一个新的连接时，都会从文件中读取一次. 指定该选项后，*svnserve* 将不会再去读取仓库目录内的 *conf/svnserve.conf*. 关于 *FILENAME* 的格式，见 “svnserve, 一个定制化的服务器”一节.

`--daemon` (`-d`)

使得 *svnserve* 以守护进程模式运行. *svnserve* 把自己放到后台运行，接受从端口 3690（默认端口）到来的 TCP/IP 连接请求.

`--foreground`

和选项 `--daemon (-d)` 一起 使用时，*svnserve* 将保持在前台运行，通常情况 下是为了调试.

`--inetd (-i)`

使用 *inetd* 模式，在这种模式下 *svnserve* 将会被 *inetd* 调用运行.

`--help (-h)`

输出帮助信息.

`--listen-host HOST`

使得 *svnserve* 只接受来自 主机 ***HOST*** 的请求，***HOST*** 可以是一个 IP 地址或主机名.

`--listen-once (-X)`

使得 *svnserve* 在服务完一次连接后就 退出，这主要用于调试.

`--listen-port PORT`

当 *svnserve* 以守护进程模式运行时，让它 监听端口 ***PORT***.（如果是 FreeBSD 系统，则 *svnserve* 默认只监听 tcp6 的端口，加上该 选择后，*svnserve* 还会同时监听 tcp4 的端口）

`--log-file FILENAME`

告诉 *svnserve* 在必要时创建文件 ***FILENAME***，并把 Subversion 的 操作性日志输出到此文件中，这些日志和 *mod_dav_svn* 所生成的日志属于同一类型. 更 多的信息见 “高层日志记录”一节.

`--memory-cache-size (-M) ARG`

配置额外的内存缓存大小（以 MB 为单位），这些缓存用于减少 冗余操作，默认值是 `16`（该选项只适用于以 FSFS 作为后端存储的仓库）.

`--pid-file FILENAME`

告诉 *svnserve* 把进程 ID 写到文件 ***FILENAME*** 里，进程 *svnserve* 的用户必须对文件具有写权限.

`--prefer-ipv6 (-6)`

在解析监听的主机名时，优先使用 IPv6，默认情况下是优先使用 IPv4.

`--quiet`

只输出与错误有关的信息.

`--root (-r) ROOT`

为仓库设置一个虚拟根目录，设置后，由客户端在 URL 中指定的 路径将被解释成相对于根目录的路径，并且不能超出根目录所限定的 文件系统范围.

`--threads (-T)`

以守护进程模式运行时，告诉 *svnserve* 为每一个连接创建一个线程，而不是创建一个进程. 但是 *svnserve* 进程本身仍然会在启动时把自己放 到后台运行.

`--tunnel (-t)`

使得 *svnserve* 以隧道模式运行，和 *inetd* 模式基本相同（两种模式都是在 *stdin*/*stdout* 上为新 连接提供服务，服务完成后退出），唯一的不同点是在隧道模式下，新连接会预先用当前 UID 的用户名进行认证. 如果 *svnserve* 运行在一个隧道代理（例如 *ssh*）之上，那么客户端会自动添加该选项，也 就是说用户几乎不用亲自为 *svnserve* 添加选项 `--tunnel (-t)`. 如果你发现自己 自己在命令行输入了 `svnserve --tunnel`，并且想 知道接下来该做什么，见 “SSH 隧道” 一节.

`--tunnel-user` *`NAME`*

该选项和 `--tunnel (-t)` 一起使用，告诉 *svnserve* 使用用户 *`NAME`* 作为认证用户，而不是进程 *svnserve* 的用户. 如果管理员希望共享同一 SSH 系统账户，而且单独管理提交用户的身份，那么这个选项就会很 有用.

`--version`

显示版本信息，以及支持的后端存储模块.

svnversion 参考手册—
Subversion 工作副本版本信息

目录

名称

svnversion — 输出工作副本的本地版本号总结信息.

大纲

`svnversion [OPTIONS] [WC_PATH [TRAIL_URL]]`

描述

svnversion 用于输出工作副本的本地版本号总结 信息.

在程序的编译过程中，人们经常用它来生成程序的版本号（指的是软件 的版本号，而不是 Subversion 提交日志的版本号）.

参数 *TRAIL_URL* 指的是 URL 中处于末尾 的分量，如果指定了该参数，它将被用于判断 *WC_PATH* 是否是切换过的（对于判断 *WC_PATH* 中的路径是否是切换过的，不依赖 于参数 *TRAIL_URL*）.

如果没有显式指定 *WC_PATH*，将默认使用当 前工作目录，而且此时不能再指定参数 *TRAIL_URL*.

选项

和 *svnserve* 一样，*svnversion* 没有子命令，只有选项：

`--no-newline (-n)`

不要打印换行符.

`--committed (-c)`

使用最近一次产生修改的版本号，而不是当前版本号（当前版本号是 本地可获得的，值最大的版本号）.

`--help (-h)`

输出帮助信息.

`--quiet (-q)`

只输出必要的信息.

`--version`

输出 *svnversion* 的版本信息.

示例

如果工作副本中每个路径的版本号都相同（例如刚执行完 *svn update*），*svnversion* 就会输出这个 共同的版本号：

```
$ svnversion
```

```
4168
```

为了判断工作副本是否是切换过的，就加上参数 *TRAIL_URL*，注意这时候必须显式地指定 *WC_PATH*：

```
$ svnversion . /var/svn/trunk
4168
```

对于版本号混合的工作副本，将会输出版本号的范围：

```
$ svnversion
4123:4168
```

如果工作副本含有本地修改，就会在末尾添加字符 'M'：

```
$ svnversion
4168M
```

如果工作副本是切换过的，就会在末尾添加字符 'S'：

```
$ svnversion
4168S
```

svnversion 还会指出工作副本是否是稀疏的（见 “稀疏目录”一节），方法是在末尾添加 字符 'P'：

```
$ svnversion
4168P
```

如果有一个工作副本是稀疏的，切换过的，含有本地修改和混合的 版本号，则 *svnversion* 的输出将会是：

```
$ svnversion
4123:4168MSP
```

svnsync 参考手册—Subversion 仓库镜像工具

svnsync 是 Subversion 的远程仓库镜像工具，简单 地说，它允许你在仓库中重放另一个仓库的版本号.

在做镜像时，总是存在两个仓库：源仓库和镜像仓库. *svnsync* 从源仓库获取版本号，再到镜像仓库中重放版本 号. 任意一个仓库都可以是远程的或本地的—*svnsync* 只是通过 URL 对它们进行寻址.

svnsync 只要求对源仓库具有读取权限，它不会试图 去修改源仓库的任何数据. 当然，*svnsync* 要求对镜像 仓库具有读写权限.

svnsync 对于不是由镜像操作产生的修改非常 敏感，因此最好的做法是仅允许 *svnsync* 对镜像 仓库进行修改.

svnsync 的选项都是全局的：

svnsync 选项

`--allow-non-empty`

不去核实被初始化的仓库是否是一个版本历史为空的仓库（默认情况下，*svnsync initialize* 会去核实）.

`--config-dir` *`DIR`*

告诉 Subversion 从指定的目录中读取配置信息，而不是从默认目录（用户家目录里的 *.subversion*）中读取.

`--config-option` *`CONFSPEC`*

在命令运行期间，设置运行时配置选项的值. *`CONFSPEC`* 是一个字符串，指定了运行时 配置选项的名字空间，选项名和选项值，格式是 *`FILE`*:*`SECTION`*:*`OPTION`*=[*`VALUE`*]. 其中，*`FILE`* 和 *`SECTION`* 分别是运行时配置文件（*config* 或 *servers*）和节，它们包含了用户希望修改的选项. *`OPTION`* 是选项名，*`VALUE`* 是选项值（如果有的话）. 例如，为了临时禁止 HTTP 压缩，可以写成 `--config-option=servers:global:http-compression=no`. 选项 `--config-option` 可以在命令行上出现多次，从而同时修改多个选项.

`--disable-locking`

告诉 *svnsync* 不要使用自己的互斥访问机制，而是假定镜像仓库的互斥访问已经通过其他带外机制实现了.

`--no-auth-cache`

禁止在 Subversion 运行时配置目录中缓存认证信息（例如用户名和密码）.

`--non-interactive`

如果认证失败，或者证书不充分，将不再提示输入证书（例如 用户名和密码）. 如果在一个自动化运行的脚本中使用 Subversion，那么这个选项就会很有用，当遇到错误时，更好的做法是立刻失败退出，而不是请求输入更多的数据.

`--quiet (-q)`

只输出重要的信息.

`--revision (-r) ARG`

为 *svnsync copy-revprops* 指定版本号或 版本号范围.

`--source-password PASSWD`

指定源仓库的密码，如果没有指定或者密码不正确，Subversion 在需要时会提示用户输入密码.

`--source-prop-encoding ARG`

告诉 *svnsync* 在源仓库发现的版本号属性，如果它是可翻译的，则假定属性使用的字符编码是 *ARG*，把这些属性复制到镜像仓库时，把字符编码转换成 UTF-8.

`--source-username NAME`

指定源仓库的用户名，如果没有指定或用户名不正确，Subversion 在需要时会提示用户输入用户名.

`--steal-lock`

使得 *svnsync* 在必要时去窃取锁，从而保证对 镜像仓库的互斥访问.（用户应该只在以下情况下才去使用该选项— 锁在镜像仓库中存在，并且不再新鲜了，即没有其他 *svnsync* 进程正在访问仓库.）

`--sync-password PASSWD`

指定镜像仓库的密码. 如果没有指定或者密码不正确，Subversion 在需要时会提示用户输入密码.

`--sync-username NAME`

指定镜像仓库的用户名，如果没有指定或用户名不正确，Subversion 在需要时会提示用户输入用户名.

`--trust-server-cert`

和 `--non-interactive` 一起使用，告诉 Subversion 接受任意一个未知的 SSL 服务器证书，不要向用户提示.

目录

名称

svnsync copy-revprops — 把源仓库中的某个特定版本号或版本号范围上的所有版本号属性都 复制到镜像仓库中.

大纲

```
svnsync copy-revprops DEST_URL [SOURCE_URL]

svnsync copy-revprops DEST_URL REV[:REV2]
```

描述

由于 Subversion 的版本号属性可以在任意时刻发生变化，因此在把版本 号属性复制到镜像仓库后，源仓库中的版本号属性可能又发生了变化. 由于 *svnsync synchronize* 只能处理还未被同步的版本号 范围，它不会注意到范围之外的版本号属性是否发生了变化，这就造成了源 仓库和镜像仓库在版本号属性上出现了不一致. 命令 *svnsync copy-revprops* 正是这一问题的解决办法，可以用它重新同步 某个版本号或版本号范围上的版本号属性.

如果指定了 *SOURCE_URL*，它将作为 *svnsync* 的源仓库. 通常来说，*SOURCE_URL* 和命令 *svnsync initialize* 中的源仓库 URL 是相同的. 如果省略了 *SOURCE_URL*，*svnsync* 将通过询问镜像仓库来确定源仓库的 URL.

我们强烈建议在命令行上显式地指定源仓库的 URL，尤其是当不受 信任的用户对版本号 0 的版本号属性具有写权限时，这是因为 *svnsync* 通过版本号 0 的版本号属性来协调很多 工作.

选项

```
--config-dir DIR
--config-option CONFSPEC
--disable-locking
--no-auth-cache
--non-interactive
--quiet (-q)
--revision (-r) ARG
--source-password PASSWD
--source-prop-encoding ARG
--source-username NAME
--steal-lock
--sync-password PASSWD
--sync-username NAME
--trust-server-cert
```

示例

重新同步版本号 6 的版本号属性:

```
$ svnsync copy-revprops -r 6 file:///var/svn/repos-mirror \
```

```
                            http://svn.example.com/repos
Copied properties for revision 6.
$
```

名称

svnsync help — 获取帮助信息.

大纲

```
svnsync help
```

描述

如果你被关到了一座监狱中，不能访问因特网，手中也没有这本书，那 就只能通过该命令来学习如何使用 *svnsync*.

选项

无

名称

svnsync info — 打印目标仓库中，与同步有关的信息.

大纲

`svnsync info` *`DEST_URL`*

描述

打印的信息有源仓库的 URL, 源仓库的 UUID, 以及最后一个被同步的 版本号.

描述

`--config-dir` *`DIR`*
`--config-option` *`CONFSPEC`*
`--no-auth-cache`
`--non-interactive`
`--source-password` *`PASSWD`*
`--source-username` *`NAME`*
`--sync-password` *`PASSWD`*
`--sync-username` *`NAME`*
`--trust-server-cert`

示例

打印镜像仓库中，与同步有关的信息:

```
$ svnsync info file:///var/svn/repos-mirror
Source URL: http://svn.example.com/repos
Source Repository UUID: e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
Last Merged Revision: 47
$
```

名称

svnsync initialize (init) — 初始化一个镜像仓库.

大纲

`svnsync initialize` *`MIRROR_URL SOURCE_URL`*

描述

svnsync initialize 先检查目标仓库是否符合 作为镜像仓库的基本要求，然后把初始的管理信息记录到镜像仓库中，初始的管理信息把镜像仓库与源仓库（由参数 *`SOURCE_URL`* 指定）关联起来. 这是针对 镜像仓库的第一步操作.

