

采用 Subversion 进行版本控制

Maven(nickrpc@gmail.com)

目 录

[前言](#)

- [目标读者](#)
- [本书结构](#)
- [排版惯例](#)
- [本书是自由的](#)
- [致谢](#)

[1. 简介](#)

- [什么是 Subversion?](#)
- [Subversion 的历史](#)
- [Subversion 的功能](#)
- [安装 Subversion](#)
- [Subversion 的组件](#)
- [客户端组件 \(供使用者使用\)](#)
- [服务器组件 \(供管理员使用\)](#)

[2. 基本概念](#)

- [档案库](#)
- [各种版本控制的模型](#)
- [档案分享的问题](#)
- [锁定-修改-解锁的解决方案](#)
- [复制-修改-合并的解决方案](#)
- [Subversion 实务](#)
- [工作复本](#)
- [修订版本](#)
- [工作复本如何追踪档案库](#)
- [混合修订版的限制](#)
- [摘要](#)

[3. 导览](#)

- [帮帮我!](#)
- [汇入](#)
- [修订版: 数字, 关键词, 与日期. 我的天啊!](#)
- [修订版号](#)
- [修订版关键词](#)
- [修订版日期](#)
- [最初的取出动作](#)
- [基本工作流程](#)

[更新工作复本](#)
[对工作复本产生更动](#)
[检视你的更动](#)
[svn status](#)
[svn diff](#)
[svn revert](#)
[解决冲突 \(合并他人的更动\)](#)
[手动合并冲突](#)
[将档案复制并盖过你的工作档](#)
[弃踢: 使用 svn revert](#)
[送交更动](#)
[检视历史纪录](#)
[svn log](#)
[svn diff](#)
[检视本地端更动](#)
[比较档案库与本地复本](#)
[档案库与档案库之间的比较](#)
[svn cat](#)
[svn list](#)
[对历史纪录的最后叮咛](#)
[其它有用的命令](#)
[svn cleanup](#)
[svn import](#)
[摘要](#)

[4. 分支与合并](#)

[何谓分支?](#)
[使用分支](#)
[建立一个分支](#)
[与分支共事](#)
[事情的内涵](#)
[在分支之间复制更动](#)
[复制特定的更动](#)
[重复合并问题](#)
[合并整个分支](#)
[从档案库移除一个更动](#)
[切换工作复本](#)
[标记](#)
[建立一个简单的标记](#)
[建立一个复杂的标记](#)
[分支维护](#)
[档案库配置](#)
[资料生命周期](#)
[摘要](#)

[5. Repository 管理](#)

[档案库的基本知识](#)

[了解异动与修订版](#)
[无版本控制的性质](#)
[档案库的建立与设定](#)

[Hook scripts](#)

[Berkeley DB 设定](#)

[档案库维护](#)

[管理员的工具箱](#)

[svnlook](#)

[svnadmin](#)

[svnshell.py](#)

[Berkeley DB 工具](#)

[档案库善后](#)

[档案库回复](#)

[汇入档案库](#)

[档案库备份](#)

[网络档案库](#)

[httpd, Apache HTTP 服务器](#)

[你需要什么, 才能设定基于 HTTP 的档案库存取](#)

[基本 Apache 设定](#)

[权限, 认证, 以及授权](#)

[服务器名称与 COPY 要求](#)

[浏览档案库的 HEAD 修订版](#)

[杂项的 Apache 功能](#)

[svnserve, 自订的 Subversion 服务器](#)

[设定匿名 TCP/IP 存取](#)

[设定使用 SSH 存取](#)

[使用哪一个服务器?](#)

[档案库权限](#)

[新增专案](#)

[选择一种档案库配置](#)

[建立配置, 汇入起始数据](#)

[摘要](#)

[6. 进阶主题](#)

[执行时期的设定区域](#)

[设定区域配置](#)

[设定与 Windows 登录档](#)

[设定选项](#)

[Servers](#)

[Config](#)

[性质](#)

[为什么要用性质?](#)

[使用性质](#)

[特殊性质](#)

[svn:executable](#)

[svn:mime-type](#)

[svn:ignore](#)
[svn:keywords](#)
[svn:eol-style](#)
[svn:externals](#)
[外部定义](#)
[供货商分支](#)
[通用供货商分支管理程序](#)
[svn-load-dirs.pl](#)

[7. Developer Information](#)

[Layered Library Design](#)
[Repository Layer](#)
[Repository Access Layer](#)
[RA-DAV \(Repository Access Using HTTP/DAV\)](#)
[RA-SVN \(Proprietary Protocol Repository Access\)](#)
[RA-Local \(Direct Repository Access\)](#)
[Your RA Library Here](#)
[Client Layer](#)
[Using the APIs](#)
[The Apache Portable Runtime Library](#)
[URL and Path Requirements](#)
[Using Languages Other than C and C++](#)
[Inside the Working Copy Administration Area](#)
[The Entries File](#)
[Pristine Copies and Property Files](#)
[WebDAV](#)
[Programming with Memory Pools](#)
[Contributing to Subversion](#)
[Join the Community](#)
[Get the Source Code](#)
[Become Familiar with Community Policies](#)
[Make and Test Your Changes](#)
[Donate Your Changes](#)

[8. 完整 Subversion 参考手册](#)

[Subversion 命令列客户端: svn](#)
[svn 选项](#)
[svn 子命令](#)
[svn add](#)
[svn cat](#)
[svn checkout](#)
[svn cleanup](#)
[svn commit](#)
[svn copy](#)
[svn delete](#)
[svn diff](#)
[svn export](#)
[svn help](#)
[svn import](#)
[svn info](#)
[svn list](#)

[svn log](#)
[svn merge](#)
[svn mkdir](#)
[svn move](#)
[svn propdel](#)
[svn propedit](#)
[svn propget](#)
[svn proplist](#)
[svn propset](#)
[svn resolved](#)
[svn revert](#)
[svn status](#)
[svn switch](#)
[svn update](#)
[svnadmin](#)
[svnadmin 选项](#)
[svnadmin 子命令](#)
[svnadmin list-unused-dblogs](#)
[svnadmin create](#)
[svnadmin dump](#)
[svnadmin help](#)
[svnadmin load](#)
[svnadmin lstxns](#)
[svnadmin recover](#)
[svnadmin rmtxns](#)
[svnadmin setlog](#)
[svnlook](#)
[svnlook 选项](#)
[svnlook author](#)
[svnlook cat](#)
[svnlook changed](#)
[svnlook date](#)
[svnlook diff](#)
[svnlook dirs-changed](#)
[svnlook help](#)
[svnlook history](#)
[svnlook info](#)
[svnlook log](#)
[svnlook proplist](#)
[svnlook tree](#)
[svnlook youngest](#)

A. 给 CVS 使用者的 Subversion 指引

[不同的修订版号](#)
[目录版本](#)
[更多不需网络的动作](#)
[区分状态与更新](#)
[分支与标记](#)
[中介资料性质](#)
[冲突排解](#)

[二进制档案与转换](#)

[Versioned Modules](#)

[B. 汇入 CVS 档案库](#)

[需求](#)

[执行 cvs2svn.py](#)

[C. 故障排除](#)

[常见问题](#)

[使用 Subversion 的问题](#)

[每当我想要存取档案库时, 我的 Subversion 客户端会停在那里.](#)

[当我想要执行 svn 时, 它就说我的工作复本被锁定了.](#)

[寻找或开启档案库时有错误发生, 但是我确定我的档案库 URL 是正确的.](#)

[我要如何在 file:// URL 中指定 Windows 的磁盘驱动器代号?](#)

[我没有办法经由网络写入数据至 Subversion 档案库.](#)

[在 Windows XP 中, Subversion 服务器有时会送出损坏的数据.](#)

[要在 Subversion 客户端与服务器进行网络传输的检查, 最好的方法是什么?](#)

[编译 Subversion 的问题](#)

[我把执行档编辑好了, 但是当我想要取出 Subversion 时, 我得到](#)

[Unrecognized URL scheme. 的错误.](#)

[当我执行 configure, 我得到像 subs-1.sed line 38: Unterminated `s' command 的错误.](#)

[我无法在 Windows 以 MSVC++ 6.0 来编译 Subversion.](#)

[D. WebDAV 与自动版本](#)

[基本 WebDAV 概念](#)

[简易 WebDAV](#)

[DeltaV 扩充](#)

[Subversion 与 DeltaV](#)

[将 Subversion 对映至 DeltaV](#)

[自动版本支持](#)

[mod dav lock 的替代品](#)

[自动版本互通性](#)

[Win32 网络数据夹](#)

[Mac OS X](#)

[Unix: Nautilus 2](#)

[Linux davfs2](#)

[E. 其它 Subversion 客户端](#)

[Out of One, Many](#)

[F. 协力厂商工具](#)

[ViewCVS](#)

[SubWiki](#)

[Glossary](#)

List of Figures

2.1. [典型的主从式系统](#)

2.2. [应避免的问题](#)

2.3. [锁定-修改-解锁的解决方案](#)

- 2.4. [复制-修改-合并的解决方案](#)
- 2.5. [...复制-修改-合并的解决方案 \(续\)](#)
- 2.6. [档案库的档案系统](#)
- 2.7. [档案库](#)
- 4.1. [发展的分支](#)
- 4.2. [起始档案库的配置](#)
- 4.3. [有新复本的档案库](#)
- 4.4. [档案历史的分支](#)
- 5.1. [一种建议的档案库配置.](#)
- 5.2. [另一种建议的档案库配置.](#)
- 7.1. [Subversion's "Big Picture"](#)
- 7.2. [Files and Directories in Two Dimensions](#)
- 7.3. [Revisioning Time—the Third Dimension!](#)

List of Tables

- 2.1. [档案库存取的 URL](#)
- 7.1. [A Brief Inventory of the Subversion Libraries](#)
- E.1. [Subversion 的图形客户端](#)

List of Examples

- 5.1. [利用 svnshell, 在档案库之中巡行](#)
- 5.2. [txn-info.sh \(回报未处理异动\)](#)
- 5.3. [使用渐进式档案库倾印](#)
- 6.1. [登录项目 \(.REG\) 档案的范例.](#)
- 7.1. [Using the Repository Layer](#)
- 7.2. [Using the Repository Layer with Python](#)
- 7.3. [A Simple Script to Check Out a Working Copy.](#)
- 7.4. [Contents of a Typical .svn/entries File](#)
- 7.5. [Effective Pool Usage](#)

前言

Table of Contents

- [目标读者](#)
- [本书结构](#)
- [排版惯例](#)
- [本书是自由的](#)
- [致谢](#)

“如果 C 给了你够多的绳子来吊死自己, 那么 Subversion 可视为是一种收纳绳子的器具.”—Brian Fitzpatrick

在开放原码软件的世界中, **Concurrent Versions System (CVS)** 长久以来, 一直都是版本控制的不二选择. **CVS** 本身是自由软件, 而且它是“非锁定式”的系统 — 这让分布广泛的程序设计人员能够分享彼此的工作 — 完全符合开放原码世界的合作模式. **CVS**, 以及它那半混乱式的发展模式, 已经成为开放原码文化的基石.

但是就像许多的工具, **CVS** 已经开始显露疲态. 比较起来, **Subversion** 是一个新的工具, 是设计来成为 **CVS** 的后继者. 设计者要以两个方法来赢得 **CVS** 使用者的心: 产生一个设计 (还有 "外观与感觉") 类似 **CVS** 的开放原码系统, 以及试着修正 **CVS** 中最广为人知的缺点. 虽然结果不见得会是版本控制设计的下一个伟大革命, 但是 **Subversion** 绝对会是个强力, 可用性高, 而且深具弹性的工具.

目标读者

本书是写给那些想要以 **Subversion** 来管理数据的计算机使用者. 虽然 **Subversion** 可以在许多不同的操作系统上执行, 不过主要的使用界面还是命令列. 由于这样的原因, 本书的例子都假设使用者使用的是类似 **Unix** 的操作系统, 而且熟悉 **Unix** 与其命令列的界面.

大多数的使用者可能是程序设计师或系统管理员, 需要追踪原始码的变更; 这是 **Subversion** 最常见的用法, 因此也是本书例子的情景. 但是请记住, **Subversion** 可以用来管理任何类似的信息: 图形, 音乐, 数据库, 文件等等. 对 **Subversion** 而言, 所有的数据就只是数据而已.

虽然本书撰写时, 我们假设读者都没有使用过版本控制软件, 但是我们也试着让 **CVS** 的使用者能够很快地上手. 一些 sidebar 会随时出现讨论 **CVS**, 而且也有一章附录用来概述 **CVS** 与 **Subversion** 之间的不同.

本书结构

本书的前三章对 **Subversion** 有概括性的介绍. 我们一开始先介绍 **Subversion** 的功能, 讨论它的设计与使用者模型, 然后进行一个导引. 不管是否有相关的经验, 所有的读者都应该读这几章的内容. 它们是本书其它章节的基础.

第四, 五, 六章讨论比较复杂的议题, 像是分支, 管理档案库 (repository), 以及进阶的功能, 像是性质, 外部定义, 以及存取控制. 系统管理员与进阶使用者绝对会希望读这几章的.

第七章是特别写给那些想要在他们的软件里使用 **Subversion** 的 API, 或者是想要修改 **Subversion** 的程序设计师.

本书最后以参考数据结束: 第八章是所有 **Subversion** 命令的参考指南, 而附录则涵盖了许多有用的主题. 这几章大概是你读完本书之后, 最常使用的部份.

排版惯例

O'Reilly almost certainly needs to fill this in, depending on how they typeset the book.

请注意这里使用的程序代码例子, 就只是一单纯的例子. 在适当的编译器设定之下, 它们都能够正常被编译, 但是只是用来示范手边的问题而已, 而不是用来作为程序设计的范例.

本书是自由的

本书起初只是 Subversion 计划的发展人员所写的文件而已, 后来就成为独立的工作, 并且重新改写. 因为如此, 这个文件也和 Subversion 一样, 使用的是自由的开放原码授权. 事实上, 本书是在公众面前写成的, 是 Subversion 的一部份. 这意味着两件事:

- 你可以在 Subversion 的源码树中, 找到本书的最新版本.
- 你可以任意散布、修改本书 — 它用的是自由授权. 当然了, 除了散布自己私有的版本, 我们比较希望你能够将响应与修正送回给 Subversion 开发者社群. 请参见 [the section called “Contributing to Subversion”](#), 以了解如何加入本社群.

You can send publishing comments and questions to O'Reilly here: ###insert boilerplate.

在 <http://svnbook.red-bean.com> 可以找到还满新的在线版本.

致谢

Huge list of thanks to the many svn developers who sent patches/feedback on this book.

Also, individual-author acknowledgements to specific friends and family.

Chapter 1. 简介

Table of Contents

[什么是 Subversion?](#)

[Subversion 的历史](#)

[Subversion 的功能](#)

[安装 Subversion](#)

[Subversion 的组件](#)

[客户端组件 \(供使用者使用\)](#)

[服务器组件 \(供管理员使用\)](#)

版本控制是管理信息变化的技术. 对于程序设计者来说, 它已经成为不可或缺的工具, 他们常常将花时间修改软件, 产生部份的变更, 然后第二天再取消所作的变更. 想象有一群程序设计人员同时工作的情况, 你就能够理解, 为什么需要一个良好的系统来管理可能的混乱.

什么是 Subversion?

Subversion 是一个自由/开放源码的版本控制系统, 也就是说 Subversion 管理着随时间改变的档案. 这些档案放置在一个中央 *档案库(repository)* 中. 这个档案库很像一个寻常的档案服务器, 不过它会记住每一次档案的变动. 这样你就可以把档案回复到旧的版本, 或是浏览档案的变动历程. 许多人会把版本控制系统想象成某种“时光机器”.

某些版本控制系统也是 software configuration management (SCM) 系统. 这些系统是特别设计来管理大量程序代码的, 而且具有许多功能, 专门用在软件开发之用——像是可完全了解程序语言, 或是提供编译软件的工作. 不过 Subversion 并不是这样的系统; 它是一个泛用系统, 可用来管理 *任何* 类型的档案, 其中包括了程序源码.

Subversion 的历史

在 1995 年时, Karl Fogel 与 Jim Blandy 成立了 Cyclic Software, 提供 Concurrent Versions System (CVS) 的商业支持, 并着手改良它. Cyclic 作出了第一个具网络功能的 CVS 公开版本 (由 Cygnus 软件公司捐赠). 在 1999 年, Karl Fogel 出版了一本书, 讲的是 CVS, 以及它所促成的开放源码发展模式. Karl 与 Jim 很早前就提过, 要制作一个 CVS 的取代软件的构想; Jim 甚至还起草了一个新的, 理论性的档案库设计, 而且还想到了一个不错的计划名称. 最后, 在 2000 年二月, CollabNet (<http://www.collab.net>) 的 Brian Behlendorf 提供 Karl 全职的工作, 专职发展 CVS 的替代程序. Karl 集合了一个团队, 于五月开始发展. 由于 Subversion 是以自由授权撰写的, 它很快就吸引了一堆发展人员.

Subversion 的原始设计团队定下了几个简单的目标. 他们决定它必须在功能上可取代 CVS. 也就是说, 所有 CVS 可达成的事, 它都要能够作到. 在修正最显而易见的瑕疵的同时, 还要保留相同的发展模式. 还有, Subversion 应该要和 CVS 很相像, 任何 CVS 使用者只要花费少许的力气, 就可以很快地上手.

经过十四个月的撰写之后, Subversion 于 2001 年 8 月 31 号开始“自行管理”. 也就是说, 发展人员不再使用 CVS 来管理 Subversion 的程序代码, 而以 Subversion 自己来管理.

虽然起始这个计划, 与提供大部份成果的资金都归功于 CollabNet (它付出几位全职 Subversion 开发人员的薪水), 这还是个开放源码计划, 由一般开放源码界所公认的规则所支配. CollabNet 拥有程序代码的版权, 不过程序代码是以 Apache/BSD 风格的版权发行, 完全符合 Debian Free Software Guidelines. 换句话

说, 每个人都可以随意地自由下载、修改、以及重新散播 Subversion; 完全不需要经过 CollabNet, 或是任何人的允许.

Subversion 的功能

Subversion 哪里比 CVS 的设计更好? 这里是个简短的列表, 以满足你的好奇心. 如果你不熟悉 CVS 的话, 可能不了解这些特色在哪里. 别害怕: 第二章会提供你版本控制的简单介绍.

目录版本控制

CVS 只能追踪单独档案的历史, 不过 Subversion 实作了一个“虚拟”的版本控管档案系统, 能够依时间追踪整个目录的更动. 目录和档案都被纳入版本控管. 最后, 客户端有真正可用的 **move** (移动) 与 **copy** 指令.

不可分割的送交

一个送交动作, 不是导致所有更动都送入档案库, 就是完全不会送入. 这让发展人员以逻辑区段建立更动, 并送交更动.

纳入版本控管的描述数据 (Meta-data)

每一个档案与目录都附有一组隐形“性质 (property)”. 你可以自己发明, 并储存任何你想要的键值对. 性质是随着时间来作版本控管的, 就像档案内容一样.

选择不同的网络层

Subversion 有抽象的档案库存取概念, 可以让人很容易地实作新的网络机制. Subversion “先进”的网络服务器, 是 Apache 网页服务器的一个模块, 它以称为 WebDAV/DeltaV 的 HTTP 变体协议与外界沟通. 这对 Subversion 的稳定性与互通性有很大的帮助, 而且额外提供了许多重要功能: 举例来说, 有身份认证, 授权, 在线压缩, 以及档案库浏览. 另外也有小而独立的 Subversion 服务器程序, 使用的是自订的通讯协议, 可以很容易地透过 ssh 以 tunnel 方式使用.

一致的数据处理方式

Subversion 使用二进制差异运算法, 来表示档案的差异, 它对文字 (人类可理解的) 与二进制档案 (人类无法理解) 两类的档案都一视同仁. 这两类的档案都同样地以压缩形态储存在档案库中, 而且档案差异是以两个方向在网络上传送的.

更有效率的分支 (branch) 与标记 (tag)

分支与标记的花费并不必一定要与计划大小成正比. Subversion 建立分支与标记的方法, 就只是复制该计划, 使用的方法就像 **hard-link** 一样. 所以这些动作只会花费很小, 而且是固定的时间.

Hackability

Subversion 没有任何的历史包袱; 它主要是一群共享的 C 链接库, 具有定义完善的 API. 这使得 Subversion 便于维护, 并且可被其它应用程序与程序语言使用.

安装 Subversion

Subversion 建立在一个可移植的 layer, 称为 APR (Apache Portable Runtime 链接库) 上. 这表示 Subversion 应该可以在任何可以执行 Apache 的 httpd 服务器的操作系统上: Windows, Linux, 所有的 BSD 分支, Mac OS X, Netware, ... 等等.

取得 Subversion 最简单的方式, 就是下载为你的操作系统所编译的二进制套件. Subversion 的网站 (<http://subversion.tigris.org>) 常常有自愿者提供的, 可供下载的可执行档. 网站上, 也常有供微软操作系统使用者使用的图形接口安装套件. 如果你使用的是 Unix 系的操作系统, 你可以使用系统内定的套件发行系统 (rpm, deb, ports), 来取得 Subversion.

另外, 你也可以直接从源码建立 Subversion. 你可以从网站上, 下载最新的源码发行档. 解开之后, 请遵循 INSTALL 档案里的说明, 把它建立起来. 请注意发行的源码套件中, 包含了所有你需要建立命令列客户端, 可与远程档案库沟通的套件 (尤其是 apr, apr-util, 以及 neon 链接库). 但是 Subversion 还有许多其它相依套件, 像是 Berkeley DB, 另外 Apache httpd 也是个可能性. 如果你想要建立一个“完整的”的编译, 请确定你具备了所有记载在 INSTALL 里的套件. 如果你计划要与 Subversion 工作, 你可以使用你的客户端程序, 抓取最新的, 流血前线的源码. 这记载在 [the section called “Get the Source Code”](#) 内.

Subversion 的组件

安装好之后, Subversion 会有数个不同的部份. 以下是你取得的程序的快速综览.

客户端组件 (供使用者使用)

svn

命令列客户端程序. 这是用来管理数据的主要工具, 在第 2, 3, 4, 以及第 6 章有详细说明.

svnversion

用来回报工作复本的混合版本状态. (请参考 [Chapter 2, 基本概念](#), 以了解混合版本的工作复本.)

服务器组件 (供管理员使用)

这些会在 [Chapter 5, Repository 管理](#) 中讨论.

svnlook

用来检阅 Subversion 的档案库的工具.

svnadmin

用来调整与修整 Subversion 的档案库的工具.

mod_dav_svn

给 Apache-2.X 网页服务器使用的外挂模块; 可以用来将你的档案库透过网络对外开放, 以供他人进行存取。

svnserve

一个独立的服务器程序, 可以作为服务器行程执行, 或是被 SSH 启动; 另一个让你的档案库在网络上可供其它人存取的方法.

假设你已经正确地安装 Subversion, 你应该可以开始使用了. 接下来的两章, 我们会带领你涵盖 Subversion 的命令列客户端程序 **svn** 的使用.

Chapter 2. 基本概念

Table of Contents

[档案库](#)

[各种版本控制的模型](#)

[档案分享的问题](#)

[锁定-修改-解锁的解决方案](#)

[复制-修改-合并的解决方案](#)

[Subversion 实务](#)

[工作复本](#)

[修订版本](#)

[工作复本如何追踪档案库](#)

[混合修订版的限制](#)

[摘要](#)

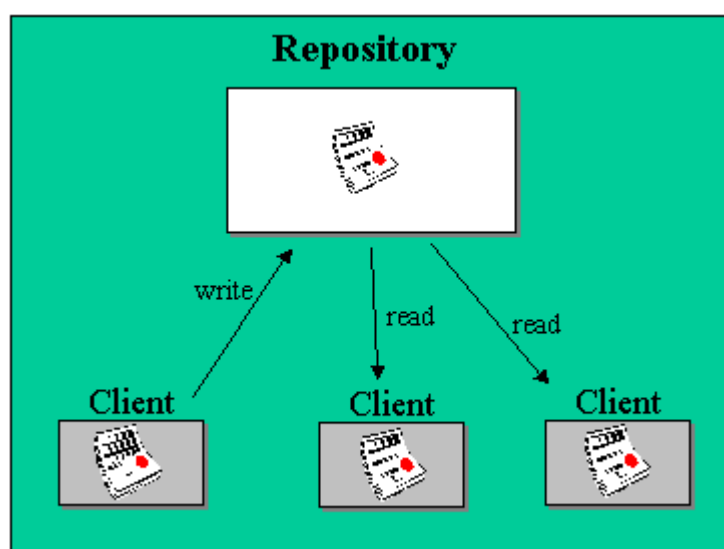
本章对 Subversion 有简短与非正式的描述. 如果你是版本控制的新手, 本章相当适合你. 我们一开始会讨论一般性的版本控制, 慢慢导向 Subversion 背后的概念, 并且利用 Subversion 举出简单的例子.

即使本章都以人们共同一群程序源码来作为例子, 但是请记住 Subversion 可用来管理任何种类的档案 — 并非仅仅局限在帮助程序设计师而已.

档案库

Subversion 是一个用以分享信息的中央系统, 核心为 *档案库(repository)*, 作为储存数据的集散地. 档案库储存数据的形式是 *档案系统树(filesystem tree)* — 也就是典型的目录与档案的架构. 许多的 *客户端* 会先连上档案库, 然后对这些档案作读取或写入的动作. 藉由写入数据, 一个客户端能够让信息为他人所用; 藉由读取数据, 该客户端能够撷取他人的信息.

Figure 2.1. 典型的主从式系统



为什么这样会很有趣? 到目前为止, 这些听起来就像一个典型的档案服务器. 事实上, 档案库 *就* 是一种档案服务器, 但是与你所见的不太相同. 让 Subversion 档案库如此不同的原因, 在于 *它会记住所有的更动*: 每个档案的每一个更动, 甚至是每一个目录所作的更动, 像是目录与档案的新增, 删除, 以及重新编排.

当一个客户端自档案库读取数据时, 它通常只会看到最新版本的档案系统树. 但是客户端也可以看到 *早先* 的档案系统. 举例来说, 客户端可以询问历史性的问题, 像是 "上个星期三, 这个目录里有什么东西?", 或 "谁是最后一个更动这个档案的人, 而且作了哪些更动?" 这就是任何 *版本控制系统* 的核心问题: 记录并追踪随着时间对数据所作的更动.

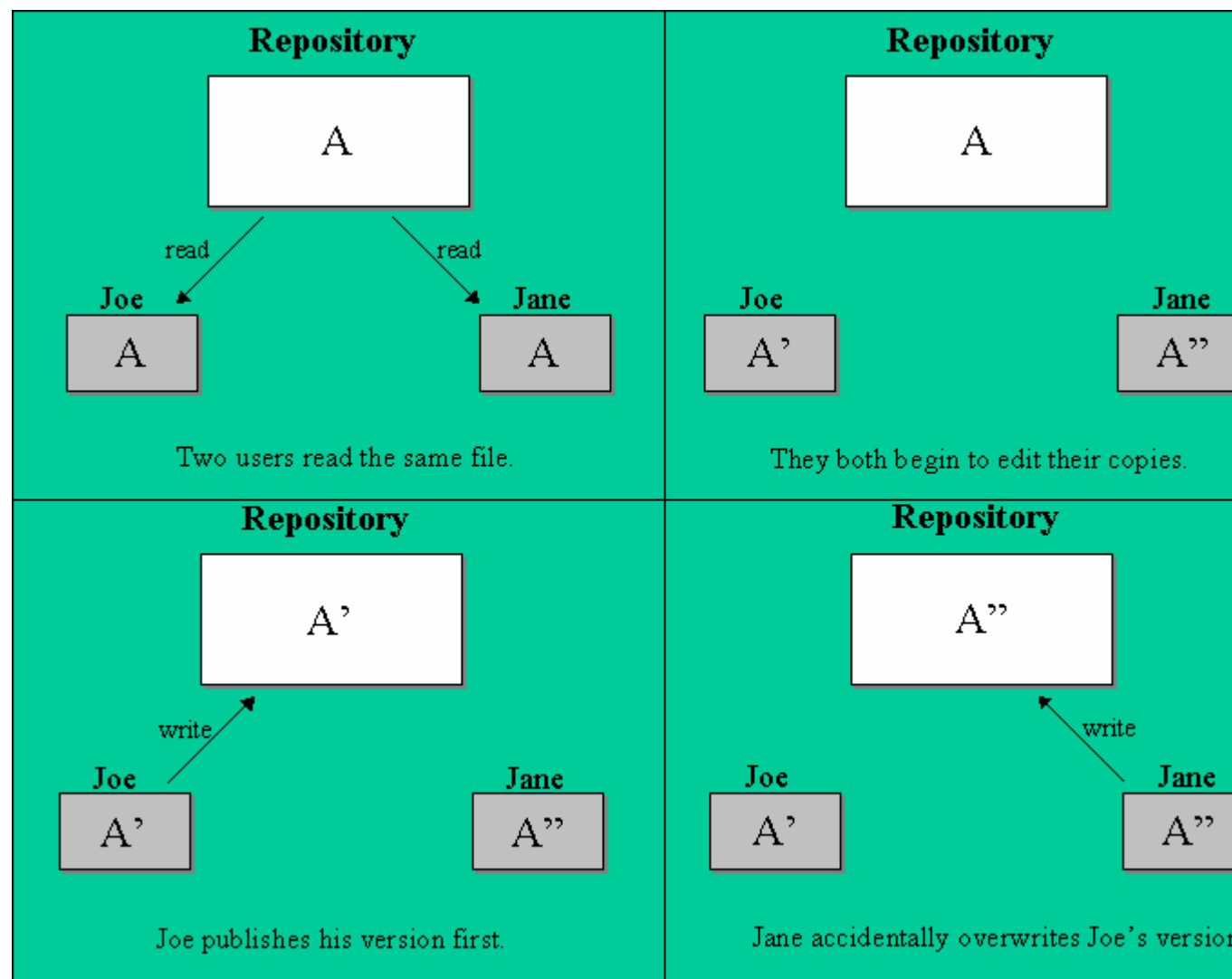
各种版本控制的模型

档案分享的问题

所有的版本控制系统都必须解决同样的基本问题: 如何让使用者分享数据, 但是不让他们不小心成为彼此的阻碍? 使用者要不小心覆写掉彼此在档案库里的更动, 实在是太容易了.

考虑一下以下的情景: 假设我们有两个协同工作人员, Harry 与 Sally. 他们决定同时编辑同一个储存在档案库的档案. 如果 Harry 先存入档案库, (几个月之后) Sally 很有可能以她自己的新版档案覆写过去. 虽然 Harry 的版本并不会就此消失 (因为系统记得每一次的更动), 但是任何 Harry 所作的更动不会出现在 Sally 的新版档案中, 因为她打从一开始就没看过 Harry 的更动. 总地来说, Harry 的心血就这么消失了——至少从该档案的最新版就遗漏掉了——这绝对是我们想要避免的情况!

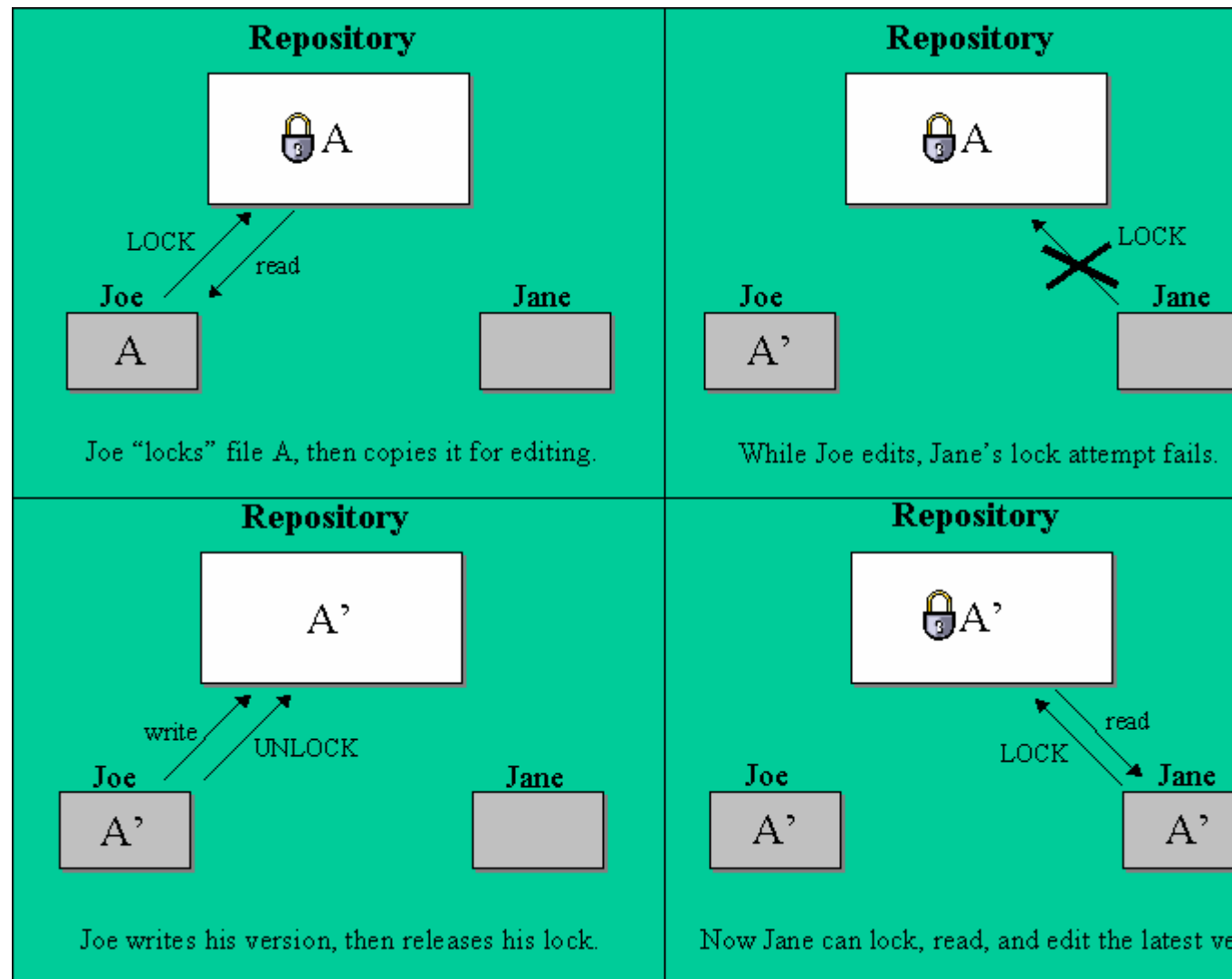
Figure 2.2. 应避免的问题



锁定-修改-解锁的解决方案

许多版本控制系统以 **锁定-修改-解锁** 的方式 来解决这个问题, 这是个很简单的解决方案. 在这样的系统中, 档案库在同一时间只允许一个人修改一个档案. 首先 Harry 必须在他开始更动之前, 先 "锁定" 该档案. 锁定档案就像从图书馆借书; 如果 Harry 已锁定一个档案, 那么 Sally 就无法对其进行任何更动. 如果她想要锁定该档案, 档案库会拒绝她的要求. 她所能作的, 就只是读取这个档案, 然后等着 Harry 完成他的更动, 解除他设下的锁定. 在 Harry 解除该档案的锁定之后, 他的回合就结束了, 现在轮到 Sally 可以对其进行锁定并编辑内容.

Figure 2.3. 锁定-修改-解锁的解决方案



锁定-修改-解锁模型的问题, 在于它的限制多了点, 而且经常会成为使用者的绊脚石:

- **锁定可能会造成管理上的问题.** 有的时候 Harry 在锁定一个档案之后, 然后就忘了这件事. 在此同时, 由于 Sally 还在等着编辑这个档案, 她什么事也不能作, 然后 Harry 就跑去休假了. 现在 Sally 必须找个管理员, 才能解除 Harry 的锁定. 这样状况, 最后导致了许多的延迟与时间的浪费.

- *锁定可能造成不必要的工作瓶颈* 如果 Harry 编辑的是文字文件的开头部份, 而 Sally 只是要修改同一档案的结尾部份呢? 这样的更动完全不会重迭在一起. 他们可以同时修改同一个档案, 而不会造成任何的伤害, 只要这些更动适当地合并在一起. 像这样的情况, 他们并不需要轮流进行才能完成.
- *锁定可能会造成安全的假象* 假设 Harry 锁定并编辑档案 A, 同时 Sally 锁定并编辑档案 B. 但是假设 A 与 B 彼此相依, 而对各档案所作的修改是完全不兼容的语意. 突然之间 A 与 B 彼此无法执行. 锁定系统完全无法避免这样的状况 — 但是它在某种程度上提供了一种安全的假象. 它很容易让 Harry 与 Sally 认为藉由锁定档案, 每个人都有个好的开始, 工作互不干涉, 如此让他们不会在早期就讨论他们互不兼容的更动

复制-修改-合并的解决方案

Subversion, CVS, 还有其它的版本控制系统使用一种 *复制-修改-合并* 的模型, 作为锁定的取代方法. 在这种模型下, 每一个使用者客户端会读取档案库, 然后建立档案或计划的 *工作复本*. 然后使用者进行各自的工作, 修改他们自己的私有复本. 最后, 私有复本会合并在一起, 以产生一个新的, 最后的版本. 版本控制系统通常会协助合并的动作, 但是最后人类还是负有让它能够产生正确结果的责任.