一般情况下，*`SOURCE_URL`* 是源仓库 根目录的 URL. Subversion 1.5 及更新的版本允许用户针对仓库的子目录 进行同步—方法是把 *`SOURCE_URL`* 写成欲同步的仓库子目录的 URL.

默认情况下，上面提到的作为镜像仓库的基本要求指的是仓库 允许修改版本号属性，并且不含有任何版本历史. 然而，从 Subversion 1.7 开始，用户可以通过选项 `--allow-non-empty` 来 允许含有版本历史的仓库作为镜像仓库. 但是用户不应该习惯性地使用该 选项（因为它同时还会关掉保护机制），而是只在这样一种情景下使用：初始化一个已有的仓库副本，用它作为源仓库的镜像仓库. 如果仓库已经 包含了大量的版本历史，那么直接把它作为镜像仓库将会非常方便. 为了 创建一个镜像仓库，比较普通的做法是先初始化一个空仓库，然后再同步 全部的版本历史，但是更好的做法是先复制出一个仓库副本（例如使用命令 *svnadmin hotcopy*），再用 *svnsync initialize --allow-non-empty* 把仓库副本 初始化成一个镜像仓库，管理员将会看到后面这种做法会快得多.

选项

`--allow-non-empty`
`--config-dir` *`DIR`*
`--config-option` *`CONFSPEC`*
`--disable-locking`
`--no-auth-cache`
`--non-interactive`
`--quiet (-q)`
`--source-password` *`PASSWD`*
`--source-prop-encoding` *`ARG`*
`--source-username` *`NAME`*
`--steal-lock`
`--sync-password` *`PASSWD`*
`--sync-username` *`NAME`*
`--trust-server-cert`

示例

如果目标仓库禁止修改版本号属性，*svnsync initialize* 将会失败：

```
$ 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
$
```

把一个仓库初始化镜像仓库，该仓库已经通过钩子脚本 pre-revprop-change 来允许修改版本号属性：

```
$ svnsync initialize file:///var/svn/repos-mirror \
                    http://svn.example.com/repos
Copied properties for revision 0.
$
```

名称

svnsync synchronize (sync) — 把源仓库中未同步过的版本号同步到镜像仓库中.

大纲

```
svnsync synchronize DEST_URL [SOURCE_URL]
```

描述

命令 *svnsync syncronize* 负责镜像操作的大 部分工作. 命令首先询问镜像仓库已经复制了哪些版本号, 然后再从源 仓库复制那些还未被同步的版本号.

svnsync synchronize 可以被任意地中止或重启.

如果指定了 *SOURCE_URL*, *svnsync* 将把它作为源仓库的 URL. 通常来说, *SOURCE_URL* 和命令 *svnsync initialize* 中的源仓库 URL 是相同的. 如果省略了 *SOURCE_URL*, *svnsync* 将通过询问镜像仓库来确定源仓库的 URL.

我们强烈建议在命令行上显式地指定源仓库的 URL, 尤其是当不受 信任的用户对版本号 0 的版本号属性具有写权限时, 这是因为 *svnsync* 通过版本号 0 的版本号属性来协调很多 工作.

选项

```
--config-dir DIR
--config-option CONFSPEC
--disable-locking
--no-auth-cache
--non-interactive
--quiet (-q)
--source-password PASSWD
--source-prop-encoding ARG
--source-username NAME
--steal-lock
--sync-password PASSWD
--sync-username NAME
--trust-server-cert
```

示例

把源仓库中未同步过的版本号同步到镜像仓库中:

```
$ svnsync synchronize file:///var/svn/repos-mirror \
                      http://svn.example.com/repos
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.
$
```

svnrdump 参考手册—Subversion 远程仓库数据迁移

svnrdump 在 Subversion 1.7 引入，它相当于 *svnadmin dump* 和 *svnadmin load* 的网络版，作为一个单独的程序被发布出来. 我们在 “迁移仓库数据”一节 介绍了如何使用 *svnadmin* 和 *svnrdump* 转储和加载 仓库数据.

svnrdump 的选项都是全局的：

svnrdump 选项

`--config-dir` *`DIR`*

告诉 Subversion 从指定的目录内读取配置信息，而不是从默认的目录（用户家目录中的 *.subversion*）中读取.

`--config-option` *`FILE`*:*`SECTION`*:*`OPTION`*=[*`VALUE`*]

在命令运行期间，设置运行时配置选项的值. *`CONFSPEC`* 是一个字符串，指定了运行时 配置选项的名字空间，选项名和选项值，格式是 *`FILE`*:*`SECTION`*:*`OPTION`*=[*`VALUE`*]. 其中，*`FILE`* 和 *`SECTION`* 分别是运行时配置文件（*config* 或 *servers*）和节，它们包含了用户希望修改的选项. *`OPTION`* 是选项名，*`VALUE`* 是选项值（如果有的话）. 例如，为了临时禁止 HTTP 压缩，可以写成 `--config-option=servers:global:http-compression=no`. 选项 `--config-option` 可以在命令行上出现多次，从而同时修改多个选项.

`--incremental`

在转储版本号或版本号范围时，对范围中的第一个版本号按照增量 格式进行转储，而不是默认行为—转储第一个版本号的完整内容.

`--no-auth-cache`

禁止在 Subversion 运行时配置目录中缓存认证信息（例如用户名和密码）.

`--non-interactive`

如果认证失败，或者证书不充分，将不再提示输入证书（例如 用户名和密码）. 如果在一个自动化运行的脚本中使用 Subversion，那么这个选项就会很有用，当遇到错误时，更好的做法是立刻失败退出，而不是请求输入更多的数据.

`--password` *`PASSWD`*

指定 Subversion 用户密码，如果没有指定密码，或者密码不正确，在必要时 Subversion 将会提示用户再次输入密码.

`--quiet` (`-q`)

在执行过程中，只打印重要的信息.

`--revision` (`-r`) *`ARG`*

指定待操作的版本号或版本号范围.

`--trust-server-cert`

和 `--non-interactive` 一起使用，告诉 Subversion 接受任意一个未知的 SSL 服务器证书，不要向用户提示.

`--username` *`NAME`*

指定 Subversion 用户名，如果没有指定或用户名不正确，Subversion 在必要时会提示用户重新输入.

目录

名称

svnrdump dump

大纲

`svnrdump dump` *`SOURCE_URL`*

描述

把位于 *`SOURCE_URL`* 的仓库的版本号 转储到标准输出中. 除非用选项 `--revision (-r)` 指定了待转储的版本号或版本号范围, 否则的话, *svnrdump dump* 将转储全部的版本号.

选项

`--config-dir` *`DIR`*
`--config-option` *`FILE:SECTION:OPTION=[VALUE]`*
`--incremental`
`--no-auth-cache`
`--non-interactive`
`--password` *`PASSWD`*
`--quiet (-q)`
`--revision (-r)` *`ARG`*
`--trust-server-cert`
`--username` *`NAME`*

示例

下面的例子展示了如何转储一个远程仓库的全部历史 (假设执行该 命令的用户对仓库的全部路径都具有读取权限).

```
$ svnrdump dump http://svn.example.com/repos/calc > full.dump
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
```

仍然是同一个仓库, 但这次是增量地转储一个单独的版本号:

```
$ svnrdump dump http://svn.example.com/repos/calc \
           -r 21 --incremental > inc.dump
* Dumped revision 21.
$
```

名称

svnrdump help — 帮助!

大纲

```
svnrdump help
```

描述

如果既无法上网，手上也没有这本书，那你就只能依靠这个子命令来学习 如何使用 *svnrdump*.

选项

无

名称

svnrdump load

大纲

```
svnrdump load DEST_URL
```

描述

从标准输入读取转储流，并加载到位于 *DEST_URL* 的仓库中.

选项

```
--config-dir DIR
--config-option FILE:SECTION:OPTION=[VALUE]
--no-auth-cache
--non-interactive
--password PASSWD
--quiet (-q)
--trust-server-cert
--username NAME
```

示例

在转储一个本地仓库的同时，用 *svnrdump load* 把转储流加载到一个远程仓库中：

```
$ svnadmin dump -q /var/svn/repos/new-project | \
      svnrdump load http://svn.example.com/repos/new-project
* Loaded revision 0
* Loaded revision 1.
* Loaded revision 2.
…
```

为了保证 *svnrdump load* 能正常运行，要求 目标仓库允许修改版本号属性，这要通过钩子脚本 pre-revprop-change 加以实现，关于钩子脚本 pre-revprop-change 的更多信息，见 pre-revprop-change.

svndumpfilter 参考手册—Subversion 历史过滤工具

svndumpfilter 是一个命令行工具，用于从 Subversion 的转储文件中移除某些历史（即版本号），方法是排除或包含具有给定前缀的路径，关于 *svndumpfilter* 的更多信息，见 “svndumpfilter”一节.

svndumpfilter 的选项都是全局的：

svndumpfilter 选项

`--drop-empty-revs`

如果当前的 *svndumpfilter* 调用导致某个 版本号为空（即版本号不会对仓库产生修改），则从最终的转储文件中 删除该版本号.

`--drop-all-empty-revs`

从最终的转储文件中删除所有的空版本号（除了版本号 0).

`--pattern`

把命令行上所指定的路径前缀看成是文件名通配符模式，而不是显式的 路径子字符串.

`--renumber-revs`

对过滤后的版本号进行重新编号.

`--skip-missing-merge-sources`

忽略由于过滤而被删除的合并源. 如果没有加上这个选项， 并且保留 下来的路径的合并源由于过滤被删除了， 那么 *svndumpfilter* 就会报错退出.

`--preserve-revprops`

如果版本号内的所有路径都由于过滤而被删除了，并且没有指定 选项 `--drop-empty-revs`，则 *svndumpfilter* 的默认行为是删除除了日期和日志 消息外的所有版本号属性（仅仅是为了表明该版本号是空的）. 添加 选项 `--preserve-revprops` 后，*svndumpfilter* 将会保留所有已存在的版本号属性（由于与版本号相关的路径已经从转储文件中删除了，所以这样做可能没 什么意义）.

`--targets` *`FILENAME`*

告诉 *svndumpfilter* 从文件 *`FILENAME`* 内读取额外的路径前缀— 每行一个. 如果过滤操作非常复杂，复杂到难以在命令行上直接写出 全部的路径前缀，那么这个选项将会非常有用.

`--quiet`

不要打印与过滤有关的统计信息.

目录

名称

svndumpfilter exclude — 从转储流中删除含有指定前缀的路径.

大纲

```
svndumpfilter exclude PATH_PREFIX...
```

描述

从转储流中删除以 *PATH_PREFIX* 开始的 路径. 可以指定多个 *PATH_PREFIX*.

选项

```
--drop-empty-revs
--drop-all-empty-revs
--pattern
--preserve-revprops
--quiet
--renumber-revs
--skip-missing-merge-sources
--targets FILENAME
```

示例

假设我们有了一个仓库的转储文件, 里面包含了与野餐有关的各种目录, 但是人们不喜欢吃三明治 (sandwiches), 因此我们打算删除以 *sandwiches* 开始的路径:

```
$ 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'
$
```

从 Subversion 1.7 开始, *svndumpfilter* 不仅可以把 *PATH_PREFIX* 看成显式的子字符串, 还能看成是文件通配符模式. 例如, 如果用户想排除所有以 *.OLD* 结尾的路径, 则可以写成:

```
$ svndumpfilter exclude --pattern "*.OLD" < dumpfile > filtered-dumpfile
Excluding prefix patterns:
```

```
   '/*.OLD'

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):
   '/condiments/salt.OLD'
   '/condiments/pepper.OLD'
   '/toppings/cheese.OLD'
$
```

名称

svndumpfilter include — 从转储流中删除不含有指定前缀的路径.

大纲

```
svndumpfilter include PATH_PREFIX...
```

描述

可以用于选出那些以 *PATH_PREFIX* 开始的 路径 (从而排除掉其他路径), 可以指定多个 *PATH_PREFIX*.

选项

```
--drop-empty-revs
--drop-all-empty-revs
--pattern
--preserve-revprops
--quiet
--renumber-revs
--skip-missing-merge-sources
--targets FILENAME
```

示例

假设我们有了一个仓库的转储文件, 里面包含了与野餐有关的各种目录, 但是我们只想留下三明治 (sandwiches):

```
$ 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 12 node(s):
   '/condiments'
   '/condiments/pepper'
   '/condiments/pepper.OLD'
   '/condiments/salt'
   '/condiments/salt.OLD'
   '/drinks'
   '/snacks'
   '/supplies'
```

```
   '/toppings'
   '/toppings/cheese'
   '/toppings/cheese.OLD'
   '/toppings/lettuce'
$
```

从 Subversion 1.7 开始, *svndumpfilter* 不仅可以把 `PATH_PREFIX` 看成显式的子字符串, 还能看成是文件通配符模式. 例如, 如果用户想留下所有以 *ks* 结尾的路径, 则可以写成:

```
$ svndumpfilter include --pattern "*ks" < dumpfile > filtered-dumpfile
Including prefix patterns:
   '/*ks'

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 11 node(s):
   '/condiments'
   '/condiments/pepper'
   '/condiments/pepper.OLD'
   '/condiments/salt'
   '/condiments/salt.OLD'
   '/sandwiches'
   '/supplies'
   '/toppings'
   '/toppings/cheese'
   '/toppings/cheese.OLD'
   '/toppings/lettuce'
$
```

名称

svndumpfilter help — 帮助!