这里举个例子. 假设 Harry 与 Sally 各自从档案库 建立同一个项目的工作复本. 他们同时工作, 并对同一个复本中的档案 "A" 作了修改. Sally 先将她的更动送回档案库. 当 Harry 试着要存回他的更动时, 档案库会通知他, 他的档案 A 已经 *过时*. 换句话说, 档案库里的档案 A 在他上次产生复本后已经更动过了. 所以 Harry 要求他的客户端程序, 将档案库里新的更动与他的档案 A 工作复本 *合并* 起来. 通常 Sally 的更动与他的更动不会重迭; 所以只要让两组更动整合在一起, 他就可以将他的工作复本回存至档案库.

Figure 2.4. 复制-修改-合并的解决方案

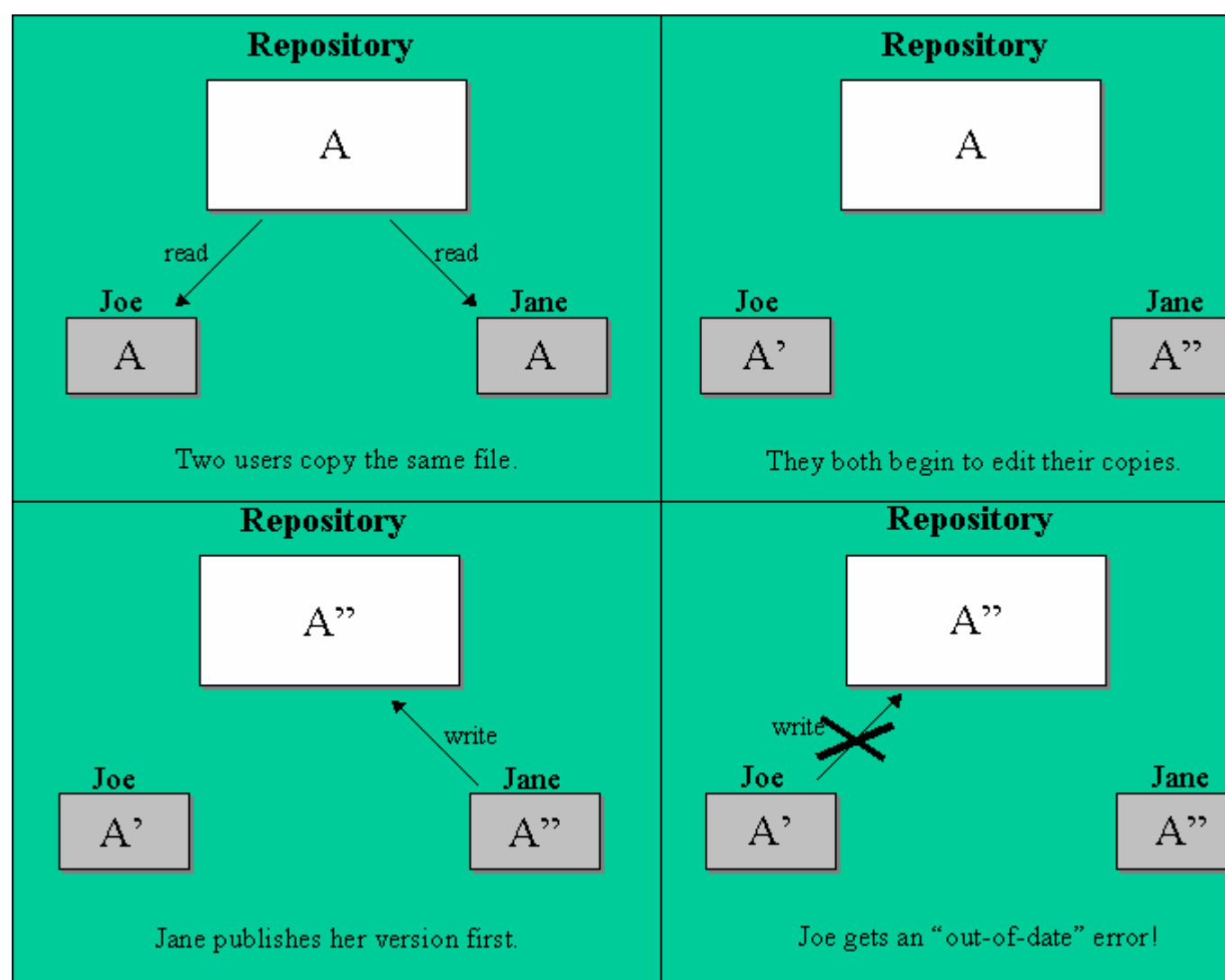
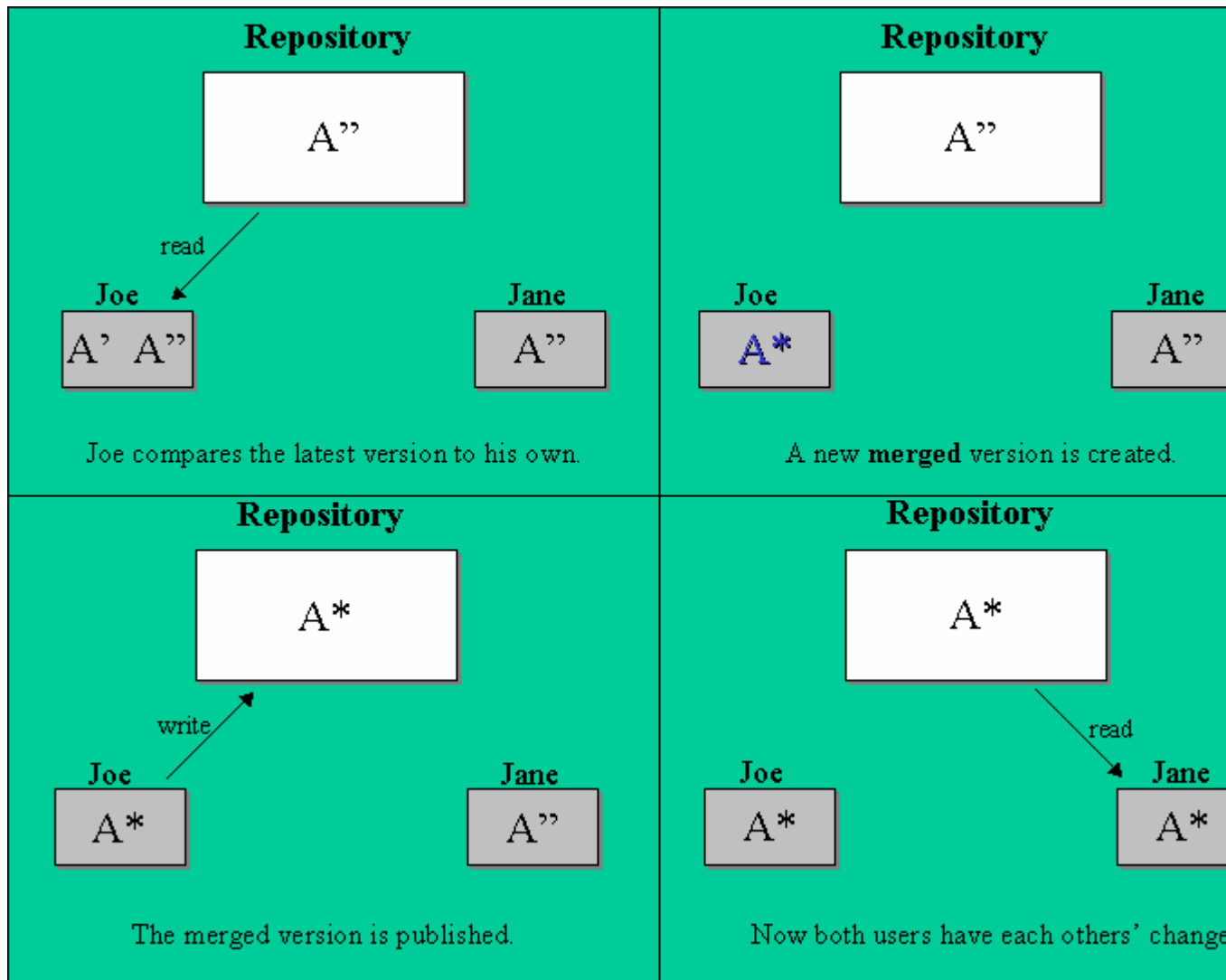


Figure 2.5. ...复制-修改-合并的解决方案 (续)



要是 Sally 的更动真的与 Harry 的更动重迭的话呢? 这该怎么办呢? 这种情况称为 *冲突(conflict)*, 通常也不会是多大的问题. 当 Harry 要求客户端程序将最新的档案库更动, 合并到他自己的工作复本时, 他自己的档案 A 会被标示为冲突的状态: 他可以看到两边互相冲突的更动, 然后手动在这两者之间作选择. 请注意, 软件不会自动地解决冲突; 只有人类才有理解能力, 进而作出明智的决定. 当 Harry 手动地解决重迭的更动之后 (也许就是和 Sally 讨论这个冲突!), 他就可以安全地将手动合并的档案存回档案库.

复制-修改-合并模型听起来有点混乱, 但是实务上跑起来可是相当地平顺. 使用者可以同时各自工作, 不需等待他人. 当他们同时处理同一个档案时, 大多数的情况下, 这些同时产生的更动都不会互相重迭; 冲突是相当少见的. 而且解决冲突所需要的时间, 也远低于锁定系统所损失的时间.

最后, 这些通通都回归到一个最重要的因素: 使用者沟通. 当使用者彼此沟通不良时, 语法与语意冲突都会增加. 没有任何系统能够强迫使用者完美地沟通, 而且也没有任何系统能够侦测出语意冲突. 所以并没有任何论点, 能够导出锁定系统可减少冲突发生的虚假承诺; 实务上, 锁定系统似乎比其它系统对生产力有更大的伤害.

Subversion 实务

工作复本

你已经读过有关工作复本的部份;现在我们要示范 Subversion 如何建立并使用它.

Subversion 工作复本就只是一个普通的目录树,位于你的本地系统中,其中包含了一堆档案.你可以依你喜好,自由地编辑这些档案.而且如果这些是源码档,你可以依一般的方法编译它们.你的工作复本就是你自己的私有工作空间: Subversion 不会将其它人的更动合并进来,也不会让你的更动让他人取得,除非你明确地要求要这样作.

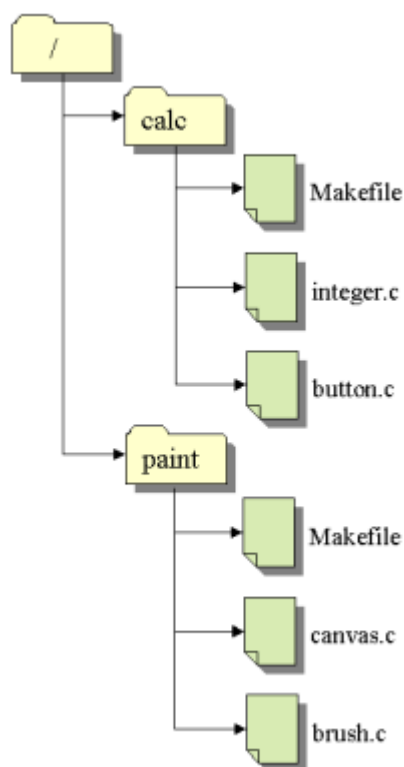
你在自己的工作复本档案中作了一些更动,并且确认它们都能正常地工作,此时 Subversion 提供你许多命令,让你将这些更动 "发表" 给同一计划的其它人使用 (藉由写至档案库). 如果别人发表了他们的更动, Subversion 提供你许多命令,可将这些更动合并至你的工作目录中 (藉由读取档案库).

工作复本也包含了其它额外的档案, Subversion 建立并维护这些档案,让它自己可以执行这些命令.特别一提,工作复本中的每个目录都会有一个名为 `.svn` 的子目录,也就是工作复本的 *管理目录*. 每个工作目录中的档案,都能够帮助 Subversion 了解哪些档案有未出版的更动,哪些与其它人的工作相比是过时的档案.

一个典型的 Subversion 档案库通常会包含数个项目使用的档案 (或源码档); 一般来讲,每一个项目是档案库档案树中的子目录. 在这样的安排中,一个使用者的工作复本,通常对应到档案库里的某一特定的子目录.

举例来说,假设你有一个包含两个软件项目的档案库.

Figure 2.6. 档案库的档案系统



换句话说, 档案库的根目录有两个子目录: `paint` 与 `calc`.

要取得一个工作复本, 你必须 *取出* (*check out*) 某个档案库里的子目录. ("check out" 听起来有点像是锁定或预约某项资源, 但是它不是; 它就只是为你建立一个计划的私有复本.) 举个例子, 如果你取出 `/calc`, 你会得到一个像这样的工作复本:

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

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

这一串字母 **A**, 表示 **Subversion** 已经加入几个对象到你的工作复本之中. 现在你有档案库的 `/calc` 目录的个人复本, 外加一些额外的东西——`.svn`——这里面有 **Subversion** 所需的额外信息, 早先有提到过.

档案库 URL

Subversion 的档案库可以经由数种不同的方法存取——本地端磁盘, 或是经由不同的网络协议. 但是一个档案库位置, 一定都是 **URL**. **URL schema** 说明了存取的方法:

Table 2.1. 档案库存取的 URL

Schema	存取方法
file:///	直接存取档案库 (位于本地端磁盘上)
http://	经由 WebDAV 通讯协议, 连接到知晓 Subversion 的 Apache 服务器.
https://	与 http:// 相同, 但是有 SSL 加密.
svn://	经由自订通讯协议的未授权 TCP/IP 联机, 连接到 svnserve 服务器.
svn+ssh://	经由自订通讯协议的已认证加密 TCP/IP 联机, 连接到 svnserve 服务器.

大多数的情况下, Subversion 的 URL 使用标准的语法, 可让主机名称与连接端口编号被指定为 URL 的一部份. 请记住 file: 存取方法只可用与服务器在相同机器的客户端—事实上, 就一般的传统而言, URL 的服务器名称部份必须是省略, 或是 localhost:

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

另外, 在 Windows 平台的 file: schema 使用者, 必须使用非正式的“标准”语法, 以存取在同一台机器上, 但是在不同磁盘驱动器的档案库. 以下两种语法的 URL 路径语法, 都可用来存取位在 x 磁盘驱动器上的档案库:

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

第二个语法中, 你必须将 URL 以括号包起来, 以便垂直棒字符不会被解译成管道. 请注意, 即使 Windows 上的路径使用的是倒斜线, URL 还是使用一般的斜线.

假设你更动了 button.c. 由于 .svn 目录会记得档案的修改日期与原始的内容, Subversion 能够知道你改了档案. 但是 Subversion 不会让你的更动公诸于世, 除非你明确地表明要这么作. 发表你的更动的行为, 通常称为 送交 (或 存入 (check in)) 更动至档案库.

要发表你的更动让其它人知道, 可以使用 Subversion 的 **commit** 命令:

```
$ svn commit button.c
Sending button.c
Transmitting file data..
```

Committed revision 57.

现在对 `button.c` 所作的更动, 已经送交至档案库了; 如果其它使用者取出 `/calc` 的工作复本, 他们会在最新版的档案中看到你的更动.

假设你有个合作伙伴 **Sally**, 他和你同一时间取出 `/calc` 的工作复本. 当你送交你对 `button.c` 的更动, **Felix** 的工作复本并没有改变; **Subversion** 只会依使用者要求来变更工作复本.

要让她的项目也能跟上变动, **Sally** 可以藉由使用 **Subversion** 的 **update** 命令, 要求 **Subversion** 更新他的工作复本. 这会让你的更动合并到她的工作复本中, 外加他人自上次取出档案以后所送交的更动.

```
$ pwd
/home/sally/calc

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

$ svn update
U button.c
```

以上取自 **svn update** 命令的输出, 表示 **Subversion** 更新了 `button.c` 的内容. 请注意 **Felix** 不必指定要更新哪个档案; **Subversion** 使用 `.svn` 目录与档案库里的额外信息, 来决定哪些档案必须更新到最新版.

修订版本

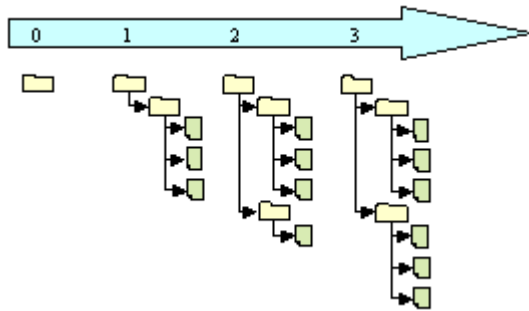
svn commit 的动作, 可以将不限数目的档案与目录的更动, 视为单一不可分割的异动以进行发表. 在你的工作复本中, 你可以修改档案的内容, 建立, 删除, 更名, 以及复制档案与目录, 然后将所有的更动视为一个单位, 一口气送交回去.

在档案库中, 每一次的送交都被视为是一个不可分割的异动: 不是所有送交的更动都成功, 就是全部都不成功. **Subversion** 会试着维持这样的不可分割特性, 不管是遭遇程序失败, 系统当机, 网络有问题, 还是其它使用者进行的动作.

每一次档案库接受一个送交的更动, 就会让档案树进入一个新的状态, 称之为 *修订版本*. 每一个修订版本都会被赋与一个唯一的, 比前一个修订版本号大一的自然数. 一个新建立的档案库的修订版号为零, 其中除了空的根目录外, 什么都没有.

有一个把档案库具象化的好方法, 就是将之视为一系列的树. 想象有一系列的修订版本号, 自左至右, 由 0 开始. 每一个修订版本号下都有一个档案系统的树状结构, 每一个档案树都是每一次送交之后的档案库“快照”.

Figure 2.7. 档案库



全面性的修订版号

不同于许多其它的版本控制系统, Subversion 的修订版号是对整个树有效, 而不仅只于个别的档案. 每一个修订版号都可选取整个树, 它对应到某个更动送交之后的特定状态的档案库. 另一个思考的方式, 就是版号 N 表示是第 N 次送交之后的档案库的档案系统状态. Subversion 使用者说 ``foo.c 的修订 5 版" 时, 实际上的意义是 ``出现在修订 5 版的 foo.c". 请注意, 基本上一个档案的修订 N 与 M 版并不见得是不同的! 由于 CVS 使用的是档案自各的修订版号, CVS 使用者可以先看看附录 A, "对 CVS 使用者介绍 SVN", 以了解更多的细节.

请特别注意, 工作复本并不见得一定会符合档案库某一特定的修订版; 每个档案可能会对对应到不同的修订版本. 举个例子, 假设你的工作目录, 是从档案库注销了 4 号修订版:

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

此时工作目录对应的是档案库的 4 号修订版. 不过要是你修改了 button.c, 送交这个更动. 假设此时其它人没有送交任何更动, 你所送交的更动就会变成档案库的第 5 号修订版, 然后你的工作复本就会变成像这样:

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

假设在这个时候, Sally 送交了 integer.c 的更动, 产生了 6 号修订版. 如果你以 **svn update** 更新你的工作目录, 它看起来就会像这样:

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

Sally 对 integer.c 的更动会出现在你的工作复本中, 而你的更动还是在 button.c 之中. 在这个例子中, 版本 4, 5, 6 的 Makefile 内容都是一样的, 不过

Subversion 会将工作副本中的 `Makefile` 标示为版本 6, 表示它还是最新版本. 所以, 在你对工作副本作了一次完整更新之后, 它基本上就是完全对应到档案库的某一个版本.

工作副本如何追踪档案库

对每个工作目录中的档案, Subversion 会在管理区域 `.svn/` 中, 记录两个重要的信息:

- 工作档案的基本修订版本 (亦称为档案的 *工作版本*), 以及
- 时间戳记, 记录本地副本最近一次被档案库更新的时间.

有了这些信息, 再藉由咨询档案库, Subversion 就可以决定某个工作档案是处于下列四个状态何者之一:

未更动, 现行版本

本档案在工作目录中未被更动, 而且自工作版本之后, 也没有任何该档案的更动被送交回去. 对它执行 **`svn commit`** 不会发生任何事, 执行 **`svn update`** 也不会发生任何事.

本地修改, 现行版本

这个档案在工作目录中被修改过, 而自其基础修订版号后, 也没有任何更动送交回档案库. 由于有尚未送交回去的本地端修改, 所以对它的 **`svn commit`** 会成功地发表你的更动, 而 **`svn update`** 则不会作任何事.

未更动, 过时版本

这个档案在工作目录中并未更动, 但是档案库已被更动. 本档案应该要更新, 以符合公开修订版. 对它的 **`svn commit`** 不会发生任何事, 而 **`svn update`** 会让工作目录中的档案更新至最新版本.

本地修改, 过时版本

这个档案在工作目录与档案库都受到了更动. 对它执行 **`svn commit`** 会产生 "out-of-date" 错误. 这个档案应该要先被更新; **`svn update`** 会试着将已发表的更动, 与本地的更动合并在一起. 如果 Subversion 无法自动无误地完成它, 那么就会留给使用者, 让他来解决这个冲突.

听起来好像要注意很多东西, 但是 **`svn status`** 的命令会显示任何在工作副本里的项目的状态. 欲取得该命令更详细的信息, 请参见 [the section called “svn status”](#).

混合修订版的限制

Subversion 的一个基本原则,就是要尽量地保有弹性. 有一种特别的弹性,就是工作复本可包含混合修订版的能力.

一开始可能无法理解,为什么这样的弹性会被视为一项特色,而不是责任. 在完成至档案库的送交动作后,刚送交的档案与目录的修订版,会比工作复本其它部份的修订版还要新. 这看起来有点混乱. 就像我们早先示范过的,我们永远都可藉由 **svn update**, 将工作复本带回至单一的工作修订版. 为什么会有人要故意混合不同的工作修订版呢?

假设你的项目够复杂,你发现有时强制将工作复本的某些部份带回到“旧版本”反而更好;在第 3 章,你会学到怎么达成. 也许你想要对早先版本的子模块进行测试,也许你想要在最新的档案树中,检视某个档案几个过去的版本.

但是,当你在工作复本中使用混合修订版时,这项弹性有几个限制.

首先,如果档案或目录不全是最新版本时,对它们的删除动作是无法送交的. 如果一个项目在档案库中,存在着比目前更新的版本,你想要送交的删除动作会被拒绝,以防止你不小心毁了还没看过的更动.

第二,你无法送交一个对目录的描述信息更动,除非它完全是最新版本的. 你会在第 6 章学到如何将“性质”附加到项目. 一个目录的工作修订版,会定义出特定的实体与性质的集合,因此送交一个对过时目录的性质更动,很有可能会毁掉你还没检视过的性质.

摘要

本章涵盖了几个 Subversion 的基本概念:

- 我们介绍了中央档案库,客户端工作复本,以及档案库修订版树的数组.
- 本章示范几个例子,说明两个协同工作者如何利用 '复制-修改-合并' 模式,透过 Subversion 来发表并接收彼此所作的更动.
- 我们也谈了一些有关 Subversion 如何追踪与管理工作复本的数据.

现在,你应该对 Subversion 如何工作有个清楚的概念. 有了这样的知识,你已经准备好进行到下一章,对 Subversion 的命令与功能来个详细的巡礼.

Chapter 3. 导览

Table of Contents

[帮帮我!](#)

[汇入](#)

[修订版: 数字, 关键词, 与日期. 我的天啊!](#)

[修订版号](#)

[修订版关键词](#)

- [修订版日期](#)
- [最初的取出动作](#)
- [基本工作流程](#)
 - [更新工作复本](#)
 - [对工作复本产生更动](#)
 - [检视你的更动](#)
 - [svn status](#)
 - [svn diff](#)
 - [svn revert](#)
 - [解决冲突 \(合并他人的更动\)](#)
 - [手动合并冲突](#)
 - [将档案复制并盖过你的工作档](#)
 - [弃踢: 使用 svn revert](#)
 - [送交更动](#)
- [检视历史纪录](#)
 - [svn log](#)
 - [svn diff](#)
 - [检视本地端更动](#)
 - [比较档案库与本地复本](#)
 - [档案库与档案库之间的比较](#)
 - [svn cat](#)
 - [svn list](#)
 - [对历史纪录的最后叮咛](#)
- [其它有用的命令](#)
 - [svn cleanup](#)
 - [svn import](#)
- [摘要](#)

现在我们将详细讲解如何使用 Subversion. 当你看完这一章后, 应该就能够进行每日会用到的功能. 一开始会先取出程序代码, 更动程序, 然后检视这些更动. 你也会看到如何将别人的更动, 取回至自己的工作复本. 检视这些更动, 然后处理可能发生的冲突.

请注意本章并不打算列出所有 Subversion 的命令—更确切地说, 它只是个对话式的介绍, 讲解最常遇到的 Subversion 工作. 本章假设你已经读过并了解 [Chapter 2, 基本概念](#), 也熟悉 Subversion 的基本模型. 想要取得所有命令列表, 请参照 [Chapter 8, 完整 Subversion 参考手册](#).

帮帮我!

在继续下去之前, 以下是你在使用 Subversion 时, 最重要、最常用到的命令: **svn help**. Subversion 的命令列客户端提供自己的说明文件 — 在任何时间, 只要打 **svn help <子命令>**, 就会有 **子命令** 的文件, 选项, 以及运作方式的说明.

汇入

你可使用 **svn import** 来汇入一个新的计划至 Subversion 的档案库中. 虽然在设定你的 Subversion 服务器时, 这可能是你第一件作的事, 但是它很少会使用到. 欲取得更详细的汇入功能说明, 请参见 本章稍后的 [the section called “svn import”](#).

修订版: 数字, 关键词, 与日期. 我的天啊!

在我们继续下去之前, 你应该要知道如何指定档案库中的特定修订版. 如你在 [the section called “修订版本”](#) 学到的, 修订版是档案库在某一特定时间的“快照”. 只要你会继续送交更动, 让档案库成长下去, 你就需要一个指定这些快照的机制.

使用 `--revision (-r)` 选项, 再加上你想要的修订版号 (**svn --revision REV**), 就可以指定修订版, 或是以分号隔开两个修订版号 (**svn --revision REV1:REV2**), 就可以指定一个修订版范围. Subversion 还可以让你以数字, 关键词, 或是日期来指定这些修订版号.

修订版号

当你建立了一个新的 Subversion 档案库, 它就以修订版号 0 开始它的生命. 其后的送交动作, 都会让修订版号加一. 在你的送交动作完成后, Subversion 客户端会告知你新的修订版号:

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

在未来的某一时间, 如果你想参考到这个版号时 (稍后在本章, 我们会看看如何, 以及为什么我们会想这样作), 你可以以 “3” 来指定它.

修订版关键词

Subversion 客户端懂得几个 *修订版关键词*. 这些关键词可在 `--revision` 选项中, 用来取代整数自变量, 它们将会被 Subversion 转换成特定的修订版号:

Note

每一个工作复本的目录内, 都有一个 `.svn` 管理区域. Subversion 会在管理区域中, 存放目录里每一个档案的复本. 这个复本是你上次进行更新时, 所取得的修订版 (称为 “BASE” 修订版) 内的无更动 (没有关键词展开, 没有列尾字符展开, 什么都没有) 档案复本. 我们称这个档案为 “原始未更动 (preistine)” 复本.

HEAD

档案库内的最新版本.

BASE

一个对象在工作复本中的“原始未更动”修订版.

COMMITTED

一个对象距 **BASE** 修订版之前 (包含), 最近一次修改的修订版.

PREV

最近一次修改的修订版的 前一个 修订版. (技术上来讲, 就是 COMMITTED - 1.)

以下是几个使用修订版关键词的范例 (如果不懂这些命令, 没有关系; 藉由本章的引导, 我们会逐步解释这些命令):

```
$ svn diff --revision PREV:COMMITTED foo.c
# 显示最近一次送交 foo.c 所产生的更动

$ svn log --revision HEAD
# 显示最新的档案库送交的记录讯息

$ svn diff --revision HEAD
# 将工作档案 (含有本地的修改) 与档案库的最新版本作比较

$ svn diff --revision BASE:HEAD foo.c
# 将你的 “原始未更动” foo.c (未含本地修改)
# 与档案库里的最新版本作比较

$ svn log --revision BASE:HEAD
# 显示自你上次更新后的所有送交讯息

$ svn update --revision PREV foo.c
# 复原上一个 foo.c 的更动
# (foo.c 的工作修订版本会被减少.)
```

这些关键词感觉并不是非常重要, 但是它们可以让你进行一些常见 (而且有用) 的动作, 而不必先找出确切的修订版号, 或是记住工作复本确实的修订版本.

修订版日期

任何你可以指定修订版号或修订关键词的地方, 你也可以在大括号 “{}” 内指定日期, 就可以指定修订版日期. 你还可以一起使用日期与修订版, 以存取档案库里一个范围的更动.

Subversion 可接受相当多的日期格式— 只要记得将有空白的日期以引号包起来就好. 这里只是几个 Subversion 接受的格式的例子而已:

```
$ svn checkout --revision {2002-02-17}
$ svn checkout --revision {2/17/02}
$ svn checkout --revision {"17 Feb"}
$ svn checkout --revision {"17 Feb 2002"}
$ svn checkout --revision {"17 Feb 2002 15:30"}
$ svn checkout --revision {"17 Feb 2002 15:30:12 GMT"}
$ svn checkout --revision {"10 days ago"}
$ svn checkout --revision {"last week"}
$ svn checkout --revision {"yesterday"}
...
```

当你将日期指定为修订版时, Subversion 会找出与该日期最接近的档案库的修订版:

```
$ svn log --revision {11/28/2002}
-----
---
r12:  ira | 2002-11-27 12:31:51 -0600 (Wed, 27 Nov 2002) | 6 lines
...
```

Subversion 都早一天吗?

如果你指定单独一天为修订版号, 而没有指定时间的话 (例如 11/27/02), 你可能会以为 Subversion 会给十一月 27 日那天发生的最后修订版. 相反的, 你得到的是 26 日的, 甚至还要更早. 请记住 Subversion 会找出离你指定日期 *最近的档案库修订版*. 如果你给定一个日期, 而没有时间, 像是 11/27/02 的话, Subversion 会假设时间为 00:00:00, 所以找出最近的修订版的结果, 并不会包含 27 日那天的.

如果你想要找出的结果包含 27 日的话, 你可以指定 27 日里的时间 ("27 Nov 2002 23:59"), 或直接指定下一天 ("28 Nov 2002").

你也可以使用一个范围的日期. Subversion 会找出所有两个日期之间的修订版, 包含这两个日期在内:

```
$ svn log --revision {2002-11-20}:{2002-11-29}
...
```

就像我们前面指出的, 你也可以混合日期与修订版号:

```
$ svn log -r {11/20/02}:4040
```

最初的取出动作

大部份开始使用 Subversion 的档案库的动作, 就是对你的项目进行 *取出* (*checkout*) 的动作. “取出” 档案库会在你的机器里建立一份复本. 这个复本包含了你在命令列里指定的档案库的 HEAD (也就是最新的) 版本:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A  trunk/subversion.dsw
A  trunk/svn_check.dsp
A  trunk/COMMITTERS
A  trunk/configure.in
A  trunk/IDEAS
...
Checked out revision 2499.
```

档案库配置

如果你在猜上面 URL 中的 trunk 是作什么用的, 它是我们对你的档案库配置的一部份建议, 在 [Chapter 4, 分支与合并](#) 会有更详细的解释.

虽然上面例子取出的是 trunk 目录, 但是你可以取出任何档案库深层的目录, 只要在取出的 URL 指定副目录即可:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk/doc/book/tools
A  tools/readme-dblite.html
A  tools/fo-stylesheet.xsl
A  tools/svnbook.el
A  tools/dtd
A  tools/dtd/dblite.dtd
...
Checked out revision 3678.
```

由于 Subversion 使用的是“复制-修改-合并”模式, 而不是“锁定-修改-解锁”模式 (参见 [Chapter 2, 基本概念](#)), 你现在已经可以修改已取出的档案与目录 (整个称之为 *工作复本*).

[换句话说](#), 现在你的“工作复本”就像其它在系统中的目录与档案的集合一样 ^[1]. 你可以编辑或更动它们, 把它们移来移去, 甚至还可以删掉整个工作复本, 然后完全把它抛诸脑后.

Note

虽然你的工作复本“就只是跟其它在系统上的目录与档案的集合没有两样”, 但是你要重新摆放工作复本里的东西的话, 你还是得让 Subversion 知道. 如果你要复制或移动工作复本中的某个项目, 你应该使用 **svn copy** 或 **svn move**, 而非操作系统所提供的复制与移动的指令. 本章稍后会再讨论到这些指令.

除非你准备要 *送交* (*commit*) 新档案或目录, 或是对现有项目的更动, 否则你无需通知 Subversion 服务器你所作的事情.

.svn 目录有什么作用?

每一个工作复本的目录都有 *管理区域*, 一个名为 `.svn` 的子目录. 通常列出目录的指令不会显示这个子目录, 但是它是一个相当重要的目录. 不管你作什么, 千万别删除或是修改管理区域里的数据! Subversion 倚靠它来管理你的工作复本.

虽然你可以用档案库的 `url` 作为唯一的参数, 来取出工作复本, 你还是可以在档案库 `url` 之后指定一个目录, 如此会将你的工作复本置于你命名的新目录中. 举个例子:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subv
A  subv/subversion.dsw
A  subv/svn_check.dsp
A  subv/COMMITTERS
A  subv/configure.in
A  subv/IDEAS
...
Checked out revision 2499.
```

这会将你的目录置于 `subv` 中, 而不是我们之前的 `trunk` 目录.

基本工作流程

Subversion 有许多特色, 选项, 以及其它有的没有的功能, 不过对每天例行的工作来说, 你会用到的只有其中的一小部份而已. 在本节中, 我们会详加介绍所有你会在每天的工作中, 最常会使用到的功能.

典型的工作流程, 看起来像这样:

- 更新工作复本
 - **svn update**
- 产生更动
 - **svn add**
 - **svn delete**
 - **svn copy**
 - **svn move**
- 检视你的更动
 - **svn status**
 - **svn diff**
 - **svn revert**
- 合并其它人的更动
 - **svn merge**

- **svn resolved**
- 送交更动
 - **svn commit**

更新工作复本

与一组团队同时修改同一个项目时, 你会想先 *更新(update)* 你的工作复本: 也就是说, 取回同一项目中, 其它发展人员所作的更动. 你可以使用 **svn update**, 将你的工作复本里的版本同步至档案库的最新修订版.

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

在这个例子中, 有人从你上次更新后, 登录了 `foo.c` 与 `bar.c` 所产生的更动, 而且 Subversion 已经更新了工作目录, 以包含这些新的更动.

让我们更详细地检视 **svn update** 的输出. 当服务器送出更动至你的工作目录时, 在每一个项目的旁边, 都有一个字母代码, 让你知道 Subversion 会执行什么动作来更新你的工作目录.

U foo

档案 `foo` 会被更新 (update) (自服务器取得更动).

A foo

档案或目录 `foo` 会被新增 (Add) 到工作目录中.

D foo

档案或目录 `foo` 会自工作目录删除 (Delete).

R foo

工作复本的档案或目录 `foo` 被取代 (Replace); 也就是说, `foo` 被删除, 然后新增同一名称的新项目. 虽然它们的名称是相同的, 但是档案库会认为它们是不同的, 而且有着不同的历史进程.

G foo

档案 `foo` 自档案库取得新的更动, 但是本地复本的档案含有你的更动. 不过这些更动并没有重迭的部份, 所以 Subversion 可以毫无困难地合并 (merge) 档案库的更动.

C foo

档案 `foo` 自服务器收到冲突的 (conflict) 更动. 从服务器来的更动, 与你对该档案的更动有重迭的部份, 不过不必太惊慌失措. 冲突必须由人类 (也就是你) 来解决; 我们在本章稍后会讨论这个状况.

对工作复本产生更动

现在你可以开始工作, 修改工作复本了. 比较好的作法, 是产生特定的一个更动 (或是一组更动), 像是加个新功能, 修正臭虫等等. 这里会用到的 Subversion 指令, 有 **svn add**, **svn delete**, **svn copy**, 以及 **svn move**. 如果你只是要修改一个已在 Subversion 里的档案 (或是不只一个档案), 可能在送交更动前都不会用到这些指令. 你能对工作复本所作的更动:

变更档案

这是最简单的更动了. 你不需要先和 Subversion 说你要修改档案; 直接更动它即可. Subversion 有能力自动侦测哪些档案被更动过.

目录结构更动

你可以要求 Subversion 先“标示”预定要移除、新增、复制、或是移动的目录或档案. 虽然这些更动会马上出现在工作复本中, 不过在你送交更动前, 档案库并不会跟着有所变动.

要产生档案的更动, 请使用文字编辑程序, 文书处理程序, 图形程序, 或是你平常使用的任何工具. Subversion 可以很容易地处理二进制档案, 如同文本文件一般——而且一样有效率.

以下是你最常用来更动目录结构的四个常用的 Subversion 指令 (我们稍后会讲到 **svn import** 与 **svn mkdir**).

svn add foo

预定将档案 `foo` 新增至档案库中. 下一次送交时, `foo` 会成为其父目录的子项目. 请注意如果 `foo` 是一个目录的话, 所有在 `foo` 的东西都预定会被加入档案库. 如果你只想要 `foo` 本身而已, 请使用 `--non-recursive (-N)` 选项.

svn delete foo

预定将 `foo` 自档案库删除. 如果 `foo` 是档案, 那么它会马上自工作复本中删除. 如果 `foo` 是目录的话, 它不会被删除, 但是 Subversion 会预定将其删除. 当你送交你的更动时, `foo` 会自工作复本与档案库删除.^[2]

svn copy foo bar

建立一个新的项目 `bar`, 成为 `foo` 的复本. `bar` 会自动被预定新增至档案库中. 当 `bar` 在下次被送交时, 它的复制历程就会被纪录下来 (也就是来自 `foo`).

svn move foo bar

这个命令就像执行 **svn cp foo bar; svn delete foo** 一样. 也就是说, `bar` 预定被新增为 `foo` 的复本, 而 `foo` 则预定被删除.

毋需工作复本而变更 repository

早先在本章中, 我们曾提过任何的更动都必须进行送交动作, 以使档案库反映所作的更动. 实际上并不尽然如此— 还是有某些状况, 档案树的变更会马上送交回档案库. 只有子命令是直接处理一个 URL, 而非工作复本路径时, 这样的行为才会发生. 尤其是可用于 URL 的 **svn mkdir**, **svn copy**, **svn move**, 以及 **svn delete** 的某些特定使用方式.

URL 动作是如此运作的原因, 是因为那些针对工作复本的命令, 可以把工作复本当成是更动送交前的“集结区”, 而针对 URL 的命令就无法享用这样的好处, 所以当你直接对 URL 下命令时, 上述任何一个动作, 都会导致立即发生的送交行为.

检视你的更动

当你修改完毕后, 你必须将更动送交至档案库, 不过在这样作之前, 先看看自己作了哪些更动, 是个不错的主意. 在送交前先检视更动的部份, 让你可以写出更清楚的记录讯息. 你还可能会发现意外更动到的档案, 这让你在送交前还有回复这项错误的机会. 除此之外, 这也是在发表之前, 仔细检视并检查所作的更动的时机. 想要更清楚地了解所作的更动, 你可以藉由 **svn status**, **svn diff**, 以及 **svn revert** 这些命令, 来看看你所作的更动是什么. 通常你会用前两个命令, 来找你对工作复本所作的更动, 然后可能用第三个命令来复原这些更动.

Subversion 经过特别调校, 可在不与档案库沟通的情况下, 帮你进行以下的工作, 外加许多其它的工作. 尤其是你的工作复本里, `.svn` 区域拥有每一个受版本控制的档案的“原始未更动”复本. 也因为如此, Subversion 可以很快地告诉你档案是如何被更动, 甚至让你可以取消你所作的变更, 而不需使用到档案库.

svn status

你使用 **svn status** 的次数, 可能会远大于其它的 Subversion 命令.

CVS 使用者: 别急着更新!

你可能曾用过 **cv**s **update** 来看看你对工作复本所作的更动. **svn status** 就可以得知所有你对工作复本所作过的更动的信息— 不必存取档案库, 而且还不会不小心合并其它使用者发表的更动.

在 **Subversion** 中, **update** 就是作更动的动作— 它会将从你上次更新工作复本后所产生的所有更新, 通通更新到你的工作复本中. 你必须戒掉使用 **update** 以得知自己作过什么修改的习惯.

如果你对工作目录不带任何自变量执行 **svn status**, 它会侦测出所有在档案树里所产生的更动. 以下的例子, 是用来显示所有 **svn status** 可传回的状态码. 接在 # 后的文字, 并不是由 **svn status** 所产生的.

```

L      ./abc.c                # svn 在 .svn 目录中, 有 abc.c 的锁定
M      ./bar.c                # bar.c 的内容, 有本地端的变更
M      ./baz.c                # baz.c 已变更了性质, 但是没有内容的更动
?      ./foo.o                # svn 并未管理 foo.o
!      ./some_dir             # svn 管理它, 但是不是不见了, 就是不完整
~      ./qux                  # 纳入管理的是目录, 但是这里是档案, 或是相反的情况
A +    ./moved_dir            # 新增项目, 并以来源的历史纪录为其历史纪录
M +    ./moved_dir/README     # 新增项目, 使用来源的历史纪录, 再加上本地更动
D      ./stuff/fish.c         # 本档案已预定要被删除
A      ./stuff/loot/bloo.h    # 本档案已预定要被新增
C      ./stuff/loot/lump.c    # 这个档案有因更新而产生的冲突
S      ./stuff/squawk         # 这个档案或目录已切换到分支
```

在这个 **svn status** 的输出格式中, 先印出了五栏字符, 后接几个空白, 再接一个档案或目录的名称. 第一栏表示档案或目录本身, 以及/或是其内容的状态. 这里显示的代码有:

A file_or_dir

目录或档案 file_or_dir 已预定要被新增至档案库.

M file

档案 file 的内容已被修改.

D file_or_dir

目录或档案 file_or_dir 已预定要自档案库删除.

X dir

目前 dir 未纳入版本控制, 但是关联到一个 **Subversion** 的外部定义. 欲了解更多外部定义, 请参考 [the section called “外部定义”](#).

? file_or_dir

目录或档案 你可以藉由 **svn status** 命令的 `--quite (-q)` 选项, 或是对父目录设定其 `svn:ignore` 性质, 就不会显示问号代码. 想知道有哪些档案会被忽略, 请参照 [the section called “svn:ignore”](#).

! file_or_dir

目录或档案 `file_or_dir` 已纳入版本控制之中, 但是它不是消失, 就是不完整. 如果这个档案或目录被非 **Subversion** 的命令所删除, 就会被判定为消失. 如果是目录的话, 如果你在取出或是更新时中断, 那么它就会变成不完整. 只要执行 **svn update**, 就可以从档案库重新取得目录或档案, 或者以 **svn revert file_or_dir**, 回存消失的档案.

~ file_or_dir

目录或档案 `file_or_dir` 在档案库里是一种对象, 但是实际在工作复本中又是另一种. 举个例子, **Subversion** 可能在档案库里有一个档案, 但是你删除了这个档案, 而以原名称建立了一个目录, 但是都没有使用 **svn delete** 或 **svn add** 命令来处理.

C file

`file_or_dir` 处于冲突的状态. 也就是说, 在更新时, 来自服务器的更新与工作复本中的本地更动, 有重迭的部份. 在送交更动回档案库之前, 你必须先解决这个冲突.

第二栏指示的是目录或档案的性质 (请参见 [the section called “性质”](#), 以了解更多性质的信息). 如果第二栏出现的是 `M`, 那么表示其性质曾变更过, 不然显示的是空格符.

第三栏只会显示空格符, 或是 `L`, 表示 **Subversion** 在 `.svn` 工作区中有对象的锁定. 如果你处于正在执行 **svn commit** 的目录中, 此时执行 **svn status** 的话, 你就会看到 `L`——大概那时正在编辑记录讯息. 如果 **Subversion** 并没有在运行的话, 那么很有可能 **Subversion** 因故中断. 这个锁定可以利用 **svn cleanup** 来清除 (稍后本章会提到).

第四栏只会显示空格符, 或是 `+`, 表示这个档案或目录已预定要加入档案库, 或是以额外附加的历史纪录修改. 通常发生的时机, 是你对档案或目录执行 **svn move** 或 **svn cp** 命令. 如果你看到 `A +`, 表示这个项目已预定被加入档案库, 并有历史纪录. 这可能是一个档案, 也可能是目录复制的根目录. `+` 表示这是个预定被加入, 并有历史纪录的子档案树, 也就是其某个父目录被复制, 而它是连带被复制的. `M +` 表示这是个预定被加入, 并有历史纪录的子档案树, 而且它有本地端的更动. 当你送交的时候, 其父对象会先以附加历史纪录的方式加入 (复制), 也就是该档案会自动出现在复本中, 接着本地的修改也会跟着上传至复本中.

第五栏只会显示空白或是 s. 它表示这个档案或目录, 已经自该工作复本里的路径 (利用 **svn switch**) 切换到一个分支.

如果你指定路径给 **svn status**, 它只会提供给你该对象的信息:

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

svn status 也有 **--verbose (-v)** 选项, 它会显示 *所有* 在工作目录中的对象数据, 就算还没有修改过的亦同:

```
$ svn status --verbose
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”输出. 第一栏的输出不变, 但是第二栏显示的是对象的工作修订版. 第三与第四栏显示上次修改的修订版, 还有谁修改了它.

以上的 **svn status** 的行为都不会存取到档案库, 他们只会比较工作复本与本地端的 **.svn** 目录的描述数据. 最后要提的是 **--show-updates (-u)** 选项, 这就会存取档案库, 并且显示已经 *过时* 的信息:

```
$ svn status --show-updates --verbose
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
```

请注意那两个星号: 如果你现在执行 **svn update**, 那么你会取得 **README** 与 **trout.c** 的更新. 这告诉你一件很重要的事情— 你需要在你送交变更之前, 先作更新, 自服务器取得 **README** 档案的更新, 不然档案库 会因档案过时而拒绝你的送交动作. (稍后会着墨这个课题.)

svn diff

另一个检视更动的方法, 是使用 **svn diff** 命令. 你可以得知 *确切* 修改的地方, 只要不带自变量执行 **svn diff** 即可, 它会以统一差异格式 (**unified diff**) 格式显示档案的更动: [\[3\]](#)

```

$ svn diff
Index: ./bar.c
=====
--- ./bar.c
+++ ./bar.c      Mon Jul 15 17:58:18 2002
@@ -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
+++ ./README      Mon Jul 15 17:58:18 2002
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: ./stuff/fish.c
=====
--- ./stuff/fish.c
+++ ./stuff/fish.c  Mon Jul 15 17:58:18 2002
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: ./stuff/things/bloo.h
=====
--- ./stuff/things/bloo.h
+++ ./stuff/things/bloo.h  Mon Jul 15 17:58:18 2002
+Here is a new file to describe
+things about bloo.

```

svn diff 命令藉由比较你的工作复本, 以及 .svn 里的“原始未更动”复本, 以产生这个输出. 预定要加入的档案, 会以全部新增的方式显示, 而预定要删除的档案则以全部删除的方式显示.

输出是以 **统一差异格式** 显示的. 也就是说, 删除的文字列是以 - 开头的, 新增的文字列是以 + 开头的. **svn diff** 也会显示可供 **patch** 使用的文件名与偏移信息, 所以你可以将 diff 输出重导向至档案, 以产生“修补档”:

```
$ svn diff > patchfile
```

举例来说, 你可以在送交之前, 先把修补文件以电子邮件寄给另一个发展人员检阅, 或是测试有没有问题.

svn revert

假设你现在看了以上的 `diff` 输出, 发现你对 `README` 所作的修改是不对的; 也许你不小心把那段文字打错了档案.

现在就是使用 **svn revert** 的绝佳机会.

```
$ svn revert README
Reverted ./README
```

Subversion 会使用 `.svn` 里的“原始未更动”复本来盖写档案, 将它回复至修改之前的状态. 不过 **svn revert** 也能取消 *任何* 预定的动作—举个例子, 你可能会决定根本不必增加一个新档案:

```
$ svn status foo
?      foo

$ svn add foo
A      foo

$ svn revert foo
Reverted foo

$ svn status foo
?      foo
```

Note

svn revert `ITEM` 的效果, 就跟自工作复本删除 `ITEM`, 然后执行 **svn update** `ITEM` 是一样的. 但是如果你要回复一个档案, **svn revert** 有一个很重要的差异 — 它不需要与档案库沟通, 就能回复你的档案.

或者你不该从版本控制之中删除一个档案:

```
$ svn status README
      README

$ svn delete README
D      README

$ svn revert README
Reverted README

$ svn status README
      README
```

看吧! 不用网络!

这三个命令 (**svn status**, **svn diff**, 以及 **svn revert**) 都可以在没有网络可用的情况下使用. 在你没有网络联机的情况下, 还是可以管理进行中的更动, 像是在旅途的飞机上, 在通勤电车中, 或是在海滩上 **hack** 程序.

Subversion 提供如此功能的方法, 是将每一个受版本控制的档案的原始未更动复本, 储存于 `.svn` 这个管理目录中. 如此, Subversion 就可以在没有 *网络存取* 的情况下, 对这些档案的本地修改进行报告—或是复原—. 这样的快取 (称为 "文件参考基础 (text-base)") 也让 Subversion 在送交的动作中, 能够将使用者的本地端修改, 以与原始未更动档案的差异的压缩形式来传送. 有这样的快取是相当大的好处—就算你有快速的网络联机, 只传送档案差异还是要比传送整个档案要快得多. 你可能不会觉得这有什么重要的, 但是请想象一下, 如果要送交的是 400MB 大的档案中的一行更动, 而你还是得将整个档案传回服务器的情况!

解决冲突 (合并他人的更动)

我们之前已经看过 **svn status -u** 如何预测冲突. 假设你执行了 **svn update**, 然后有趣的事情发生了:

```
$ svn update
U  ./INSTALL
G  ./README
C  ./bar.c
```

代码 `u` 与 `g` 没什么好担心的; 这几个档案都很顺利地接受了来自档案库的更动. 标示有 `u` 的档案, 表示它没有本地端的更动, 但是更新了档案库的更动. `g` 代表的它已经合并更动, 也就是说有本地端的更动, 但是来自档案库的更动并没有与它重迭.

但是 `c` 表示有冲突, 也就是说来自服务器的更动与你的更动有重迭的地方, 而现在你必须手动在这两者之间作选择.

不管冲突于哪里发生, 客户端的 Subversion 会作三件事:

- Subversion 在更新时, 会显示 `c`, 并且记得这个档案 “有冲突”.
- Subversion 会将 *冲突标记* 置于档案中, 明确地将重迭的部份标示出来.
- 对每一个冲突的档案而言, Subversion 会额外放置三个档案在你的工作复本中:

`filename.mine`

这是你在更新工作复本前, 就在工作复本中的档案—也就是说, 它没有冲突标记. 这个档案就只有你的最新更动, 没有包含其它的东西.

`filename.r 旧版号`

这是在你更新工作副本前, **BASE** 修订版的档案. 也就是在你开始进行修改之前所取出的档案.

`filename.r` 新版号

这是 **Subversion** 客户端在你更新工作副本时, 刚从服务器取得的档案. 这个档案对应的是档案库中的 **HEAD** 修订版的档案.

这里的 旧版号 是该档案在 `.svn` 目录中的修订版号, 而 新版号 是档案库中的 **HEAD** 修订版号.

举例来说, Sally 对档案库中的 `sandwich.txt` 作了修改. Harry 则刚在他的工作副本中, 修改了这个档案, 然后将它登录进去. Sally 在将档案登录时, 先执行更新动作, 结果她得到了一个冲突状况:

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

此时, **Subversion** 将 不允许你再送交档案 `sandwich.txt`, 除非这三个临时的档案都被删除.

```
$ svn commit --message "Add a few more things"
svn: A conflict in the working copy obstructs the current operation
svn: Commit failed (details follow):
svn: Aborting commit: '/home/sally/svn-work/sandwich.txt' remains in conflict.
```

如果你遇到了冲突, 你必须进行以下三项之一:

- “手动” 合并发生冲突的文字 (藉由检视与编辑档案里的冲突标记).
- 将某一个临时档案复制并盖过你的工作档.
- 执行 **svn revert <filename>**, 将你的本地更动全部舍弃.

当你解决了冲突之后, 你必须执行 **svn resolved**, 让 **Subversion** 知道. 如此会删除这三个临时档案, **Subversion** 就不会认为这个档案还是处于冲突的状态. [\[4\]](#)

```
$ svn resolved sandwich.txt
Resolved conflicted state of sandwich.txt
```

手动合并冲突

第一次尝试手动合并冲突可能会觉得很恐怖,但是经过少许的练习后,就会觉得和从脚踏车上摔下来一样,没什么大不了的。

以下是个范例. 假设因为你与协同工作人员 Sally 的沟通有误,两个人都同时编辑了 sandwich.txt. Sally 送交了她的更动,而你更新工作复本时就会得到一个冲突. 我们要编辑 sandwich.txt 以解决冲突. 首先,让我们先看看这个档案:

```
$ 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
```

这些小于符号, 等于符号, 以及大于符号, 称为 *冲突标记*. 夹在前两种标记之间的文字, 就是你在冲突区域所作的变更:

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

夹在后两组冲突标记的文字, 来自于 Sally 的送交更动:

```
=====
Sauerkraut
Grilled Chickenn
>>>>>>> .r2
```

通常你不会直接删掉冲突标记与 Sally 的更动—当她看到 sandwich, 发现与她预期的不同时, 可是会吓个老大一跳. 所以现在请拿起你的电话, 或是走过办公室, 跟她解释在意大利食品店, 是买不到德国泡菜的。^[5] 当你们都同意你将存入的变更, 请编辑你的档案, 将冲突标记移除.

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

现在执行 **svn resolved**, 你就可以送交你的更动.

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

请记住, 如果在编辑冲突档案时, 搞不清楚该怎么改, 你还可以参考 **Subversion** 在工作复本中建立的三个档案— 还包括你在更新前, 已修改过的档案.

将档案复制并盖过你的工作档

如果你遇到了冲突, 但是决定要舍弃你所作的更动, 你可以直接将 **Subversion** 建立的暂存盘之一复制并盖过你的工作档:

```
$ svn update
C sandwich.txt
Updated to revision 2.
$ ls sandwich.*
sandwich.txt sandwich.txt.mine sandwich.txt.r2 sandwich.txt.r1
$ cp sandwich.txt.r2 sandwich.txt
$ svn resolved sandwich.txt
$ svn commit -m "Go ahead and use Sally's sandwich, discarding my
edits."
```

弃踢: 使用 **svn revert**

如果你遇到了冲突, 经检视之后, 决定要丢弃你的更动, 再重新开始你的编辑工作, 只要回复所作的更动即可:

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

请注意, 你在回复一个冲突的档案时, 不必再执行 **svn resolved**.

现在你可以登录你的更动了. 请注意 **svn resolved** 并不像本章其它的命令一样, 它需要一个自变量来执行. 不管在什么情况下, 你必须非常小心, 只有当你非常确定已经解决了档案中的冲突, 才执行 **svn resolved** — 当临时档案被移除后, 就算档案中还有冲突标记, Subversion 还是会让你送交档案.

送交更动

终于到这里了! 你已经完成了你的编辑, 从服务器合并更动的部份, 并且准备要将你的更动送交至档案库.

svn commit 命令会将所有的更动送至档案库. 当你送交了一个更动, 你需要给它一个 *记录讯息*, 说明一下你的更动. 你的记录讯息会附在你建立的新修订版上. 如果记录讯息很简短的话, 你可能会希望透过命令列的 `--message` (或 `-m`) 选项来指定:

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

但是如果你是边工作边写记录讯息的话, 你大概会希望直接告诉 Subversion, 从 `-file` 选项所指定的档案来取得记录讯息:

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

如果你并没有指定 `--message` 或 `--file`, 那么 Subversion 会自动叫用你偏好的编辑器 (定义于环境变量 `$EDITOR` 中) 来编辑记录讯息.

Tip

如果你在编辑器中写了送交讯息, 然后决定要取消这次的送交, 只要不储存更动, 直接结束编辑器即可. 如果你已经储存了送交讯息, 只要把文字都删除, 然后再储存一次即可.

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

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

档案库并不会知道,也不管你的更动是否有意义;它只会检查是否有人在你没注意时,也修改了你所更动的档案.如果有人 *也* 修改到了,那么整个送交就会失败,并以一则讯息提示你,有一个或多个档案已经过时了:

```
$ svn commit --message "Add another rule"
Sending          rules.txt
svn: Transaction is out of date
svn: Commit failed (details follow):
svn: out of date: `rules.txt' in txn `g'
$
```

此时,你需要执行 **svn update**, 处理产生的合并或冲突,然后再试着送交一次.

这样就涵盖了 **Subversion** 的基本工作流程. **Subversion** 还有许多功能,可用来管理你的档案库与工作复本,但是我们在本章所介绍的命令,已足以让你轻松地应付日常工作所需.

检视历史纪录

我们早先曾说过,档案库就像时光机器一样,它会记录所有曾送交过的更动,而且允许你检视档案与目录,以及伴随的描述数据的过去修订版,以了解过往的历史.透过一个 **Subversion** 的命令,你可以取出档案库 (或回复现有的工作复本),让它完全符合过去某一个修订版号,或是某一天的样子.不过呢,有的时候你只是想 *看看* 过去,而不是真的 *回到* 过去.

有几个命令,都能够从档案库提供历史数据:

svn log

给你比较广的信息: 附加在修订版的记录讯息,以及每一个修订版所更动的路径.

svn diff

提供档案随着时间的确切变更内容.

svn cat

这是用来取得任何存于某一特定修订版的档案,并将档案内容显示在屏幕上.

svn list

用来显示任何指定修订版的目录内的档案.

svn log

要找出某一档案或目录的历史纪录信息, 请用 **svn log** 命令. **svn log** 可告诉你谁改了档案或目录, 该修订版的日期与时间. 另外, 如果有提供送交时的记录讯息, 它也会一并显示出来.

```
$ svn log
-----
---
r3:  sally | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line
Added include lines and corrected # of cheese slices.
-----
---
r2:  harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line
Added main() methods.
-----
---
r1:  sally | Mon, 15 Jul 2002 17:40:08 -0500 | 2 lines
Initial import
-----
---
```

请注意预设记录讯息是以 *反向时间顺序* 显示的. 如果你希望以特定顺序显示某个范围的修订版, 或是仅仅某一个修订版而已, 请使用 `--revision (-r)` 选项:

```
$ svn log --revision 5:19      # 以时间顺序, 显示 5 到 19 的纪录讯息
$ svn log -r 19:5              # 以反向时间顺序, 显示 5 到 19 的纪录讯息
$ svn log -r 8                  # 显示修订版 8 的纪录讯息
```

你也可以检视某一个档案或目录的讯息历程. 举个例子:

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

这些就 *只会* 显示指定档案 (或是 URL) 变动过的修订版纪录讯息.

如果你还想要更多有关于档案或目录的信息, **svn log** 也有 `--verbose (-v)` 详细讯息输出选项. 由于 Subversion 允许你移动复制档案或目录, 追踪档案系统内的路径更动是很重要的, 所以开启详细讯息输出的话, *svn log* 会在修订版的输出中, 包含一串 *变动路径* 的列表:

```
$ svn log -r 8 -v
```

```
-----  
---  
r8:  sally | 2002-07-14 08:15:29 -0500 | 1 line  
Changed paths:  
U /trunk/code/foo.c  
U /trunk/code/bar.h  
A /trunk/code/doc/README
```

```
Frozzled the sub-space winch.  
-----  
---
```

为什么 **svn log** 给我一个空白的回应??

在使用 Subversion 一阵子之后,大部份的使用者会遇到像这样的状况:

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

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

第一眼会觉得这是个错误,但是你要知道,修订版是适用于整个档案库的, **svn log** 运作的目标是档案库里的路径 (如果你没有指定路径, Subversion 预设为 "."). 如果你工作的工作复本子目录之下并没有更动档案, 结果就是取得空白的纪录讯息. 如果你想知道该修订版里更动了什么, 试着在工作复本的上层目录执行同样的命令.

svn diff

我们在前面已经看过 **svn diff**— 它会以统一差异格式来显示档案的差异; 在我们送交至档案库之前, 可用来显示我们对本地的工作复本所作的修改.

事实上, **svn diff** 总共有 三种 不同的用法:

- 检视本地端更动
- 比较档案库与本地复本
- 档案库与档案库之间的比较

检视本地端更动

我们已经看过, 不带自变量执行 **svn diff**, 会比较 `.svn` 区域里的“原始未更动”复本与工作复本之间的差异:

```
$ 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
$
```

比较档案库与本地复本

如果只有传递一个 `--revision (-v)` 的修订版号, 那么你的工作复本会与指定的档案库修订版作比较.

```
$ svn diff --revision 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
$
```

档案库与档案库之间的比较

如果透过 `--revision (-r)`, 传递两个以冒号隔开的修订版号, 那么这两个修订版号会直接拿来比较.

```
$ svn diff --revision 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 closed
$
```

svn diff 不但能拿来比较工作复本与档案库的档案, 如果你的自变量是 URL 的话, 就可以检视档案库中的两个对象, 甚至完全不需要先有工作复本. 如果你想看的更动, 是本地机器没有工作复本的档案, 这样就非常有用:

```
$ svn diff --revision 4:5 http://svn.red-  
bean.com/repos/example/trunk/text/rules.txt  
...  
$
```

svn cat

如果你想要检视某一档案早先的版本, 而不是两个档案之间的差异, 你可以使用 **svn cat**:

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

你可以将输出直接重导到一个档案中:

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

你大概在想, 为何我们不使用 **svn update --revision** 将档案更新至旧的修订版. 有几个理由, 让我们会想用 **svn cat**.

首先, 你可能想要使用外部的 **diff** 程序, 来看一个档案两个不同版本之间的差异 (也许是图形界面的, 也许你的档案已经是这样的格式, 输出统一差异格式是完全没有意思的). 在这种情况下, 你需要取得旧版本的复本, 重导向至一个档案中, 然后将它与在你的工作目录中的档案一起传给你的外部 **diff** 程序.

有的时候直接看整个先前版本的档案, 要比只看它与其它修订版的差异要方便得多.

svn list

svn list 命令可用来显示档案库的目录中有什么档案, 而不必实际将档案下载至本地机器中:

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

```
branches/  
clients/  
tags/  
trunk/
```

如果你想要更详细的列表, 加上 `--verbose (-v)` 选项, 就可以取得像这样的输出.

```
$ svn list --verbose http://svn.collab.net/repos/svn  
  2755 harry          1331 Jul 28 02:07 README  
  2773 sally          Jul 29 15:07 branches/  
  2769 sally          Jul 29 12:07 clients/  
  2698 harry          Jul 24 18:07 tags/  
  2785 sally          Jul 29 19:07 trunk/
```

这些字段告诉你, 这个档案或目录最后一次修改的修订版, 修改它的使用者, 档案则会有档案大小, 上一次更新的日期, 以及该项目的名称.

对历史纪录的最后叮咛

[除了以上的命令](#), 你还可以使用 **svn update** 与 **svn checkout**, 加上 `--revision` 选项, 就可以将整个工作复本带回到“过去的时间”^[6]:

```
$ svn checkout --revision 1729 # 取出修订版 1729 为新的工作复本  
...  
$ svn update --revision 1729 # 更新现有的工作复本为修订版 1729  
...
```

其它有用的命令

虽然不像本章前面讨论过的命令那么常用到, 这些命令还是有需要的时候.

svn cleanup

当 Subversion 修改你的工作复本时 (或是任何在 `.svn` 的信息) 时, 它会试着尽量以安全的方式进行. 在修改任何东西前, 会先将其意图写入一个纪录文件, 执行纪录文件里的命令, 然后删除纪录文件 (这很像日志档案系统的设计). 如果一个 Subversion 的动作被中断 (像是你按了 **Control-C**, 或是机器当掉了), 那么纪录文件会继续存在磁盘之中. 藉由重新执行纪录文件, Subversion 可以完成先前进行的动作, 而你的工作复本也能回复到一致的状态.

而这就是 **svn cleanup** 所作的事: 它会搜寻你的工作复本, 执行任何遗留下来的纪录文件, 移除动作进行时所使用的锁定档. 如果 Subversion 曾说过工作复本的某些地方被“锁定”了, 那么这就是你该执行的命令. 另外, **svn status** 也会在被锁定的项目旁显示 **L**:

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

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

svn import

svn import 汇入命令, 是用来将未纳入版本控制的档案树放进档案库的快速方法.

```
$ svnadmin create /usr/local/svn/newrepos
$ svn import mytree file:///usr/local/svn/newrepos/fooproject
Adding  mytree/foo.c
Adding  mytree/bar.c
Adding  mytree/subdir
Adding  mytree/subdir/quux.h
Transmitting file data....
Committed revision 1.
```

以上的例子, 是将目录 `mytree` 的内容, 直接放至档案库的 `fooproject` 目录里:

```
/fooproject/foo.c
/fooproject/bar.c
/fooproject/subdir
/fooproject/subdir/quux.h
```

摘要

现在我们已经涵盖了大多数的 Subversion 客户端命令. 显而未提的部份, 是处理分支与合并 (参见 [Chapter 4, 分支与合并](#)), 以及处理性质 (参见 [the section called “性质”](#)) 的命令. 不过, 你可能会想花点时间略读 [Chapter 8, 完整 Subversion 参考手册](#), 以了解 Subversion 还有哪些不同的命令— 还有你能如何使用它们, 让你的工作能够轻松点.

^[1] 嗯, 除了一件事, 就是“工作复本”中的每个目录都会有一个 `.svn` 子目录. 不过现在讲这个言之过早.

^[2] 当然啰, 没有任何东西会真正地自档案库中删除— 它只是在档案库的 `HEAD` 版本之后被删除了. 你还是可以取回被删除的东西, 只要取出 (或更新工作复本) 到你删除的修订版还早的版本即可.

^[3] Subversion 使用它自己内部的差异引擎, 预设使用的是统一差异格式. 如果你想要使用不同的差异格式输出, 请以 `--diff-cmd` 指定外部差异程序, 并以 `--extensions` 选项传递任何你想要使用的旗标. 举个例子, 要以文脉输出格式 (context output format) 来看 `foo.c` 的本地端差异, 但是要忽略空格符的更动, 你可以执行 “`svn diff --diff-cmd /usr/bin/diff --extensions '-bc' foo.c`”.

^[4] 你绝对可以自行移除暂存盘, 不过 Subversion 可以帮你作好的话, 你还会想要自己来吗? 我们不这么认为.

^[5] And if you ask them for it, they may very well ride you out of town on a rail.

^[6] 看到了吧? 我们跟你讲过, Subversion 是一个时光机器.

Chapter 4. 分支与合并

Table of Contents

[何谓分支?](#)

[使用分支](#)

[建立一个分支](#)

[与分支共事](#)

[事情的内涵](#)

[在分支之间复制更动](#)

[复制特定的更动](#)

[重复合并问题](#)

[合并整个分支](#)

[从档案库移除一个更动](#)

[切换工作复本](#)

[标记](#)

[建立一个简单的标记](#)

[建立一个复杂的标记](#)

[分支维护](#)

[档案库配置](#)

[资料生命周期](#)

[摘要](#)

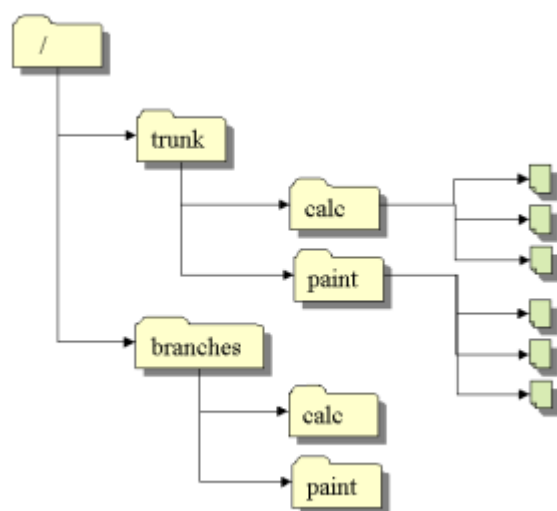
分支, 标记, 以及合并, 几乎都是所有版本控制系统的共通概念. 如果你对这些概念不了解, 我们在这一章提供了一个不错的介绍. 如果你熟悉这些概念的话, 那么我们希望你能对 Subversion 如何实作这些概念的作法感到有兴趣.

分支是版本控制的基本部份. 如果你要让 Subversion 来管理你的数据, 那么你一定依赖这个功能. 本章假设你已经了解 Subversion 的基本概念. ([Chapter 2, 基本概念](#)).

此时,你应该了解,每一次的送交都会在档案库建立一个全新的档案系统树(称为“修订版”).如果不是的话,请回去 [the section called “修订版本”](#) 阅读有关修订版的概念.

我们会使用第 2 章的范例,作为本章的例子.请记得你和你的协同工作者 Sally,两个人都共享同一个档案库,其中包含 paint 和 calc 两个计划.不过这一次有人在档案库中,建立了两个最顶层的目录,称为 trunk 与 branches.这两个计划本身是 trunk 的子目录,我们稍后会解释.

Figure 4.2. 起始档案库的配置



就像前面的一样,假设你与 Sally 两个人都有 `/trunk/calc` 计划的工作复本.

假设你被指定了一个工作,要对该计划进行全面性的重新整理.这需要花费许多时间撰写,而且会影响计划中全部的档案.问题是你并不想妨碍到 Sally 的工作,此时她正在修正随处可见的臭虫.她所依赖的,就是计划的每一份最新修订版都是可用的.如果你开始送交你所作的更动,肯定会打断 Sally 的工作.

有个方法,就是与世隔绝:在一两个星期中,你和 Sally 不分享信息.也就是说,开始在工作复本中修改并重整所有的档案,但是在完成工作之前,不进行任何送交或更新.这样会有一些问题.首先,它并不安全.大多数的人喜欢时常将他们的工作存回至档案库中,以防工作复本发生什么致命的意外.如果你在不同的计算机上工作(也许你在两台不同的计算机上,都有 `/trunk/calc` 的工作复本),你就需要在这两台计算机之间手动复制你的修改,不然就是只在一台计算机上工作.最后,当你完成了之后,你可能会发现送交更动是相当困难的. Sally (或其它人)也许会在档案库作一些更动,而它们很难合并到你的工作复本中 — 尤其是全部一口气来的时候.

较好的方式,就是在档案库中建立你自己的分支,或称为发展支线.这让你能够常常储存进行到一半的工作,又不会妨碍到其它人,而你还可以选择性地与其它协同工作者分享信息.稍候你就可以了解,这是如何运作的.

建立一个分支

建立一个分支相当地容易 — 利用 **svn copy** 命令, 在档案库中建立计划的复本. Subversion 不仅能够复制单一档案, 整个目录也没有问题. 既然如此, 你会想要产生一份 `/trunk/calc` 目录的复本. 新的复本该放在哪里? 哪里都可以 — 这实际上是计划的原则. 我们假设你所属团队的原则, 是将分支置于档案库里的 `/branches/calc` 区域, 而你想将分支命名为 `"my-calc-branch"`. 你想要建立一个新的目录 `/branches/calc/my-calc-branch`, 它一开始为 `/trunk/calc` 的复本.

建立一份复本有两种不同的方法. 我们先从麻烦的方法开始, 让这个概念能够更清楚. 一开始, 请先取出档案库根目录 (`/`) 的工作复本:

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

现在, 建立复本就只是将两个工作复本路径传给 **svn copy** 命令:

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

在这里, **svn copy** 命令会将 `trunk/calc` 工作目录里所有的东西复制到 `branches/calc/my-calc-branch`. 你可以从 **svn status** 命令看得出来, 新的目录现在预计要被新增至档案库中. 不过请注意字母 A 旁边的 + 号. 它表示这预计要新增的项目, 实际上是某项目的复本, 而不是全新的东西. 当你送交你的更动时, Subversion 会在档案库复制 `/trunk/calc` 以建立 `/branches/calc/my-calc-branch`, 而不是将所有工作复本数据再经由网络重传一次.

```
$ svn commit -m "Creating a private branch of /trunk/calc."
Adding branches/calc/my-calc-branch
Committed revision 341.
```

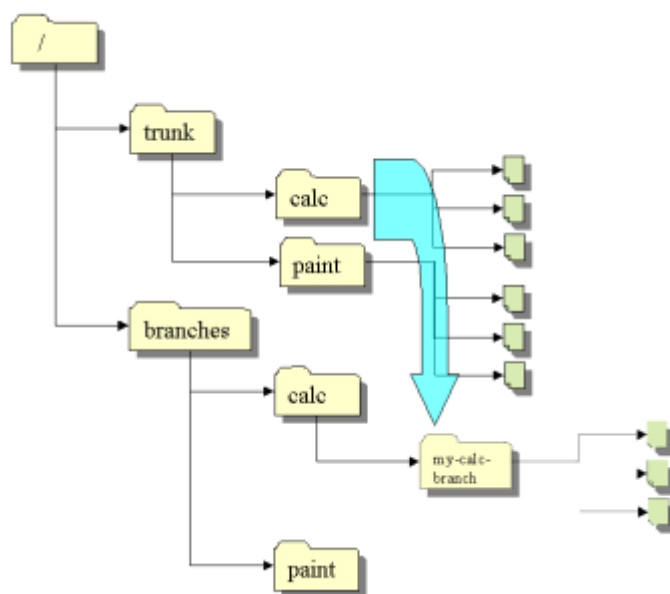
现在介绍的是另一个建立分支较容易的方法, 我们该在一开始就告诉你: **svn copy** 可以处理两个 URL.


```
$ svn copy http://svn.example.com/repos/trunk/calc \
    http://svn.example.com/repos/branches/calc/my-calc-branch \
    -m "Creating a private branch of /trunk/calc"
```

Committed revision 341.