大纲

```
svndumpfilter help [SUBCOMMAND...]
```

描述

如果既无法上网，手上也没有这本书，那你就只能依靠这个子命令来学习 如何使用 *svndumpfilter*.

选项

无

svnmucc 参考手册— Subversion 多 URL 命令行客户端

Subversion 多 URL 命令行客户端 (*svnmucc*) 允许用户在没有工作副本的情况下，向仓库提交任意地修改. 对于远程提交 这个特性而言，*svnmucc* 提供的功能类似于 *svn*, 但要远远强于后者，例如 *svnmucc* 可以在一次提交中执行多种不同类型的操作. *svnmucc* 甚至可以在没有工作副本的前提下修改文件 内容和版本化属性，而这是 *svn* 所不具备的.

本章介绍命令行工具 *svnmucc*, 以及用户可以用它 完成哪些远程操作.

目录

名称

svnmucc — 基于 URL, 对 Subversion 仓库执行一个或多个操作，在一个版本 号中提交所有的修改.

大纲

`svnmucc` *`ACTION...`*

描述

利用 *svnmucc*，用户可以在没有工作副本的情况下向 仓库提交修改. 它允许用户基于 URL 直接操作仓库. 每一次调用 *svnmucc* 都可以执行一个或多个 *`ACTION`*，并且操作产生的所有修改都是在同一个 版本号中完成提交.

操作

svnmucc 支持的操作（及其参数）有以下这些，它们 可以以各种组合出现在命令行上：

cp *`REV SRC-URL DST-URL`*

把版本号为 *`REV`* 且位于 *`SRC-URL`* 的文件或目录复制到 *`DST-URL`*.

mkdir *`URL`*

在 *`URL`* 创建一个新目录. *`URL`* 的父目录必须已经存在（或者说 父目录将会被前面的 *svnmucc* 操作创建），命令无法自动创建中间目录.

mv *`SRC-URL DST-URL`*

把位于 *`SRC-URL`* 的文件或目录移动到 *`DST-URL`*.

rm *`URL`*

删除位于 *`URL`* 的文件或目录.

put *`SRC-FILE URL`*

在 *`URL`* 添加一个新文件，或者修改 位于 *`URL`* 的已经存在的文件，命令把 本地文件 *`SRC-FILE`* 的内容复制到新 文件或已有的文件上. 如果 *`SRC-FILE`* 是单连字符 `-`，*svnmucc* 将从标准输入中读取文件内容.

propset *`NAME VALUE URL`*

把 *`URL`* 的属性 *`NAME`* 的值设置成 *`VALUE`*.

propsetf *`NAME FILE URL`*

把 *`URL`* 的属性 *`NAME`* 的值设置成文件 *`FILE`* 的内容.

propdel *NAME URL*

删除 *URL* 上的属性 *NAME*.

选项

svnmucc 的选项都是全局的，包括：

`--config-dir` *DIR*

告诉 *svnmucc* 从指定的目录中读取配置信息，而不是从默认目录（用户家目录里的 *.subversion*）中读取.

`--config-option` *CONFSPEC*

在命令运行期间，设置运行时配置选项的值. *CONFSPEC* 是一个字符串，指定了运行时 配置选项的名字空间，选项名和选项值，格式是 *FILE*:*SECTION*:*OPTION*=[*VALUE*]. 其中，*FILE* 和 *SECTION* 分别是运行时配置文件（*config* 或 *servers*）和节，它们包含了用户希望修改的选项. *OPTION* 是选项名，*VALUE* 是选项值（如果有的话）. 例如，为了临时禁止 HTTP 压缩，可以写成 `--config-option=servers:global:http-compression=no`. 选项 `--config-option` 可以在命令行上出现多次，从而同时修改多个选项.

`--extra-args` (`-X`) *ARGFILE*

从文件 *ARGFILE* 读取额外的命令行 参数，每行一个. 如果 *ARGFILE* 是 单连字符 `-`，*svnmucc* 将从标准输入中读取额外的命令行参数.

`--file` (`-F`) *MSGFILE*

从文件 *MSGFILE* 中读取日志消息.

`--help` (`-h`, `-?`)

显示命令的帮助信息，然后退出.

`--message` (`-m`) *MSG*

使用 *MSG* 作为日志消息.

`--no-auth-cache`

禁止在 Subversion 运行时配置目录中缓存认证信息（例如用户名和密码）.

`--non-interactive`

禁止所有的交互性提示（例如请求用户输入认证证书）.

`--revision` (`-r`) *REV*

把版本号 *REV* 作为 *svnmucc* 操作所导致的所有修改的基础版本号. 如果用户需要使用 *svnmucc* 修改已存在的版本 化数据，那么你应该习惯性地使用该选项，从而避免在无意间撤消 其他同事几乎在同一时间提交的修改.

`--root-url(-U)` *`ROOT-URL`*

把 *`ROOT-URL`* 作为其他 URL 目标 的基础 URL, 其他 URL 目标将被看作是相对于 *`ROOT-URL`* 的路径. *`ROOT-URL`* 并不一定就是仓库的根 URL（例如由 *svn info* 输出的仓库根 URL), 它可以 是任意的 URL, 只要它是 *svnmucc* 命令行上所指定的 URL 目标的公共前缀部分即可.

`--password(-p)` *`PASSWD`*

指定 Subversion 用户密码, 如果没有指定密码, 或者密码不正确, 在必要时 Subversion 将会提示用户再次输入密码.

`--username` *`NAME`*

指定 Subversion 用户名, 如果没有指定或用户名不正确, Subversion 在必要时会提示用户重新输入.

`--version`

输出程序的版本信息, 然后退出.

`--with-revprop` *`NAME`*`=`*`VALUE`*

把版本号属性 *`NAME`* 的值设置成 *`VALUE`*.

示例

为了在没有工作副本的情况下安全地修改文件，我们可以这样做：先用 *svn cat* 把文件的当前内容下载到本地，然后再用 *svnmucc put* 把更新后的文件提交到仓库中.

```
$ # Calculate some convenience variables.
$ export FILEURL=http://svn.example.com/projects/sandbox/README
$ export BASEREV=`svn info ${FILEURL} | \
                  grep '^Last Changed Rev' | cut -d ' ' -f 2`
$ # Get a copy of the file's current contents.
$ svn cat ${FILEURL}@${BASEREV} > /tmp/README.tmpfile
$ # Edit the (copied) file.
$ vi /tmp/README.tmpfile
$ # Commit the new content for our file.
$ svnmucc -r ${BASEREV} put README.tmpfile ${FILEURL} \
          -m "Tweak the README file."
r24 committed by harry at 2013-01-21T16:21:23.100133Z
# Cleanup after ourselves.
$ rm /tmp/README.tmpfile
```

使用类似的思路修改文件或目录的属性, 只不过这次要把命令换成 *svn propget* 和 *svnmucc propsetf*.

```
$ # Calculate some convenience variables.
$ export PROJURL=http://svn.example.com/projects/sandbox
$ export BASEREV=`svn info ${PROJURL} | \
                  grep '^Last Changed Rev' | cut -d ' ' -f 2`
$ # Get a copy of the directory's "license" property value.
$ svn -r ${BASEREV} propget license ${PROJURL} > /tmp/prop.tmpfile
```

```
$ # Tweak the property.
$ vi /tmp/prop.tmpfile
$ # Commit the new property value.
$ svnmucc -r ${BASEREV} propsetf prop.tmpfile ${PROJURL} \
          -m "Tweak the project directory 'license' property."
r25 committed by harry at 2013-01-21T16:24:11.375936Z
# Cleanup after ourselves.
$ rm /tmp/prop.tmpfile
```

下面再看看如何在一次调用中完成多个操作.

为了实现 “移动标签”（重复使用同一个标签名，指向 代码的不同版本，例如总是指向当前最新的稳定版），可以使用 *svnmucc rm* 和 *svnmucc cp*:

```
$ svnmucc -U http://svn.example.com/projects/doohickey \
          rm tags/latest-stable \
          cp HEAD trunk tags/latest-stable \
          -m "Slide the 'latest-stable' tag forward."
r134 committed by harry at 2013-01-12T11:02:16.142536Z
$
```

在上面的例子里，我们通过使用选项 `--root-url(-U)` 指定了一个基础 URL，其他参数中的 URL 将被当 成相对 URL，从而减少打字工作.

下面的例子展示了如何使用 *svnmucc* 创建一个新 标签，这个标签删除了在发布版中不需要的一个目录，同时还添加了一个用于描述发布版的文件：

```
$ echo "This is the 1.2.0 release." | \
       svnmucc -U http://svn.example.com/projects/doohickey \
               -m "Tag the 1.2.0 release." \
               -- \
               cp HEAD trunk tags/1.2.0 \
               rm tags/1.2.0/developer-notes \
               put - tags/1.2.0/README.tag
r164 committed by cmpilato at 2013-01-22T05:26:15.563327Z
$ svn log -c 164 -v http://svn.example.com/projects/doohickey
------------------------------------------------------------------------
r164 | cmpilato | 2013-01-22 00:26:15 -0500 (Tue, 22 Jan 2013) | 1 line
Changed paths:
   A /tags/1.2.0 (from /trunk:163)
   A /tags/1.2.0/README.tag
   D /tags/1.2.0/developer-notes

Tag the 1.2.0 release.
$
```

上面的例子不仅展示了如何在 *svnmucc* 的一次调用 中完成多种不同的操作，还展示了如何从标准输入读取文件的内容. 注意我 们还在命令行中使用了双连字符 `--`，用于告诉 *svnmucc* 在 `--` 的右边已经没有 更多的选项了，从而避免右边的单连字符 `-` 被 *svnmucc* 当成选项开始的标志.

Subversion 仓库钩子参考手册

Subversion 仓库提供了很多事件钩子, 利用这些钩子, 管理员可以在 特定操作的特定时间点扩展 Subversion 的功能. 仓库钩子被实现成由 Subversion 在特定时间点执行的程序, 这些时间点包括在提交之前或之后, 用户锁定文件之前或之后, 等等.

对于每一种钩子, Subversion 都会尝试去执行以钩子命名的程序, 这些 程序文件位于仓库目录的 *hooks/* 子目录内. 例如, 在一个 Unix 系统中, 钩子 start-commit 对应的程序文件是 *REPOS_PATH/hooks/start-commit*, 它可以是一个二进制可执行程序, shell 脚本 或 Python 脚本等. 在 Windows 系统中, 钩子 start-commit 对应的程序文件仍然在相同的目录内, 但文件的名 字变成了 *START-COMMIT.EXE* 或 *START-COMMIT.BAT*, 而不是 Unix 中的 *start-commit*.

本章介绍 Subversion 提供的各种钩子, 包括这些钩子何时被调用, 钩子 的输入参数, 以及钩子的行为将会如何影响 Subversion 的工作流.

目录

名称

start-commit — 开始一个新提交的通知.

大纲

`start-commit` *`REPOS-PATH USER CAPABILITIES TXN-NAME`*

描述

提交事务创建完, 并且初始属性设置完成后, 紧接着就开始执行钩子 start-commit. 它的典型用法是作为早期的终止机制, 避免浪费大量的 时间等待一个已经确定最终会失败的提交结束, 提交失败的原因可能是用户 缺少提交权限, 或者是某些提交元数据验证失败.

如果钩子 start-commit 的退出值不为零, 提交过程就会中止, 提交 事务也会被销毁, 任何打印到 *stderr* 的信息 都会返回给客户端.

钩子 start-commit 并非是评价特定提交的实质内容的合适地方, 因为它在传送文件或目录的修改信息之前被调用, 此时管理员应该使用 钩子 pre-commit (见本章的 pre-commit).

在 Subversion 1.8 之前, Subversion 是在创建提交事务 之前 调用钩子 start-commit, 如果钩子返回 失败, 提交事务就不会被创建. 从 Subversion 1.8 开始不再如此, 这是为了让钩子 start-commit 能够访问事务的属性, 事务的属性包括 提交的日志消息等.

输入参数

传递给钩子程序的命令行参数, 按照出现的顺序来说, 有:

1. 仓库路径
2. 尝试向仓库提交的已认证的用户名
3. 以冒号分隔的特性 (capabilities) 列表, 这些特性由客户端传递 给服务器, 包括 `depth`, `mergeinfo` 和 `log-revprops` (`log-revprops` 在 Subversion 1.5 引入)
4. 提交事务名 (在 Subversion 1.8 引入)

常见用法

访问控制 (例如临时禁止向仓库提交修改).

只允许支持某些特性的客户端向仓库提交修改.

名称

pre-commit — 提交即将完成的通知.

大纲

`pre-commit` *`REPOS-PATH TXN-NAME`*

描述

钩子 pre-commit 在提交事务即将生成一个新的版本号之前被调用. 它的典型用法是禁止内容不符合要求的提交 (例如你的公司可能要求所有 的提交日志消息都要包含来自问题跟踪系统的单号, 或者要求日志消息不 能为空).

如果钩子 pre-commit 的退出值不为零, 提交过程就会中止, 提交事务 也会被销毁, 任何打印到 *stderr* 的信息都会返回 给客户端.

输入参数

传递给钩子程序的命令行参数, 按照出现的顺序来说, 有:

1. 仓库路径

2. 提交事务名

此外, Subversion 还会把客户端可能提供的所有锁令牌通过标准输入 *stdin* 传递给钩子程序. 如果提供了锁令牌, 它们 的格式将会是这样: 首先是一行 `LOCK-TOKENS`, 接下 来是锁令牌, 每个锁令牌占据单独的一行. 每个锁令牌行都包含了以下这信息: 与锁有关的仓库文件系统路径, 这些路径已经是经过转码后的 URI; 然后是管道符 (|); 最后是锁令牌字符串.

常见用法

控制和检查修改

名称

post-commit — 成功提交的通知.

大纲

`post-commit` *`REPOS-PATH REVISION TXN-NAME`*

描述

事务提交并且新的版本号生成后，Subversion 就会执行钩子 post-commit. 大多数管理员都会利用 post-commit 向团队成员发送关 于新提交的邮件，或者通知其他工具（例如一个问题跟踪系统）有新的提交 生成. 有些管理员还会用 post-commit 触发备份操作.

如果钩子 post-commit 的退出值不为零，提交过程将 *不会* 中止，因为这时候提交已经完成了. 但是，钩子打印到 *stderr* 的信息都会返回给客户端，以 便分析钩子失败的原因.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有：

1. 仓库路径
2. 新版号的整数编号
3. 已经变成版本号的事务名，正是该版本号触发了本次钩子的运行.

常见用法

提交通知；工具集成

名称

pre-revprop-change — 版本号属性即将被修改的通知.

大纲

`pre-revprop-change` *`REPOS-PATH REVISION USER PROPNAME ACTION`*

描述

钩子 pre-revprop-change 在版本号属性即将被修改之前调用，这 里所说的版本号修改不是通常提交的一部分. 和其他钩子不同的是，pre-revprop-change 的默认行为是禁止修改版本号属性. 为了允许修改 版本号属性，pre-revprop-change 必须显式地以零作为退出值.

如果 pre-revprop-change 没有退出，或者不可执行，或者退出值不 为零，Subversion 将禁止修改版本号属性，钩子打印到 *stderr* 的信息都会返回给客户端.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有:

1. 仓库路径
2. 即将被修改的版本号的整数编号
3. 试图修改版本号属性的已认证的用户名
4. 即将被修改的版本号属性的名字
5. 用于描述修改类型的字符: `A` (新增的)，`D` (被删除的)，或 `M` (被修改的)

此外，Subversion 还会把版本号属性的新值通过标准输入 *stdin* 传递给钩子程序.

常见用法

访问控制; 控制和检查修改

名称

post-revprop-change — 版本号属性被成功修改的通知.

大纲

`post-revprop-change` *`REPOS-PATH REVISION USER PROPNAME ACTION`*

描述

钩子 post-revprop-change 在版本号属性被修改完成后立即执行，这 里所说的版本号修改不是通常提交的一部分. 从前面对钩子 pre-revprop-change 的介绍读者应该可以很容易推断出，除非提供了 pre-revprop-change，否则的话 post-revprop-change 根本没机会执行. 这个钩子的典型用途是发邮件通知版本号属性被修改了.

如果钩子 post-revprop-change 的退出值不为零，那么针对版本号属性 的修改将 不会 被中止，因为这时候修改已经完成了. 但是，钩子打印到 *stderr* 的信息都会返回给客户端，以便分析钩子失败的原因.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有：

1. 仓库路径
2. 属性即将被修改的版本号的整数编号
3. 修改版本号属性的已认证的用户名
4. 被修改的版本号属性的名字
5. 用于描述修改类型的字符：`A`（新增的），`D`（被删除的），或 `M`（被修改的）

此外，Subversion 还会把版本号属性的旧值通过标准输入 *stdin* 传递给钩子程序.

常见用法

通知人们有版本号属性被修改了

名称

pre-lock — 有人试图锁定某一路径的通知.

大纲

`pre-lock` *`REPOS-PATH PATH USER COMMENT STEAL`*

描述

每当有人尝试对某个路径进行锁定时，就会触发钩子 pre-lock. 它可以 用于禁止锁定，或者根据策略来决定哪些用户可以锁定特定的路径. 如果钩子 发现路径已经被其他人锁定了，它还可以决定用户是否可以 “窃取” 其他人的锁.

如果钩子 pre-lock 的退出值不为零，锁定操作将被中止，任何打印到 *stderr* 的信息都会返回给客户端.

钩子 pre-lock 可以口述锁令牌，方法是把锁令牌打印到标准输出中，这个锁令牌将会被分配给锁. 正因为如此，在实现钩子 pre-lock 时，注意 不要往标准输出中打印不必要的信息.

如果钩子 pre-lock 往标准输出中打印了锁令牌，那么钩子程序自己 要负责保证生成的锁令牌是 独一无二 的. 如果 不能生成独一无二的锁令牌，那么将导致未定义的—很可能是不希望 看到的—行为.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有：

1. 仓库路径
2. 将被锁定的路径
3. 尝试加锁的已认证的用户名
4. 由用户提供的锁注释
5. `1`（如果用户试图窃取一个已存在的锁）或 `0`（用户不想窃取锁）

常见用法

访问控制

名称

post-lock — 路径被成功锁定的通知.

大纲

`post-lock` *`REPOS-PATH USER`*

描述

钩子 post-lock 在一个或多个路径被成功锁定后执行，它的典型用法 是发送路径被锁定的通知邮件.

如果钩子 post-lock 的退出值不为零，那么锁定操作将 不会 被中止，因此这时候锁定操作已经完成了，但是钩子程序打印到 *stderr* 的所有信息都会返回 给客户端，以便分析钩子失败的原因.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有：

1. 仓库路径

2. 锁定路径的已认证的用户名

另外，被锁定的路径列表将通过标准输入传递给钩子程序，每行一个.

常见用法

路径被锁定的通知

名称

pre-unlock — 有人试图解锁某一路径的通知.

大纲

`pre-unlock` *`REPOS-PATH PATH USER TOKEN BREAK-UNLOCK`*

描述

如果有人试图解决某一路径，就会触发钩子 pre-unlock. 它可以用于 决定哪些用户可以解锁特定的路径，尤其是决定锁的破坏策略，例如当用户 A 已经锁定了一个文件时，这时候是否应该允许用户 B 破坏锁? 如果锁已经 持有一周了呢? 这些考虑都可以放在钩子 pre-unlock 里实现.

如果钩子 pre-unlock 的退出值不为零，解锁操作将被中止，任何打印到 *stderr* 的信息都会返回给客户端.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有:

1. 仓库路径
2. 将被解锁的路径
3. 试图解锁路径的已认证的用户名
4. 与锁有关的锁令牌
5. `1` (如果用户试图破坏锁) ; `0` (用户不想破坏锁)

常见用法

访问控制

名称

post-unlock — 路径被成功解锁的通知.

大纲

`post-unlock` *`REPOS-PATH USER`*

描述

钩子 post-unlock 在一个或多个路径被成功解锁后执行，它的典型用法 是发送路径被解锁的通知邮件.

如果钩子 post-unlock 的退出值不为零，那么解锁操作将 不会 被中止，因此这时候解锁操作已经完成了，但是钩子程序打印到 *stderr* 的所有信息都会返回 给客户端，以便分析钩子失败的原因.

输入参数

传递给钩子程序的命令行参数，按照出现的顺序来说，有：

1. 仓库路径

2. 解锁路径的已认证的用户名

另外，被解锁的路径列表将通过标准输入传递给钩子程序，每行一个.

常见用法

解锁通知

部分 III. 附录

目录

附录 A. Subversion 快速入门

如果读者想要马上开始使用 Subversion（并且很享受在使用的过程中学习），那么本附录将快速地教会你如何创建仓库，导入代码，并检出工作副本. 在介绍 的过程中我们会给出详细内容的章节链接.

如果你对版本控制的概念，或者对 CVS 和 Subversion 的 “复制–修改–合并” 模型感到完全陌生，那么建议你先去阅读 第 1 章 基本概念.

安装 Subversion

Subversion 构建在一个可移植函数库之上，这个函数库叫做 APR (Apache Portable Runtime, Apache 可移植运行库). APR 提供了 Subversion 在不同操作系统平台上运行所需的各种接口：磁盘访问，网络访问，内存管理等. 利用 APR 提供的抽象层，只要是能运行基于 APR 的应用程序的 操作系统—例如 Windows, Linux, 所有的 BSD, Mac OS X 和 NetWare 等 —都能运行 Subversion.

虽然 APR 函数库是 Apache HTTP 服务器（即 *httpd*) 的一部分，并且 *httpd* 也可以作为 Subversion 仓库的托管服务器，但 *httpd* 并不是 Subversion 的 必要组件，即使你没有 *httpd*, 仍然可以安装 Subversion.

获取 Subversion 最方便的方式就是下载你所用的操作系统对应的 二进制安装包. 在 Subversion 官网 (http://subversion.apache.org) 经常能够看到由志愿者提供的 安装包，还包括针对 Windows 的图形化安装包. 如果你的操作系统是类 Unix 系统，还可以用操作系统自带的软件包管理器 (Yum, APT 等) 下载安装 Subversion.

当然，你也可以自己从源代码编译安装 Subversion, 虽然这通常不是一件 很容易的事.（如果你没有构建开源软件包的经验，那么最好还是直接下载已 经编译好的二进制安装包）构建的第一步是从 Subversion 官网下载最新的源 码包，把源码包解压后，按照文件 *INSTALL* 列出的 步骤来编译和安装 Subversion.

如果你是一个追求新技术的极客，还可以自己从 Subversion 源码的托管 站点下载源代码，当然，在下载之前你必须已经有了一个 Subversion 客户端. 一旦有了客户端，你就可以从 http://svn.apache.org/repos/asf/subversion 检出 Subversion 的 工作副本[1]:

```
$ svn checkout http://svn.apache.org/repos/asf/subversion/trunk subversion
A    subversion/HACKING
A    subversion/INSTALL
A    subversion/README
A    subversion/autogen.sh
A    subversion/build.conf
…
```

上面的命令在当前工作目录中创建了一个工作副本目录 *subversion*, 工作副本里存放的是当前最新（未发布）的 Subversion 源代码，你可以按照自己的需求修改检出后的工作副本目录名. 检出 后，无论你的工作副本目录叫什么名字，你都拥有了一份最新的 Subversion 源代码. 当然，为了构建 Subversion, 你还要下载一些函数库（例如 apr, apr-util 等)—具体的细节见工作副本顶层目录中的 *INSTALL* 文件.

[1]注意在下面的例子中，URL 的最后一个路径分量是 `trunk`, 而不是 `subversion`, 这背后的原因可以看一下我们关于 Subversion 分支与标签模型的讨论.

快速入门教程

> “旅客朋友们，请检查你们的座椅靠背是否完好，方向是否竖直，杂物 桌是否已经固定．空乘人员，现在准备起飞….”

接下来的内容是 Subversion 的快速入门教程，在这个教程里我们将为 读者介绍最基本的 Subversion 配置与操作．阅读后，读者应该对 Subversion 的典型用法具备了最基本的了解．

本附录内的例子假定读者的类 Unix 系统已经安装了 *svn* 和 *svnadmin*，它们分别是 Subversion 命令行客户端工具和管理工具．（本附录内的例子对 Windows 同 样适用，不过要做一些小调整．）我们假设读者使用的是 Subversion 1.2 或 更新的版本（可能通过执行 `svn --version` 查看 Subversion 的版本）．

Subversion 把所有的版本化数据都存放在一个中央仓库中．为了开始我 们的教程，首先创建一个新仓库：

```
$ cd /var/svn
$ svnadmin create repos
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
$
```

上面的命令在目录 */var/svn/repos* 创建了一 个 Subversion 仓库，如果目录 *repos* 事先不存在，*svnadmin* 将自动创建该目录．这个目录含有一个数据 库文件集合（以及其他文件），你不会在目录中直接看到那些版本化文件．关于仓库创建和维护的更多信息，见 第 5 章 仓库管理．

Subversion 没有 “项目” (project) 这个概念，仓库仅 仅是一个虚拟的版本化文件系统，它可以存放你希望的任何数据．有些管理员 喜欢在一个仓库中只存放一个项目，而其他管理员则喜欢在一个仓库中存放多个 项目，我们在 “规划仓库的组织方式”一节 讨论了各种仓库组织方式的利弊．无论采用何种组织方式，仓库都只是在管理 文件和目录，因此是否要将某些目录解释成 “项目” 完全取决于 用户．因此，本附录在谈到项目时，请记住我们只是在谈论仓库里的目录（或 目录集合）．

在下面的例子里，我们假设读者已经准备好了可以被导入到新仓库中的 项目（文件和目录的集合）．在开始前把项目里的文件和目录都放到一个名 为 *myproject*（或其他你喜欢的名字）的目录中，由于 第 4 章 分支与合并 介绍的原因，你的项目内包含了 3 个顶层目录：*branches*, *tags* 和 *trunk*. 目录 *trunk* 应该 包含了项目的所有数据，而 *branches* 和 *tags* 则是空目录：

```
/tmp/
  myproject/
    branches/
    tags/
    trunk/
      foo.c
      bar.c
      Makefile
      …
```