这两个方法实际上没什么差别. 这两个方法都在修订版 341 中建立一个新的目录, 而这个新目录是 /trunk/calc 的复本. 但是呢, 请注意第二个方法进行的是 *立即* 送交.^[7] 这个方法比较容易的原因, 是它不需要你先取出档案库的东西. 事实上, 这个技巧甚至不要求你得先有工作复本.

Figure 4.3. 有新复本的档案库



廉价复制

Subversion 的档案库设计相当地好. 当你复制一个目录时, 你不必担心档案库会变得非常大 — Subversion 不会真的复制数据. 相反地, 它建立一个指向 *现存* 档案树的目录项目. 如果你是一个 Unix 使用者, 这个概念就像 *hard-link* 一样. 从这开始, 这个复本被称为是 "延迟 (lazy)" 的. 也就是说, 如果对一个复制目录里的档案送交更动, 那么就只有那个档案会被更动 — 其余的档案还是以源目录里的源文件的连结形式存在.

这就是为什么你会常常听到 Subversion 使用者在谈论 "廉价复制". 它完全不管目录有多大 — 建立复本, 只需要极少而固定的时间. 事实上, 这个功能是 Subversion 的送交如何工作的基础: 每一个修订版是前一个修订版的 "廉价复本", 而其中只有几个项目被延迟修改. (想知道更多的细节, 请上 Subversion 的网站, 看看 Subversion 设计文件里的 "bubble up" 方法).

这里的重点, 就是复制是很廉价的, 不管是时间还是空间. 请依你所需, 尽量建立分支.

与分支共事

现在你已经建立了一个新的计划分支, 你可以取出新的工作复本, 然后开始使用它:

```
$ svn checkout http://svn.example.com/repos/branches/calc/my-calc-branch
A my-calc-branch/Makefile
A my-calc-branch/integer.c
A my-calc-branch/button.c
Checked out revision 341.
```

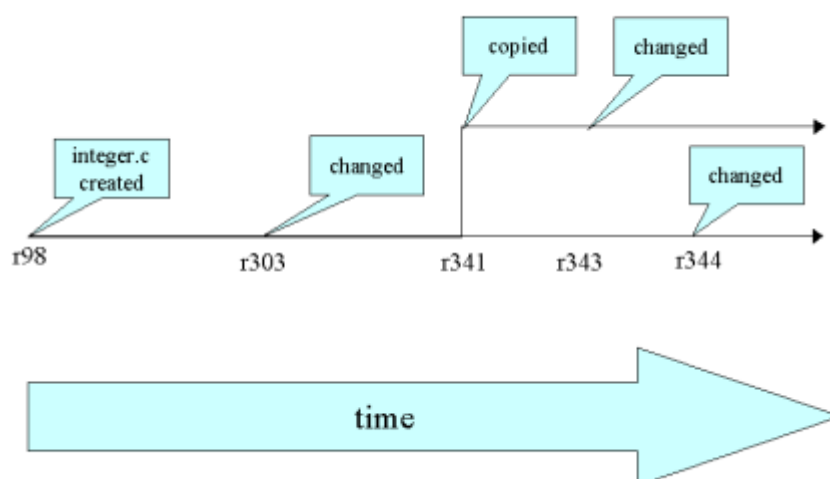
这个工作复本没有什么特殊的地方; 它只是映像出另一个档案库的位置. 但是当你送交更动时, Sally 在更新时也不会看到它们. 她的工作复本是在 `/trunk/calc`.

让我们假装一个星期已经过去了, 而我们有以下的送交更动:

- 你修改了 `/branches/calc/my-calc-branch/button.c`, 产生修订版 342.
- 你修改了 `/branches/calc/my-calc-branch/integer.c`, 产生修订版 343.
- Sally 修改了 `/trunk/calc/integer.c`, 产生修订版 344.

现在 `integer.c` 有两条独立的发展支线:

Figure 4.4. 档案历史的分支



如果你看看你自己的 `integer.c` 复本的历程纪录, 事情变得满有趣的:

```

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

$ svn log integer.c
-----
---
r343:  user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
* integer.c:  frozzled the wazjub.

-----
---
r303:  sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
* integer.c:  changed a docstring.

-----
---
r98:   sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
* integer.c:  adding this file to the project.

-----
---

```

请注意 Subversion 会依时间回溯 integer.c 的历程纪录, 直到它被复制之时. (请记住你的分支于修订版 341 时建立.) 现在看看 Sally 对她自己的档案复本执行相同命令时, 会发生什么事:

```

$ pwd
/home/sally/calc

$ svn log integer.c
-----
---
r344:  sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
* integer.c:  fix a bunch of spelling errors.

-----
---
r303:  sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
* integer.c:  changed a docstring.

-----
---
r98:   sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
* integer.c:  adding this file to the project.

-----
---

```

Sally 会看到她自己的修订版 344 的更动, 但是不会看到你在修订版 343 的更动. 就 Subversion 的认知, 这两个送交更动影响的是档案库里不同位置的不同档案.

但是 Subversion 会显示这两个档案有共同的历史. 在修订版 341 建立分支复本之前, 它们两个是同一个档案. 这也是为什么你和 Sally 两个都会看到修订版 303 与 98.

事情的内涵

这一节中, 有两点课题是你应该要记住的.

1. 不像其它的版本控制系统, Subversion 的分支是以 *正常的档案系统目录* 存在于档案库里, 而不是其它独立的维度.
2. Subversion 内部并没有 "分支" 的概念—只有复本而已. 如果你复制了一个目录, 结果的目录会是 "分支", 只是因为 你赋与它这个意义. 你可以把它想成不同的意义, 或是以不同的方式处理它, 但是对 Subversion 而言, 它只是一个因复制而产生的普通目录而已.

在分支之间复制更动

现在你与 Sally 在同一项目的不同平行分支上工作: 你的是私有分支, 而 Sally 则是 *主干(trunk)*, 也就是主要的发展线.

对于有众多贡献人员的项目来说, 大多数的人都有主干的工作复本是很常见的. 要是有哪个人需要进行可能会妨碍到主干的长期变动, 标准的作法是建立一个私有的分支, 然后将更动都送交至该分支, 直到工作结束为止.

所以, 好消息就是你和 Sally 两个不会互相影响, 坏消息是这两个很容易就跑得太远了. 请记得 "与世隔绝" 策略的一个问题, 就是在你结束分支的工作之后, 几乎不可能将你所作的更动, 在不产生大量冲突的情况下, 把它合并回主分支去.

相反地, 你和 Sally 应该在你工作时, 也一直分享彼此的更动. 哪些更动应该被分享, 完全由你来决定; Subversion 提供你在各个分支之间 "复制" 选择性更动的能力. 而当你已经处理完你的分支时, 整个分支的所有更动就可以复制回主干去.

复制特定的更动

在前一节中, 我们提到你与 Sally 两个在不同的分支中, 都修改了 `integer.c`. 如果你看看 Sally 在修订版 334 的记录讯息, 你可以看到她修改了几个拼字错误. 毫无疑问的, 你这相同档案的复本也还有同样的拼字错误. 很有可能你未来对这个档案的更动, 是针对有拼字错误的地方, 那么某一天你要合并分支的时候, 也就很有可能会产生冲突. 所以最好是现在就取得 Sally 的更动, 在你开始在同一地方进行大改 *之前*.

现在该是使用 `svn merge` 命令的时候. 这个命令结果是很像 `svn diff` 命令 (这在第 3 章就介绍过了). 这两个命令都可以比较两个档案库里对象, 并表示它们之间

的差异. 举个例子, 你可以要求 **svn diff** 为你显示 Sally 在修订版 344 时所作的更动:

```
$ svn diff -r 343:344 http://svn.example.com/repos/trunk/calc

Index: integer.c
=====
--- integer.c      (revision 343)
+++ integer.c      (revision 344)
@@ -147,7 +147,7 @@
     case 6:  sprintf(info->operating_system, "HPFS (OS/2 or NT)");
break;
     case 7:  sprintf(info->operating_system, "Macintosh"); break;
     case 8:  sprintf(info->operating_system, "Z-System"); break;
-   case 9:  sprintf(info->operating_system, "CPM"); break;
+   case 9:  sprintf(info->operating_system, "CP/M"); break;
     case 10: sprintf(info->operating_system, "TOPS-20"); break;
     case 11: sprintf(info->operating_system, "NTFS (Windows NT)");
break;
     case 12: sprintf(info->operating_system, "QDOS"); break;
@@ -164,7 +164,7 @@
     low = (unsigned short) read_byte(gzfile); /* read LSB */
     high = (unsigned short) read_byte(gzfile); /* read MSB */
     high = high << 8; /* interpret MSB correctly */
-   total = low + high; /* add them together for correct total */
+   total = low + high; /* add them together for correct total */

     info->extra_header = (unsigned char *) my_malloc(total);
     fread(info->extra_header, total, 1, gzfile);
@@ -241,7 +241,7 @@
     Store the offset with ftell() ! */

     if ((info->data_offset = ftell(gzfile)) == -1) {
-   printf("error: ftell() returned -1.\n");
+   printf("error: ftell() returned -1.\n");
     exit(1);
   }

@@ -249,7 +249,7 @@
     printf("I believe start of compressed data is %u\n", info-
>data_offset);
     #endif

-   /* Set position eight bytes from the end of the file. */
+   /* Set position eight bytes from the end of the file. */

     if (fseek(gzfile, -8, SEEK_END)) {
     printf("error: fseek() returned non-zero\n");
```

svn merge 几乎是完全一样的. 但是它不会在终端机上显示差异, 而是直接当成 *本地修改* 套用到你的工作复本上:

```
$ svn merge -r 343:344 http://svn.example.com/repos/trunk/calc
U  integer.c

$ svn status
M  integer.c
```

svn merge 的输出, 显示了你的 `integer.c` 副本已经修补过了. 它现在包含了 Sally 的更动 — 这个更动从主干 "复制" 到你的私有分支内, 并以本地修改的形式存在. 此时, 你可检视本地的更动, 确认它没有问题.

在另一个场景中, 事情可能就不会这么顺利, 使得 `integer.c` 进入了冲突的状态. 你必须使用标准的步骤 (参见第 3 章) 来解决冲突, 如果你认为合并根本不是一个好主意的话, 直接舍弃它, 以 **svn revert** 来回复本地更动.

不过假装你已经检视过合并的更动, 你可以一如往常地使用 **svn commit** 来送交更动. 此时, 这个更动就被合并到你的 档案库分支. 在版本控制的术语中, 这个在不同的分支之间复制更动的动作, 称为 *移植* 更动.

为什么不改用修补呢?

你心中可能会有个疑问, 尤其你是 Unix 使用者的话: 为什么要使用 **svn merge** 呢? 为什么不使用操作系统中的 **patch** 命令来达成相同的工作呢? 举个例子:

```
$ svn diff -r 343:344 http://svn.example.com/repos/trunk/calc >
patchfile
$ patch -p0 < patchfile
Patching file integer.c using Plan A...
Hunk #1 succeeded at 147.
Hunk #2 succeeded at 164.
Hunk #3 succeeded at 241.
Hunk #4 succeeded at 249.
done
```

在这特定的情况, 没错, 是没有什么差别. 但是 **svn merge** 有一个优于 **patch** 程序的特殊功能. **patch** 所使用的档案格式有其限制; 它只能处理档案的内容而已. 它没有办法表达 档案树的变动, 诸如目录与档案的新增, 删除, 或是更名. 假如说 Sally 的更动新增了一个目录, 那么 **svn diff** 的输出完全无法显示. **svn diff** 只能输出功能有限制的修补格式, 所以有些东西就是无法显示出来.^[8] 但是 **svn merge** 命令可以藉由直接套用档案树的更动到你的工作副本中, 来表示这样的更动.

有件事要提醒一下: 虽然 **svn diff** 与 **svn merge** 的概念很类似, 但是它们在许多情况下的语法并不相同. 请阅读第 8 章, 或是使用 **svn help** 以了解细节. 举个例子, **svn merge** 需要一个工作副本路径作为目标, 也就是要套用档案树更动的地方. 如果目标没有指定的话, 它会假设你要执行以下的动作之一:

1. 你想要合并目录更动至目前的工作目录中.
2. 你想要合并特定档案的更动至目前工作目录中的同名档案中.

如果你要合并一个目录, 但是没有指定目标路径的话, **svn merge** 会假设第一种情况, 试着将更动套用到目前的工作目录中. 如果你要合并一个档案, 而那个档案 (或同名档案) 存在于目录的工作目录中, 那么 **svn merge** 会假设第二种情况, 并试着将更动套用到同名的本地端档案.

如果你要让更动套用到别的地方, 你要这样下:

```
$ svn merge -r 343:344 http://svn.example.com/repos/trunk/calc my-  
calc-branch  
U    my-calc-branch/integer.c
```

重复合并问题

合并更动听起来很简单, 但是实务上, 它可能让你很头痛. 问题在于如果你将更动从一个分支重复复制到另一个分支的话, 你很有可能不小心合并相同的更动 *两次*. 如果发生的话, 有可能并不会出什么问题. 当 **Subversion** 套用更动时, 一般来说, 它会帮你记住这个档案是已经有这个更动, 如果有的话, 不会发生任何事. 不过这个已套用的更动已经被更动的话, 你就会得到冲突. 理想的状况下, **Subversion** 会自动防止重复套用更动.

这是个困扰许多版本控制软件的问题, 包括 **CVS** 与 **Subversion**. 目前而言, 在 **Subversion** 中, 唯一防止这个问题的方法, 就是记录哪些更动已经合并了, 哪些还没. 当你建立一个分支目录时, 你必须记录它是从哪个修订版产生出来的 — 自己建立在别的地方. 当你合并一个修订版 (或是一个范围的修订版) 到工作复本时, 你也得把它们记录下来. 如果你忘了任何一个这样的信息, 你可以利用 **svn log -v 分支目录** 的输出来重新找出这个信息. 但是这里的重点, 是每一个后续的合并必须小心翼翼地手动建立, 以确定之前合并过的修订版不会再重新合并一遍.

当然啰, **Subversion** 预计在发行版 1.0 之后, 再来解决这个问题. 所有这样的合并信息可以在性质描述数据中找到 (参见 [the section called “性质”](#)), 这样 **Subversion** 有一天就能够自动地防止重复的合并.

合并整个分支

要让我们所举的实例更完整, 让我们把时间往后拉. 几天过去了, 主干与你的私人分支都有了许多更动. 假设你已经完成你的私有分支; 它该有的功能, 或是臭虫修正终于完成了, 现在你想把分支里所有的更动都合并回主干, 让别人也能享用.

所以, 这种情况我们该如何使用 **svn merge** 呢? 请记住这个命令会比较两个档案树, 然后把更动套用到工作复本去. 所以要接收这些更动, 你需要有主干的工作复本. 我们假设你手边还有一份 (完全更新的), 或者你已取出一份新的 /trunk/calc 工作复本.

但是要比哪两个档案树? 随便一想, 答案似乎很明显: 比较最新的主干树与你最新的分支树就好了. 但是注意 — 这个想法是 *错的*, 而且重创不少初学者! 由于 **svn merge** 的行为与 **svn diff** 相似, 比较最新的主干与分支树并 *不会* 直接给你你对分支所产生的更动: 它不只显示出你新加入的分支更动, 还会 *移除* 你从未在分支作过的主干更动.

想要表达只在你的分支中产生的更动, 你必须比较分支的初始状态, 以及其最终状态. 对你的分支执行 **svn log**, 你可以发现分支是在修订版 341 产生的. 而分支的最终状态, 只要使用修订版 HEAD 即可.

那么, 这就是最后的合并手续:

```
$ cd trunk/calc

$ svn merge -r 341:HEAD
http://svn.example.com/repos/branches/calc/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn status
M   integer.c
M   button.c
M   Makefile

[examine the diffs, compile, test, etc.]

$ svn commit -m "Merged all my-calc-branch changes into the trunk."
...
```

从档案库移除一个更动

svn merge 一个常用的用法, 就是回复一个已经送交的更动. 假设你以前在修订版 303 所作的更动, 修改了 `integer.c`, 这根本就是错误的, 它根本就不该被送交. 你可以在工作复本中使用 **svn merge** 来 "反悔" 这个更动, 然后再将这个本地更动送交回档案库. 你只要指定一个 *反向* 的差异即可:

```
$ svn merge -r 303:302 http://svn.example.com/trunk/calc
U   integer.c

$ svn status
M   integer.c

$ svn commit -m "Undoing change committed in r303."
Sending      integer.c
Transmitting file data .
Committed revision 350.
```

一种想象档案库修订版的方法, 就是将它们视为一群更动 (某些版本控制系统称这为 *更动组*). 藉由使用 `-r` 选项, 你可以要求 **svn merge** 将一个更动组, 或是一个范围的更动组, 套用到工作复本中. 在我们反悔更动的情况中, 我们要求 **svn merge** *逆向* 套到更动组 #303 到我们的工作复本中.

你要谨记在心, 像这样回复一个更动, 就跟其它的 **svn merge** 动作没什么两样, 所以你应该要使用 **svn status** 与 **svn diff** 来确认你的所作的跟你想要的是相同的,

然后再使用 **svn commit** 将最终的版本送交回档案库. 在送交之后, 该特定的更动组就不会再出现在 **HEAD** 修订版中.

当然啰, 你可能会想: 呃, 这根本就不是反悔先前的送交, 不是吗? 这个更动还是存在于修订版 303 中. 如果某人取出了修订版 303 与 349 之间的 某个 `calc` 计划的版本, 就会看到这个错误的更动, 对吧?

是的, 没有错. 当我们讲到 "移除" 一个更动时, 我们真正讲的是 "从 **HEAD** 移除". 原先的更动还是会存在于档案库的历程中. 大多数的情况下, 这样已经够好了. 大多数的人只对追踪一个计划的 **HEAD** 版本有兴趣而已. 不过还是可能存在着特殊的情况, 你会希望完全地消除所有该送交的证据 (也许某人不小心送交了一个机密文件). 但是它非常地不容易, 因为 **Subversion** 的设计, 就是不会丢弃任何信息. 修订版是不可变动的档案树, 它们一个个建立在另一个之上. 从历史进程移除一个修订版会产生骨牌效应, 会对所有后续的修订版造成混乱, 还有可能让所有的工作复本都无效.^[9]

切换工作复本

svn switch 命令可将现有的工作复本切换到不同的分支去. 虽然这个命令与分支运作没什么很直接的关系, 但是对使用者来说, 是个满不错的快捷方式. 我们早先的例子中, 在建立了自己的私有分支之后, 你要从新的档案库目录取出一个全新的工作复本. 但是你也可以改为要求 **Subversion**, 将你的 `/trunk/calc` 的工作复本改为映像到新的分支位置去:

```
$ cd calc

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

$ svn switch http://svn.example.com/branches/calc/my-calc-branch
U   integer.c
U   button.c
U   Makefile
Updated to revision 341.

$ svn info | grep URL
URL: http://svn.example.com/branches/calc/my-calc-branch
```

在 "切换" 到指定的分支之后, 你的工作复本与取出全新的目录没有什么差别. 而且使用这个命令是更有效率的, 因为分支通常都只有小部份不同而已. 服务器只会送出最小的, 足以让你的工作复本反映出分支目录的更动集合.

svn switch 命令也接受 `--revision (-r)` 选项, 所以工作复本并不一定得是分支的 "尖端".

当然啰, 大多数的项目要比你的 `calc` 范例要复杂得多了, 包括了许多子目录. **Subversion** 的使用者在使用分支时, 通常都依循一个特定的算法:

1. 将整个项目 '主干' 复制到新的分支目录.
2. 只切换 部份的主干工作复本, 以反映分支.

换句话说, 如果使用者知道分支的工作只需要在某个特定子目录下发生的话, 他们可以用 **svn switch**, 将他们工作复本的子目录转移到某特定的分支去. (有的时候, 使用者可能只要将某个工作档案切换到分支去!) 这样, 他们大部份的工作复本还是可以继续接受正常的 '主干' 更新, 但是被切换的部份就会保持不动 (除非有对该分支的送交更动). 这个功能对 "混合工作复本" 的概念, 提供了全新的维度, 工作复本不只是可以混合不同的工作修订版, 还可以混合不同的档案库位置.

如果你的工作复本包含了来自不同档案库位置的切换子档案树, 它们还是能够正常地工作. 当你更动时, 你会从每个子档案树收到修补更新. 当你送交更动时, 你的本地修改还是会以单一的, 不可分割的更动送回档案库.

请注意一点, 虽然你的工作复本可以对映到混合的档案库位置, 但是它们必须全在 同一个档案库中. Subversion 的档案库还无法彼此沟通; 这个功能预计在 Subversion 1.0 之后加入. [\[10\]](#)

切换与更新

你注意到 **svn switch** 与 **svn update** 看起来是一样的吗? **switch** 命令实际上是 **update** 命令的超集.

当你执行 **svn update** 时, 你是要求档案库比较两个档案树. 档案库会这么作, 然后再将两者差异的描述送回客户端. **svn switch** 与 **svn update** 之间的差异, 只是 **svn update** 比较的一定是两个相同的路径.

也就是说, 如果你的工作复本是映射到 `/trunk/calc`, 那么 **svn update** 会自动将你的 `/trunk/calc` 工作复本与 `/trunk/calc` 的 HEAD 修订版作比较. 如果你将工作复本切换到分支的话, 那么 **svn switch** 会比较你的 `/trunk/calc` 工作复本, 与 某一个在 HEAD 修订版的分支目录作比较.

换句话说, 更新是在时间轴上移动你的工作复本, 而切换则是在时间轴 与 空间轴上移动.

由于 **svn switch** 实际上是 **svn update** 的变形, 它们有相同的行为; 当有新的数据从档案库来的时候, 任何在工作复本里的本地修改还是会保存下来. 这可以让你进行各式各样的妙技.

举个例子, 假设你有一个 `/trunk` 的工作复本, 并且对它进行了一些修改, 然后你突然发现, 应该修改的是分支才对. 没问题! 当你以 **svn switch** 来切换工作复本到分支时, 本地更动还是会存在. 此时你就可以进行测试, 然后将它们送交回分支去.

标记

另一个常见的版本控制概念,就是 *标记*. 标记只是一个计划在时间轴上的 "快照". 在 **Subversion** 中, 这个概念已经随处都是了. 每一个档案库的修订版, 就是档案系统在每一次送交之后的快照.

但是呢, 人们通常会想要标记一个便于记忆的名称, 像是 "release-1.0". 而且他们只想要的, 只是一部份档案系统子目录的快照而已. 要记住某一个软件的 release-1.0 版, 对应的是修订版 4822 里的某个特定子目录, 毕竟是不太容易的.

建立一个简单的标记

再一次地, **svn copy** 又来解救众生了. 如果你想要建立一个完全符合 HEAD 修订版的 /trunk/calc 的快照, 那就建立一个它的复本:

```
$ svn copy http://svn.example.com/repos/trunk/calc \
           http://svn.example.com/repos/tags/calc/release-1.0 \
           -m "Tagging the 1.0 release of the 'calc' project."
```

Committed revision 351.

这个例子假设 /tags/calc 的目录已经存在了. 在复制动作完成后, 新的 release-1.0 目录就永远是这个计划的快照, 完全符合你建立复本时的 HEAD 修订版的内容. 当然啰, 你可能想要更准确地指定你所复制的修订版, 以免别人在你没注意时, 又送交了几个更动. 所以如果你知道 /trunk/calc 的 350 修订版, 就是你想要的快照的话, 你可以藉由传给 **svn copy** 命令的 -r 350 选项来指定它.

不过先等一下: 这个建立标记的步骤, 不是和我们用来建立分支的步骤是一样的吗? 没错, 事实上, 根本就是一样的. 在 **Subversion** 中, 标记与分支是没有任何分别的. 两者都只是透过复制而建立的寻常目录而已. 就像分支一样, 被复制的目录之所以为 "标记", 只是因为人们决定它是: 只要没有人再送交更动到这个目录中, 它就永远是个快照而已. 如果有人开始送交更动进去, 它就会变成一个分支.

如果你管理一个档案库的话, 有两种方法可以用来管理标记. 第一个方法是 "放任": 把它视为计划的原则, 决定标记放置的位置, 确定所有的使用者都知道如何处理复制进去的目录. (也就是说, 确定他们都知道别送交更动进去.) 第二个方法就很偏执: 你可以利用 **Subversion** 内附的存取控制的脚本档, 让别人只能在标记区域建立新的复本, 但是无法送交东西进去. (###cross-ref a section that demonstrates how to use these scripts?). 不过呢, 偏执的方法并不是那么必要的. 如果一个使用者不小心送交更动到标记目录, 你只要像前一节所讲的, 把它回复回来即可. 毕竟, 这可是版本控制软件.

建立一个复杂的标记

有的时候, 你想要你的 "快照" 要比单一修订版的单一目录要来得复杂一点.

举例来说, 假设你的计划比我们的 `calc` 例子要大得多: 也许它包含了几个子目录, 以及许多的档案. 在你工作之间, 你可能决定要建立一个工作复本, 它有特定的功能, 以及特定的臭虫修正. 想要达到这样的目的, 你可以藉由选择性地将档案与目录固定在某一个修订版, (大方地使用 **svn update -r**), 或是将目录与档案切换到某一个分支 (使用 **svn switch**). 当你完成之后, 你的工作复本就会变成来自不同版本的不同档案库位置的大杂烩. 但是在测试过后, 你确定这就是你需要的数据组合.

现在是作快照的时候了. 这里无法将一个 URL 复制到另一个去. 在这个情况中, 你想要建立一个你现在调整的工作复本的快照, 并将它存到档案库中. 很幸运的, **svn copy** 事实上有四种不同的使用方法 (你可以参考第 8 章), 包括将工作复本的档案树复制到档案库的功能:

```
$ ls
./      ../      my-working-copy/

$ svn copy my-working-copy
http://svn.example.com/repos/tags/calc/mytag

Committed revision 352.
```

现在在档案库中, 有了一个新的目录 `/tags/calc/mytag`, 它就是完全符合你的工作复本的快照 — 混合的修订版, url, 所有的东西.

有别的使用者找出这个功能的有趣用法. 有的情况下, 你的工作复本可能有一群本地更动, 而你想让你的协同工作者看看. 除了执行 **svn diff**, 将修补档送过去以外 (这没办法捕捉到档案树的变化), 你可以改用 **svn copy** 来 "上载" 你的工作复本至某一个档案库的私有区域. 你的协同工作者就可以取出一个一模一样的工作复本, 或是利用 **svn merge** 来接收你所作的更动.

分支维护

你大概注意到, Subversion 相当地有弹性. 因为它以相同的机制 (目录复本) 来实作分支与标记, 也因为分支与标记是出现在正常的档案系统空间中, 有许多人觉得 Subversion 真是太可怕了. 它可以说是 太有弹性了. 在本节中, 我们提供了几点建议, 作为你依时间来安排与管理数据的参考.

档案库配置

管理档案库有数种标准建议的方法. 大多数人会建立一个 `trunk` 的目录来放置 "主要" 的发展途径, 建立一个 `branches` 目录来放置分支复本, 以及建立一个 `tags` 目录来放置标记复本. 如果一个档案库只放置一个计划, 那么大多数的人会建立这样的最顶层目录:

```
/trunk
/branches
/tags
```

如果一个档案库包含了多个计划, 那么人们建立目录配置的方法, 会以分支为准:

```
/trunk/paint
/trunk/calc
/branches/paint
/branches/calc
/tags/paint
/tags/calc
```

...或是依计划:

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

当然啰, 你还是有忽略这些常用配置的自由. 你可以自己建立各种变化, 只要最能符合你与你的团队需要即可. 请记住, 不管你作什么样的选择, 都不表示就永远都不可变更. 不管何时, 你都可以重新整理你的档案库. 由于分支与标记只是普通的目录, 你可以利用 **svn move** 命令来移动或更名这些目录. 从一种配置换到另一种, 只是对伺服器端进行一连串的移动而已; 如果你不喜欢档案库现在的目录结构, 直接动它们吧.

资料生命周期

另一个 Subversion 模型的特色, 就是分支与标记可以是有限生命周期的, 就像其它受控管的项目一样. 举个例子, 假设你最后完成了个人 `calc` 计划分支里的工作. 在将你所有的更动送交回 `/trunk/calc` 后, 你的私有分支目录就没有必要再存在了:

```
$ svn delete http://svn.example.com/repos/branches/calc/my-calc-
branch \
    -m "Removing obsolete branch of calc project."
```

Committed revision 375.

现在, 你的分支就消失了. 它当然不是真的不见了: 这个目录只是从 **HEAD** 修订版消失, 不会再引起别人的注意. 如果你检视更早一点的修订版 (使用 **svn checkout -r**, **svn switch -r**, 或是 **svn list -r**), 你还是可以看到以前的分支.

如果只是浏览被删除的目录还不够, 你还是可以再让它回来. Subversion 要恢复旧资料相当地容易. 如果你想把被删除的目录 (或档案) 叫回 **HEAD** 修订版, 只要使用 **svn copy -r** 将它从以前的修订版复制回来就好了:

```
$ svn copy -r 374 http://svn.example.com/repos/branches/calc/my-calc-branch \
    http://svn.example.com/repos/branches/calc/my-calc-branch
```

Committed revision 376.

在我们的例子中, 你的个人分支有个满短的生命周期: 你可能只是把它建立起来, 用来修正臭虫, 或是实作一个新的功能. 在工作完成后, 这个分支也没有存在的必要了. 不过在软件开发的过程中呢, 长时间同时有两个 "主要" 的发展主线也是很常见的. 举个例子, 假设现在该对外发布一个稳定的 `calc` 的版本, 而你也知道大概要花好几个月才能把臭虫都清掉. 你不想让别人对这个计划加新功能, 但是 you 也不想叫所有的发展人员都停止工作, 所以你改为计划建立一个不会有太大变更的 "stable" 分支:

```
$ svn copy http://svn.example.com/repos/trunk/calc \
    http://svn.example.com/branches/calc/stable-1.0
    -m "Creating stable branch of calc project."
```

Committed revision 377.

现在设计人员可以继续自由地将最先进的 (或实验性的) 功能加到 `/trunk/calc` 中, 你也可以宣布 `/branches/calc/stable-1.0` 只接受臭虫修正的计划原则. 也就是说, 大家还是可以继续对主体工作, 但是选择性将臭虫修正移植到稳定分支去. 就算在稳定分支已经送出去了, 你可能还是得继续维护该分支很长一段时间 — 换句话说, 只要你继续为客户支持该发行版本, 就得继续维护.

摘要

我们在本章中, 涵盖了相当大的范围. 我们讨论了标记与分支的概念, 也示范了 Subversion 如何藉由透过 **svn copy** 命令复制目录, 实作这样的概念. 我们示范如何使用 **svn merge** 命令, 从一个分支复制更动到另一个分支, 或是回复错误的更动. 我们也展示了以 **svn switch** 来建立混合位置的工作复本. 我们还提到在档案库中, 如何管理分支的结构与其生命周期.

请注意 Subversion 的 mantra: 分支是廉价的. 所以请大方地使用它们吧!

^[7] Subversion 不支持跨档案库的复制. 当 **svn copy** 或 **svn move** 与 URL 一同使用时, 你只能复制同一个档案库里的项目.

^[8] 未来呢, Subversion 打算使用 (或发明) 可以描述档案树更动的扩充修补格式.

^[9] 不过 Subversion 计划还是有个计划, 打算在某天实作 **svnadmin obliterate** 命令, 可以达到永久删除信息的任务.

[10] 要是服务器的 URL 变更了, 而你又不想抛弃现有的工作复本, 你 可以将 **svn switch** 与 `--relocate` 一同使用. 请参阅 [Chapter 8. 完整 Subversion 参考手册](#) 的 **svn switch** 一节, 以了解更多的信息.

Chapter 5. Repository 管理

Table of Contents

[档案库的基本知识](#)

[了解异动与修订版](#)

[无版本控制的性质](#)

[档案库的建立与设定](#)

[Hook scripts](#)

[Berkeley DB 设定](#)

[档案库维护](#)

[管理员的工具箱](#)

[svnlook](#)

[svnadmin](#)

[svnshell.py](#)

[Berkeley DB 工具](#)

[档案库善后](#)

[档案库回复](#)

[汇入档案库](#)

[档案库备份](#)

[网络档案库](#)

[httpd, Apache HTTP 服务器](#)

[你需要什么, 才能设定基于 HTTP 的档案库存取](#)

[基本 Apache 设定](#)

[权限, 认证, 以及授权](#)

[服务器名称与 COPY 要求](#)

[浏览档案库的 HEAD 修订版](#)

[杂项的 Apache 功能](#)

[svnserve, 自订的 Subversion 服务器](#)

[设定匿名 TCP/IP 存取](#)

[设定使用 SSH 存取](#)

[使用哪一个服务器?](#)

[档案库权限](#)

[新增专案](#)

[选择一种档案库配置](#)

[建立配置, 汇入起始数据](#)

[摘要](#)

Subversion 的档案库是个中央仓储, 用来存放任意数量项目的受版本控管数据. 因为如此, 它成为管理员集中注意的焦点. 档案库一般并不需要太多的照顾, 但是

了解如何适当地设定它,照顾它是很重要的,如此才能避免一些潜在性的问题,而实际的问题得以安全地解决.

在本章中,我们会讨论如何建立与设定 **Subversion** 的档案库,以及如何开放档案库的网络存取. 我们也会提到档案库的维护,包括 **svnlook** 与 **svnadmin** 工具的使用方法 (它们都包含 **Subversion** 中). 我们也会说明常见的问题与错误,并提供几个如何安排档案库数据的建议.

如果你存取 **Subversion** 的档案库的目的,只是打算成为一个单纯将数据纳入版本控管的使用者的话 (也就是透过 **Subversion** 客户端),那你可以完全跳过本章.但是如果你是,或想要成为 **Subversion** 的档案库管理员,^[1] 本章绝对是你要投注注意力的地方.

当然了,一个人无法成为档案库的管理员,除非他有档案库可管理.

档案库的基本知识

了解异动与修订版

就概念来讲, **Subversion** 的档案库是一连串的目录树. 每一个目录树,就是档案库的目录与档案于不同时间点的快照. 这些快照是使用者进行作业的结果,称为修订版.

每一个修订版,是以异动树 (transaction tree) 开始其生命周期. 在送交发生时, **Subversion** 客户端会建立一个异动,其中反映了本地端的更动 (以及自客户端开始进行送交的任何额外更动),然后指示档案库将该树储存为下一个快照. 如果送交成功的话,这个异动就会实际成为新的修订版树,并被赋与新的修订版号. 如果送交因为某些原因失败的话,这个异动会被舍弃,客户端会被通知该动作失败.

更新的动作也类似这样. 客户端会建立一个暂时的异动树,反映工作复本的状态. 接着,档案库会将异动树与指定的修订版树作比较 (通常是最新的,或是“最年轻的”树),然后送回指示客户端该如何进行何种更动的信息,以将工作复本转换成该修订版树的样子. 在更新完成后,这个暂时的异动树就会被删除.

使用异动树,是对档案库的版本控制档案系统产生永久更动的唯一方法. 但是,了解异动的生命周期极富弹性是很重要的. 在更动的情况下,异动只是马上会被消灭的暂时档案树而已. 在送交的情况下,异动会变成固定的修订版 (如果失败的情况下,则是被移除). 如果有错误或是臭虫的话,异动很有可能不小心遗留在档案库之中 (虽然不会影响什么东西,但是会占用空间).

理论上,某一天整个流程能够发展出对异动生命周期能够有更细部的控制. 可以想象一下成一个系统,在客户端已经描述完它对档案库所产生的更动后,每个无法成为修订版的异动会先放到某个地方. 如此,每个新的送交就可以被某个人检阅,也许是主管,也许是工程品管小组,他们可以决定是要接受这个异动成为修订版,还是舍弃它.

这些跟档案库管理有什么关系呢? 答案很简单: 如果你要管理一个 **Subversion** 的档案库, 除了监控档案库的情况外, 你还必须检视修订版与异动情况.

无版本控制的性质

Subversion 档案库的异动与修订版也可以附加性质上去. 这些性质只是基本的键值对应, 可以用来储存与对应档案树有关的资料. 这些键值与你的档案树数据一样, 都是储存在档案库的档案系统之中.