Subversion 并不要求仓库顶层目录下必须是 *branches*, *tags* 和 *trunk* 这 3 个子目录，但这是最流行的仓库目录布局．

项目数据一旦准备好，接下来就可以用命令 *svn import*（见 “往仓库中添加数据”一节）把它们都导入到仓库中：

```
$ 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，仓库中就已经包含了项目的初始数据．前面已经说过，用户无法 在仓库目录中直接看到导入的那些数据，它们都存放在一个数据库中．不过，仓库的虚拟文件系统已经包含了一个名为 *myproject* 的顶层目录，导入的项目数据就存放在这个目录里．

注意，被导入的原始目录 */tmp/myproject* 没有发 生任何变化，Subversion 不会对它做出任何修改．（实际上，在导入完成后你 甚至可以把它删除．）在开始操作仓库的数据之前，你需要创建一个 “工作副本” (working copy)，它是用户的一个私有工作空间．下面的命令请求 Subversion 为仓库中的目录 *myproject/trunk* “检出” (check out) 一个工作副本：

```
$ svn checkout file:///var/svn/repos/myproject/trunk myproject
A    myproject/foo.c
A    myproject/bar.c
A    myproject/Makefile
…
Checked out revision 1.
$
```

现在你就有了一份仓库部分数据的个人拷贝，即新创建的目录 *myproject*．你可以编辑工作副本里的文件，然后把 它们提交到仓库中．

- 进入工作副本并修改文件．
- 执行 `svn diff` 查看修改的标准差异输出．
- 执行 `svn commit` 把修改后的文件提交到 仓库中．
- 执行 `svn update`，把工作副本 “更新” 到仓库的最新版本．

至于用户能在工作副本中完成哪些工作，见 第 2 章 基本用法．

到这里为止，你可以选择把自己的仓库通过网络共享给其他人，阅读 第 6 章 服务器配置，查看有哪些服务器进程可供使用，以及 如何配置它们．

附录 B. 针对 CVS 用户的 Subversion 介绍

本附录是一篇针对 CVS 用户的 Subversion 介绍，实际上是从较高的层次 介绍 Subversion 与 CVS 的区别. 在每一小节，（如果可能的话）我们都会给出 相关章节的链接.

虽然 Subversion 的目标是完全取代 CVS，但是为了克服 CVS 的某些 “缺陷”，新的特性和设计上的变化要求 CVS 用户必须改掉某些 习惯—否则的话，CVS 用户在刚开始学习 Subversion 时会感到某些方面 有点奇怪.

版本号的编号不再相同

在 CVS 中，版本号的编号是特定于每个文件的，这是因为 CVS 把数据 存放在 RCS 文件中，每个文件在仓库中都有一个对应的 RCS 文件，而且 仓库的目录结构基本上等同于项目的目录结构.

而在 Subversion 中，仓库看起来像是一个单一的文件系统，每次提交 都会生成一棵全新的文件系统树，从本质上讲仓库就是文件系统树组成的数 组. 每一棵文件系统树都由一个单一的版本号进行标记，当人们在谈论 “版本号 54” 时，其实他们谈论的是一棵特定的文件系统树（或者说在第 54 次提交后的文件系统）.

从技术上讲，“文件 *foo.c* 的版本号 5” 是不正确的说法，正确的说法是 “在版本号为 5 时，文件 *foo.c*”. 另外，在假定文件的演变时还要注意，在 CVS 中，*foo.c* 的版本号 5 和版本号 6 肯定是不同 的，但是在 Subversion 中，从版本号 5 到版本号 6，*foo.c* 可能根本 没 发生什么变化.

类似的，在 CVS 中，标签和目录只不过是文件或文件版本信息上的注释，而对于 Subversion，标签和分支是整棵文件系统树的拷贝（按照惯例，标签和 分支通常放在仓库顶层目录的 *tags/* 和 *branches/* 子目录内，和 *tags/*，*branches/* 同级的目录还有 *trunk/*）. 在整个仓库中，每个文件的多个版本都是可见的：每个分支的最新版，每个标签，当然还有主干的最新版. 因此，为了让术语更加准确，人们经常会说成 “在版本号为 5 时，分支 *branches/REL1* 里的文件 *foo.c*”.

关于版本号的更多内容，见 “版本号” 一节.

目录的版本

Subversion 跟踪的是整棵目录树的变化，而不仅仅是文件的内容，这是 用 Subversion 替换 CVS 的一个重大理由.

对于前 CVS 用户而言，能够对目录进行版本控制意味着：

- 命令 *svn add* 和 *svn delete* 可以对目录进行操作，就像它们操作普通文件那样，*svn copy* 和 *svn move* 同样如此. 但是这些命令 不会 导致仓库马上被修改，项目的添加或删除 只是被排到了日程表中，只有在执行完 *svn commit* 后，仓库才会发生变化.
- 目录不再是一个哑容器，它和文件一样具有版本号.（因此我们可以 放心地说 “在版本号为 5 时，目录 *foo/*.”）

关于最后一点我们再多说一些. 对目录进行版本控制是一件很麻烦的 事，因为我们希望工作副本能够支持混合的版本号，这就使得我们在对目录 进行版本控制时有些限制需要遵循.

从理论上讲，我们把 “目录 *foo* 的版本号 5” 的涵义定义成目录项和属性的特定集合. 现在假设我们从目录 *foo* 里添加并删除了一些文件，然后提交，这时候 如果再说 “我们仍然拥有目录 *foo* 的版本号 5” 是不对的. 再者，如果我们在提交后更新

了目录（仅 目录）*foo* 的版本号，这时候再说 “我们 仍然拥有目录 *foo* 的版本号 5” 还是不对，因为目录中可能还有其他一些文件的更新还没有收到.

Subversion 解决该问题的方法是在目录 *.svn* 内 安静地跟踪已提交的添加和删除. 当用户最终执行 `svn update` 时，所有的信息都将与仓库同步，目录的新版本号也 会被正确地设置. 因此，只有在更新后才能放心地说你已经拥有了 一个具有 “完美” 版本号的目录. 在大多数时候，你的工作目录总是存在 “不完美” 的目录版本号.

类似的，当你试图提交目录的属性修改时可能会引起另一个问题. 通常情况 下，提交操作将会更新目录的本地版本号，但这仍然是在撒谎，因为还未执行更新 操作，因此可能存在目录项的添加和删除未在目录中体现出来. 因此，只有在目录完全更新之后，你才能提交目录的属性修改.

关于对目录进行版本控制而产生的限制，其更详细的讨论见 “版本号混合的工作副本”一节.

更多的无连接操作

最近几年，磁盘的存储空间越来越大，价格也更加便宜，但网络带宽却 并非如此，因此 Subversion 的工作副本专门针对贫乏的网络资源进行了优化.

Subversion 工作副本的管理目录 *.svn* 与 CVS 的管理目录 *CVS* 具有相同的目的，但有所不同的是 *.svn* 还存放着文件的只读 “原始” 副本，这就允许用户在离线的情况下执行多种操作：

svn status

显示本地修改（见 “查看修改的整体概述”一节）

svn diff

显示本地修改的细节（见 “查看修改的细节”一节）

svn revert

撤消本地修改（见 “修正错误”一节）

另外，缓存在本地的原始文件允许 Subversion 客户端在提交时只发送 差异部分，而这是 CVS 所不支持的.

相对于 CVS，上面列表中的最后一个子命令—*svn revert*—是一条全新的子命令. 它不仅可以撤消本地修改，还能撤消未提交的添加和删除. 虽然在删除后再执行 `svn update` 可以达到同样的效果，但是这样做实际上已经曲解了 命令 *svn update* 原本的功能.

状态与更新的区别

Subversion 尽最大的努力去消除 *cvs status* 与 *cvs update* 之间的混乱关系.

命令 *cvs status* 包含两个目的：第一是显示工作 副本里的本地修改，第二是显示哪些文件是过时的. 但不幸的是，由于 *cvs status* 的输出信息难以阅读，因此很多用户根本不 会用它，而是渐渐地养成了一种习惯，那就是用 *cvs update* 或 *cvs -n update*

来替代 *cvs status*. 如果用户忘记加上选项 `-n`, 那么 *cvs update* 将会把仓库中的修改合并到本地, 而用户此时可能还没有 准备好合并.

为消除这种混乱的状况, Subversion 让 *svn status* 的输出信息既容易被人阅读, 也容易被程序解析. 而且 *svn update* 只打印被更新的文件信息, 不再 输出与本地修改有关的信息.

状态

svn status 会输出所有的具有本地修改的文件, 而且在默认情况下, 它不会去访问仓库. 该命令可授受的选项比较多, 但 其中最常用的选项有以下这些:

`-u`

访问仓库, 从而判断出工作副本中的哪些文件已经过时了, 并 输出这些过时的文件.

`-v`

显示 所有的, 被版本控制的项目.

`-N`

非递归执行 (即不要对子目录执行操作).

命令 *svn status* 的输出格式有 2 种, 默认是 “短” 格式, 此时含有本地修改的文件将显示成:

```
$ svn status
M       foo.c
M       bar/baz.c
```

如果添加了选项 `--show-updates` (`-u`), 将使用 “长” 格式:

```
$ svn status -u
M             1047   foo.c
        *     1045   faces.html
        *            bloo.png
M             1050   bar/baz.c
Status against revision:   1066
```

在上面的例子中新出现了两列. 如果文件或目录已经过时, 那么输出中的第 2 列将显示一个星号; 第 3 列是项目在工作副本中的版本号. 从输出中可以看 到, 如果我们更新工作副本, *faces.html* 将会被更新, 而 *bloo.png* 则是仓库中新增的一个文件. (*bloo.png* 的左边没有出现版本号明该文件在工作副本 中还不存在.)

关于 *svn status* 的更多介绍, 包括如何理解 输出信息中的状态码, 见 “查看修改的整体概述”一节.

更新

命令 *svn update* 会更新你的工作副本, 并输出 与被更新文件有关的信息.

Subversion 把 CVS 的 `P` 和 `U` 这两个状态码合并成一个状态码, `U`. 如果在更新时发生 合并或冲突, Subversion 将只打印状态码 `G` 或 `C`, 而不是一整句话.

关于 *svn update* 的更多介绍，见 “更新工作副本”一节.

分支与标签

Subversion 并不区分文件系统空间与 “分支” 空间，分支与 标签仅仅是文件系统中的普通目录而已. 这大概是 CVS 用户需要跨越的最大的 心理障碍. 关于分支与标签的更多信息，见 第 4 章 分支与合并.

由于 Subversion 把分支和标签当成普通的目录看待，项目的各个开发 线通常位于项目根目录的子目录内，因此用户在为特定的开发线进行开发时，需要使用开发线对应的子目录 URL 进行检出，而不是项目的根 URL. 如果 用户不小心检出了项目的根目录，那么用户将得到一个具有项目全部内容的 工作副本，其中包括了所有的分支与标签. [1]

元数据属性

相对于 CVS, Subversion 的一项新特性是允许用户在文件和目录上添加 任意的元数据（或者说 “属性”). 属性是一对任意的 名字/值 组合，它们可以被关联到工作副本的文件与目录上.

为了设置或获取一个属性的值，要用到命令 *svn propset* 和 *svn propget*. 为了得到某个文件或目录上的所有属性，使用 *svn proplist*.

关于属性的更多信息，见 “属性”一节.

冲突解决

CVS 使用内联的 “冲突标记” 来标记冲突，并在更新或 合并过程中打印状态码 `C`. 从历史角度来看，这种做法 造成了很多问题，因为 CVS 做得还不够多. 很多用户会忘记（或者根本不看）在终端上匆匆闪过的 `C`, 导致文件里的冲突标记还没被清除就把文件给提交了.

Subversion 通过两种方式解决这一问题. 首先，如果文件发生了冲突，Subversion 将把文件置于冲突状态下，除非用户显式地清除文件的冲突状态，否则的话 Subversion 将禁止用户提交修改. 第二，Subversion 提供了交互式的 冲突解决过程，它允许用户在冲突发生时就开始处理冲突，而不是在更新与合并 全部做完后，再回过头来处理冲突. 关于冲突解决的更多信息，见 “解决冲突”一节.

二进制文件与转换

在通常情况下，与 CVS 相比，Subversion 能够更加高效地处理二进制文件，因为 CVS 使用了 RCS, 它只能存放被修改的二进制文件的完整副本. 而对于 Subversion 而言，无论文件是否包含二进制数据，它都能使用二进制差异算法 表示文件之间的差异，这就意味着所有被修改的文件在仓库中都只存放差异 部分（而且还是压缩过的）.

为了防止二进制文件发生错乱（由于关键字展开和行结束符转换），CVS 用户必须为它们指定选项 `-kv`, 但用户有时候会忘记这件 事.

[1] 假设在检出完成 前，磁盘的存储空间不会被耗尽.

Subversion 采取了更为偏执的路线. 首先, 除非用户显式地要求 Subversion, 否则的话它不会执行任何关键字展开或行结束符转换 (更多的 信息见 “关键字替换” 一节 和 “行结束标记” 一节). 默认情况下, Subversion 把所有文件的内容都看成是字节序列, 而且在把文件存放到仓库 中时不会做任何转换.

第二, 对于某个文件是 “文本” 文件, 还是 “二进制” 文件, Subversion 有它自己的一套内部概念, 但这套 概念 只在 工作副本中才会显现出来. 执行 *svn update* 时, Subversion 将为含有本地修改的文本文件执行基于上下 文的合并, 但不会对含有本地修改的二进制文件做相同的合并操作.

为了判断是否可以对某个文件执行基于上下文的合并, Subversion 将查看 文件的 `svn:mime-type` 属性. 如果文件没有设置属性 `svn:mime-type`, 或者属性值表示文件的 MIME 类型是 文本的 (例如 `text/*`), Subversion 就会认为它是个文本文件; 否则的话则认为它是二进制文件. 当用户执行 *svn import* 和 *svn add* 时, Subversion 也会 通过自己的二进制检测算法来判断文件的 MIME 类型, 如果 Subversion 猜出 了文件的 MIME 类型, 它 (可能) 会自动地为新增的文件设置 `svn:mime-type` 属性值. (如果 Subversion 猜错了, 用户总是可以删除或手动设置 `svn:mime-type` 属性.)

版本化的模块

与 CVS 不同, Subversion 的工作副本能够意识到它是否检出了一个 模块, 这就意味着如果有人修改了模块的定义 (例如添加或删除了某些组件), 执行 *svn update* 将使得工作副本相应地添加或删除那些 组件.

Subversion 把模块定义成目录属性内的一系列目录列表, 见 “外部定义” 一节.

认证

由于 CVS pserver 的要求, 用户在读写仓库—甚至包括匿名操作— 前必须先登录服务器 (使用命令 *cvs login*). 而 Subversion 使用的服务器程序是 Apache *httpd* 或 *svnserve*, 在一开始用户不用提供任何凭证, 只有在用户的 操作要求认证时, 服务器才会要求用户提供凭证 (这些凭证可以是用户名和密码, 客户端证书, 或两者都要提供). 所以说如果你的仓库对外是可读的, 那你在执行 读操作时就不必认证.

和 CVS 一样, Subversion 仍然会在本地 (具体的位置是 *~/.subversion/auth/*) 缓存用户的证书, 除非用户显式 地用选项 `--no-auth-cache` 告诉 Subversion 禁止缓存 证书.

然而这里有个例外. 如果 *svnserve* 运行在 SSH 隧道 之上, 访问仓库的 URL 模式将会变成 `svn+ssh://`, 这时候 *ssh* 程序将会无条件地要求用户认证, 仅仅是为了启动隧道.

把 CVS 仓库转换成 Subversion 仓库

为了让 CVS 用户熟悉 Subversion, 或许最重要的方法就是让他们使用 Subversion 继续原来的工作. 虽然可以简单地把 CVS 仓库中的最新数据直接 导入到 Subversion 仓库里, 但最好的做法是同时也把全部的修改历史导入到 Subversion 仓库. 这种问题解决起来非常困难, 因为它涉及到在缺少原子性 的前提下推断出变更集, 而且还要在两种完全正交的分支策略之间完成转换, 其中还会出现各种复杂的情况. 不过, 仍然存在一些工具可以部分支持从 CVS 仓库到 Subversion 仓库的转换.

最流行 (并且最成熟) 的转换工具是 cvs2svn (http://cvs2svn.tigris.org/), 它由 Subversion 社区成员使用 Python 开发而成. 这个工具只需运行一次: 它会多次扫描 CVS 仓库, 尽最大 的努力去推断出提交, 分支和标签. 命令执行结束后, 最终得到的是一个 Subversion 仓库或可移植的 Subversion 转储文件. 关于命令的详细用法 和注意事项, 请浏览官网.

附录 C. WebDAV 与自动版本控制

WebDAV 是一个 HTTP 扩展，作为一种文件共享标准，它正变得越来越流行. 现在的操作系统越来越注重网络方面的支持，并且很多系统都把挂载 WebDAV 服务器导出的目录作为内建功能.

如果你使用 Apache 作为 Subversion 的服务器，那么这同时也意味着 你使用了 WebDAV 服务器. 本附录将介绍 WebDAV 协议的背景知识，Subversion 如何使用它，以及 Subversion 如何与支持 WebDAV 的其他软件进行互操作.

什么是 WebDAV?

DAV 表示 “分布式创作与版本控制” (Distributed Authoring and Versioning). RFC 2518 为 HTTP 1.1 定义了 一系列的概念和相应的扩展，使得 Web 成为一个更加通用的读写媒介. RFC 2518 的基本思想是支持 WebDAV 的服务器看起来就像是一个普通的文件服务器，客户端可以 “挂载” 基于 HTTP 协议的共享目录，这些共享目录 就像其他网络文件系统那样（例如 NFS 和 SMB).

但不幸的是，撇开名字不说，RFC 2518 并没有介绍任何形式的版本控制，最基本的 WebDAV 客户端和服务器都假设文件或目录只存在一个版本，而且可以 被重复写入.

由于 RFC 2518 没有介绍与版本控制相关的内容，几年后，另一个委员会 开始负责起草 RFC 3253. 新的 RFC 为 WebDAV 添加了与版本控制相关的内容，在 “DAV” 后面加上了 “V”—于是有了术语 “DeltaV”. WebDAV/DeltaV 客户端和服务器经常被称作 “DeltaV” 程序，因为 DeltaV 是 WebDAV 的超集.

原始的 WebDAV 标准获得了广泛的支持. 每一个现代化的操作系统都自带了 一个通用的 WebDAV 客户端（后面会谈到细节），还有很多流行的独立应用程序 都支持 WebDAV—Microsoft Office, Dreamweaver, 和 Photoshop. 在服务器端，Apache HTTP 服务器从 1998 年起就开始提供 WebDAV 服务，除 了开源软件，还有商业 WebDAV 服务器可供使用，例如 Microsoft 的 IIS.

但不幸的是，DeltaV 就没这么成功. 支持 DeltaV 的客户端或服务器 非常少，即使有，也是不怎么知名的商业产品，而且在不同产品之间进行互操作非 常困难. DeltaV 如此不受待见的原因不是非常清楚，但有人觉得这是因为规范实 在太复杂了，还有人觉得是 WebDAV 的功能已经足够吸引人了（即使是最不懂技术 的用户也非常喜欢网络文件共享功能），对于大多数用户而已，版本控制功能并 不是非常有必要，还有人觉得是因为还未出现支持 DeltaV 的开源服务器.

当 Subversion 还处于设计阶段时，设计人员就觉得把 Apache 作为网络 服务器是一个好主意，那时它已经具备了一个可以提供 WebDAV 服务的模块. DeltaV 在当时是一个相对较新的规范，开发人员希望 Subversion 的服务器 模块 (*mod_dav_svn*) 最终可以演变成 DeltaV 的开源实现. 但不幸的是，DeltaV 有一个非常明确的版本控制模型，而这个模型与 Subversion 并非完全契合，有些概念可以互相映射，但有些不行.

那么这会导致什么后果?

首先，Subversion 客户端并非是一个 DeltaV 客户端的完整实现. DeltaV 无法提供 Subversion 客户端所需的某些特定信息，后者非常依赖于 Subversion 特有的 HTTP `REPORT` 请求，而这些请求只有 *mod_dav_svn* 能够支持.

然后，*mod_dav_svn* 也不是一个 DeltaV 服务器的完整 实现. DeltaV 规范中的很多内容对 Subversion 没什么用，于是 *mod_dav_svn* 就没有实现它们.

对于是否要去纠正上面所说的两种结果, Subversion 开发者社区最终达成了一致, 他们正式宣布放弃完全支持 DeltaV 的计划. 在 Subversion 1.7, 客户端和服务器引入了大量的, 不标准的 DeltaV 简化实现[1] 以后可能还会出现更多的定制化实现. 新版 Subversion 仍然会继续提供旧版 Subversion 已有的 DeltaV 功能, 但不会继续增加对 DeltaV 标准规范的支持—Subversion 已经放弃把严格的 DeltaV 作为它的主要的, 基于 HTTP 的协议.

自动版本控制

虽然 Subversion 客户端和服务器都不是一个完整的 DeltaV 客户端和服务器, 但仍然存在值得人们高兴的 WebDAV 互操作性: 自动版本控制 (*autoversioning*).

自动版本控制是 DeltaV 标准定义的可选功能. 如果客户端尝试在一个 被版本控制的文件上执行 `PUT` 操作, 一个典型的 DeltaV 服务器将会拒绝这个操作. 为了修改被版本控制的文件, 服务器希望收到一系列 合理的版本控制请求, 就像 `MKACTIVITY`, `CHECKOUT`, `PUT` 和 `CHECKIN`. 但是, 如果 DeltaV 服务器支持自动版本控制, 来自 WebDAV 客户端的写请求将会被接受, 服务器的表现就像是客户端 已经 发送了一系列的版本控制请求, 并且在底层执行 了提交操作. 换句话说, 自动版本控制允许 DeltaV 服务器与普通的, 不了解 版本控制的 WebDAV 客户端实现互操作.

由于很多操作系统已经内建了 WebDAV 客户端, 因此自动版本控制对于 经常和非技术用户打交道的管理员来说非常有吸引力. 假设在一个由普通人组成 的办公室中, 人们都是使用 Microsoft Windows 和 Mac OS, 每个人都 “挂载” 了 Subversion 仓库, 看起来就是一个普通的网络共享 目录, 他们可以按照和原来一样的步骤使用共享目录: 打开文件, 编辑文件, 然后保存. 同时, 版本控制将由服务器自动完成. 管理员 (或懂得技术的用户) 仍然可以使用 Subversion 客户端来搜索修改历史和检索数据的旧版本.

上面所说的场景并非虚构—它是真实存在的, 而且工作得很好. 为了 开启 *mod_dav_svn* 的自动版本控制功能, 在 Apache 配置文件 *httpd.conf* 的 `Location` 配置块里, 加上配置指令 `SVNAutoversioning`, 就像:

```
<Location /repos>
  DAV svn
  SVNPath /var/svn/repository
  SVNAutoversioning on
</Location>
```

当 Subversion 的自动版本控制功能开启后, 来自 WebDAV 客户端的写 请求将会生成自动的提交, 其日志消息也是自动生成.

在开启自动版本控制之前, 管理员需要明白这会导致什么后果. WebDAV 客户端倾向于执行 “很多” 写请求, 这会产生更多的自动提交. 例如, 保存文件时, 许多 WebDAV 客户端将先做一次空文件的 `PUT` 操作 (为了占住文件名), 接下来再为真正的文件 内容做另一次 `PUT` 操作, 一次单独的写操作将生成两 次分开的提交. 同时还要考虑到很多应用程序每隔几分钟, 就会自动保存一次, 这会产生更多的自动提交.

如果管理员使用钩子 post-commit 来发送邮件, 那么他可能想要完全禁止 或者只在仓库的某些部分上禁止邮件通知, 这取决于管理员是否还觉得大量 涌入的通知邮件仍然值得阅读. 另外, 一个聪明的 post-commit 应该能够 区分某个提交是由自动版本控制生成的, 还是由通常的 Subversion 提交操作 生成的, 区分的技巧是查看版本号属性 `svn:autoversioned`, 如果该属性存在, 则说明该版本号是由 WebDAV 客户端导致的.

[1] Subversion 开发人员把这种 DeltaV 标准的变体非正式地称为 “HTTPv2”.

为了使 Subversion 的自动版本控制更加完整，需要用到 Apache 的 *mod_mime* 模块. 如果 WebDAV 客户端为仓库添加了一个 新文件，用户将没有机会设置新文件的 `svn:mime-type` 属性，这可能会导致当用户在 WebDAV 共享目录中浏览文件时，文件的图标 是一个通用图标，不与任何打开程序关联. 一种解决办法是由管理员（或懂得 技术的用户）检出一个工作副本，然后手工修改文件的 `svn:mime-type` 属性. 但这种做法似乎很麻烦，特别是当 文件数量特别巨大时，更好的做法是在 Apache 配置文件 *httpd.conf* 的 `Location` 配置块里，加上配置指令 `ModMimeUsePathInfo`:

```
<Location /repos>
  DAV svn
  SVNPath /var/svn/repository
  SVNAutoversioning on