对于储存某些档案树的数据, 但是这些数据并不完全与这些目录与档案相关时, 修订版与异动性质是很有用的 — 像是不被客户端工作复本管理的性质. 例如, 当一个新的送交异动于档案库中建立时, **Subversion** 会对这个异动新增一个名为 `svn:date` 的性质 — 一个用来表示异动何时建立的时间戳记. 当送交程序结束后, 而该异动被提升为一个固定的修订版时, 这个档案树会再多出储存修订版作者的使用者名称性质 (`svn:author`), 以及一个用来储存该修订版的纪录讯息性质 (`svn:log`).

修订版与异动性质都是 *无版本控制的性质* (*unversioned property*) — 因为它们被修改后, 原先的值就永远被舍弃了. 另外, 虽然修订版树本身是不会再变的, 附加在它们上的性质却不是. 你可以在日后对修订版性质作新增, 移除, 以及修改的动作. 如果你送交了一个新的修订版, 但是日后发现纪录讯息写错了, 或是有拼字错误的话, 你可以直接以正确的新讯息盖过 `svn:log` 的值.

档案库的建立与设定

建立一个 **Subversion** 的档案库出乎意料地简单. **Subversion** 所提供的 **svnadmin** 工具, 有个专门处理这件事的子命令. 要建立一个新的档案库, 只要执行:

```
$ svnadmin create path/to/repos
```

这会在目录 `path/to/repos` 里建立一个新的档案库. 这个新的档案库会以修订版 0 开始其生命周期, 里面除了最上层的根目录 (`/`), 什么都没有. 刚开始, 修订版 0 还有一个单一的修订版性质 `svn:date`, 会设定在档案库初建立的时间.

你可能注意到 **svnadmin** 的路径自变量只是一个普通的档案系统路径, 而不是像 **svn** 客户端程序用来表示档案库的 URL. **svnadmin** 与 **svnlook** 都被视为是服务器端的工具 — 它们是在档案库所在的机器上, 对档案库作检视或修改之用. **Subversion** 新手常犯的错误, 就是试着将 URL (即使是 "本地端" 的 `file:` 路径) 传给这两个程序.

所以, 在你执行 **svnadmin create** 命令之后, 这个目录中就会有全新的 **Subversion** 档案库. 让我们看一下在这个目录里产生了什么东西.

```
$ ls repos
```

dav/ db/ format hooks/ locks/ README.txt

除了 README.txt 与 format 檔以外, 档案库是由一群子目录组成. 就像 Subversion 其它部份的设计一样, 模块化是很重要的原则, 而且阶层式组织要比杂乱无章好. 以下是新的档案库目录中, 各个项目的简单叙述:

dav

提供给 Apache 与 mod_dav_svn 使用的目录, 让它们储存内部数据.

db

主要的 Berkeley DB 环境, 里面都是储存 Subversion 档案系统 (就是你置于版本控制的全部数据所在) 的数据库表格.

format

一个内容为一个整数的档案, 表示档案库配置的版本号码.

hooks

一个放置 hook 脚本文件模板的目录 (如果你有安装的话, 还有脚本档本身的档案).

locks

用来放置 Subversion 档案库锁定数据的目录, 用来追踪存取档案库的客户端.

README.txt

这个档案只是用来告知使用者, 他们在看的是 Subversion 的档案库.

一般来说, 你不需要自己“手动”处理档案库. **svnadmin** 工具就足以用来处理对档案库的任何改变, 不然也可以使用协力厂商的工具 (像是 Berkeley DB 工具组) 来调整部份的档案库. 不过还是有些例外, 我们会在这里提到.

Hook scripts

Hook scripts

挂勾(hook) 是某些档案库事件所触发的程序, 像是建立新的修订版时, 修改未纳入版本控制的性质时. 每一个挂勾都会得到足够的信息, 可以分辨出得到的是什么事件, 针对哪个 (哪些) 目录, 以及被谁触发. 依挂勾输出或传回状态的不同, 挂勾程序可以继续, 停止, 或是暂停该动作.

hook 子目录中, 预设是放置各个档案库挂勾的范本.

```
$ ls repos/hooks/  
post-commit.tmpl          pre-revprop-change.tmpl  
post-revprop-change.tmpl  start-commit.tmpl  
pre-commit.tmpl
```

每一个 Subversion 档案库实作出来的挂勾, 都各自有对应的范本, 只要检视这些范本的内容, 你就知道每一个命令稿是被什么触发的, 什么数据会传给命令稿. 每个模板还包含包含一些范例, 示范如何与其它 Subversion 随附工具一同完成某些常见工作. 要安装一个真正可用的挂勾程序, 只要把可执行档案或命令稿置于 repos/hooks 目录下, 并且让它能以挂勾的名称 (像是 **start-commit** 或 **post-commit**) 执行即可.

在 Unix 平台上, 这表示提供与挂勾完全同名的命令稿或是程序即可 (可以是 shell 命令稿, Python 程序, 编译过的 C 二进制程序, 或是任何的可能). 当然啰, 模板档案存在的目的, 并不只有提供信息而已 — 在 Unix 平台上安装挂勾最简单的方法, 就是把适当的模板拷贝成去掉 .tmpl 扩展名的新档案, 修改挂勾的内容, 然后确定命令稿是可执行的. 但是 Windows 是利用档案扩展名来决定它是不是可执行档, 所以你必须提供主档名为挂勾名称的程序, 再加上 Windows 认为是可执行档的扩展名, 像是视为程序的 .exe 或 .com, 以及批次档的 .bat.

目前 Subversion 档案库实作的挂勾有五个:

start-commit

这个挂勾在送交异动还没建立之前就会执行, 通常用来决定使用者是否有送交的权限. 档案库会传递两个自变量给这个程序: 档案库的路径, 以及想要进行送交的使用者名称. 如果程序传回一个非零的结束值, 在送交异动还没建立之前, 送交就会结束.

pre-commit

本挂勾执行的时间为异动完成之后, 送交之前. 一般来讲, 这个挂勾是用来阻止因内容或位置而不被允许的送交 (举个例子, 你的站台可能要求某个分支的送交都必须包含臭虫追踪的申请单号码, 或是进来的记录讯息不可为空白的). 档案库会传递两个自变量给这个程序: 档案库的路径, 以及准备送交的异动名称. 如果程序传回一个非零的结束值, 送交会被中止, 而异动会被删除.

Subversion 的发行档案包含了几个存取控制的命令稿 (位于 Subversion 源码树的 tools/hook-scripts 目录中), 可在 **pre-commit** 中使用, 以进行更细微的存取控制. 在这个时候, 除了 httpd.conf 所提供的以外, 这是管理员唯一可以进行细微的存取控制的方法. 在未来的 Subversion 中, 我们计划直接在档案系统实作 ACL。

post-commit

本挂勾执行的时间是在异动送交, 新修订版被建立之后. 大多数的人用这个挂勾来寄出关于本次送交的电子邮件, 或是建立档案库的备份. 档案库会传递两个自变量给这个程序: 档案库的路径, 以及新建立的修订版号. 本程序的结束码会被忽略.

Subversion 的发行档案包含了一个 **commit-email.pl** 命令稿 (位于 Subversion 源码树的 `tools/hook-script` 目录中), 可以用来寄送包含描述指定送交的电子邮件. 这个邮件包含了更动路径列表, 该送交所对应的记录讯息, 使用者, 送交日期, 以及一个以 GNU diff 样式表示的本次更动差异.

另一个 Subversion 随附的有用工具是 **hot-backup.py** 命令稿 (位于 Subversion 源码树的 `tools/backup/` 目录内). 这个命令稿可以进行 Subversion 档案库的实时备份 (这是 Berkeley DB 数据库后端支持的功能), 可以用来建立每一次送交的档案库快照, 作为归档纪录, 或是紧急回复之用.

`pre-revprop-change`

由于 Subversion 的修订版性质并未纳入版本控制, 修改这样的性质 (举个例子, `svn:log` 送交讯息性质) 将会永远地覆写原先的数值. 由于数据有可能会消失, Subversion 提供了这个挂勾 (以及相对应的 `post-revprop-change`), 以便档案库管理员能够依其意愿, 利用其它外部机制来记录曾作过的更动.

这个挂勾在档案库即将发生更动之前执行. 档案库会传递四个自变量给这个挂勾: 档案库的路径, 更动性质所在的修订版, 产生更动的认证使用者名称, 以及性质名称本身.

`post-revprop-change`

如同我们早先提过的, 这个挂勾是 `pre-revprop-change` 的对应挂勾. 事实上, 为了偏执狂着想, 如果 `pre-revprop-change` 不存在的话, 这个命令稿不会执行. 这两个挂勾都存在的话, `post-revprop-change` 挂勾只会在修订版性质更动之后才会执行, 一般是用来寄送包含更动性质的新数值的电子邮件. 档案库会传递四个自变量给这个挂勾: 档案库的路径, 该性质所在的修订版, 产生更动的认证使用者名称, 以及性质名称.

Subversion 的发行档案包括了一个 **propchange-email.pl** 命令稿 (位于 Subversion 源码树的 `tools/hook-scripts/` 目录中), 可用来寄送包含修订版性质更动细节的电子邮件 (或/且附加至记录文件中). 这个邮件会包含更动性质所在的修订版与名称, 产生更动的使用者, 以及新的性质数值.

Subversion 会试着以正在存取 Subversion 档案库的使用者身份来执行挂勾程序. 在大多数的情况下, 档案库是透过 Apache HTTP 服务器与 `mod_dav_svn` 存取的, 所以使用者会与 Apache 执行的使用者身份相同. 挂勾程序本身必须以操作系统

层级的权限进行设定,以便让使用者可以执行.另外,这也表示任何会被挂勾程序直接或间接存取的档案或程序(包括 **Subversion** 档案库本身),都会以同一个使用者的身份来进行存取.换句话说,请注意任何因档案权限而可能造成的潜在问题,以免挂勾无法进行你想进行的工作.

Berkeley DB 设定

Berkeley DB 环境有它自己预设的设定值,像是任何时间可使用的锁定数目,或是 Berkeley 日志记录文件的截止大小等等. **Subversion** 档案系统的程序会额外地为几个 Berkeley DB 设定选项选择其它的默认值.不过,你的档案库可能会有自己独特的数据集与存取型态,也许就需要不同的设置选项数值.

Sleepycat (Berkeley DB 的制造厂商)的人员了解不同的数据库有不同的需求,所以他们提供了一种执行时期的机制,可以更动许多 Berkeley DB 环境的设定选项数值. Berkeley 会检查每个环境目录是否存在着一个名为 `DB_CONFIG` 的档案,然后剖析其中的为某个特定 Berkeley 环境所用的选项.

档案库的 Berkeley 设定文件位于 `db` 环境目录中,也就是 `repos/db/DB_CONFIG`. **Subversion** 在建立档案库时,就会自行建立这个档案.这个档案一开始包含了几十个默认值,也包含了几十个 Berkeley DB 在线文件的参照,这样你可以自己去查这些选项所代表的意义.当然啰,你可以任意在 `DB_CONFIG` 档案里加入任何支持 Berkeley DB 选项.不要请记得,虽然 **Subversion** 不会试着去读取或解读这个档案的内容,或是使用任何里面的选项,你还是得避免某些设定选项,让 Berkeley DB 产生出 **Subversion** 不知如何处理的行为.另外,对 `DB_CONFIG` 的更动并不会马上生效,除非你回复了数据库环境(使用 **svnadmin recover**.)

档案库维护

管理员的工具箱

svnlook

svnlook 是 **Subversion** 提供的工具,用来检视档案库不同的修订版与异动.本程序完全不会试着去修改档案库 — 这是“只读”工具. **svnlook** 通常用在档案库挂勾程序中,用来回报档案库即将送交的更动(用在 **pre-commit** 挂勾时),或刚送交的更动(用在 **post-commit** 挂勾时).档案库管理员也许会将这个工具用于诊断之用.

svnlook 的语法很直接:

```
$ svnlook help
general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
Note: any subcommand which takes the '--revision' and '--transaction'
      options will, if invoked without one of those options, act on
      the repository's youngest revision.
Type "svnlook help <subcommand>" for help on a specific subcommand.
...
```

几乎每一个 **svnlook** 的子命令可以针对修订版或异动树运作, 显示档案树的信息, 或是它与档案库的前一个修订版之间有什么差异. 你可以使用 `--revision` 与 `--transaction` 选项来指定要检视的修订版与异动. 请注意, 虽然修订版号看起来像是自然数, 但是异动名称是包含英文字母与数字的字符串. 请记得档案系统只允许浏览未送交的异动 (未变成新修订版的异动). 大多数的档案库不会有这样的异动, 因为异动不是已送交了 (这让它们失去被检视的资格), 就是被中止然后移除.

如果完全没有 `--revision` 或 `--transaction` 选项的话, **svnlook** 会检视档案库中最年轻的 (或 “HEAD”) 修订版. 所以这两个命令作的事情都一样, 如果 19 是 `/path/to/repos` 档案库的最年轻修订版:

```
$ svnlook info /path/to/repos
$ svnlook info /path/to/repos --revision 19
```

这些子命令规则的唯一例外, 就是 **svnlook youngest** 子命令, 它不需要指定选项, 就只会印出 HEAD 修订版号.

```
$ svnlook youngest /path/to/repos
19
```

svnlook 的输出是设计为人类与机器都易读的. 以 `info` 子命令的输出作为例子:

```
$ svnlook info path/to/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
```

`info` 的输出命令定义如下:

1. 作者, 后接换行字符.
2. 日期, 后接换行字符.
3. 纪录讯息的字符数目, 后接换行字符.
4. 纪录讯息, 后接换行字符.

这个输出是人类可解读的, 像是日期戳记等具有意义的项目, 皆以文字形式表示, 而不是用看不懂的方式 (像是某个可爱外星人出现至今的毫微秒计数). 但是这个输出也是机器可读 — 因为纪录讯息可以有好几行, 没有长度的限制, 所以 **svnlook** 在讯息之前提供了讯息的长度. 这让命令稿与其它包装这个程序的命令稿, 可以对记录讯息作出聪明的决定, 要是这段数据不是串流的最后一个部份时, 至少也知道要略过几个字节.

另一个 **svnlook** 常见的用法, 就是检视某个修订版或异动树的内容. **svnlook tree** 会显示要求档案树的目录与档案系统 (还可以选择显示每一个路径的档案系统节

点修订版编号), 藉由检视其输出, 对于管理员决定是否可以移除某个看起来无效的异动是很有用的, 对 **Subversion** 发展人员在诊断发生档案系统相关的问题时也很有用.

```
$ svnlook tree path/to/repos --show-ids
/ <0.0.1>
A/ <2.0.1>
  B/ <4.0.1>
    lambda <5.0.1>
  E/ <6.0.1>
    alpha <7.0.1>
    beta <8.0.1>
  F/ <9.0.1>
mu <3.0.1>
C/ <a.0.1>
D/ <b.0.1>
  gamma <c.0.1>
G/ <d.0.1>
  pi <e.0.1>
  rho <f.0.1>
  tau <g.0.1>
H/ <h.0.1>
  chi <i.0.1>
  omega <k.0.1>
  psi <j.0.1>
iota <l.0.1>
```

svnlook 还可以进行其它的查询, 显示我们早先提到过的信息的一部份, 回报某个指定的修订版或异动中更动的路径, 显示档案与目录产生的文字与性质的差异, 诸如此类的信息. 以下是目前 **svnlook** 所支持的子命令, 简单的描述, 以及他们输出的东西:

author

显示档案树的作者.

cat

显示档案树里的档案内容.

changed

列出档案树中, 所有更动的档案与目录.

date

显示档案树的日期戳记.

diff

显示更动档案的统一差异格式.

dirs-changed

显示档案树里本身更动的目录,或是其下子档案有更动的目录.

history

显示某个纳入版本控制路径的历程纪录点 (更动或复制发生的地方).

info

显示档案树的作者,日期戳记,纪录讯息字符计数,以及纪录讯息.

log

显示档案树的纪录讯息.

propget

显示设定于档案树路径的性质内容.

proplist

显示设定于档案树路径的性质名称与内容.

tree

显示档案树列表,还可选择显示关联到每一个路径的档案系统节点修订版编号.

uuid

显示档案树的唯一使用者代号 (UUID).

youngest

显示最年轻的修订版号.

svnadmin

svnadmin 程序是档案库管理员最好的朋友.除了可以建立 Subversion 档案库,这个程序还可以让你对档案库进行几种维护动作.**svnadmin** 的语法与 **svnlook** 很类似:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
Type "svnadmin help <subcommand>" for help on a specific subcommand.
```

Available subcommands:

```
    create
    dump
    help (?, h)
```

...

我们已经提过 **svnadmin** 的 `create` 子命令 (见 [the section called “档案库的建立与设定”](#)). 在本章中, 我们会仔细讲解大多数其它的命令. 现在, 我们先简单地看看每个可用的子命令提供什么样的功能.

`create`

建立一个新的 Subversion 档案库.

`deltify`

在指定的修订版范围中, 对其中变动的路径作 **deltification**. 如果没有指定修订版的话, 则直接对 **HEAD** 修订进行.

`dump`

以可移植倾印格式, 倾印指定范围修订版的档案库内容.

`list-dblogs`

列出关联至档案库的 **Berkeley DB** 纪录档案的路径. 这个列表包含了所有的纪录文件 — Subversion 仍在使用的, 以及不再使用的.

`hotcopy`

产生档案库的实时备份. 你可以在任何时候执行这个命令以安全地产生档案库的复本, 毋须理会是否有其它的行程也同时在存取档案库.

`list-unused-dblogs`

列出所有关联到档案库的, 但是已经不再使用的 **Berkeley DB** 纪录文件. 你可以安全地自档案库的目录结构中移除这些纪录文件, 也可以将其备份下来, 日后要在遭遇灾难事件后, 可于回复档案库时使用.

`load`

从一个使用可移植倾印格式、由 `dump` 子命令产生的数据串流中, 加载一组修订版至档案库中.

`lstxns`

列出目前存在于档案库中, 尚未送交的 Subversion 异动.

`recover`

对一个有需要的档案库进行回复步骤, 可能因为曾发生重大的错误, 让某个行程无法正常地中止与档案库的沟通.

`rmtxns`

安全地从档案库中移除 Subversion 异动 (可以直接使用 `lstxns` 子命令的输出).

`setlog`

将档案库中指定修订版的 `svn:log` (送交纪录讯息) 性质的现值, 以指定的新值取代.

`verify`

验证档案库的内容. 这包括了比较存放于档案库里的版本控制数据的总和检查码.

svnshell.py

Subversion 源码树还包含了一个类似 shell, 与档案库沟通的界面. **svnshell.py** 这个 Python 命令稿 (位于源码树的 `tools/examples/` 目录中), 它使用 Subversion 的语言系统 (因此你必须正确地编译并安装它们, 以便让这个命令稿能正常执行), 以连接上档案库与档案系统链接库.

执行之后, 这个程序就像 `shell` 一样, 让你可以浏览档案库里的目录. 一开始, 你“位于”档案库的 **HEAD** 修订版的根目录, 并给你一个提示符号. 你可以在任何时候使用 `help` 命令, 它会显示可用命令的列表, 以及它们功用为何.

```
$ svnshell.py /path/to/repos
<rev: 2 />$ help
Available commands:
  cat FILE      : dump the contents of FILE
  cd DIR        : change the current working directory to DIR
  exit          : exit the shell
  ls [PATH]     : list the contents of the current directory
  lstxns        : list the transactions available for browsing
  setrev REV    : set the current revision to browse
  settxn TXN    : set the current transaction to browse
  youngest      : list the youngest browsable revision number
<rev: 2 />$
```

在档案库的目录结构之间移动, 就像在普通的 Unix 或 Windows shell 作法一样 — 请用 `cd` 命令. 在任何时候, 命令指示符号都会显示目前检视的修订版 (前置 `rev:`) 或异动 (前置 `txn:`), 以及该修订版或异动的路径位置. 你可以利用 `setrev` 或 `settxn`, 来变更目前你的修订版或异动. 就像在 Unix shell, 你可以使用 `ls` 命令来显示目前目录的内容, 也可以使用 `cat` 命令来显示档案的内容.

Example 5.1. 利用 `svnshell`, 在档案库之中巡行

```
<rev: 2 />$ ls
  REV    AUTHOR  NODE-REV-ID    SIZE    DATE NAME
-----
-----
```

```

1      sally <      2.0.1>          Nov 15 11:50 A/
2      harry <      1.0.2>          56 Nov 19 08:19 iota
<rev: 2 />$ cd A
<rev: 2 /A>$ ls
      REV  AUTHOR  NODE-REV-ID  SIZE  DATE NAME
-----
1      sally <      4.0.1>          Nov 15 11:50 B/
1      sally <      a.0.1>          Nov 15 11:50 C/
1      sally <      b.0.1>          Nov 15 11:50 D/
1      sally <      3.0.1>          23 Nov 15 11:50 mu
<rev: 2 /A>$ cd D/G
<rev: 2 /A/D/G>$ ls
      REV  AUTHOR  NODE-REV-ID  SIZE  DATE NAME
-----
1      sally <      e.0.1>          23 Nov 15 11:50 pi
1      sally <      f.0.1>          24 Nov 15 11:50 rho
1      sally <      g.0.1>          24 Nov 15 11:50 tau
<rev: 2 /A>$ cd ../../
<rev: 2 />$ cat iota
This is the file 'iota'.
Added this text in revision 2.

<rev: 2 />$ setrev 1; cat iota
This is the file 'iota'.

<rev: 1 />$ exit
$

```

如你在前一个范例中看到的, 在一个命令指示中可以用分号隔开, 指定数个命令. 还有, **shell** 了解绝对路径与相对路径的表示法, 它可以正确地处理 "." 与 ".." 的特殊路径部份.

youngest 命令会显示最年轻的修订版号, 它很适合用来决定你可用在 **setrev** 命令自变量的有效修订版范围 — 你可以浏览包含 0 到最年轻的修订版内容 (请回想一下, 修订版号是以整数表示的). 决定可浏览的有效异动就不是那么容易了, 请使用 **lstxns** 命令来列出你可浏览的异动. 可浏览的异动列表就跟 **svnadmin lstxns** 传回的是一样的, 它们也可以用在 **svnlook** 的 **--transaction** 选项中.

当你完成了 **shell** 的工作, 你可以使用 **exit** 命令, 以便正确地结束. 另外, 你也可以输入文件尾字符 — **Control-D** (不过某些 Win32 的 Python 发行版本改用 Windows 传统的 **Control-Z**.)

Berkeley DB 工具

目前 Subversion 档案库只有一种数据库后端 — **Berkeley DB**. 所有你的档案系统结构与数据都存在于档案库的 **db** 子目录的一组数据表格中. 这个子目录是一般的 **Berkeley DB** 环境目录, 因此可以跟任何 **Berkeley** 数据库工具一起使用 (你可以在 **SleepyCat** 的网站看看这些工具的文件, <http://www.sleepycat.com/>). 日常使用 Subversion 是不需要这些工具的, 但是他们提供的一些重要功能, 是目前 Subversion 本身所没有的.

举个例子, 因为 **Subversion** 使用 **Berkeley DB** 的日志功能, 数据库本身会先写下任何它打算进行的更动的描述, 然后再进行实际的变动. 这是为了确定在出错的情况下, 数据库系统还是可以回溯到前一个 *检查点 (checkpoint)* — 一个已知没有问题的纪录文件位置 — 然后再重新进行异动, 直到数据回复到一个可用的状态. 因为这个功能, 就是为什么选择 **Berkeley DB** 作为 **Subversion** 一开始的主要数据库后端的主要原因之一.

这些纪录档案会随着时间累积. 这实际上是数据库系统的一项特色 — 你应该可以只藉由这些纪录文件, 就可以重建整个数据库, 所以这些档案对于灾后的数据库重建是很重要的. 但是一般来说, 你会想要把 **Berkeley DB** 不再需要的纪录档案给归档, 然后把它们从磁盘删去以节省空间. **Berkeley DB** 提供了一个 **db_archive** 工具, 其中之一的功能就是列出关联到指定数据库, 但是不再被使用的纪录文件. 如此, 你就可以知道哪些数据可以归档, 然后将其移除. **svnadmin** 工具程序对这个 **Berkeley DB** 工具提供了一个方便的包装:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## 释放磁盘空间!
```

Subversion 的档案库使用 `post-commit` 挂勾命令稿, 它在执行档案库的“实时备份”之后, 就会移除这些多余的纪录档案. 在 **Subversion** 的源码树, `tools/backup/hot-backup.py` 命令稿示范了一个安全的方法, 如何在 **Berkeley DB** 数据库环境仍被存取的情况下, 还能进行备份: 以递归方式复制整个档案库目录, 然后重新复制 **db_archive -l** 列出的档案.

一般来说, 只有真正的偏执狂才真的需要在每次送交时进行实时备份. 但是我们假设任何一个档案库都有某种程度的冗余机制, 可到某一细微的程度 (例如每一次的送交), 档案库管理员仍有可能想要进行数据库的实时备份, 作为系统层级的每日备份. 就大多数的档案库来说, 归档的送交电子邮件本身就足以成为回复的来源, 至少可用于最后几个送交. 但是这是你的数据; 请以你希望的程度来保护它.

Berkeley DB 还包含了两个用来转换成 ASCII 文字文件, 以及自它转换成数据库的工具. **db_dump** 与 **db_load** 这两个程序, 各是用来写入与读取一个自订的档案格式, 可用来描述 **Berkeley DB** 数据库里的数据键与数据值. 由于 **Berkeley** 数据库在不同的机器平台并不具移植性, 这个格式在不同的机器之间传输数据库就很有用了, 完全不必烦恼机器架构与操作系统.

档案库善后

一般来讲, 在你的 **Subversion** 档案库依你需要作好设定之后, 就不需要太多的注意. 但是有的时候还是会需要管理员的介入. **svnadmin** 工具提供了几个有用的功能, 可帮助你进行以下的工作:

- 修改送交纪录讯息,
- 移除无效的异动,
- 修复“卡住的”档案库, 以及
- 将档案库内容移至不同的档案库.

也许 **svnadmin** 最常使用的子命令是 `setlog`. 当一个异动送交到档案库, 并且提升成为修订版之后, 关联到新修订版的描述纪录讯息 (由使用者提供) 会被储存为附加到修订版的未纳入版本控制的性质. 换句话说, 档案库只会记得该性质的最新值, 直接舍弃前一个值.

有的时候, 使用者的纪录讯息会出错 (也许是拼错字, 或是写错信息). 如果档案库设定为 (使用 `pre-revprop-change` 与 `post-revprop-change` 挂勾; 请参照 [the section called “Hook scripts”](#)) 送交之后还可以接受纪录讯息的更动, 那么使用者可以利用 **svn** 程序的 `propset` 命令 (请参照 [Chapter 8, 完整 Subversion 参考手册](#)), 从远程“修正”纪录讯息. 但是呢, 因为有永远失去信息的可能, **Subversion** 档案库的预设设定是不允许更动未纳入版本控制的性质 — 除非由管理员为之.

如果纪录讯息必须由管理员来更正, 这可由 **svnadmin setlog** 来达成. 这个命令会修改档案库中, 指定修订版的纪录讯息 (`svn:log` 性质), 并且由提供的档案中读取新值.

```
$ echo "Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

别搞丢了那个纪录讯息!

管理员应该要警觉到, 透过 **svnadmin setlog** 会跳过所有可能在档案库开启时执行的修订版性质挂勾命令稿. 使用这个子命令需要极度小心, 请确定你指定要变更的是正确的修订版号.

另一个 **svnadmin** 常用的用法, 就是查询档案库中未处理的 — 有可能是已经不再使用了的 — **Subversion** 异动. 在送交注定失败的时候, 异动一般都会被清除, 也就是说, 异动本身会自档案库移除, 任何与该异动有关 (也只有与其有关) 的资料会被移除. 但是有的时候, 发生的错误不会清除异动. 发生的原因有几种: 也许客户端的动作被使用者无预期中断, 或是网络在运作之中突然中断等等. 不管原因为何, 这些未处理的异动只会分散档案库, 只会占用资源而已.

你可以使用 **svnadmin** 的 `lstxns` 命令, 列出目前未处理异动的名称.

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

产生的输出中, 每一个项目都可以供 **svnlook** 使用 (以及它的 `--transaction` 选项), 看看是谁建立这个异动, 何时建立, 异动中有哪种类型的更动 — 换句话说, 就是这些异动是不是可以安全地被移除! 如果这样的话, 异动的名称可以传给 **svnadmin rmtxns**, 用来清除异动. 事实上, `rmtxns` 子命令可以直接以 `lstxns` 的输出作为输入.

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`  
$
```

如果你像这样使用这两个子命令的话, 你应该考虑暂时不让客户端存取档案库. 如此, 在你开始进行清除之前, 没有人可以进行有效的异动. 以下是简单的 shell 命令稿, 可以快速地产生产每一个档案库里的未处理异动的信息:

Example 5.2. txn-info.sh (回报未处理异动)

```
#!/bin/sh  
  
### 产生 Subversion 档案库中, 未处理异动的相关讯息.  
  
SVNADMIN=/usr/local/bin/svnadmin  
SVNLOOK=/usr/local/bin/svnlook  
  
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} --transaction ${TXN}  
done
```

你可以以 `/path/to/txn-info.sh /path/to/repos` 来使用前面的命令档. 这个输出基本上是 **svnlook info** 输出区块合并起来而已 (请参考 [the section called “svnlook”](#)), 看起来就像这样:

```
$ 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
```

一般来讲, 如果你看到一个没有关联纪录讯息的无效异动, 这是一个失败的更新 (或类似更动) 作业的结果. 这些动作是偷偷使用 **Subversion** 异动来模仿工作复本的状态. 由于它们并不打算送交, **Subversion** 并不要求这些异动要有纪录讯息. 有纪录讯息的异动几乎可以肯定是某种失败的送交. 还有, 异动的日期戳记可以提供一些有趣的信息 — 举个例子, 一个九个月前开始的作业, 仍为有效的可能性有多大?

简单地说, 是否要清除异动的决定不可轻率为之. 不同的数据来源 — 包括 **Apache** 的错误与存取纪录, 成功的 **Subversion** 送交纪录, 诸如此类的 — 都可以用来帮助你作决定. 最后, 管理员常常只要简单地跟疑似异动的拥有者沟通 (像是透过电子邮件), 就可以确定这个异动实际上是处于僵尸状态.

档案库回复

为了保护档案库中的数据, 数据库后端使用了锁定的机制. 这个机制保证不会同时有多个数据库存取者同时修改数据库的某一部份, 而数据从数据库读取出来时, 每一个行程都会看到数据是在正确的状态. 当一个行程需要修改数据库里的东西, 首先会检查目标数据是否有锁定. 如果数据没有被锁定的话, 行程就会锁定数据, 产生它想要的更动, 然后解除锁定. 另一个行程会被迫等待, 直到锁定解除之后, 它才可以继续存取该部份的数据库.

在使用 **Subversion** 档案库的过程中, 严重的错误 (像是磁盘空间, 或是可用的内存耗尽) 或中断会导致行程没有机会移除它在数据库产生的锁定, 结果就是后端数据库会“卡住”. 当这样的情况发生时, 任何存取档案库的尝试都会无限期停住 (因为每一个新的存取者都在等锁定解除 — 而这件事不会发生).

如果你的档案库发生这样的情况, 首先请勿慌张. [Subversion](#) 的档案系统充份地利用了数据库的异动, 检查点, 以及事先写入日志的优点, 以确保只有最严重的灾难事件 ^[12] 才会永久地毁坏数据库环境. 一个够偏执的档案库管理员会作出某种形式的 off-site 档案库备份, 但是还别忙着找你的系统管理员将磁带回存回来.

第二, 使用以下的步骤, 试着“解开”卡住的档案库:

1. 确定没有别的行程在存取 (或是试着去存取) 档案库. 对网络档案库来说, 这表示也要把 **Apache HTTP** 服务器给关掉.
2. 成为拥有与管理档案库的使用者身份.
3. 执行 **svnadmin recover /path/to/repos** 命令. 你会看到像这样的输出:
- 4.
5. Acquiring exclusive lock on repository db, and running recovery procedures.
6. Please stand by...
7. Recovery completed.
8. The latest repos revision is 19.
9. 重新启动 **Subversion** 服务器.

这个程序几乎能够解决所有档案库死锁的情况. 请确定你以拥有与管理数据库的身份来执行这个命令, 而不是只以 `root` 执行. 修复程序的某些部份会从新建立数个数据库档案 (举例来说, 共享的内存部份). 以 `root` 重建的话, 这些档案将为 `root` 所拥有, 这表示就算你重新开放档案库供人存取, 一般的使用者也无法存取它.

如果前述的步骤因为某些原因而无法解开你的档案库, 你应该作两件事. 首先, 将已损坏的数据库移到别的地方, 然后回存最新的备份. 然后, 寄一封电子邮件到 Subversion 发展人员邮件论坛 (在 [<dev@subversion.tigris.org>](mailto:dev@subversion.tigris.org)), 详细地描述你的问题. 对 Subversion 的发展人员来说, 资料完整性有着极高的优先权.

汇入档案库

Subversion 档案系统将其数据散布在数个数据库表格之中, 通常只有 Subversion 管理员了解 (也只有他们想了解). 但是有的时候, 我们需要将所有的数据, 或是部份的数据, 集合在单一可移植的平面档. Subversion 提供这样的机制, 由两个 **svnadmin** 子命令实作出来: `dump` 以及 `load`.

倾印与载入 Subversion 档案库的最常见原因, 就是 Subversion 本身的改变. 随着 Subversion 日渐成熟, 有时会因后端数据库 `schema` 的改变, 导致 Subversion 与前一版的档案库不兼容. 当你升级遇到这样的兼容性问题时, 我们建议以下列简单的步骤来进行:

1. 使用你 现有版本的 **svnadmin**, 将档案库倾印至倾印档案.
2. 升级至新版的 Subversion.
3. 将旧的档案库移开, 在原处以 新版本的 **svnadmin**, 建立新的空档案库.
4. 再利用 新版本的 **svnadmin**, 将你的倾印档载入至刚刚建立的档案库.
5. 最后, 请记得将原来档案库的自订部份复制到新的去, 包括 `DB_CONFIG` 与挂勾命令稿. 你应该要检查一下新版本 Subversion 的发行备注, 看看从你上次更新后, 有没有影响这些挂勾或设定选项的更动.

svnadmin dump 会以 Subversion 自订的档案系统倾印文件格式, 输出一个范围的档案库修订版. 倾印档格式会输出到标准输出串流, 而信息讯息会送至标准错误串流. 这让你可以将输出串流转向到一个档案, 但是还能在终端机看到状况输出. 举个例子:

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
...
* Dumped revision 25.
* Dumped revision 26.
```


作业结束之后,你会有一个档案(在前述的例子中,就是 `dumpfile`),其中包含所有储存于要求的档案库修订版范围的数据。

相对的另一个子命令是 **svnadmin load**,它会将标准输入串流以 Subversion 档案库倾印档案进行剖析,可以有效地将这些倾印的修订版回放到目标的档案库.它也会提供信息讯息,但是这次是送到标准输出串流:

```
$ 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) >>>
```

请注意,因为 **svnadmin** 使用标准输入与输出串流,作为倾印与加载作业,胆子很大的人可以这样试试(可能还在管道两旁使用不同版本的 **svnadmin**):

```
$ svnadmin create newrepos
$ svnadmin dump myrepos | svnadmin load newrepos
```

我们在前面说过,**svnadmin dump** 会输出一个范围的修订版.透过 `--revision` 选项,可以指定倾印单一修订版,或是一个范围的修订版.如果你省略这个选项,所有档案库里的现有修订版都会被倾印出来.

```
$ svnadmin dump myrepos --revision 23 > rev-23.dumpfile
$ svnadmin dump myrepos --revision 100:200 > revs-100-200.dumpfile
```

虽然 Subversion 会倾印所有的修订版,它只会输出够用的信息,让后来的加载程序可以依此来进行重建.换句话说,对任何一个倾印档里的修订版来说,只有在该修订版被更动的项目会出现在倾印档中.本规则的唯一例外,就是目前 **svnadmin dump** 命令所输出的第一个修订版.

Subversion 预设并不会将第一个倾印的修订版, 以可用于一个修订版的差异输出. 第一, 在倾印档中并没有前一个修订版! 第二个, **Subversion** 无法得知后来加载倾印档时 (如果真的有的话), 载入它的档案库的状态为何. 要确定每一次 **svnadmin dump** 都能自己自给自足的话, 预设第一个倾印出来的修订版会完整地表示出该档案库修订版的目录, 档案, 以及性质.

但是, 你可以变更这样的预设行为. 如果在倾印档案库时, 加上了 `--incremental` 选项, **svnadmin** 会将档案库的第一个倾印修订版, 与档案库中前一个修订版作比较, 就像其它被倾印的修订版的处理方法一样. 第一个修订版的输出, 就像倾印档里其它的修订版输出一样 — 只会显示该修订版的更动部份而已. 这样的好处, 是你可以建立几个可连续加载的小倾印档, 而不是一个很大的, 像这样:

```
$ svnadmin dump myrepos --revision 0:1000 > dumpfile1
$ svnadmin dump myrepos --revision 1001:2000 --incremental >
dumpfile2
$ svnadmin dump myrepos --revision 2001:3000 --incremental >
dumpfile3
```

这些倾印档可以透过下列一连串的命令, 加载到新的档案库中:

```
$ svnadmin load newrepos < dumpfile1
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

另一个可以透过 `--incremental` 选项的技巧, 就是你可以把新范围的倾印修订版附加至现有的倾印档. 举个例子, 你可能有一个 `post-commit` 挂勾, 它就会附加触动它的修订版的倾印内容. 或是你可以每晚执行像下面的命令稿, 将档案库中, 自上一次执行后新增的修订版附加至倾印档.

Example 5.3. 使用渐进式档案库倾印

```
#!/usr/bin/perl -w

use strict;

my $repos_path = '/path/to/repos';
my $dumpfile    = '/usr/backup/svn-dumpfile';
my $last_dumped = '/var/log/svn-last-dumped';

# Figure out the starting revision. Use 0 if we cannot read the
# last-dumped file, else use the revision in that file incremented
# by 1.
my $new_start = 0;
if (open LASTDUMPED, $last_dumped)
{
    my $line = <LASTDUMPED>;
    if (defined $line and $line =~ /\^(\\d+)/)
    {
        $new_start = $1 + 1;
    }
}
```

```

    close LASTDUMPED;
}

# Query the youngest revision in the repos.
my $youngest = `svnlook youngest $repos_path`;
defined $youngest && $youngest =~ /^\\d+$/
    or die "$0: 'svnlook youngest $repos_path' cannot get youngest
revision.\\n";
chomp $youngest;

# Do the backup.
system("svnadmin dump $repos_path --revision $new_start:$youngest --
incremental >> $dumpfile.tmp") == 0
    or die "$0: svnadmin dump to '$dumpfile.tmp' failed.\\n";

# Store a new last-dumped revision.
open LASTDUMPED, "> $last_dumped.tmp"
    or die "$0: cannot open '$last_dumped.tmp' for writing: $!\\n";
print LASTDUMPED "$youngest\\n";
close LASTDUMPED
    or die "$0: error in closing '$last_dumped.tmp' for writing: $!\\n";

# Rename to final locations.
rename("$dumpfile.tmp", $dumpfile)
    or die "$0: cannot rename '$dumpfile.tmp' to '$dumpfile': $!\\n";
rename("$last_dumped.tmp", $last_dumped)
    or die "$0: cannot rename '$last_dumped.tmp' to '$last_dumped':
$!\\n";

# All done!

```

以这样子使用的话, **svnadmin** 的 `dump` 与 `load` 命令就成了相当有用的工具, 可进行档案库更动的备份, 以避免系统当机, 或是其它灾难事件所带来的损害。

最后, 另一个 **Subversion** 档案库倾印档案格式的可能应用, 就是从不同的储存机制, 或是不同的版本控制系统进行转换。因为倾印档案格式绝大部份都是人类可懂的, ^[13]

档案库备份

从现代计算机诞生以来, 不过技术有多进步, 有件事情还是很不幸地一直不变 — 有的时候, 事情会有非常非常严重的问题。电力中断, 网络断线, 烂掉的内存, 坠毁的硬盘, 无论管理员是如何努力作好他们的工作, 还是有可能遇到这些暗黑命运的魔爪。所以我们遇到了最重要的课题 — 如何备份你的档案库的数据。

基本上, **Subversion** 档案库管理员可以使用两种备份 — 渐进式与完整式。我们在本章稍早的章节曾讨论过, 如果使用 **svnadmin dump --incremental** 来进行渐进式备份 (请参见 [the section called “汇入档案库”](#))。基本上, 它的概念就是只备份自上次制作备份后, 一定时间内的档案库更动。

就跟字面上的意思一样, 档案库的完整备份是整个档案库目录 (这包括了 **Berkeley** 数据库环境) 的复制。现在, 除非你暂时关闭所有其它对档案库的存取,

不然直接进行递归式的目录复制的话, 会有得到无效备份的风险存在, 这是因为别人仍有可能正在写入数据库.

很幸运地, **Sleepcat** 的 **Berkeley DB** 文件载明了以怎么样的步骤复制数据库档案, 就可以保证取得有效的备份复本. 更好的是你不必亲自实作, 因为 **Subversion** 发展团队已经作好了. 在 **Subversion** 源码的 `tools/backup/` 目录中, 可以找到 **hot-backup.py** 命令稿. 只要指定一个档案库的路径与备份的位置, **hot-backup.py** 就会进行必要的步骤, 进行档案库的实时备份 — 完全不需要中止所有的公共存取 — 然后会清除档案库中无用的 **Berkeley** 记录文件.

就算你已经有了渐近式备份, 你还是会想要定期执行这个程序. 举例来说, 你会想把 **hot-backup.py** 加进定时执行的程序里 (像是 **Unix** 系统的 **crond**). 或者你偏好可微调的备份方案的话, 可以让你的 `post-commit` 挂勾命令稿呼叫 **hot-backup.py** (请参见 [the section called “Hook scripts”](#)), 这样在每次有新的修订版出现时, 就会建立档案库的新备份. 只要将以下这一行加进你的档案库目录的 `hooks/post-commit` 命令稿即可:

```
(cd /path/to/hook/scripts; ./hot-backup.py ${REPOS} /path/to/backups
&)
```

产生出来的备份是完全可用的 **Subversion** 档案库, 在事情无法收拾时, 可以立即用来作为档案库的替代品.

这两种备份方式都有其长处. 目前最简单的是完整备份, 它一定能够产生正确无误的档案库复制. 这表示不管在线档案库发生多糟糕的事, 只要一个简单的递归目录复制指令, 就能从备份回复回来. 不过很不幸地, 如果你维护多个档案库的备份, 这些完整备份会占用跟你在线档案库一样多的磁盘空间.

在 **Subversion** 前后版本的数据库纲要变动的时候, 使用档案库倾印格式的渐进式备份在手边是很方便的. 由于将档案库升级到新版的数据库纲要需要进行完整的档案库倾印与加载, 如果已经有一半的步骤 (倾印的部份) 已经完成的话, 事情就方便多了. 不过很不幸地, 渐进式备份的建立 — 以及回复 — 要花较长的时间, 因为实际上每一个送交都是回放至倾印档或是档案库之中.

不管是哪一种备份方式, 档案库管理员必须知道未纳入版本控制的性质更动如何影响它们的备份. 由于这些更动不会产生新的修订版, 它们不会触发 `post-commit` 挂勾, 甚至也不会触发 `pre-revprop-change` 与 `post-revprop-change` 挂勾.^[14] 由于你能够不依时间顺序更动修订版性质 — 你可以在任何时间修改任何修订版的性质 — 最新几个修订版的渐进式备份也许不会知道原先备份的性质更动.

通常最好的修订版备份是包含多种方式的. 你可以妥善运用完整备份与渐进备份的组合, 再加上电子邮件的送交档案. 举个例子, **Subversion** 的发展人员在每一次新的修订版建立后, 就会备份 **Subversion** 的源码, 并且保留所有的送交与性质的电子邮件通知. 你的方案可能很类似, 但是应该配合你的需要, 在便利与偏执之间

取得平衡. 虽然这些都无法从暗黑命运的魔拳 [\[15\]](#) 中解救你的硬件, 它应该能在她肆机而动的时候解救你.

网络档案库

一个 Subversion 档案库可能同时被其所在机器上的多个客户端同时存取, 不过常见的 Subversion 组态, 牵涉到一个可被其它办公室客户端存取的服务器 — 当然也有可能是被全世界存取.

本节描述如何让你的 Subversion 档案库开放给远程用户使用. 我们会涵盖所有可用的服务器机制, 讨论每一个的设定与用法. 读完本章后, 你应该能够描述哪一种网络设定适合你的需求, 并且了解如何在你的计算机上启用这样的设定.

httpd, Apache HTTP 服务器

Subversion 的主要网络服务器为 Apache HTTP 服务器 (**httpd**), 可以 WebDAV/deltaV 通讯协议沟通. 这个通讯协议 (它是 HTTP 1.1 的扩充; 请参照 <http://www.webdav.org/>) 采用广为使用的 HTTP 通讯协议, 这是全球信息网的核心, 再加上写入 — 更具体地说, 有版本控制的写入 — 的能力. 其结果就是一个标准的, 坚固的系统, 包装成一个 Apache 2.0 软件的一部份, 常用操作系统与协力厂商都支持, 又不需要网络管理员开启另一个自订的连接埠. [\[16\]](#)

以下的讨论中, 提到许多 Apache 设定的指令. 虽然有些范例中提供了指令的用法, 但是完整的描述并不在本书的范围中. Apache 团队维护一份很不错的文件, 可在他们的网站 <http://httpd.apache.org> 取得. 例如, 一般性的设定指令位于 <http://httpd.apache.org/docs-2.0/mod/directives.html>.

另外, 在你修改 Apache 的设定时, 很有可能会在过程中引入其它的错误. 就算你不熟悉 Apache 的纪录子系统, 你也应该要知道有它的存在. 在你的 `httpd.conf` 档中, 有着指定 Apache 产生的存取纪录文件与错误纪录文件应放置的位置 (各为 `CustomLog` 与 `ErrorLog` 指令). Subversion 的 `mod_dav_svn` 亦使用 Apache 的错误纪录界面. 你都可以浏览这些档案的内容, 它们可能会显示出其它方法无法发生的错误来源.

你需要什么, 才能设定基于 HTTP 的档案库存取

要让你的档案库透过 HTTP 供他人存取, 基本上你需要四个组件, 可从两个套件中取得. 你需要 Apache 的 **httpd 2.0**, 它也包含了 **mod_dav** 的 DAV 模块. Subversion, 以及包含的 **mod_dav_svn** 档案系统供应模块. 当你有了这些组件后, 将档案库放到网络上的步骤就只是简单的:

- 让 httpd 2.0 跑起来, 并与 `mod_dav` 模块一并执行,
- 将 `mod_dav_svn` 安插模块安装至 `mod_dav`, 它会透过 Subversion 链接库来存取档案库, 以及,
- 设定你的 `httpd.conf` 档案, 汇出 (让他人可存取) 档案库.

要完成前两项, 你可以自源码编译 **httpd** 与 **Subversion**, 或是在系统上安装事先编译好的二进制套件. 欲得知如何将 **Subversion** 编译成与 **Apache HTTPD** 服务器一同使用, 以及如何编译与设定 **Apache** 以达成这样效果的最新信息, 请看看 **Subversion** 源码树最上层的 **INSTALL** 档案.

基本 Apache 设定

当你把所有需要的组件都安装到系统上之后, 剩下的只就是透过 `httpd.conf` 档案设定 **Apache**. 请使用 `LoadModule` 指令, 让 **Apache** 加载 `mod_dav_svn` 模块, 这个指令必须出现在其它的 **Subversion** 相关指令之前. 如果你的 **Apache** 是以预设的目录配置安装的, 你的 `mod_dav_svn` 模块应该会安装在 **Apache** 安装位置 (通常是 `/usr/local/apache2`) 的 `modules` 子目录内. `LoadModule` 指令的语法很简单, 就是将一个具名模块对映到共享链接库在磁盘上的位置:

```
LoadModule dav_svn_module      modules/mod_dav_svn.so
```

请注意, 如果 `mod_dav` 是编译成共享 object (而不是直接以静态连结到 **httpd** 可执行档), 你也需要对它使用一个类似的 `LoadModule` 叙述.

稍后于设定档中, 你需要告诉 **Apache**, **Subversion** 档案库 (或多个档案库) 的位置. `Location` 指令有类似 **XML** 的表示法, 以一个起始标签开始, 以完结标签结束, 中间包含了其它不同的指令. `Location` 指令的目的, 是告诉 **Apache** 在处理指向指定 **URL** 或其子项目之一的要求时, 对它进行特别的处理. 在 **Subversion** 的情况中, 你要让 **Apache** 把指向具版本控制资源的要求, 直接交给 **DAV** 去处理. 你可以利用以下的 `httpd.conf` 语法, 指示 **Apache** 将所有对路径部份 (就是 **URL** 中, 在服务器名称与可能出现的连接埠之后的部份) 以 `/repos/` 开始的 **URL** 的处理, 通通由 **DAV** 提供者来处理, 其档案库位于 `/absolute/path/to/repository`:

```
<Location /repos>
    DAV svn
    SVNPath /absolute/path/to/repository
</Location>
```

如果你计划支持多个 **Subversion** 档案库, 而它们都有着共同的本地磁盘路径, 你可以使用另一种指令 `SVNParentPath`, 指示它们共同的父路径. 举个例子, 如果你知道你会在路径 `/usr/local/svn` 之下建立多个 **Subversion** 档案库, 并以类似 `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2` 等等的 **URL** 供人存取, 你可以使用下列例子中的 `httpd.conf` 设定语法:

```
<Location /svn>
    DAV svn
    SVNParentPath /usr/local/svn
</Location>
```

使用前述的语法, Apache 会将所有路径以 `/svn/` 开始的 URL 都交给 Subversion DAV 供应模块处理, 它会假设任何以 `SVNParentPath` 指令指定的目录都是 Subversion 档案库. 不像 `SVNPath`, 这个相当便利的语法可以让你在建立新的档案库时, 仍旧不必重跑 Apache.

权限, 认证, 以及授权

在这个阶段, 你应该要强烈地考虑权限的问题. 如果你已经让 Apache 以普通的网页服务器执行了一阵子, 你应该已经有一堆的内容 — 网页, 命令稿, 诸如此类的. 这些项目都已经以一组权限执行, 让他们能够与 Apache 一起使用, 更適切地讲, 让 Apache 与这些档案一起工作. 当 Apache 以 Subversion 服务器运作时, 它也需要正确的权限, 以便能够读取与写入你的 Subversion 档案库.

你需要决定一套权限系统设定, 能够满足 Subversion 的要求, 又不会弄乱先前安装的网页或命令稿. 这表示你要变更 Subversion 档案库的权限, 以符合 Apache 其它一起协同工作的部份, 或是利用 `httpd.conf` 的 `User` 与 `Group` 指令, 让 Apache 以拥有 Subversion 档案库的使用者与群组的身分来执行. 设定权限并没有一个绝对正确的方法, 而且每一个管理员有自己行事的标准. 你只要注意, 要让 Subversion 档案库与 Apache 一起使用时, 权限会是最常发生的问题.

既然我们在谈论权限的问题, 我们也应该谈谈 Apache 提供的认证与授权的机制可以如何运用. 除非你对这些有某种系统层面的设定, 基本上, 你透过 Location 开放的 Subversion 档案库是所有人都可以存取的. 换句话说,

- 任何人都可以用它们的 Subversion 客户端, 取出档案库 URL (或其任意的子目录) 的工作复本.
- 任何人只要将他们的浏览器指向档案库 URL, 就可以交互式地浏览档案库最新的修订版, 以及,
- 任何人可以送交至档案库.

如果你要限制整个档案库的读取或写入存取, 你可以使用 Apache 内建的存取控制功能. 在这些功能中, 最简单的是基本设证机制, 它只会使用使用者名称与密码, 用以确认使用者是他所声称的身份. Apache 提供了 **htpasswd** 工具程序, 来管理接受的使用者名称与密码, 也就是你想要授与存取 Subversion 档案库权限的使用者. 让我们授与 Sally 与 Harry 送交存取的权限. 首先, 我们必须把它们加入到密码档案.

```
$ ### 第一次: 以 -c 建立档案
$ htpasswd -c /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd /etc/svn-auth-file sally
New password: *****
Re-type new password: *****
Adding password for user sally
$
```

接着,你需要在 `httpd.conf` 的 `Location` 区块中新增几个指令,告诉 **Apache** 如何处理你的新密码文件. `AuthType` 指令指定应使用何种认证系统. 在目前状况中,我们想要指定 `Basic` 认证系统. `AuthName` 是一个任意的名称,让你用来指定认证领域 (**authentication domain**). 大多数的浏览器在向使用者询问使用者代号与密码时,会将这个名称显示在弹出的对话框中. 最后,使用 `AuthUserFile` 指令,指定你以 **htpasswd** 产生的密码文件.

在新增这三个指令后,你的 `<Location>` 区块看起来应该像这样:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

现在如果你重新启动 **Apache**,任何需要认证的 **Subversion** 动作都会从 **Subversion** 客户端取得使用者代号与密码,这可能是使用先前置于快取的值,或是向使用者询问. 剩下要作的,就是告诉 **Apache** 哪些动作需要这样的认证.

你可以藉由将 `Require valid-user` 指令加进 `<Location>` 区块,对所有存取档案库的动作进行限制. 以先前的例子而言,这表示只有声称他们自己是 `harry` 或 `sally`,并且能够提供这些代号的正确密码的客户端,才可以对 **Subversion** 档案库作任何事.

有的时候,你不需要这么地严格. 举例来说, **Subversion** 源码所在的 `http://svn.collab.net/repos/svn` 档案库就允许世界上的任何人对它进行只读的存取 (像是取得工作复本,或是透过浏览器来浏览档案库),但是所有写入的动作仅限由认证用户为之. 要达到这样的选择性限制,你可以使用 `Limit` 与 `LimitExcept` 设定指令. 就像 `Location` 指令,这些区块都有起始与完结标签,而且你应该把它们放置在你的 `<Location>` 区块.

在 `Limit` 与 `LimitExcept` 出现的参数,是该区块中受到影响的 **HTTP** 要求类别. 举个例子,如果除了目前支持的只读动作,其它的存取通通不允许,你可以使用 `LimitExcept` 指令,并且使用 `GET`, `PROPFIND`, `OPTIONS`, 以及 `REPORT` 的要求类别参数. 然后前述的 `Require valid-user` 指令就应该摆在 `<LimitExcept>` 区块中,而不只是 `<Location>` 中而已.

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
  <LimitExcept GET PROPFIND OPTIONS REPORT>
```



```
    Require valid-user
</LimitExcept>
</Location>
```

这只是几个简单的例子而已. 欲得知更详细的 Apache 存取控制的信息, 请参考 <http://httpd.apache.org/docs-2.0/misc/tutorials.html> 的 Apache 的导览文件中的 安全 一节.

服务器名称与 COPY 要求

Subversion 使用 COPY 要求类别, 进行服务器端的目录与档案的复制. 由于 Apache 模块要进行的完整性检查, 复制来源必须与目标处于同一台机器上. 要达到这个要求, 你需要告诉 mod_dav 用以作为服务器主机的名称. 一般来说, 你可以在 httpd.conf 中使用 ServerName 指令来达到这样的目的.

```
ServerName svn.red-bean.com
```

如果你透过 NameVirtualHost 使用 Apache 的虚拟主机功能, 那么你必须使用 ServerAlias 指令来指定服务器的其它名称. 当然啰, 请参考 Apache 的文件, 以了解详细的细节.

浏览档案库的 HEAD 修订版

对你的 Subversion 档案库进行 Apache/WebDAV 设定, 最有用的功效之一, 就是可以马上使用普通的浏览器来存取最年轻修订版、纳入版本控制的目录与档案. 由于 Subversion 使用 URL 来指定受版本控制的资源, 这些用来存取基于 HTTP 的档案库存取的 URL, 可直接输入到网页浏览器之内. 你的浏览器会为这个 URL 送出一个 GET 要求, 依该 URL 所代表的是受版本控制的目录还是档案, mod_dav_svn 会以目录列表或档案内容进行响应.

由于 URL 中没有包含你想知道的资源的版本信息, mod_dav_svn 都会以最年轻的版本响应. 这个功能有个很棒的副作用, 就是你可以将 Subversion 的 URL 当作文件参照, 丢给同事使用, 然后这些 URL 永远都指向该文件的最新版本. 当然啰, 你还可以将这些 URL 当作其它网页的超级链接之用.

你通常会找到指向受版本控制档案的 URL 的其它用法 — 毕竟, 通常那也是人家有兴趣的内容所在. 但是你可能偶而也看过 Subversion 的目录列表, 很快就会发现产生出来的 HTML 功能很阳春, 一点也不漂亮 (甚至一点也不有趣). 如果想要自订目录列表, Subversion 提供了 XML 索引功能. 在 httpd.conf 中的档案库的 Location 区块, 一个 SVNIndexXSLT 指令就会告诉 mod_dav_svn, 在显示目录列表时, 产生 XML 的输出, 并且使用你所指定的 XSLT 式样表:

```
<Location /svn>
    DAV svn
    SVNParentPath /usr/local/svn
```

```
SVNIndexXSLT "/svnindex.xsl"  
...  
</Location>
```

透过 `SVNIndexXSLT` 指令与一个有创意的 **XSLT** 式样表, 你可以让你的目录列表符合网站其它部份的色彩运用与影像. 或者你喜欢的话, 也可以使用 **Subversion** 源码中, 置于 `tools/xslt/` 目录里的式样表范例. 请记住 `SVNIndexXSLT` 指令所提供的路径, 实际上是 **URL** 路径 — 浏览器必须要能够读取你的式样表, 才能使用它们!

杂项的 Apache 功能

作为一个坚固的网页服务器, **Apache** 本身已有的几项功能, 也可以用在提供 **Subversion** 的功能性与安全性. **Subversion** 透过 **Neon** 与 **Apache** 进行沟通, 这是一个通用型的 **HTTP/WebDAV** 链接库, 支持了几个像是 **SSL** (**secure socket layer**) 与 **Deflate** 压缩 (和 **gzip** 与 **PKZIP** 用来将档案“缩小 (shrink)”成较小的区块的算法相同) 的机制. 你要作的, 就只是把你想要的功能与 **Subversion** 和 **Apache** 编译在一起, 然后正确地设定程序, 让它们使用这些功能.

这表示启用 **SSL** 的 **Subversion** 客户端可以存取开启 **SSL** 的 **Apache** 服务器, 透过加密的通讯协议进行所有的沟通, 仅仅需要把 `http://` 以 `https://` 的 **URL** 取代即可. 有些需要把它们档案库对防火墙外开放的公司, 必须要考虑到恶意第三方可能会“窃听”网络封包. **SSL** 让这样不受欢迎的行为不容易造成敏感数据外泄. **Apache** 可以设定成只让开启 **SSL** 的 **Subversion** 客户端与档案库进行沟通.

Deflate 压缩会在客户端与服务器加诸小小的负担, 对网络传递进行压缩与解压缩, 以将实际的数据传输量降至最低. 在网络频宽相当不足的情况下, 这样的压缩功能能够大幅地加速服务器与客户端之间的传输速度. 在很极端的情况下, 像这样被减少的网络传输, 就可以决定动作会逾时, 还是成功地完成.

有些比较不引人注意, 但是同样有用的, 是 **Apache** 与 **Subversion** 之间的功能, 像是指定自订连接埠 (而不使用预设 **HTTP** 的 80 连接埠), 可使用虚拟网域名称对 **Subversion** 档案库进行存取的能力, 或是透过代理服务器存取档案库的功能. 这些都被 **Neon** 支持, 所以 **Subversion** 不费吹灰之力就得到这些功能.

svnserve, 自订的 Subversion 服务器

除了 **Apache** 之外, **Subversion** 还提供另一个单独的服务器程序, 也就是 **svnserve**. 这个程序要比 **Apache** 更轻便, 而且更容易设定. 它会与 **Subversion** 客户端透过 **TCP/IP** 联机以自订的通讯协议沟通.

svnserve 有两种基本的使用方法:

未授权 (匿名) 存取

在这个情况下, 一个 **svnserve** 的背景监控行程会在服务器上执行, 聆听外部的联机. **svn** 客户端会使用自订的 **svn:// URL schema** 联机. 客户端联机会无条件地接受, 而档案库会以未授权的使用者名称进行存取. 大多数的情况下, 管理员会将这个背景监控程序设定为允取只读动作.

授权 (SSH) 存取

在这个情况下, **svn** 客户端使用的是自订的 **svn+ssh:// URL schema**; 这会启动一个本地端的 **Secure Shell (SSH)** 行程, 它会联机至服务器, 然后进行授权. 使用者必须在服务器有系统账号, 才能这样作. 在授权完成后, **SSH** 行程会在服务器上执行一个暂时的私有 **svnserve** 行程, 以授权使用者的身份执行. 服务器与客户端会透过加密的 **ssh** 通道进行沟通.

请注意, 这些 **svnserve** 的使用方法并不互相冲突; 你可以很简单地同时在你的服务器上, 使用这两种技巧.

设定匿名 TCP/IP 存取

当 **svnserve** 不带自变量执行时, 它会将数据写至标准输出, 并自标准输入读取数据, 试着与一个 **svn** 客户端商议出一个进程:

```
$ svnserve
( success ( 1 1 ( ANONYMOUS ) ( ) ) )
```

这对任何人没有立即的效果; 因为 **svnserve** 的运作如此, 它才能被 **inetd** 背景监控程序执行. 但是要将 **svnserve** 以背景监控程序执行的话, 方法还有很多种.

有一个方法就是对服务器主机的 **inetd** 背景监控程序登记一个 **svn** 服务. 那么当一个客户端试着要连接到连接埠 3690 时, ^[17] **inetd** 就会起动一个“只用一次”的 **svnserve** 行程来处理该用户的要求.

当你以这种方式设定的话, 请记住你不该以 **root** 使用者 (或其它有无限权限的使用者) 的身份来执行 **svnserve** 行程. 依你汇出档案库的所有权与权限的不同, 不同的 — 也许是自订的 — 的使用者可能更合适. 举个例子, 你可能会想要建立一个名为 **svn** 的新使用者, 给这个使用者可使用 **Subversion** 档案库的专有权限, 然后设定你的 **svnserve** 行程以该使用者的身份执行.

当然啰, 第一个方法只适用于 **inetd** (或类似 **inetd** 的) 背景监控程序的机器. 通常这只限于 **Unix** 平台上的. 另一个方法是 **svnserve** 以单一个背景监控程序执行. 以 **-d** 选项执行的话, **svnserve** 会马上自目前的 **shell** 行程分离, 以背景行程一直地执行下去, 当然还是在连接埠 3690 上等待新进的联机.

```
$ svnserve -d
$ # svnserve 仍在执行, 但是使用者已经回到提示字符
```

当一个客户端透过网络与 **svnserve** 行程 (以背景监控程序执行, 或是“只用一次”的处理行程) 联机时, 完全没有认证发生. 服务器行程会以它执行的使用者身份来处理档案库, 如果客户端进行送交的话, 新的修订版完全不会设定 `svn:author` 性质.

只要 **svnserve** 服务器开始执行, 它会让整个档案库都开放给网络使用. 换句话说, 如果一个客户端试着要对在 `example.com` 上执行的 **svnserve** 行程取出

`svn://example.com/usr/local/repos/project`, 它会试着去找位于绝对路径 `/usr/local/repos/project` 上的档案库. 如果你想增加安全性, 你应该以 `-r` 选项执行 **svnserve**, 它会限制只开于该路径下的档案库:

```
$ svnserve -d -r /usr/local
...
```

透过 `-r` 选项, 可以有效地变更程序认为是远程档案系统空间的根目录. 客户端就可以把相同的路径拿掉, 就会得到比较短的 (也比较不明显的) URL:

```
$ svn checkout svn://example.com/repos/project
...
```

要取消档案库的写入存取, 请以 `-R` 选项起动 **svnserve**. 这样就只允许读取档案库里的数据.

设定使用 SSH 存取

通常来说, 我们都想知道 (在几近无限的使用者中) 哪一个使用者该为哪些更动负责, 但是限制哪些使用者有档案库的写入能力要来得更重要.^[18] 要达到这两个目的, 使用 `libsvn_ra_svn` 的客户端可以透过 SSH 信道来使用网络进程.

在这种情况下, 每一次客户端想要连上 Subversion 档案库, 本地端的 SSH 层就会在服务器机器上启动一个新的 **svnserve**. **svnserve** 背景监控程序并没有必要执行起来 — 已认证过的 SSH 用户联机会在服务器上启动一个私有的 **svnserve** 行程, 会以认证的使用者身份执行. (事实上, CVS 在使用 `:ext:` 存取方式与 SSH 时, 也是这么作的.) SSH 本身需要认证过的使用者, 所以不会有匿名的身份. 由于存取不是匿名的, 已认证的使用者代号会被用来当作档案库更动的作者储存起来.

一个客户端可以藉由使用 `svn+ssh:// schema`, 以及指定档案库的绝对路径, 就可以进行 SSH 通道, 像这样:

```
$ svn checkout svn+ssh://example.com/usr/local/repos/project
Password:
...
```

`svn` 客户端预设会在使用者的 `$PATH` 中, 寻找并启动一个名为 **ssh** 的本地端程序, 不过 SSH 可由以下两种方法之一置换掉. 你可以将 `SVN_SSH` 环境变量设为新的

值,或是可以设定客户端执行时期设定档 `config` 的 `[tunnels]` 一节中的 `ssh` 变数.

举个例子, **ssh** 行程在试着与服务器进行认证时,会使用你目前的使用者名称. 你可能希望 **ssh** 使用不同的使用者名称. 要达成这个目的,你可以这样执行本命令

```
$ export SVN_SSH="ssh -l username"
```

... 或者你也可以修改执行时期设定档 `config`, 让它包含

```
[tunnels]
ssh = ssh -l username
```

欲知更多有关修改 `config` 执行时期设定文件的信息, 请参考 [the section called “Config”](#).

使用哪一个服务器?

当你在 Apache HTTP 服务器与自订的 **svnserve** 程序之间作抉择时, 其实并没有一个标准的答案. 依你自己的要求, 某一个可用的方案可能看起来会比较适合. 事实上, 这些服务器都可以一起使用, 每个都以它们自己的方式存取档案库, 不会互相干扰. 作为前两章的总结, 以下是两个 Subversion 可用的服务器的简单比较表 — 请选择最适合你与使用者之用的:

Apache/mod_dav_svn

- 认证: http basic/digest auth, certificates, LDAP 等等. Apache 有为数众多的认证模块. 不需要为使用者建立真实的系统账号.
- 认证: 读写权限可以依档案库 (使用 `httpd.conf` 指令), 或是依目录 (使用 `mod_authz_svn`) 为基础进行设定.
- 毋需开启新的防火墙连接埠.
- 内建的档案库网页浏览功能 (有限度)
- 与其它 WebDAV 客户端的有限配合
- 使用快取 HTTP 代理服务器的能力
- 通过时间验证的强大扩充能力 — Apache 是无数大型企业网站的网页服务器首选. *It takes a lickin' and keeps traffickin'*
- 强大的扩充能力: 可使用许多现有的认证与授权方式, 以及加密, 压缩等等的功能.

svnserve

- 认证: 只能透过 SSH 通道. 使用者必须要有系统账号.
- Authorization: via shared ownership/permissions on repository DB files.

授权: 透过档案库 DB 档案的共享拥有者/档案权限.

- 比 Apache 要更轻巧地多, 而且大多数的动作也更快.
- 比 Apache 要更容易设定.
- 可以使用现有的 SSH 安全架构.

档案库权限

你已经看到档案库如何能以许多不同的方式存取. 但是有可能 — 或是能够安全地 — 让档案库同时被多种不同的档案库存取方法来存取吗? 答案是肯定的, 前提是你得有些先见之明.

在任何时间, 这些行程也许都需要对档案库进行读取与/或写入存取:

- 一般的系统使用者, 会使用 Subversion 客户端 (以他们自己的身份) 直接存取档案库;
- 一般的系统使用者, 连接到 SSH 执行的私有 **svnserve** 行程 (以他们自己的身份执行), 经由它来存取档案库;
- 一个 **svnserve** 行程 — 无论是背景监控程序, 还是被 **inetd** 执行的 — 会以某一个固定的使用者身份执行;
- 一个 Apache 的 **httpd** 行程, 会以某一个固定的使用者身份执行.

绝大部份管理员会遇到的共同问题, 是档案库的拥有权与权限. 每个前项列表中的行程 (或使用者) 都有权限可以读取与写入 Berkeley DB 档案吗? 假设你有一个类似 Unix 的操作系统, 最直接的方法, 大概就是把每一个可能的档案库使用者加入到新的 **svn** 群组, 并且确定该群组拥有档案库. 但是即使如此也还不够, 因为一个行程可能以不友善的 **umask** 写入数据库档案 — 一个会让其它使用者无法存取的 **umask**.

在为档案库使用者设定一个共同群组之后, 接下来是为每一个存取档案库的行程设定一个合理的 **umask**. 对于直接存取档案库的使用者, 你可以将 **svn** 程序放在包装程序中, 先设定 **umask 002**, 然后再执行真正的 **svn** 用户程序. 你可以为 **svnserve** 程序写一个类似的包装器, 并且将 **umask 002** 加在 Apache 自己的启动命令稿 **apachectl** 中.

在你搞定这些东西之后, 你的档案库就应该可以被所有有需要的行程存取. 这看起来可能有点乱, 有点复杂, 不过让多个使用者对共同档案都可进行写入存取, 一直都是经典的问题, 而且通常都没有很优雅解决方法.

很幸运的, 大多数的档案库管理员根本不需要有这么复杂的设定. 使用者想要存取同一台机器上的档案库时, 其实并不限只能使用 **file:// URL** 而已 — 他们也可以透过 Apache HTTP 服务器或 **svnserve**, 只要将 **http://** 或 **svn:// URL** 中的主机名称改为 **localhost** 即可. 为你的 Subversion 档案库维护多个服务器行程, 大概都是头痛大于需要. 我们建议你使用最适合需要的服务器, 然后就别再三心二意了.

新增专案

在档案库建立并设定好之后,剩下的就是使用它了.如果你有一群现有的数据,准备要纳入版本控制的话,你会想要用 **svn** 客户端程序的 `import` 子命令来进行.不过在你这么作之前,你应该仔细地考虑档案库的长期计划.在本节中,我们会提供一些如何规画档案库配置的建议,以及如何在这个配置中放置你的数据.

选择一种档案库配置

虽然 **Subversion** 可以让你随意移动受版本控管的目录与档案,而又不会遗失信息,不过这么作还是会打乱经常存取档案库,习惯什么东西会在什么地方的使用者的工作流程.请试着稍微考虑一下未来;在把数据纳入版本控制之前,先作好计划.藉由一开始就对档案库的内容先作好“规划”,你可以免去未来的头痛事件.

在设定 **Subversion** 档案库时,有几件事得考虑.假设你是档案库管理员,你得负责用于数个项目的版本控制系统的支持.第一个决定,你要对多个项目使用一个档案库,还是每个项目都有它自己的档案库,还是这两者的折衷方案.

对多个项目使用一个档案库有一些好处,最明显的是不需作重复的维护动作.一个档案库表示它有一组挂勾命令稿,一个例行的备份,如果 **Subversion** 发行一个不兼容的新版本,也只要进行一个倾印与加载,诸如此类的.再者,你也可以在各个不同的项目之间搬移档案,不会漏失掉任何版本信息的历程纪录.

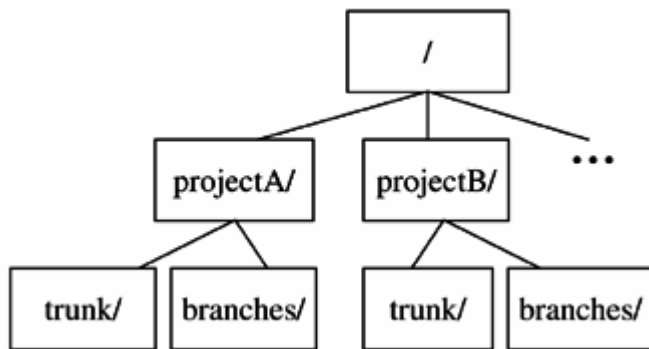
使用单一档案库的缺点,就是不同的项目可能会有不同的送交邮递论坛,或是不同的认证与授权的要求.另外,请记得 **Subversion** 使用的档案库范围的修订版号.有人对于别的项目中正如如火如荼在进行修改,但是自己的项目却因为全域修订版的关系,其最年轻的修订版号一直努力攀升会相当地反感.

不过还是有中间路线可供取舍.举个例子,不同的项目可以根据他们的相依程度归类在一起.你可以在每个档案库中,放上几个项目.这样子,很有可能会分享资料的项目可以很容易达到目的,当档案库增加修订版号时,发展人员至少可以知道,这些更动对同一个档案库上的人员都是有一些影响的.

在决定你的项目如何与档案库组织起来后,你大概就要思考档案库的目录架构.由于 **Subversion** 利用一般的目录复本来作分支与标记(请参考 [Chapter 4, 分支与合并](#)), **Subversion** 社群建议使用两个方法之一.这两种方法都使用以下的目录,分别名为 `trunk`, 表示主要项目发展的目录位置,以及 `branches`, 于其中建立主要发展线不同具名分支.

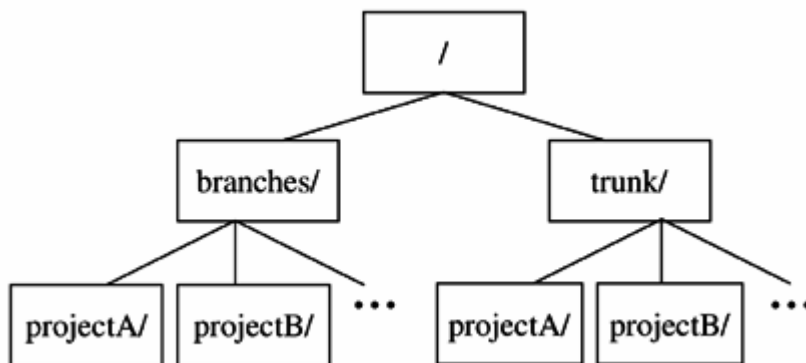
第一种方法是把每个项目放在根档案系统目录之下的子目录,每个项目计划目录其下接着为 `trunk` 与 `branches` 目录,如图 5-1 所示.

Figure 5.1. 一种建议的档案库配置.



第二种则是反过来 — 在档案系统最顶层之下的是 `trunk` 与 `branches` 目录, 其下则为所有档案库中的项目目录, 如图 5-2 所示:

Figure 5.2. 另一种建议的档案库配置.



请以你认为最适合的配置来规画档案库. **Subversion** 并不会期望或强制你要用哪一种配置 — 在它的眼中, 是目录的目录就只是目录. 归根究底, 你应该选择一种最适合你, 以及项目的档案库配置.

建立配置, 汇入起始数据

决定档案库要如何安排项目之后, 你应该就开始要以选择的配置来建立档案库, 并放置数据进去. **Subversion** 有几种方式可以达到这样的目的. 你可以使用 **svn mkdir** 命令 (请参考 [Chapter 8, 完整 Subversion 参考手册](#)) 命令, 依档案库配置的骨架, 一个一个将目录建立起来. 比较快达到相同目的的方法, 就是使用 **svn import** 命令 (请参考 [the section called “svn import”](#)). 在磁盘的暂时位置先建立好档案库配置后, 你可以利用单一送交, 将整个配置树汇入至档案库中:

```

$ mkdir tmpdir
$ cd tmpdir
$ mkdir projectA
$ mkdir projectA/trunk
$ mkdir projectA/branches
$ mkdir projectB
$ mkdir projectB/trunk

```



```
$ mkdir projectB/branches
...
$ svn import projectA file:///path/to/repos --message 'Initial
repository layout'
Adding          projectA
Adding          projectA/trunk
Adding          projectA/branches
Adding          projectB
Adding          projectB/trunk
Adding          projectB/branches

Committed revision 1.
$ cd ..
$ rm -rf tmpdir
$
```

当你的配置骨架就定位后, 如果数据还没进去的话, 你就可以开始将真正的资料汇入你的档案库. 再声明一下, 要达到这个目的有很多方法. 你可以使用 **svn import** 命令. 你可以从新的档案库取出工作复本, 将数据移到工作复本中放好, 然后使用 **svn add** 与 **svn commit**. 不过当我们开始讲到这样的事情之后, 我们就不是在讨论档案库管理. 如果你还不熟悉 **svn** 客户端程序的话, 请参考 [Chapter 3, 导览](#).

摘要

现在, 你应该已经对如何建立, 设定, 以及维护 **Subversion** 档案库有了基本的认识, 我们已经介绍几个可用来帮助你的工具. 我们还涵盖了一些基本的 **Apache** 设定步骤, 可让你在网络上开放你的档案库. 在本章中, 我们也注意到共通的管理陷阱, 以及如何避免它们的建议.

剩下的, 就是让你决定该在档案库里放些什么有趣的数据.

[11] 这听起来好像很高尚, 很飘飘然的样子, 不过我们指的, 就是任何除了工作复本以外, 还对存放所有人的数据的神秘领域有兴趣的人而已.

[12] 举例: 硬盘 + 特大号电磁铁 = 灾难

[13] **Subversion** 的档案库倾印档案格式很像 **RFC-822** 格式, 与大部份电子邮件所用的一模一样.

透过这个档案格式要描述更动集合 — 每一个都应视为是新的修订版 —, 应该是较为容易的.

[14] 例如 **svnadmin setlog** 就会完全跳过挂勾界面.

[15] 你知道的 — 就是她的魔爪的集合名词.

[16] 他们真的很讨厌这么作.

[17] 这个连接埠编号已获 Internet Assigned Numbers Authority (IANA) 指定.

[18] “大家一起来” 的模式, 有的时候是, 我们得这么说, 有点混乱的.

Chapter 6. 进阶主题

Table of Contents

[执行时期的设定区域](#)

[设定区域配置](#)

[设定与 Windows 登录档](#)

[设定选项](#)

[Servers](#)

[Config](#)

[性质](#)

[为什么要用性质?](#)

[使用性质](#)

[特殊性质](#)

[svn:executable](#)

[svn:mime-type](#)

[svn:ignore](#)

[svn:keywords](#)

[svn:eol-style](#)

[svn:externals](#)

[外部定义](#)

[供货商分支](#)

[通用供货商分支管理程序](#)

[svn-load-dirs.pl](#)

如果你是从头开始逐章阅读本书的, 你应该已经知道如何使用 Subversion 客户端, 进行大多数常见的版本控制作业. 你知道如何从 Subversion 档案库取出一份工作复本. 你已经习惯利用 **svn commit** 与 **svn update** 功能来送交与取得更动. 你甚至可能已经会反射性地使用 **svn status** 命令而不自觉. 不管从哪个角度看, 你已经相当习惯使用 Subversion 了.

但是 Subversion 的功能不仅止于 “常见的版本控制作业”.

本章着重于 Subversion 并不常用的功能. 在本章中, 我们会讨论 Subversion 的性质 (或 “描述数据”) 支持, 如何透过修改 Subversion 的执行时期设定档, 来改变它的预设行为. 我们会描述你要怎么利用外部定义, 让 Subversion 从多个档案库取得资料. 对于 Subversion 发行档中的额外客户端与服务器端的工具, 我们也会有详尽的解说.

在阅读本章之前,你应该已经熟悉 Subversion 的基本目录与档案能力. 如果你还没看过这部份,或是需要再重新熟悉一下,我们建议你看看 [Chapter 2, 基本概念](#) 与 [Chapter 3, 导览](#). 当你能够驾驭基本的功能,也了解本章的内容,你就是 Subversion 的强力使用者了 — 无效退费!^[19]

执行时期的设定区域

Subversion 提供许多选用的行为, 可让使用者自由控制. 许多像这样的选项, 都是使用者想要套用在所有的 Subversion 作业上的. 所以, Subversion 不强迫使用者记下指定这些选项的命令列选项, 然后在每个进行的作业都用上它们, 而是透过设定档, 将它们隔离在 Subversion 设定区域中.

Subversion 设定区域是一个包含选项名称与对应值的两层次结构. 一般来讲, 它们通通被挤到一个特别的目录, 其中包含 设定档(第一层结构), 就只是标准 INI 格式(以“区块”作为第二层)的纯文字文件. 这些档案可以轻易地利用你常用的文字编辑器(像是 Emacs 或 vi) 进行编辑, 其中包含被客户端读取的指令, 用来决定几个使用者偏好的选择性行为.

设定区域配置

svn 命令列客户端在第一次执行的时候, 它会建立一个使用者设定区域. 在类 Unix 系统上, 它是使用者家目录里的 .subversion 目录. 在 Win32 系统上, Subversion 会建立一个名为 Subversion 的档案夹, 一般是在使用者 profile 目录的 Application Data 区域中. 但是在这个平台上, 实际的位置会依各个系统不同而不同, 而且都是由 Windows 登录档指定的.^[20] 我们会以 Unix 上的名称 .subversion 来表示使用者设定区域.

除了使用者设定区域之外, Subversion 还会注意到系统设定区域的存在. 这让系统管理员能够在特定的机器上, 为所有使用者设定预设环境. 请注意, 系统设定区域并不具有强制性 — 使用者设定区域里的设定会盖过系统设定区域, 而在 svn 程序的命令列选项是最后决定行为的部份. 在类 Unix 平台上, 系统设定区域是在 /etc/subversion 目录; 在 Windows 平台上, 它还是会在公用的 Application Data 区域(同样地, 还是要依 Windows 登录档里的设定而定)里的 Subversion 目录. 不像使用者设定区域一样, svn 程序不会试着去建立系统设定区域.

目前 .subversion 目录包含三个档案 — 两个设定档(config 与 servers), 以及一个 README.txt 档案, 用来描述 INI 格式. 在建立它们的时候, 这些档案包含 Subversion 支持选项的默认值, 大多数都被设定为批注, 并且包括这些设定值如何影响 Subversion 行为的描述文字. 要改变某一种行为, 你只需要以文字编辑器开启适当的档案, 然后修改对应选项的值. 如果任何时候你想要回复一个或多个档案的默认值, 你只要删除这些档案, 然后执行某些无害的 svn 命令, 像是 svn --version, 那些消失的档案就会以它们的预设状态重新建立起来.

使用者设定区域也会包含认证数据的快取. `auth` 目录拥有一群子目录, 里面包含数种 Subversion 支持的认证方式所需要的快取数据. 这个目录建立时, 其权限设定为只让使用者可以读取其内容.

设定与 Windows 登录档

除了一般使用 INI 程序的设定区域, 执行于 Windows 上的 Subversion 客户端也会利用 Windows 登录文件来纪录设定数据. 选项名称与值都与 INI 档案内的相同, 而“档案/区块”的架构也是一样的, 但是说明时会有点不同 — 在该种方式下, 档案与区块只是登录的不同阶层而已.

Subversion 会在 `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` 键里找系统设定值. 举个例子, `global-ignore` 是一个在 `config` 档的 `miscellany` 区块中的选项, 可以在 `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores` 中找到. 使用者设定值则应该存于 `HKEY_CURRENT_USER\Software\Tigris.org\Subversion` 中.

登录文件设定选项会在其对应的设定档案 *之前* 被处理, 所以会被设定档案中的数值给覆盖过去. 换句话说, Windows 系统上的设定优先次序一定是以下列顺序处理的:

1. 命令列选项
2. 使用者的 INI 档案
3. 使用者的登录档数值
4. 系统的 INI 档案
5. 系统的登录文件数值

另外, Windows 的登录档并不支持“批注”的语法. 不过 Subversion 会忽略任何以井字号 (#) 作为开头的选项键. 这让你能够把一个 Subversion 选项变成批注, 而不必将其自登录档中删去, 显而易见地, 让回复该选项的动作简单多了.

`svn` 命令列用户程序永远不会写入 Windows 登录档, 也不会在那里建立预设的设定区域. 你可以利用 **REGEDIT** 来建立你需要的键. 另外, 你也可以建立一个 `.REG` 档, 然后在档案总管中双击该档, 这样会让里面的数据并入你的登录档中.

Example 6.1. 登录项目 (.REG) 档案的范例.

REGEDIT4

```
[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\groups]

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-proxy-host"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-proxy-password"=""
```

```

"#http-proxy-exceptions"=""
"#http-timeout"="0"
"#http-compression"="yes"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-trust-default-ca"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#store-password"="no"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#editor-cmd"="notepad"
"#diff-cmd"=""
"#diff3-cmd"=""
"#diff3-has-program-arg"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#global-ignores"="*.o *.lo *.la ## *.rej *.rej .*~ *~ .#"
"#log-encoding"=""
"#use-commit-times"=""
"#template-root"=""
"#enable-auto-props"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auto-props]

```

前述的范例是 .REG 档案的内容, 其中包含了几个常用的设定选项, 以及它们的默认值. 请注意里面同时包含了系统 (网络代理服务器相关的选项) 与使用者设定 (编辑器程序与储存密码的地方, 以及有的没有的). 你只需要自选项名称之前移去井字号 (#), 然后设定你想要的值.

设定选项

在本节中, 我们会讨论目前 Subversion 支持的执行时期设定选项.

Servers

servers 档案包含了有关于网络部份的 Subversion 设定选项. 本档案中有两个特定的区块名称 — groups 与 global. groups 区块实际上是个交互参照的表格. 本区块中的设定键是档案中其它区块的名称; 它们的值是 glob — 一种文字记号, 可以包含万用字符 — 它们会用来与 Subversion 提出要求的主机名称进行比较.

```

[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
...

d[collabnet]
...

```

当 Subversion 透过网络使用时, 它会试着将尝试连接的主机名称与 groups 区块的群组名称作比对. 如果比对成功, Subversion 会在 servers 档案中, 找寻名称与比对到的群组名称相同的区块. 然后由该区块中, 读取真正的网络设定区块.

所有无法与任何一个 groups 区块的 glob 比对到的服务器, 都会使用 global 区块中的设定. 本区块中所能使用的设定, 与该档案中 (当然了, 特殊的 groups 区块是例外) 的其它服务器区块内所使用的设定是相同的, 可用的如下:

http-proxy-host

用来指定代理服务器主机名称, Subversion 的 HTTP 要求会经由它传送出去. 预设为空白值, 表示 Subversion 不会将 HTTP 要求透过代理服务器传送, 而是试着直接与目标机器沟通.

http-proxy-port

用来指定代理服务器主机的连接端口, 预设为空白值.

http-proxy-username

用来指定代理服务器所用的使用者名称, 预设为空白值.

http-proxy-password

用来指定代理服务器所用的密码, 预设为空白值.

http-timeout

用来指定等待服务器响应的时间, 单位为秒. 如果你遇到因为低速网络联机所导致的 Subversion 作业逾时, 你应该增加本选项的数值. 默认值为 0, 表示让低层的 HTTP 链接库, 也就是 Neon, 直接使用它的预设逾时设定.

http-compression

指定 Subversion 是否应该压缩送往 DAV 服务器的网络要求. 默认值为 yes (不过只有连压缩的功能一起编进网络层才有用). 将它设定为 no 会关闭这个功能, 像是对网络传输进行除错时.

neon-debug-mask

这是一个整数屏蔽, 由底层 HTTP 链接库 Neon 使用, 用来选择会产生哪些类型的除错输出. 默认值为 0, 表示不会显示任何除错输出. 想要知道 Subversion 如何使用 Neon, 请参见 [Chapter 7, Developer Information](#).

ssl-authority-file

这是用来指定包含 certificate authority (或称 CA) 的档案, 当 Subversion 透过 HTTPS 存取档案库时, 这些会被客户端程序所接受.

svn-tunnel-agent

指定外部代理程序, 作为 SVN 通讯协议的信道.

Config

config 档案包含了其它目前 Subversion 可用的执行时期选项, 这些选项都与网络无关. 目前可用的选项不多, 不过它们还是分类成不同的区块, 以应付未来的需求.

auth 区块包含了与 Subversion 档案库的认证与授权有关的选项, 它包含了:

store-password

这个选项告诉 Subversion 是否要快取使用者对服务器认证响应的密码. 默认值是 yes. 将这个选项设为 no 将关闭这个储存在磁盘上的快取. 你可以透过 **svn** 命令的 `--no-auth-cache` 命令列参数 (如果子命令支持它的话), 暂时取代这个选项的效用.

helpers 区块设定 Subversion 用来完成其工作的外部应用程序. 本区块的有效选项为:

editor-cmd

指定 Subversion 在进行送交作业时, 用来向使用者要求送交讯息的程序, 像是使用 **svn commit** 而又没有提供 `--message (-m)` 或 `--file (-F)` (`-F`) 选项时. 这个程序也会用在 **svn propedit** 命令中 — 会有一个暂存档, 里面有使用者想要编辑的性质的值, 编辑的行为会直接在编辑器中发生 (参见 [the section called “性质”](#)). 这个选项的默认值为空白值. 如果这个选项没有设定的话, Subversion 会退而检查 `SVN_EDITOR`, `VISUAL` 与 `EDITOR` 环境变量 (依此顺序), 以取得编辑器命令.

diff-cmd

指示差异程序的绝对路径, 在 Subversion 产生“差异”输出 (像是使用 **svn diff** 命令时) 时使用. 默认值是 GNU diff 程序的路径, 由 Subversion 源码编译系统所决定.

diff3-cmd

指定三向差异程序的绝对路径. Subversion 使用这个程序, 将使用者产生的更动与来自档案库的更动合并在一起. 默认值是 GNU diff3 程序的路径, 由 Subversion 源码编译系统所决定.

diff3-has-program-arg

如果 `diff3-cmd` 选项接受 `--diff-program` 命令列参数的话, 本旗标应该设为 `true`. 由于 `diff3-cmd` 选项的默认值是在编译时期决定的, `diff3-has-program-arg` 的默认值亦同.

`miscellany` 区块是所有不属于其它区块的选项集合处.^[21] 在本区块中, 你可以看到:

`global-ignores`

在执行 **svn status** 命令时, 除了纳入版本控制的之外, Subversion 也会列出所有未纳入版本控制的目录与档案, 并以 `?` 字符表示 (请参见 [the section called “svn status”](#)). 有的时候, 看到列表中有一些让人不感兴趣或未纳入版本控制的项目 — 举个例子, 像是程序编译时所产生的 `object` 档 — 是很令人厌烦的. `global-ignores` 选项是一个以空格符隔开的 `glob` 列表, 用以描述 Subversion 不应显示的目录与文件名称, 除非已纳入版本控制. 默认值是 `*.o *.lo *.la ##* *.rej *.rej .*~ *~ .#*`.

你可以在单次执行 **svn status** 命令时忽略本选项, 只要透过 `--no-ignore` 命令列旗标即可. 想要知道更多忽略项目的控制细节, 请参考 [the section called “svn:ignore”](#).

性质

我们已经详细讲解 Subversion 如何在它的档案库中储存与取出纳入版本控制的档案. 有一整章的内容, 都在讲解这个工具所提供的基本功能. 就版本控制的角度来看, 如果版本控制的支持只有这些, Subversion 还是一个完整的工具. 不过它的还不止如此而已.

除了对你的目录与档案进行版本控制之外, Subversion 还提供了一个界面, 可用来新增, 修改, 以及移除已纳入版本控制的目录与档案的版本控制描述数据. 我们称这个描述数据为 *性质*, 可以把它们想成一个两栏的表格, 将每个档案库内的项目所关联的性质名称与任意的数值对映在一起. 一般来讲, 性质的名称与数值可以依你所需任意设定, 唯一的要求, 就是名称必须是人类可读的文字. 最棒的部份, 就是这些性质也是纳入版本控制的, 就像你的档案的文字内容一样. 你可以修改, 送交, 以及回复性质修改, 就像送交文字更动一样简单. 在你更新工作复本的时候, 也可以接收到别人的性质更动.

Subversion 中的其它性质

性质在 Subversion 其它地方也会出现. 就如同目录与档案可以有任何与之关联的性质名称与值一样, 每一个修订版也可以有任何与之关联的性质. 它们的限制都是一样 — 名称为人类可读的文字, 以及任何你想要的二进制数值 — 除了修订版性质并未纳入版本控制. 请参见 [the section called “无版本控制的性质”](#), 以了解更多有关这些未纳入版本控制性质的信息.

在本节中, 我们会检视性质支持的工具 — 供 Subversion 的使用者, 以及 Subversion 本身使用的都有. 你会学到与性质有关的 **svn** 子命令, 以及性质修改如何影响正常的 Subversion 工作流程. 我们希望你会发现, Subversion 的性质可以增进你在使用版本控制的经验.

为什么要用性质?

对你的工作复本来说, 性质可说是相当有用的附加价值. 事实上, Subversion 自己就利用性质来放置特殊的信息, 而且当作一种表示还需要进行某些处理的方法. 同样地, 你可以依自己所需来使用性质. 当然啰, 任何你可以透过性质达到的, 你也可以透过一般纳入版本控制的档案作到, 不过先看看以下的 Subversion 性质使用例子.

假设你想要设计一个放置了许多数字照片的网站, 并以标题与日期戳记来显示它们. 现在, 你的相片一直在变动, 所以你喜欢像大部份的站台一样, 都尽量能够自动化. 这些相片可能相当地大, 就像这类站台的共通特性一样, 你希望能够为站台访客提供缩小的图片. 你可以利用传统的档案来达到这样的目的, 像是同时摆放 `image123.jpg` 与 `image123-thumbnail.jpg`. 或者如果你想要让文件名称保持相同, 你可能会把缩小图片放在不同的目录, 像是 `thumbnails/image123.jpg`. 你也可以利用类似的方法, 储存标题与日期戳记, 一样还是存放在与原始影像不同的档案. 很快地, 你的档案树就会一团糟, 每当有新的相片放上去, 就会增加好几个档案.

现在考虑以 Subversion 的档案性质, 来进行相同的设定. 想象有一个影像文件 `image123.jpg`, 然后将该档的性质设定为 `caption`, `datestamp` 以及 `thumbnail`. 现在你的工作复本目录看起来更容易管理了 — 事实上, 里面除了影像文件以外, 就没别的了. 但是你的自动命令档了解的更多. 它们知道可以利用 **svn** (更好的, 则是利用 Subversion 的语言系统结 — 请参考 [the section called “Using Languages Other than C and C++”](#)), 挖出网站所需的额外信息, 而不必再去读取索引文件, 或是玩着操弄路径的游戏.

你该如何 (或者该说是否) 使用 Subversion 的性质, 都由你自己决定. 如前所述, Subversion 对性质有自己的用法, 我们在本章稍候会讨论到. 但是首先, 让我们讨论如何以 **svn** 程序来使用性质.

使用性质

svn 命令提供了几种新增或修改目录与档案性质的方法. 对于有着简短而可读的性质内容, 要新增性质最简单的方法, 就是利用命令列的 **propset** 来指定子命令的性质名称与数值.

```
$ svn propset copyright '(c) 2003 Red-Bean Software' calc/button.c
property 'copyright' set on 'calc/button.c'
$
```

但是我们已强调, **Subversion** 为性质的数值提供了强大的弹性. 如果你的性质内容想要有多行文字, 更甚是二进制内容的话, 你大概不会想要在命令列指定其内容, 所以 **propset** 子命令提供了一个 `--file (-F)` 选项, 用以指定一个文件名称, 将该档的内容作为性质的内容.

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

除了 **propset** 命令以外, **svn** 程序还提供了 **propedit** 命令. 这个命令会使用设定的文字编辑程序 (参见 [the section called “Config”](#)) 以新增或修改性质. 当你执行这个命令时, **svn** 会让文字编辑器修改一个暂存档, 里面包含了目前指定性质的内容 (或者什么都没有, 如果你要新增一个性质的话). 接着你只要在文字编辑器中修改这个内容, 直到它成为你想要的新内容, 然后存盘并跳离程序. 如果 **Subversion** 发现你真的修改了该性质的现有内容, 那么它会接受, 并视为新的性质内容. 如果你未作任何修改就跳离文字编辑器的话, 那么性质并不会有任何变动.

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

就像其它的 **svn** 子命令, 我们应该注意到性质也可以一次作用在多个路径上. 这让你能够利用一个命令, 一次修改一组档案的性质. 举个例子, 我们可以这样作:

```
$ svn propset copyright '(c) 2002 Red-Bean Software' calc/*
property 'copyright' set on 'calc/Makefile'
property 'copyright' set on 'calc/button.c'
property 'copyright' set on 'calc/integer.c'
...
$
```

如果你不能很容易地取得储存的性质内容的话, 这些新增与编辑性质的功能就没有多大的用处. 因此 **svn** 程序提供了两个子命令, 用以显示储存在目录与档案的性质名称与内容. **svn proplist** 命令会列出所有储存于某一路径的所有性质名称. 当你知道该节点的性质名称之后, 你就可以利用 **svn propget** 来取得个别的内容. 只要指定一个路径 (或一组路径) 以及一个性质名称, 这个命令会在标准输出串流上显示性质内容.

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
  copyright
  license
$ svn propget copyright calc/button.c
(c) 2003 Red-Bean Software
```

proplist 命令甚至还有变形, 可以列出所有性质的名称与其内容. 只要指定 **--verbose (-v)** 选项即可.

```
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2003 Red-Bean Software
  license :
=====
Copyright (c) 2003 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.
...
```

最后一个与性质有关的子命令, 是 **propdel**. 由于 Subversion 允许你储存空值的性质, 你无法利用 **propedit** 或 **propset** 来移除性质. 举个例子, 以下的命令 不会产生你想要的效果:

```
$ svn propset license '' calc/button.c
property `license' set on 'calc/button.c'
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2003 Red-Bean Software
  license :
$
```

你需要使用 **propdel** 命令来删除性质. 它的语法很接近其它性质命令:

```
$ svn propdel license calc/button.c
property `foo' deleted from ''.
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2003 Red-Bean Software
$
```

现在你已经熟悉了所有关于性质的 **svn** 子命令, 让我们看看性质修改如何影响一般的 Subversion 工作流程. 如我们早先提过的, 目录与档案的性质都被纳入版本管理, 就像你的档案内容一样. 这样的结果, 就是 Subversion 也提供合并 — 无误地合并, 或是产生冲突 — 别人更动的机会.

修改修订版性质

记得未纳入版本控制的修订版性质吗? 你也可以利用 **svn** 程序来修改它们. 只要加上 **--revprop** 命令列参数, 然后指定你想要修改的性质的修订版号即可. 由于修订版号是全域的, 只要你位于想要修改修订版性质的档案库的工作复本中, 你

就不需要指定路径名称. 举个例子, 你可能会想要更动现有修订版的送交纪录信息.[\[22\]](#)

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

请注意, 想要修改这些未纳入版本控制的性质, 必须由档案库管理员特别加入这个功能 (参见 [the section called “Hook scripts”](#)). 由于性质并未纳入版本控制, 如果你对修改掉以轻心的话, 你就有可能永远遗失该项信息. 档案库管理员可以设定防止这种损失的方法, 但是预设是关闭修改未纳入版本控制性质的功能.

就像档案内容一样, 性质的更动算是本地修改, 只有在你透过 **svn commit** 送进档案库后, 它才会永久生效. 你所作的性质更动也可以很容易地回复 — **svn revert** 命令会把你的目录与档案回复至它们尚未编辑的状态、内容、性质、所有的东西. 还有, 透过 **svn status** 与 **svn diff**, 你也可以得到关于目录与档案的一些有趣信息.

```
$ svn status calc/button.c
M      calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
```

```
Name: copyright
+ (c) 2003 Red-Bean Software
```

```
$
```

请注意 **status** 子命令是如何在第二栏, 而非第一栏中显示 **M**. 那是因为我们修改了 `calc/button.c` 的性质, 但是还没有修改它的文字内容. 要是我们两位都变更了, 我们也会在第一栏看到 **M** (请参见 [the section called “svn status”](#)).

性质冲突

与档案内容相同, 本地的性质更动也会与别人送交的相冲突而已. 如果你更新工作复本, 收到了纳入版本控制项目的更动, 但是它与你自己的相冲突, Subversion 就会回报该资源处于冲突的状态.

```
% svn update calc
M calc/Makefile.in
C calc/button.c
Updated to revision 143.
$
```

Subversion 也会在冲突资源的相同目录下, 建立一个扩展名为 `.prej` 的档案, 其中详述冲突的原因. 你应该检视这个档的内容, 以决定该如何解决该冲突. 在冲突

解决以前,你都会在该资源的 **svn status** 输出的第二栏看到 **c**,而所有的本地更动送交都会失败.

```
$ svn status calc
C      calc/button.c
?      calc/button.c.prej
$ cat calc/button.c.prej
prop `linecount': user set to '1256', but update set to '1301'.
$
```

要解决性质冲突,只要确定有冲突的性质内是应该有的数值,然后使用 **svn resolved** 命令,让 Subversion 知道你已经手动处理掉这个问题.

你也许也注意到,Subversion 使用非标准的方式来显示性质差异.你还是可以执行 **svn diff**,然后将结果重导向,以产生出适合的修补档. **patch** 程序会忽略掉性质的部份 — 一般来说,它会忽略任何它看不懂的东西.很不幸地,如果你想要完全套用 **svn diff** 的修补档的话,所有性质修改的地方,都必须手动为之.

如你所见,性质修改的存在,对于一般的 Subversion 工作流程并没有很大的影响.更新工作复本,检查目录与档案的状态,回报你所作的更动,以及送交更动回档案库的工作型态,与性质的存在与否是毫不相干的. **svn** 程序是多了几个可以用来修改性质的子命令,不过也就只有这个明显可见的差异而已.

特殊性质

Subversion 对性质没有什么特别的规定 — 你可以为任何理由而使用它们. Subversion 只要求使用者不要使用以 **svn:** 作为开头的性质.这个命名空间已经保留下来,有它自己的用途.事实上,Subversion 定义了几个特殊的性质,它对关联到的目录与档案有神奇的效用.在本节中,我们会解开这层神秘的面纱,说明这些特殊的性质是如何让你的生命稍微好过一点.

svn:executable

svn:execute 性质是以半自动的方式,用以控制一个纳入版本控制的档案在档案系统中的执行权限位.这个性质并没有指定内容 — 只要它存在,就表示 Subversion 应该开启它的执行权限位.移除它的话,就完全由操作系统控制执行权限位.

在许多操作系统中,一个档案是否能当作命令来执行,取决于执行权限位是否存在.这个位通常是被关闭的,必须由使用者对每一个需要的档案开启它.在工作复本中,在每一次更新收到现有档案的新版本时,会当作新档案来建立.这表示你可以开启一个档案的执行权限位,然后更新工作复本,如果那个档案也是更新的一部份,它的执行权限位会被关闭.为此,Subversion 提供了 **svn:executable** 性质,以作为保持执行权限位开启的方法.

在没有执行权限位概念的档案系统上,这个性质是没有作用的,像是 FAT32 与 NTFS.^[23] 另外,虽然它没有定义值,Subversion 在设定这个性质时,会强制它的值为 *. 最后,这个性质只对档案有效而已,对目录是无效的.

`svn:mime-type`

`svn:mime-type` 性质在 Subversion 中有很多作用. 除了作为储存档案的多用途因特网邮件延伸语法 (MIME) 分类之外,这个性质的内容还会决定几项 Subversion 的行为特征.

举个例子,如果 `svn:mime-type` 性质设为文字的 MIME 类别 (一般来说,即为不是以 `text/` 开始的,不过还是有例外),Subversion 会假设该档的内容是二进制 — 也就是人类看不懂 — 的数据. Subversion 提供的功能中,其中一项是在从服务器收到工作档的更新中,依文字内容与文字列进行合并. 但是对据信含有二进制数据的档案,根本就没有“文字列”的概念. 因此,Subversion 对这些档案在更新时,不会试着进行内文合并. 它改用另一种方式,更新时如果有二进制档案有本地更动的话,你的档案会改以 `.orig` 扩展名为名,然后 Subversion 会以原始档名储存一份新的工作档案,其中包含有更新时收到的更动,但是不包括你自己的本地更动. 这样的行为是要保护使用者,以避免对无法进行内文合并的档案进行内文合并.

Subversion 在执行 `svn import` 与 `svn add` 子命令时,会使用二进制侦测运算法的方式来协助使用者. 这些子命令透过聪明的方法,侦测档案的“二进制性”,然后设定据信为二进制档案的 `svn:mime-type` 性质为 `application/octet-stream` (一般的“这是一些位集合”的 MIME 类别). 如果 Subversion 猜错了,或是你希望将 `svn:mime-type` 设定成更为明确的值 — 可能是 `image/png` 或 `application/x-shockwave-flash` — 你都可以移除或是手动编辑这个性质.

最后,如果 `svn:mime-type` 性质被设定的话,在响应 GET 要求时,Subversion 的 Apache 模块会使用这个值来作为 HTTP 标头 `Content-type:` 的内容. 在以浏览器观看档案库的内容时,它对如何显示一个档案提供了决定性的提示.

`svn:ignore`

`svn:ignore` 性质包含了档案样式的列表,Subversion 处理时会忽略. 它也许是最常使用特殊性质,可以与执行时期设定的 `global-ignores` 选项 (请参见 [the section called “Config”](#)) 一起工作,以便在类似 `svn status` 的命令中过滤掉未纳入版本控制的目录与档案.

`svn:ignore` 性质的基本原理很容易解释. Subversion 并不会假设每个在工作复本目录里的子目录或档案都会被纳入版本控制. 一个资源必须透过 `svn add` 命令,才会被纳入 Subversion 的管理. 也因为如此,工作复本中常有许多并未纳入版本控制的资源.

现在, **svn status** 命令会将工作复本中每个没有被 `global-ignores` 选项 (或其默认值) 过滤掉的目录与档案给显示出来. 这样作的原因, 是让使用者可以看看是否忘了把某个资源加进版本控制.

但是 Subversion 没有任何方法可以猜测什么名字是应该忽略的. 而且某个档案库的 每一个工作复本都常常会有固定该忽略的东西. 要强迫每一个使用该档案库的使用者将这些资源的样式加进他们自己的执行时期设定档, 不只是负担而已, 还有可能会搞乱该使用者为其它已取出的工作复本所作的设定.

解决的方法, 就是把针对可能出现在某个目录里的数据的忽略样式, 储存在该目录上. 常见的专属于某个目录却又不希望纳入版本控制的例子, 包括编译过程中产生的档案, 或者 — 用个更适合本书的例子 — 像是 **HTML**, **PDF**, 或是 **PostScript** 档案, 这些由某些 **DocBook XML** 源码档案转换成更合用的输出格式的结果.

CVS 使用的忽略样式

Subversion 的 `svn:ignore` 性质的语法与功能与 CVS 的 `.cvsignore` 档案很相似. 事实上, 如果你是把 CVS 的工作复本移植到 Subversion, 你可以将 `.cvsignore` 档案作为 **svn propset** 命令的输入档案, 直接汇入忽略样式:

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

但是呢, CVS 与 Subversion 处理忽略样式还是有一些差异. 这两个系统使用忽略样式的时机不太相同, 而且忽略样式套用的范围也不太相同. 还有, Subversion 也不认识作为取消忽略样式之用的

!

样式.

因为这个缘故, `svn:ignore` 性质就是解决方案. 它的内容是多行的档案样式集合, 每一个样式一行. 这个性质是设在你希望它生效的目录之上. [\[24\]](#) 举个例子, 假设你有以下的 **svn status** 的输出:

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


在这个范例中, 你对 `button.c` 作了一些性质更动, 但是在你的工作复本中还有其它的未纳入版控制的档案. 像是在这个例子中, 有从源码编辑出来的最新 `calculator` 程序, 一个名为 `data.c` 的源码档, 以及一组除错用的输出记录文件. 现在, 你知道你的编译系统都会产生 `calculator` 程序.^[25] 然后你还知道你的测试工具都会到处留下除错用的纪录文件. 这对每一个工作复本都是既存的事实, 不单只有你的如此. 你也知道你在每次执行 `svn status` 时, 并不希望看到这些档案. 所以你透过 `svn propedit svn:ignore calc` 对 `calc` 目录加上一些忽略样式. 举个例子, 你也许会以下的值作为 `svn:ignore` 性质的新内容:

```
calculator
debug_log*
```

在你加上这个性质后, `calc` 目录就有了本地更动. 请注意你的 `svn status` 输出的不同处:

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

现在, 所有不想看到的东西都从输出中消失了! 当然啰, 这些档案都还在你的工作复本中, **Subversion** 只是不会再提醒你这些东西的存在, 还有它们尚未纳入版本控制. 现在, 这些无用的杂音自输出移开之后, 你就能专注在更有趣的项目上 — 像是你可能忘了加进版本控制的源码.

如果你想要看到被忽略的档案, 你可以传递 `--no-ignore` 选项给 **Subversion**:

```
$ svn status --no-ignore
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
```

```
svn:keywords
```

Subversion 具有取代 *关键词* — 有关纳入版本控制档案的有用信息 — 进入档案内容的功能. 一般来讲, 关键词描述了档案最近一次更动的信息. 由于这些信息会随着每次档案的更动而变更, 更重要的, 是在档案更动 *之后*. 如果不是由版本控制系统来保持这些数据的实时性的话, 不管以哪些处理方式都很麻烦. 如果留给人类使用者处理的话, 这些信息不可避免地会变成无人维护.

举个例子, 假设你有个文件, 想要在里面显示最近一次修改的日期. 你可以把这个负担加诸文件的作者身上, 只要在每一次送交他们的更动之前, 顺便把修改纪录最近一次修改日期的部份. 但是迟早有人会忘记这件事. 换个方式, 只要叫

Subversion 对 `LastChangedDate` 关键词进行关键词取代即可. 藉由摆放 *关键词定位锚*, 你可以控制关键词出现在文字中的位置. 前述的定位锚就只是格式为 `$关键词名称$` 的文字字符串.

Subversion 定义了可用来进行取代的关键词列表. 这个列表包含了以下五个关键词, 有些还可以使用较短的别名:

`LastChangedDate`

这个关键词描述了最近一次更动的日期, 看起来像 `$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $`. 它也可以缩写成 `Date`.

`LastChangedRevision`

这个关键词描述本文件于档案库中最后一次更动的修订版, 看起来像 `$LastChangedRevision: 144 $`. 它可以缩写成 `Rev`.

`LastChangedBy`

这个关键词描述了本档案于档案库最后一次修改的使用者, 看起来像 `$LastChangedBy: harry $`. 它可以缩写成 `Author`.

`HeadURL`

这个关键词描述了本档案于档案库中的最新版的完整 **URL**, 看起来像 `$HeadURL: http://svn.collab.net/repos/trunk/README $`. 它可以缩写成 `URL`.

`Id`

这个关键词是其它关键词的压缩集合. 它被取代后的样子, 看起来像 `$Id: calc.c 148 2002-07-28 21:30:43Z sally $`, 表示档案 `calc.c` 最近一次是在修订版 14, 于 2002 年七月 28 日晚上被使用者 `sally` 修改.