  ModMimeUsePathInfo on

</Location>
```

当新文件通过自动版本控制被添加到仓库中时，配置指令 `ModMimeUsePathInfo` 允许 *mod_mime* 模块去尝试推导新文件的 MIME 类型. 为了推导文件的 MIME 类型，*mod_mime* 模块会去查看文件的推展名和内容，如果符合 某种常见模式，将自动设置 `svn:mime-type` 属性.

客户端互操作性

所有的 WebDAV 客户端实现都可以分为三类—独立的应用程序，文件浏览器 扩展和文件系统实现. 这三种类型基本上规定了用户可享受到的 WebDAV 功能 类型. 表 C.1 “常见的 WebDAV 客户端软件” 给出几种 WebDAV 客户 端软件的分类，以及简短的介绍. 读者可以在下面的章节里看到关于这些软件更 详细的信息.

表 C.1. 常见的 WebDAV 客户端软件

软件	类型	Windows	Mac	Linux	描述
Adobe Photoshop	独立的 WebDAV 应用程序	X			图片编辑软件，支持直接读写由 WebDAV URL 指定的文件
cadaver	独立的 WebDAV 应用程序		X	X	命令行 WebDAV 客户端，支持文件浏览，目录树操作，以及加锁，解锁
DAV Explorer	独立的 WebDAV 应用程序	X	X	X	Java GUI 工具，用于浏览 WebDAV 共享目录
Adobe Dreamweaver	独立的 WebDAV 应用程序	X			Web 开发软件，支持直接读写由 WebDAV URL 指定的文件

软件	类型	Windows	Mac	Linux	描述
Microsoft Office	独立的 WebDAV 应用程序	X			办公套件，其中的几个组件支持直接读写由 WebDAV URL 指定的 文件
Microsoft Web Folders	文件浏览器 WebDAV 扩展	X			GUI 文件浏览器程序，支持在 WebDAV 共享目录上执行目录树 操作
GNOME Nautilus	文件浏览器 WebDAV 扩展			X	GUI 文件浏览器程序，支持在 WebDAV 共享目录上执行目录树 操作
KDE Konqueror	文件浏览器 WebDAV 扩展			X	GUI 文件浏览器程序，支持在 WebDAV 共享目录上执行目录树 操作
Mac OS X	WebDAV 文件系统实现		X		支持挂载 WebDAV 共享目录的操作系统.
Novell NetDrive	WebDAV 文件系统实现	X			驱动器映射程序，用来给挂载的远程 WebDAV 共享目录分配 Windows 驱动器盘符
SRT WebDrive	WebDAV 文件系统实现	X			文件传输软件，另外它还可以用来为挂载的远程 WebDAV 共享 目录分配 Windows 驱动器盘符
davfs2	WebDAV 文件系统实现			X	Linux 文件系统驱动程序，支持挂载 WebDAV 共享目录

独立的 WebDAV 应用程序

一个 WebDAV 应用程序是一个可使用 WebDAV 协议与 WebDAV 服务器进行 通信的程序. 下面我们将介绍最常见的几种 WebDAV 应用程序.

[2]由于某种 原因，Microsoft Access 移除了对 WebDAV 的支持，但 Office 套件内的其他程序仍然支持 WebDAV.

Microsoft Office, Dreamweaver, Photoshop

在 Windows 平台上，有几款非常知名的应用程序已经集成了 WebDAV 的客户端功能，例如 Microsoft Office,[2] Adobe Photoshop 和 Dreamweaver. 它们都支持直接读写由 WebDAV URL 指定的 文件，而且在编辑文件时，它们都倾向于大量使用 WebDAV 锁.

注意，虽然上面所说的几款应用程序也支持 Mac OS X 平台，但 它们不一定直接支持 WebDAV. 实际上，Mac OS X 的 File→Open 对话框根本就不允许用户输入 路径或 URL. 自从 OS X 从底层文件系统上支持 WebDAV 后，Macintosh 的开发人员就有意地从程序中删除了对 WebDAV 的支持.

cadaver, DAV 浏览器

cadaver 是用于浏览和修改 WebDAV 共享目录的 Unix 命令行程序，它使用 neon HTTP 函数库—不用过于惊讶，毕竟 neon 和 cadaver 的作者是同一个人. cadaver 是一个免费软件 (GPL 授权)，下载地址是 http://www.webdav.org/cadaver/.

使用 cadaver 的体验类似于使用命令行形式的 FTP 程序，因此它非常 适合用来完成基本的 WebDAV 调试. 它支持上传或下载文件，查看属性，复制和移动文件，以及对文件进行加锁和解锁.

```
$ 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 是另一款独立的 WebDAV 应用程序，使用 Java 语言 编写. 它使用的是免费的类 Apache 授权，下载地址是 http://www.ics.uci.edu/~webdav/. 它支持 cadaver 的所有 功能，并且可移植性更强，具有更加友好的 GUI 界面. 它还是第一个 支持新的 WebDAV 访问控制协议 (WebDAV Access Control Protocol, RFC 3744) 的客户端程序.

当然，DAV Explorer 对 ACL 的支持在这里没什么用，因此 *mod_dav_svn* 根本就不支持 ACL. 实际上，cadaver 和 DAV Explorer 支持的某些 DeltaV 命令其实没什么特别大 的用处，因此它们不允许 `MKACTIVITY` 请求. 但 这些都无关紧要，我们假充所有的 WebDAV 客户端都针对一个自动版本 控制的仓库进行操作.

文件浏览器 WebDAV 扩展

有些流行的文件浏览器 GUI 程序支持 WebDAV 扩展功能，允许用户 浏览一个 DAV 共享目录，就像在浏览一个普通的本地目录，而且还可以在 共享目录中执行基本的目录树编辑操作. 例如，Windows 文件浏览器可以把 WebDAV 服务器作为

一个 “网络位置” 进行浏览，用户可以 通过拖拽来上传或下载文件，还可以继续使用通常的方式来重命名，复制或 删除文件. 但是由于这仅仅是文件浏览器的功能，DAV 共享目录对其他普通 的应用程序仍然是不可见的，所有的 DAV 交互都必须通过文件浏览器接口.

Microsoft Web Folders

Microsoft 是 WebDAV 的原始赞助商之一，最早在 Windows 98 中 集成了 WebDAV 客户端，即 Web Folders. 该客户端同样出现在 Windows NT 4.0 和 Windows 2000 中.

原始的 Web Folders 其实是一个文件浏览器扩展，但它工作得已经 足够好了. 在 Windows 98, 如果在 “我的电脑” 中看不到 Web Folders, 则还要显式地安装它. 在 Windows 2000, 只需要添加一个新的 “网络位置”，输入 URL, WebDAV 共享目录就会马上跳 出来.

到了 Windows XP, Microsoft 决定推出一款新的 Web Folders 程序，即 WebDAV Mini-Redirector. WebDAV Mini-Redirector 是一个处于文件 系统层的客户端，允许 WebDAV 共享目录被挂载成一个驱动器盘符. 但 不幸的是，这个程序的问题太多. WebDAV Mini-Redirector 通常情况下 会试图把 HTTP URL (`http://host/repos`) 转换成 UNC 共享记号 (`\\host\repos`), 它还经常用 Windows Domain 认证来响应 HTTP 认证请求，把用户名转换成 `HOST\username`. 这些互操作问题非常严重，在网上随处可见，很多用户都不堪其扰. Apache WebDAV 模块的原始作者 Greg Stein 就直言不讳地说 XP Web Folders 根本就不能支持 Apache 服务器.

Windows Vista 的 Web Folders 最初版本几乎和 XP 一样，所以它也 相同的问题，但幸运的是，Microsoft 将会在 Vista Service Pack 里 修复它们.

然而，似乎还是有一些方法能够让 XP 和 Vista 的 WebDAV 客户端支持 Apache 服务器，很多用户都表示他们成功地解决了问题，于是我们打算在 这里简单地介绍一下这些方法.

如果是 Windows XP 你就有两个选择，第一个是在 Microsoft 的网站 搜索 KB907306—Web Folders 软件更新. 它可能会解决你的所有 问题，如果没有，那应该是因为系统中的 Web Folders 还是 XP 前的版本. 为了进一步挖掘下去，你需要到 Network Places 那儿添加一个新的网络 位置，在弹出的对话框中输入仓库的 URL，同时还要在 URL 中 *加上端口号*，例如，你应该把 `http://host/repos` 写成 `http://host:80/repos`, 如果需要认证，就输入 Subversion 的凭证.

如果是 Windows Vista，相同的 KB907306 更新可能会解决所有问题，但也有可能仍有问题存在. 有些用户报告说 Vista 会把所有的 `http://` 看成是不安全的连接， 除非使用 `https://`，否则的话 Vista 会让所有的来自 Apache 的认证请求都以失败返回. 如果用户无法通过 SSL 连接 Subversion 仓库， 可以通过修改系统注册表来禁止 Vista 的这种行为，修改的方法是把 `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\WebClient\Parameters\BasicAuthLevel` 的值由 `1` 改成 `2`. 最后再提醒一 点，要让 Web Folders 指向仓库的根目录，而不是仓库内的子目录（例如 */trunk*), 因为 Vista Web Folders 似乎 只有在面对仓库的根目录时才能正常工作.

通常情况下，上面所说的方法可能会解决你的问题，但使用第三方的 WebDAV 客户端程序（例如 WebDrive 或 NetDrive) 可能会获得更好的 体验.

Nautilus, Konqueror

Nautilus 是 GNOME 桌面环境 (http://www.gnome.org) 标准的文件管理与浏览器，而 Konqueror 是 KDE 桌面环境 (http://www.kde.org) 的文件管理与浏览器，它们都内 建了一个文件浏览器级别的 WebDAV 客户端，而且对自动版本控制的仓库支持得很好.

在 GNOME 的 Nautilus 里，选择菜单项 File→Open location，在弹出的对话框中输入仓库的，然后仓库就 能像其他文件系统那样展现出来.

在 KDE 的 Konqueror 里，在地址栏输入 URL 时，必须使用 webdav:// 模式. 如果用户输入了 http:// 模式的 URL，Konqueror 将表现得如同一个 网页游览器，用户将看到由 *mod_dav_svn* 生成的 HTML 目录列表. 如果输入的是 webdav://host/repos，而不是 http://host/repos，Konqueror 将变成一个 WebDAV 客户端，把仓库按照文件系统的形式展现出来.

WebDAV 文件系统实现

按理说，WebDAV 文件系统实现应该算作最优秀的一类 WebDAV 客户端. 它被实现成一个底层的文件系统模块，通常位于操作系统的内核里，这就意味 着 DAV 共享目录可以像其他网络文件系统那样被挂载，类似于 Unix 的 NFS 或 Windows 的 SMB. 这种 WebDAV 客户端为所有的程序提供了完全透明 的 WebDAV 读写访问，应用程序甚至不会意识到 WebDAV 请求正在生成.

WebDrive, NetDrive

WebDrive 和 NetDrive 是两款非常优秀的商业软件，允许 WebDAV 共享目录作为一个驱动器盘符挂载到 Windows 系统中. 它们允许用户 像访问真实的本地磁盘驱动器那样访问基于 WebDAV 的伪驱动器，而且 操作起来不会有任何区别. 为了购买 WebDrive，读者可以咨询 South River Technologies 公司 (http://www.southrivertech.com)，Novell 的 NetDrive 可 以从网上免费获取，但要求用户必须具备一个 NetWare 许可证.

Mac OS X

Apple 公司的 OS X 操作系统集成了一个文件系统级别的 WebDAV 客户端. 从 Finder 开始，选择菜单项 Go→Connect to Server，然后输入 WebDAV 的 URL，最终 WebDAV 共享目录将作为一个磁盘出现在桌面上，看起来和其他已挂载的卷没什么两样. 用户还可以从 Darwin 终端上执行 挂载操作，方法是在 *mount* 的命令行参数上把文件 系统类型指定成 webdav:

```
$ mount -t webdav http://svn.example.com/repos/project /some/mountpoint
$
```

注意，如果 *mod_dav_svn* 的版本比 1.2 旧，OS X 将不允许以读写方式挂载共享目录，而是以只读方式挂载. 这是 因为 OS X 坚持要为可读写的共享目录提供锁支持，而最早支持锁的 Subversion 版本是 1.2.

另外，OS X 的 WebDAV 客户端有时候会对 HTTP 重定向产生过度 的敏感，如果 OS X 完全无法挂载仓库，用户 需要在 Apache 服务器的 配置文件 *httpd.conf* 里开启配置指令 BrowserMatch:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Linux davfs2

Linux davfs2 是 Linux 内核的一个文件系统模块，该项目的 主页位于 http://savannah.nongnu.org/projects/davfs2. davfs2 一旦安装完成，用户就可以使用下面的命令挂载 WebDAV 共享目录:

```
$ mount.davfs http://host/repos /mnt/dav
```

附录 D. 传统的 Berkeley DB 后端存储

很久以前，当 Subversion 开始学习如何存储版本化的数据时，它的存储层 实现基于一个事务性的数据库系统—Berkeley DB（简称 BDB).[1] 随着 Subversion 的不断成熟，后端 存储又增加了一种类型—并且逐渐优于 BDB—FSFS. 如今，大部分 Subversion 仓库使用的后端存储都是 FSFS. 在 Subversion 1.8, Subversion 社区宣布 BDB 后端存储被正式弃用.

本附录将介绍如何管理基于 BDB 的仓库，这部分内容原本是本书较早版本 的重点内容之一.

配置 Berkeley DB 环境

一个 Berkeley DB 环境由一个或多个数据库，日志文件，区域文件和 配置文件封装而成. Berkeley DB 环境有一套自己的默认配置，例如在任意 时刻允许持有的数据库锁的个数，日志文件的最大大小等. Subversion 的 文件系统逻辑根据自己的需要，为某些 Berkeley DB 配置选项额外选取了默认 值. 然而，你的仓库可能存放的是非常独特的数据，而且访问模式也很特殊，这时候你可能需要一套不同的配置选项值.

Berkeley DB 的开发人员明白不同的应用程序和数据库环境具有不同的需求，所以他们提供了一种机制，支持在运行时修改 Berkeley DB 环境的众多配置选项. BDB 在环境目录（也就是仓库内的 *db* 子目录）内检查 文件 *DB_CONFIG* 是否存在，如果存在则解析该文件内 的选项.

Subversion 在创建仓库时会去创建文件 *DB_CONFIG*，初始时，文件包含了一些默认选项，以及指向 Berkeley DB 在线文档的链接，以便管理员了解选项的意义.

```
$ svnadmin create --fstype bdb /var/svn/repos
$ ls /var/svn/repos/db
changes         __db.003    __db.register  log.0000000001   revisions
checksum-reps   __db.004    format         miscellaneous    strings
copies          __db.005    fs-type        node-origins     transactions
__db.001        __db.006    locks          nodes            uuids
__db.002        DB_CONFIG   lock-tokens    representations
$
```

当然，管理员也可以往 *DB_CONFIG* 添加更多的 BDB 选项，但要注意 Subversion 从来不会去读取或解释 *DB_CONFIG* 的内容，更不会直接使用文件内的选项. 管理员需要避免的是配置上的变化可能会导致 Berkeley DB 表现出的行为 与 Subversion 所期待的不符. 另外，*DB_CONFIG* 的 变化只有在恢复数据库环境（使用 *svnadmin recover*) 时才会生效.

Berkeley DB 的限制

Berkeley DB 事务性的数据存储服务提供了能和世界级数据库系统相 媲美的数据完整性保证，但如同玫瑰上的刺，我们必须了解 Berkeley DB 的限制.

[1] 好吧，严格地说，最开始使用的后端存储是 XML 文件，但该版本从来 没有对外发布过.

体系结构上的限制

Berkeley DB 环境是不可移植的. 管理员不能简单地把在 Unix 系统中 创建的 Subversion 仓库复制到 Windows 系统中, 然后希望它能照常工作. 虽然 Berkeley DB 数据库格式的大部分都是体系结构无关的, 但仍然存在 与体系结构相关的部分.

第二, Subversion 使用 Berkeley DB 的方式在 Windows 95/98 中 无法工作—如果管理员要在 Windows 系统中创建基于 BDB 的 Subversion 仓库, 必须是 Windows 2000 或更新的版本.

网络共享目录部署

虽然 Berkeley DB 声称对于满足特定规范的网络共享目录[2], 它可以正常工作, 但大部分网络文件系统和应用程序 其实并 不满足 这些规范. 另外, 位于网络共享目录 内的, 基于 BDB 的 Subversion 不允许多个客户端同时访问 (而这却是把 仓库放在共享目录后无法避免的问题).

如果管理员试图在一个不兼容的远程文件系统中使用 Berkeley DB, 结果将是无法预测的—管理员可能会马上看到诡异的错误, 或者是 直到几个月后才发现仓库的数据库已经产生了细微的损坏. 强烈建议在 网络共享目录中使用基于 FSFS 的 Subversion 仓库.

错误容忍与恢复

由于 Berkeley DB 的库函数被静态地链接到 Subversion 中, 所以和 典型的关系数据库系统相比, Berkeley DB 对中断更加敏感. 例如大多数 SQL 系统都有一个专用的服务器进程, 负责协调对数据库表的所有访问, 如果正在 访问数据库的程序崩溃了, 数据库守护进程将会注意到断开的连接, 并清理留下 的任何杂物. 由于数据库守护进程是访问数据库表的唯一一个进程, 应用程序不 用再担心权限冲突.

然而, Berkeley DB 没有专用的进程负责协调对数据库表的访问, Subversion (和使用了 Subversion 库函数的程序) 是直接访问数据库表, 这就意味着如果程序崩溃, 数据库将临时地处于一种不一致和不可访问的状态. 如果发生了这种情况, 管理员需要请求 Berkeley DB 恢复到一个检查点, 实际做起来会有点麻烦. 除了程序崩溃, 还有些情况会造成仓库 “卡住”, 例如程序的所有者权限与数据库文件的权限相冲突.

Berkeley DB 4.4 为 Subversion (1.4 或更新的版本) 带来了自动 恢复 Berkeley DB 环境的能力. 当一个 Subversion 进程附加到仓库 的 Berkeley DB 环境上时, 它将使用进程记账机制来检测前一个进程 留下的未清理的连接, 执行必要的恢复操作, 然后继续往下执行, 就好像 什么事都没发生过. 虽然这不能完全避免仓库卡住的情况出现, 但还是 大大减少了恢复所需的人工介入.

维护 Berkeley DB 仓库

从理论上讲, 基于 BDB 的仓库的维护步骤和基于 FSFS 的仓库是一样的, 但是从历史上看, 基于 BDB 的仓库需要一点额外的关怀才能保持运转. 本节 将介绍管理 Berkeley DB 的独特之处.

[2]Berkeley DB 要求底层的文件系统实现了严格的 POSIX 锁语义, 更重要的是还要支持将文件直接映射到进程的内存中.

Berkeley DB 恢复

在 “错误容忍与恢复”一节 提到过，如果未被恰当地关闭，基于 Berkeley DB 的仓库可能会处于无法 访问的状态. 如果发生了这种情况，管理员需要把数据库回滚到一个一致的 状态. 这是基于 BDB 的仓库所特有的问题—如果仓库的后端存储是 FSFS, 则根本不会出现这种问题. 从 1.4 (Berkeley DB 版本是 4.4) 开始，Subversion 对异常情况的适应能力越来越强，但还是有可能出现仓库 被卡住的情况，这时候管理员必须知道如何安全地处理这种状况.

为了保护仓库里的数据，Berkeley DB 使用了锁机制. 锁机制保证了数据库 的各个部分不会被多个访问者同时修改，并且每个进程从数据库中读取时都能 看到处于正确状态的数据. 当一个进程需要修改数据库里的某个数据时，它首先 查看目标数据上是否存在锁，如果目标数据未被锁定，进程将会锁定数据，然后 修改数据，最后放锁. 其他进程将一直等待，直到持锁的进程放锁后，它们才能 继续访问目标数据.（这和仓库内的版本化文件上的锁一点关系也没有，我们已 在 “锁” 的多种涵义 专门向读者解释过.）

在使用 Subversion 仓库的过程中，致命的错误或中断会使得进程没 有机会去移除数据库上的锁，由此造成的结果是后端数据库系统被 “卡住” 了. 这种情况一旦发生，任何试图访问仓库的进程 都会被无限期地挂起（因为每个进程都在等待解锁，但原本要去解锁的进程 已经不存在了）.

如果你的仓库的后端数据库被卡住了，不要惊慌. Berkeley DB 文件 系统利用数据库事务，检查点和预写日志来保证只有最具灾难性的事件 才会造成数据库环境被永久地损坏[3]. 一个谨慎的仓库管理员会 经常对仓库进行备份，但现在还不用去存储柜那儿取备份，而是先按照下面 的步骤来尝试恢复仓库：

1. 先确保没有进程正在访问（或试图访问）仓库，对于连接到网络上 的仓库，这还意味着要关闭 Apache HTTP 服务器或 svnserve 守护进程.

2. 切换到拥有仓库的用户账号. 这一步很重要，如果账号不正确，在 恢复仓库时可能会改变仓库内文件的权限，使得即使仓库恢复后也无法 被其他人访问.

3. 执行命令 *svnadmin recover*:

   ```
   $ svnadmin recover /var/svn/repos
   Repository lock acquired.
   Please wait; recovering the repository may take some time...

   Recovery completed.
   The latest repos revision is 19.
   $
   ```

   命令可能会花好几分钟才会结束.

4. 重启服务器进程.

上面的步骤能够解决导致仓库卡住的大部分问题. 要确保在执行命令时 的身份是拥有仓库的用户， 而不是简单地切换到 `root` 用户. 在恢复过程中可能会从头开始创建各种数据库文件（例如共享内存区）. 如果以 `root` 身份执行命令，将导致新建的文件只属于 `root`, 这就意味着即使仓库重新上线，普通用户也没 办法访问.

[3] 例如用一个巨大的 电磁铁去接近硬盘.

清除不再有用的 Berkeley DB 日志文件

在 Berkeley DB 4.2 发布之前，对于基于 BDB 的 Subversion 仓库 而言，消耗磁盘最多的罪魁祸首就是 Berkeley DB 的日志文件，这些日志 文件是 Berkeley DB 在修改真正地数据库文件之前所写的预写日志. 预写 日志记录了数据库从一个状态到另一个状态过程中所产生的所有活动，而 数据库文件在任意一个时刻都代表了一个特定的状态，日志文件包含了 状态 之间 的所有修改，因此日志会增长地非常 迅速，消耗的磁盘存储空间也很可观.

幸运的是，从 Berkeley DB 4.2 开始，数据库环境开始支持自动删除 不再有用的日志文件. 如果 *svnadmin* 的 Berkeley DB 版本大于或等于 4.2, 那么由它创建的仓库都会把自动删除日志作为默认 配置. 如果管理员不希望日志文件被自动删除，只需要为 *svnadmin create* 加上选项 `--bdb-log-keep`. 如果 管理员忘记加上选项，或者是后面又改变了心意，只需要打开仓库子目录 *db* 里的 *DB_CONFIG* 文件，注释掉含有 `set_flags DB_LOG_AUTOREMOVE` 的那一行，然后对仓库执行 *svnadmin recover*, 使得修改后的配置 生效.

如果不支持日志文件的自动删除，随着人们不断地使用仓库，日志文件 会不断累积. 保留日志文件也可以算作数据库系统的一项特性—管理员 应该能够在只有日志文件的情况下恢复整个数据库，所以日志文件对于数据库 的灾备恢复非常有用. 但是在典型情况下，管理员希望归档不再被 Berkeley DB 使用的日志文件，然后从磁盘上删除它们，以便节省磁盘空间. 命令 *svnadmin list-unused-dblogs* 可以列出不再使用的 日志文件:

```
$ 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!
```

如果基于 BDB 的仓库把日志文件用作备份或灾备恢复计划的一部分，那么我们就 不能 允许自动删除日志文件. 根据 日志文件来恢复仓库要求 所有的 的日志都必须 留着. 如果在备份系统复制日志文件之前，日志文件被自动删除了，那么 不完整的日志文件集合对恢复仓库而言起不到半点作用.

Berkeley DB 实用工具

如果仓库的后端数据库是 Berkeley DB, 那么版本化文件系统的 所有结构和数据都存放在一系列数据库表中，而这些数据库表位于仓库 的 *db/* 子目录内. *db/* 是一 个普通的 Berkeley DB 环境目录，因此可以用 Berkeley DB 数据库工具 访问目录，数据库工具通常由 Berkeley DB 发行版提供.

在日常工作中使用 Subversion 时， 这些数据库工具都不会被用到， Subversion 仓库所需的大部分维护工作都可以由 *svnadmin* 完成，例如 Berkeley DB 实用工具 *db_archive* 的部分功能可以由 *svnadmin list-unused-dblogs* 和 *svnadmin list-dblogs* 完成，而 *svnadmin recover* 则反映了 Berkeley DB 实用工具 *db_recover* 的常见 用例.

然而，还是存在几个 Berkeley DB 实用工具可能对你有用. *db_dump* 和 *db_load* 分别用于 读写具有定制化格式的文件，该文件描述了 Berkeley DB 数据库的键值对. 由于 Berkeley DB 数据库在不同的体系结构之间是不可移植的，利用这种 格式的文件管理员就可以实现在不同的主机之间传输数据库，而不是考虑体系 结构或操作系统的差异. 虽然我们也能用 *svnadmin dump* 和 *svnadmin load* 完成类似的工作，但 *db_dump* 和 *db_load* 做得也 很好，而且快得多. 有时候，经验丰富的 Berkeley DB 用户还能利用这两个 命令在位地对仓库里的数据进行调整，而这是 Subversion 工具做不到的. 另外，*db_stat* 还能输出关于 Berkeley DB 环境的 状态信息，包括详细的，关于锁和存储子系统的统计信息.

关于 Berkeley DB 工具链的更多信息，见 Oracle 维护的 Berkeley DB 文档 https://docs.oracle.com/cd/E17275_01/html/api_reference/C/utilities.html.

附录 E. 版权


```
You are free:

    * to copy, distribute, display, and perform the work
    * to make derivative works
    * to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.

    * For any reuse or distribution, you must make clear to others the
      license terms of this work.

    * Any of these conditions can be waived if you get permission from
      the author.

Your fair use and other rights are in no way affected by the above.

The above is a summary of the full license below.

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

Creative Commons Legal Code
Attribution 2.0

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES
REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR
DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS
CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS
PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE
WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS
```

PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

   c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

   d. "Original Author" means the individual or entity who created the Work.

   e. "Work" means the copyrightable work of authorship offered under the terms of this License.

   f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

   b. to create and reproduce Derivative Works;

   c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

   d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

   e.

      For the avoidance of doubt, where the work is a musical composition:

         i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

        ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

   f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other

jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

   b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work

(e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the 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.

==========

索引

符号

A

B

C

D

E

N

O

P

R

S

T

U

V

W

X