只把关键词定位锚加进档案里的话, 什么事也不会发生. 除非你明确地表达要这么作的意愿, 不然 **Subversion** 不会试着对你的档案内容进行内容取代. 毕竟, 你可能在写一份关于如何使用关键词的文件 [\[26\]](#), 并不想要 **Subversion** 把你不需要取代的关键词定位锚的美丽例子给取代掉.

要告诉 **Subversion** 是否该对某一个档案进行关键词取代, 我们得转向使用性质相关的子命令. 当某一个纳入版本控制档案的 `svn:keywords` 性质被设定时, 它会控制该档案哪个关键词应该被取代. 这个值的内容, 是前述表格中的关键词或别名, 以空白隔开的列表.

举个例子, 假设你有一个纳入版本控制的档案, 名为 `weather.txt`, 看起来像这样:

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

如果没有设定该档案的 `svn:keywords` 性质, **Subversion** 什么事也不会作. 让我们开启关键词 `LastChangedDate` 的内容取代.

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

现在你对 `weather.txt` 产生了一个本地的性质修改. 你还不会看到档案的内容有任何的变化 (除非你已经设定过了这个性质). 请注意这个档案包含了一个 `Rev` 关键词的定位锚, 但是我们的性质内容并未包含这个关键词. **Subversion** 会很高兴地略过不在档案中的关键词取代要求, 而且也不会取代不在 `svn:keywords` 性质内容中的关键词.

关键词与虚假差异

因为使用者看得到关键词被取代了, 使用者会误认为对开启那个功能的档案进行差异比较时, 最少都会有包含该关键词的部份. 但是实际上并非如此. 当 **svn diff** 在检查本地更动时, **Subversion** 会对原先已取代的关键词进行“取消取代”. 结果就是储存在档案库里的档案, 再加上使用者真正对它作的更动的版本.

在你送交了这个性质更动之后, **Subversion** 会以新的取代文字更新你的工作档案. 你看到的不会是关键词定位锚 `$LastChangedDate$`, 而是它被取代后的结果. 结果会包含关键词的名称, 而且仍旧包含在钱字号 (\$) 里. 如我们所预测的, `Rev` 关键词并不会被取代, 因为我们并未这样要求.

```
Here is the latest report from the front lines.  
$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $  
$Rev$  
Cumulus clouds are appearing more frequently as summer approaches.
```

如果别人现在送交了 `weather.txt` 的更动, 你的档案副本还是会像原来的样子显示被取代的关键词 — 除非你更新工作复本. 在那个时候, 你的 `weather.txt` 里的关键词会重新被取代, 以反映该档案最近一次所作的送交.

```
svn:eol-style
```

除非另外指定版本控制档案的 `svn:mime-type` 性质, **Subversion** 会假设档案包含人类可读的数据. 一般来说, **Subversion** 只会这样来决定回报档案内容差异是否可行. 不然的话, 对 **Subversion** 来说, 位就只是位而已.

这表示 Subversion 预设并不会处理档案里 *列尾符号(EOL)* 标示. 很不幸地, 不同的操作系统使用不同的符号来表示一列的结尾. 举个例子, 一般用在 Windows 平台上的列尾符号是两个 ASCII 控制字符 — 返回字符 (CR) 与换行字符 (LF). 但是 Unix 软件就只使用 LF 字符来表示一列的结尾.

并不是这些操作系统上的程序, 都会认得执行平台的 *原生列尾样式* 以外的格式. 一般的结果, 是 Unix 程序把 Windows 档案的 CR 字符表示成一般的字符 (一般显示成 ^M), 而 Windows 程序会把 Unix 档案里的所有文字列合并成一个超长的文字列, 这是因为没有返回-换行 (或是 CRLF) 字符组合的存在来表示一列的结束..

像这样对外来的 EOL 符号这么敏感, 对不同操作系统之间共享档案的人很不方便. 举个例子, 想象有一个源码档, 而发展人员各在 Windows 与 Unix 系统上编辑这个档案. 如果所有的发展人员都使用会保留档案的列尾样式的工作, 那就一点问题都没有.

但是实际上, 许多常见的工具不是无法正确地读出有外来的 EOL 符号的档案, 就是在存盘时, 会转换成原生的列尾符号. 如果对发展人员是前者的话, 他就必须使用一个外部转换工作 (像是 **dos2unix** 或是 **unix2dos**), 让这个档案可以被编辑. 后者就不必作额外的处理了. 不过这两种方法, 都会让档案的每一列都与原来的不一样! 在送交他的更动之前, 使用者有两种选择. 不是使用转换工具, 把它还原成与编辑之前档案相同的列结尾样式, 就是直接送交这个档案 — 完全变成新的 EOL 符号.

像这样情况之下, 我们得到的就是时间的浪费, 以及对送交档案不必要的修改. 浪费时间已经够苦的了, 但是当送交的更动会改变档案的每一行时, 要找出到底修改了哪些地方就不是一件简单的工作. 臭虫到底在哪里被修正了? 哪一行导致了新的语法错误?

解决的方法是 `svn:eol-style` 性质. 当这个性质设定为一个有效值时, Subversion 会利用它来决定是否对该档案进行特别处理, 以免因为每一次有从不同操作系统来的送交, 而造成档案的列尾样式就要换一次. 有效的值为:

`native`

这会让档案的 EOL 符号为 Subversion 执行的平台的原生列尾符号. 换句话说, 如果一个 Windows 使用者取出的档案, 它的 `svn:eol-style` 性质设定为 `native`, 那个档案会包含 CRLF 的 EOL 符号. Unix 使用者取出相同档案的工作复本的话, 会看到该文件其中的是 LF EOL 符号.

请注意, 不管操作系统为何, Subversion 实际上是以正规化的 LF EOL 符号来储存的. 不过呢, 基本上使用者是不需要知道的.

`CRLF`

这会让档案包含 CRLF 序列作为 EOL 符号, 不管使用中的操作系统为何.

LF

这会让档案包含 LF 字符作为 EOL 符号, 不管使用中的操作系统为何.

CR

这会让档案包含 CR 字符作为 EOL 符号, 不管使用中的操作系统为何. 这种列尾样式并不常见. 它是用在旧型的麦金塔平台上 (不过 Subversion 在上面根本不能执行).

svn:externals

svn:externals 性质的值, 会指示 Subversion 以其它 Subversion 取出的工作复本来产生纳入版本控制的目录. 想要知道更多这个关键词的信息与用法, 请参见 [the section called “外部定义”](#).

外部定义

有的时候, 一个工作复本包含了数个不同来源的工作复本是很方便的. 举个例子, 你可能想要有数个不同的目录, 各来自同一个档案库的不同位置, 或是根本就是来自不同的档案库. 你当然可以手动建立 — 透过 **svn checkout**, 你就可以建立你想要建立的目录架构. 但是如果这个架构对每个使用这个档案库的人都很重要的话, 每个人就必须进行你所作过的相同动作.

很幸运地, Subversion 提供 *外部定义* 的支持. 外部定义是本地目录与外部版本控制数据的 URL. 在 Subversion 中, 你应该利用 svn:externals 性质来宣告外部定义的群组. 这个性质是对纳入版本控制的目录设定, 内容是子目录 (相对于本性质设定的纳入版本控制的目录) 对应至 Subversion 档案库 URL 的多行表格.

```
$ svn propget svn:externals calc
third-party/sounds          http://sounds.red-bean.com/repos
third-party/skins           http://skins.red-bean.com/repositories/skinproj
third-party/skins/toolkit    http://svn.red-bean.com/repos/skin-maker
```

svn:externals 性质便利之处, 在于一旦对纳入版本控制的目录设定完成之后, 每一个取出该目录的工作复本的人也会一并得到外部定义的好处. 换句话说, 一旦有人花时间定义了这些巢状工作复本架构, 别人就不必烦恼了 — 在取出原本的工作复本时, Subversion 也会一并取出外部工作复本.

请注意先前的外部定义范例. 当有人取出 calc 目录的工作复本, Subversion 还会继续取出在外部定义里的项目.

```
$ 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
...
```

如果你需要更改外部定义, 你可以使用正常的性质修改子命令来达成. 当你送交了 `svn:externals` 性质的更动, Subversion 会在你下次执行 **svn update** 时, 重新对更动过后的性质取出工作复本. 相同的事会发生在别人更新他们的工作复本, 然后收到你对外部定义更动时.

供货商分支

发展软件特别会遇到的情况, 就是你在版本控制下维护的软件, 常常是与别人数据相关, 或是根基于其上. 一般来讲, 你的项目会要求你随时更新这个外部软件, 但是又不损及你自己项目的稳定性. 这种情况一直都在发生 — 任何地方, 某一群人产生的数据, 对另一群人有着直接的影响.

举个例子, 软件开发人员可能发展一个应用程序, 它使用第三方的链接库.

Subversion 就是如此使用 Apache Portable Runtime library (请参见 [the section called “The Apache Portable Runtime Library”](#)). Subversion 的源码, 完全依靠 APR 链接库来达成它对移植性的要求. 在 Subversion 发展的早期阶段, 本项目紧密地追踪 APR 不断改动的 API, 一直盯住在链接库不停变动的“流血前线”. 现在 APR 与 Subversion 都已成熟了, Subversion 试着只对已充份测试, 稳定的发行版本的 APR 链接库的 API 同步.

现在, 如果你的项目倚靠别人的信息, 你有几种方法可以让这个信息与你自己的同步. 最麻烦的方法, 你可以口头或书面的告知项目的所有贡献者, 应该如何确认他们有项目所需的特定版本的第三方信息. 如果第三方信息是维护于 Subversion 档案库中, 你可以使用 Subversion 的外部定义, 以有效地将该信息的特定版本都“钉死在”工作复本中的相同位置 (请参见 [the section called “外部定义”](#)).

不过, 有的时候你也会想要在你自己的版本控制系统里, 维护自己对第三方数据的自订更动. 回到软件开发的例子, 程序设计师也许会依他们的需要, 对第三方程式库进行修改. 这些修改, 也许包含了新功能, 也许是臭虫修正, 只会在内部维护, 让它成为第三方程式库正式发行的一部份. 也许这个更动永远也不会回到链接库的维护人员, 它就只是让链接库更依软件开发人员所需而作的小更动.

现在你面临了一个有趣的情况. 你的项目可以将它对第三方资料的修改, 以和原来无关的方式储存, 像是修补档, 或是整个不同的目录或是档案. 但是很快地, 这些就会变成维护上的包袱, 需要某种机制将你的自订修改套用到第三方的数据上, 而且需要对每一个你追踪的第三方数据各个版本产生重新这些更动.

这个问题的解决方法, 就是使用 *供货商分支*. 供货商分支是在你的版本控制软件里的目录, 其中包含由第三方, 或是供货商所提供的信息. 每一个你决定置放在项目中的供货商数据版本, 称为 *供货商 drop*.

供货商分支有两个优点. 第一, 将现有支持的供货商 **drop** 存在你自己的版本控制系统, 项目的成员就不必烦恼他们是否有供货商资料正确版本的问题, 你会随着例行的工作复本更新而接收到正确版本的数据. 第二, 由于这些数据在你的 Subversion 档案库中, 你可以就地存放你的更动 — 你不需要自动 (或者更糟, 手动) 方法来切换你的自订更动.

通用供货商分支管理程序

管理供货商分支的方法, 一般是以这种方式处理. 你建立一个最上层的目录 (像是 /vendor) 来存放供货商分支. 然后将第三方源码汇入到这个最上层目录的一个子目录中. 然后, 你将这个子目录复制到你的主要发展分支 (举个例子, /trunk) 至适当的位置. 你的更动, 一定都是在主要发展分支里. 每次你追踪的程序代码有了新的发行版本时, 就把它放进供货商分支, 将更动合并回 /trunk, 并且解决本地更动与上游更动之间的冲突.

也许来个例子, 就更能了解这个方法. 我们使用一个案例, 你的发展团队正在制作一个计算器程序, 它需要连结到一个第三方的复数运算链接库, 叫作 libcomplex. 我们一开始先建立一个供货商分支, 然后汇入第一个供货商 **drop**.

```
...
$ svn import /path/to/libcomplex-1.0 \
http://svn.example.com/repos/calc/vendor/libcomplex/current \
    -m 'importing initial 1.0 vendor drop'
...
```

我们现在在 /vendor/libcomplex/current 中, 放置了现行版本的 libcomplex 源码. 现在我们建立这个版本的标记 (请参见 [the section called “标记”](#)), 然后将把它复制到主发展分支, 这样我们才能对它建立自订修改.

```
$ svn copy
http://svn.example.com/repos/calc/vendor/libcomplex/current \
    http://svn.example.com/repos/calc/vendor/libcomplex/1.0 \
    -m 'tagging libcomplex-1.0'
...
$ svn copy http://svn.example.com/repos/vendor/libcomplex/1.0 \
    http://svn.example.com/repos/calc/libcomplex \
    -m 'bringing libcomplex-1.0 into the main branch'
```


...

我们取出项目的主要分支 — 现在它包含了最初的供货商 **drop** 复本 — 然后开始建立 **libcomplex** 的自订源码. 在我们注意到之前, 我们修改过的 **libcomplex** 已经完全整合到我们的计算器程序.^[27]

几个星期后, **libcomplex** 的发展人员释出了新版的链接库 — 1.1 版 — 它包含了几个非常想要的特色与功能. 但是我希望升级到这个新版本, 但是又仍保有我们对现存版本的自订修正. 就如你所猜想的, 我们实际要作的, 就是以 **libcomplex** 1.1 的复本取代我们现存的基准版本 **libcomplex** 1.0, 然后再将我们对该链接库所作的自订修改套用到新的版本去.

要进行这样的升级, 我们先取得一份供货商分支的复本, 然后以新的 **libcomplex** 1.1 源码取得 **current** 版本. 在送交这个更动之后, 现在我们的 **current** 分支就包含了新的供货商 **drop**. 我们标记新的版本, 然后再将前一个版本的标记与新的现行版本之间的差异, 合并至我们的主要发展分支.

```
$ cd working-copies/calc
$ svn merge http://svn.example.com/repos/vendor/libcomplex/1.0      \
            http://svn.example.com/repos/vendor/libcomplex/current \
            libcomplex
... # resolve all the conflicts between their changes and our changes
... # 解决所有他们与我们的更动之间的冲突
$ svn commit -m 'merging libcomplex-1.1 into the main branch'
...
```

在普通的使用情况下, 从目录与档案的观点来说, 第三方工具的新版本看起来与前一个版本没什么差别. 换句话说, 没有任何一个 **libcomplex** 源码档案被删除, 更名, 或是移到新的位置 — 简单地讲, 在完美的世界中, 我们的更动可以无误地套用在新版的链接库, 一点也不复杂, 也不会有冲突发生.

但是事情并不总是那么简单, 而且事实上, 源码在不同发行版本之间常常会移动. 确认我们的更动仍能适用在新版本的源码变得复杂, 很快地, 我们会陷入必须手动对新版本重新作出我们的自订更动. 不过只要 **Subversion** 知道某个指定源码档的历史 — 包括它先前的位置 — 合并新版本链接库的过程就很简单了. 但是我们必须告诉 **Subversion** 在供货商 **drop** 版本之间的档案配置更动.

svn-load-dirs.pl

如果供货商 **drop** 包含好几个删除, 新增, 以及移动的档案的话, 更新第三方数据的连续版本就变得复杂. 所以 **Subversion** 提供了 **svn_load_dirs.pl** 脚本文件, 帮助你进行这样的作业. 这个脚本档会自动执行我们在通用供货商分支管理程序一节中所讲的汇入方法, 以将错误降到最低. 不过你还是要自己执行合并命令, 以将新版本的第三方数据合并至你的主发展分支, 但是 **svn_load_dirs.pl** 可以帮助更快地轻松地达到那个阶段.

简单地说, **svn_load_dirs.pl** 是 **svn import** 的加强, 它有以下几个重要特性:

- 它可以在任何时候, 实时地将现有档案库中的目录变得与外部目录一模一样, 并且执行所有需要的新增与删除动作, 有必要还可以执行移动的动作.
- 它会处理一连串需要 **Subversion** 先进行送交的复杂作业 — 像是在重新命名目录或档案两次之前.
- 它可以依需要, 标记新汇入的目录.
- 它可以依需要, 对符合正规表示式的目录与档案加上任意的性质.

svn_load_dirs.pl 需要三个主要的自变量. 第一个自变量是用来工作的基础 Subversion 目录的 URL. 接着这个自变量的是 URL — 相对于第一个自变量 — 目前的供货商 **drop** 会汇入至该目录下. 最后, 第三个自变量是要汇入的本地目录. 以我们前面的例子来说, **svn_load_dirs.pl** 执行起来就像这样;

```
$ svn_load_dirs.pl
http://svn.example.com/repos/calcul/vendor/libcomplex \
current
\
/path/to/libcomplex-1.1
...
```

藉由传递 **-t** 命令列选项, 并且指定一个标签名称, 你可以让 **svn_load_dirs.pl** 来标记新的供货商 **drop**. 这个标记是另一个相对于第一个程序自变量的 URL.

```
$ svn_load_dirs.pl -t libcomplex-1.1
\
http://svn.example.com/repos/calcul/vendor/libcomplex \
current
\
/path/to/libcomplex-1.1
...
```

当你执行 **svn_load_dirs.pl** 时, 它会检查你“现有”供货商 **drop** 的内容, 然后将它与新的供货商 **drop** 作比较. 在最简单的情况中, 不会有一个档案出现在一个版本, 而在另一个版本不见的情况, 命令稿会毫无问题地进行新的汇入. 不过要是版本之间有不同档案配置, **svn_load_dirs.pl** 会问你要如何解决这些差异. 举个例子, 你有机会告诉命令稿 libcomplex 1.0 版的 math.c 已经更名为 libcomplex 1.1 的 arithmetic.c. 任何没有以移动来解释的差异, 将会当作普通的新增与删除.

这个命令稿也接受一个独立的设定档, 设定符合正规表示式、*新增*到档案库的目录与档案的性质. 可以透过 **-p** 命令列, 向 **svn_load_dirs.pl** 指定设定档. 设定档的每一列, 包含以空白分隔的两个或四个值: 一个类似 Perl 风格的正规表示式, 用以比对新增的路径, 一个控制关键词 (可为 **break** 或 **cont**), 然后可以接一个性质的名称与其内容.

```
\.png$           break    svn:mime-type    image/png
\.jpe?g$         break    svn:mime-type    image/jpeg
\.m3u$           cont     svn:mime-type    audio/x-mpegurl
\.m3u$           break    svn:eol-style    LF
```



```
. *                               break    svn:eol-style    native
```

对每个新增的路径, 当依序比对的路径正规表示式比对成功时, 与其对应的性质会变更, 除非控制指定为 `break` (这表示该路径已经没有性质要变动的). 如果控制指定为 `cont` — 这是 `continue` 的缩写 — 那么比对会继续往控制档案的下一列进行.

任何在正规表示式、性质名称、或是性质内容里的空格符, 都必须以单引号或双引号字符包起来. 你可以将不是用来包起空格符的引号字符, 于其前加上倒斜线 (`\`) 将其逸出 (escape). 在剖析设定档时, 倒斜线只会用来逸出引号, 所以对正规表示式, 不必对不必要的字符保护过头.

[19] 这声明只对那些不花分文取得 Subversion 的大多数人有效.

[20] `APPDATA` 环境变量会指向 Application Data, 所以你一定可以使用 `%APPDATA%\Subversion` 来表示这个数据夹.

[21] Anyone for potluck dinner?

[22] 修正送交纪录讯息的拼字错误, 文法瑕疵, 以及“就是不正确”的问题, 大概是 `--revprop` 选项最常见的使用情况.

[23] Windows 档案系统使用档案扩展名 (像是 `.EXE`, `.BAT`, 以及 `.COM`) 来表示可执行档.

[24] 这些样式只对该目录有效而已 — 它们不会递归式地加诸在子目录上.

[25] 这不就是编译系统的重点吗?

[26] ... 也有可能是书中的一个章节 ...

[27] 而且还是没有臭虫的!

Chapter 7. Developer Information

Table of Contents

[Layered Library Design](#)

[Repository Layer](#)

[Repository Access Layer](#)

[RA-DAV \(Repository Access Using HTTP/DAV\)](#)

[RA-SVN \(Proprietary Protocol Repository Access\)](#)

[RA-Local \(Direct Repository Access\)](#)

[Your RA Library Here](#)

[Client Layer](#)
[Using the APIs](#)
 [The Apache Portable Runtime Library](#)
 [URL and Path Requirements](#)
 [Using Languages Other than C and C++](#)
[Inside the Working Copy Administration Area](#)
 [The Entries File](#)
 [Pristine Copies and Property Files](#)
[WebDAV](#)
[Programming with Memory Pools](#)
[Contributing to Subversion](#)
 [Join the Community](#)
 [Get the Source Code](#)
 [Become Familiar with Community Policies](#)
 [Make and Test Your Changes](#)
 [Donate Your Changes](#)

Subversion is an open-source software project developed under an Apache-style software license. The project is financially backed by CollabNet, Inc., a California-based software development company. The community that has formed around the development of Subversion always welcomes new members who can donate their time and attention to the project. Volunteers are encouraged to assist in any way they can, whether that means finding and diagnosing bugs, refining existing source code, or fleshing out whole new features.

This chapter is for those who wish to assist in the continued evolution of Subversion by actually getting their hands dirty with the source code. We will cover some of the software's more intimate details, the kind of technical nitty-gritty that those developing Subversion itself—or writing entirely new tools based on the Subversion libraries—should be aware of. If you don't foresee yourself participating with the software at such a level, feel free to skip this chapter with confidence that your experience as a Subversion user will not be affected.

Layered Library Design

Subversion has a modular design, implemented as a collection of C libraries. Each library has a well-defined purpose and interface, and most modules are said to exist in one of three main layers—the Repository Layer, the Repository Access (RA) Layer, or the Client Layer. We will examine these layers shortly, but first, see our brief inventory of Subversion's libraries in Table 7-1. For the sake of consistency, we will refer to the libraries by their extensionless Unix library names (e.g.: `libsvn_fs`, `libsvn_wc`, `mod_dav_svn`).

Table 7.1. A Brief Inventory of the Subversion Libraries

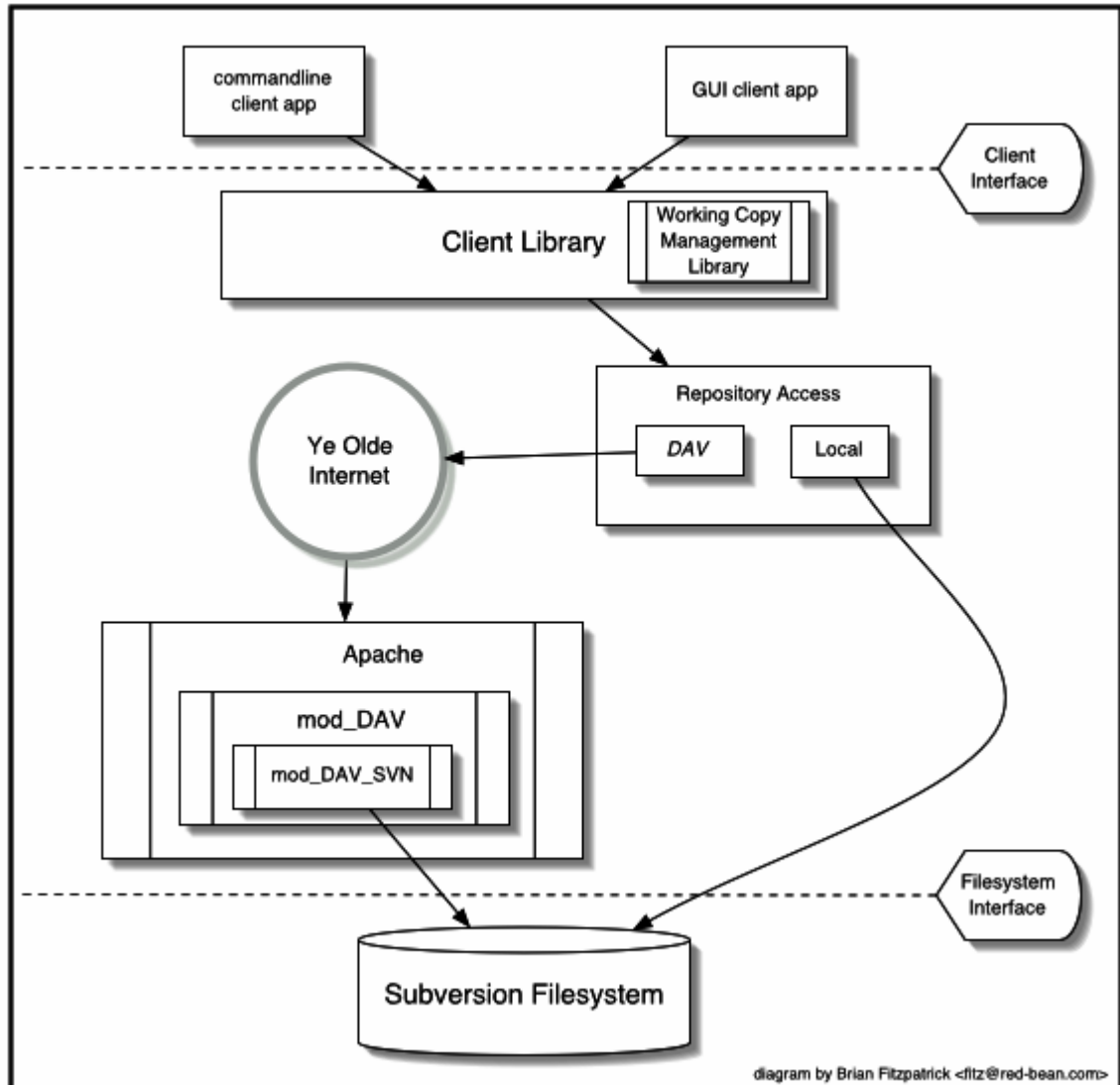
Library	Description
<code>libsvn_client</code>	Primary interface for client programs
<code>libsvn_delta</code>	Tree and text differencing routines

Library	Description
libsvn_fs	The Subversion filesystem library
libsvn_ra	Repository Access commons and module loader
libsvn_ra_dav	The WebDAV Repository Access module
libsvn_ra_local	The local Repository Access module
libsvn_ra_svn	A proprietary protocol Repository Access module
libsvn_repos	Repository interface
libsvn_subr	Miscellaneous helpful subroutines
libsvn_wc	The working copy management library
mod_dav_svn	Apache module for mapping WebDAV operations to Subversion ones

The fact that the word "miscellaneous" only appears once in Table 7-1 is a good sign. The Subversion development team is serious about making sure that functionality lives in the right layer and libraries. Perhaps the greatest advantage of the modular design is its lack of complexity from a developer's point of view. As a developer, you can quickly formulate that kind of "big picture" that allows you to pinpoint the location of certain pieces of functionality with relative ease.

And what could better help a developer gain a "big picture" perspective than a big picture? To help you understand how the Subversion libraries fit together, see Figure 7-1, a diagram of Subversion's layers. Program flow begins at the top of the diagram (initiated by the user) and flows "downward".

Figure 7.1. Subversion's "Big Picture"



Another benefit of modularity is the ability to replace a given module with a whole new library that implements the same API without affecting the rest of the code base. In some sense, this happens within Subversion already. The `libsvn_ra_dav`, `libsvn_ra_local`, and `libsvn_ra_svn` all implement the same interface. And all three communicate with the Repository Layer— `libsvn_ra_dav` and `libsvn_ra_svn` do so across a network, and `libsvn_ra_local` connects to it directly.

The client itself also highlights modularity in the Subversion design. While Subversion currently comes with only a command-line client program, there are already a few other programs being developed by third parties to act as GUIs for Subversion. Again, these GUIs use the same APIs that the stock command-line client does. Subversion's `libsvn_client` library is the one-stop shop for most of the functionality necessary for designing a working Subversion client (see [the section called “Client Layer”](#)).

Repository Layer

When referring to Subversion's Repository Layer, we're generally talking about two libraries—the repository library, and the filesystem library. These libraries provide the storage and reporting mechanisms for the various revisions of your version-controlled data. This layer is connected to the Client Layer via the Repository Access Layer, and is, from the perspective of the Subversion user, the stuff at the "other end of the line."

The Subversion Filesystem is accessed via the `libsvn_fs` API, and is not a kernel-level filesystem that one would install in an operating system (like the Linux `ext2` or `NTFS`), but a virtual filesystem. Rather than storing "files" and "directories" as real files and directories (as in, the kind you can navigate through using your favorite shell program), it uses a database system for its back-end storage mechanism. Currently, the database system in use is Berkeley DB.^[28] However, there has been considerable interest by the development community in giving future releases of Subversion the ability to use other back-end database systems, perhaps through a mechanism such as Open Database Connectivity (ODBC).

The filesystem API exported by `libsvn_fs` contains the kinds of functionality you would expect from any other filesystem API: you can create and remove files and directories, copy and move them around, modify file contents, and so on. It also has features that are not quite as common, such as the ability to add, modify, and remove metadata ("properties") on each file or directory. Furthermore, the Subversion Filesystem is a versioning filesystem, which means that as you make changes to your directory tree, Subversion remembers what your tree looked like before those changes. And before the previous changes. And the previous ones. And so on, all the way back through versioning time to (and just beyond) the moment you first started adding things to the filesystem.

All the modifications you make to your tree are done within the context of a Subversion transaction. The following is a simplified general routine for modifying your filesystem:

1. Begin a Subversion transaction.
2. Make your changes (adds, deletes, property modifications, etc.).
3. Commit your transaction.

Once you have committed your transaction, your filesystem modifications are permanently stored as historical artifacts. Each of these cycles generates a single new revision of your tree, and each revision is forever accessible as an immutable snapshot of "the way things were."

The Transaction Distraction

The notion of a Subversion transaction, especially given its close proximity the database code in `libsvn_fs`, can become easily confused with the transaction support provided by the underlying database itself. Both types of transaction exist to provide atomicity and isolation. In other words, transactions give you the ability to perform a set of actions in an "all or nothing" fashion—either all the actions in the set complete with success, or they all get treated as if *none* of them ever happened—and in a way that does not interfere with other processes acting on the data.

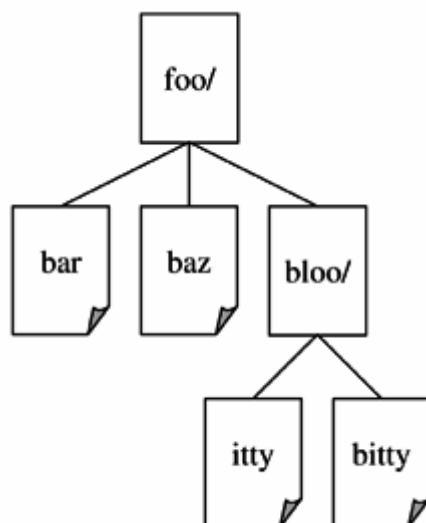
Database transactions generally encompass small operations related specifically to the modification of data in the database itself (such as changing the contents of a table row). Subversion transactions are larger in scope, encompassing higher-level operations like making modifications to a set of files and directories which are intended to be stored as the next revision of the filesystem tree. If that isn't confusing enough, consider this: Subversion uses a database transaction during the creation of a Subversion transaction (so that if the creation of Subversion transaction fails, the database will look as if we had never attempted that creation in the first place)!

Fortunately for users of the filesystem API, the transaction support provided by the database system itself is hidden almost entirely from view (as should be expected from a properly modularized library scheme). It is only when you start digging into the implementation of the filesystem itself that such things become visible (or interesting).

Most of the functionality provided by the filesystem interface comes as an action that occurs on a filesystem path. That is, from outside of the filesystem, the primary mechanism for describing and accessing the individual revisions of files and directories comes through the use of path strings like `/foo/bar`, just as if you were addressing files and directories through your favorite shell program. You add new files and directories by passing their paths-to-be to the right API functions. You query for information about them by the same mechanism.

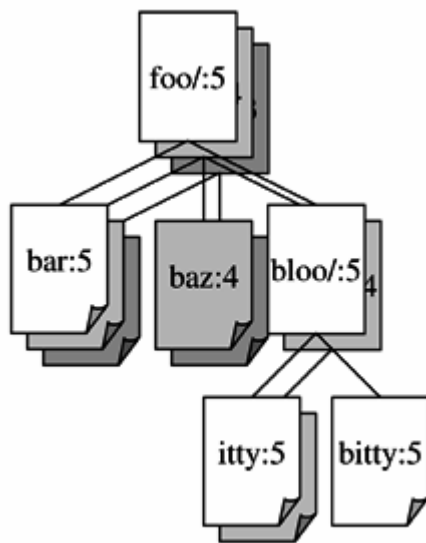
Unlike most filesystems, though, a path alone is not enough information to identify a file or directory in Subversion. Think of a directory tree as a two-dimensional system, where a node's siblings represent a sort of left-and-right motion, and descending into subdirectories a downward motion. Figure 7-2 shows a typical representation of a tree as exactly that.

Figure 7.2. Files and Directories in Two Dimensions



Of course, the Subversion filesystem has a nifty third dimension that most filesystems do not have—Time! ^[29] In the filesystem interface, nearly every function that has a `path` argument also expects a `root` argument. This `svn_fs_root_t` argument describes either a revision or a Subversion transaction (which is usually just a revision-to-be), and provides that third-dimensional context needed to understand the difference between `/foo/bar` in revision 32, and the same path as it exists in revision 98. Figure 7-3 shows revision history as an added dimension to the Subversion filesystem universe.

Figure 7.3. Revisioning Time—the Third Dimension!



As we mentioned earlier, the `libsvn_fs` API looks and feels like any other filesystem, except that it has this wonderful versioning capability. It was designed to be usable by any program interested in a versioning filesystem. Not coincidentally, Subversion itself is interested in that functionality. But while the filesystem API should be sufficient for basic file and directory versioning support, Subversion wants more—and that is where `libsvn_repos` comes in.

The Subversion repository library (`libsvn_repos`) is basically a wrapper library around the filesystem functionality. This library is responsible for creating the repository layout, making sure that the underlying filesystem is initialized, and so on. `Libsvn_repos` also implements a set of hooks—scripts that are executed by the repository code when certain actions take place. These scripts are useful for notification, authorization, or whatever purposes the repository administrator desires. This type of functionality, and other utility provided by the repository library, is not strictly related to implementing a versioning filesystem, which is why it was placed into its own library.

Developers who wish to use the `libsvn_repos` API will find that it is not a complete wrapper around the filesystem interface. That is, only certain major events in the general cycle of filesystem activity are wrapped by the repository interface. Some of these include the creation and commit of Subversion transactions, and the modification of revision properties. These particular events are wrapped by the

repository layer because they have hooks associated with them. In the future, other events may be wrapped by the repository API. All of the remaining filesystem interaction will continue to occur directly with `libsvn_fs` API, though.

For example, here is a code segment that illustrates the use of both the repository and filesystem interfaces to create a new revision of the filesystem in which a directory is added. Note that in this example (and all others throughout this book), the `SVN_ERR` macro simply checks for a non-successful error return from the function it wraps, and returns that error if it exists.

Example 7.1. Using the Repository Layer

```
/* 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. */
static svn_error_t *
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. */
    SVN_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. */
    SVN_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. */
    SVN_ERR (svn_fs_begin_txn (&txn, fs, youngest_rev, pool));

    /* Now that we have started a new Subversion transaction, get a
       root object that represents that transaction. */
    SVN_ERR (svn_fs_txn_root (&txn_root, txn, pool));

    /* Create our new directory under the transaction root, at the path
       NEW_DIRECTORY. */
```



```

SVN_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);

if (! err)
{
    /* No error?  Excellent!  Print a brief report of our success.
    */
    printf ("Directory '%s' was successfully added as new revision
"
           "'%' SVN_REVNUM_T_FMT "'.\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);
}

/* Return the result of the attempted commit to our caller.  */
return err;
}

```

In the previous code segment, calls were made to both the repository and filesystem interfaces. We could just as easily have committed the transaction using `svn_fs_commit_txn`. But the filesystem API knows nothing about the repository library's hook mechanism. If you want your Subversion repository to automatically perform some set of non-Subversion tasks every time you commit a transaction (like, for example, sending an email that describes all the changes made in that transaction to your developer mailing list), you need to use the libsvn_repos-wrapped version of that function—`svn_repos_fs_commit_txn`. This function will actually first run the "pre-commit" hook script if one exists, then commit the transaction, and finally will run a "post-commit" hook script. The hooks provide a special kind of reporting mechanism that does not really belong in the core filesystem library itself. (For more information regarding Subversion's repository hooks, see [the section called “Hook scripts”](#).)

The hook mechanism requirement is but one of the reasons for the abstraction of a separate repository library from the rest of the filesystem code. The libsvn_repos API provides several other important utilities to Subversion. These include the abilities to:

1. create, open, destroy, and perform recovery steps on a Subversion repository and the filesystem included in that repository.
2. describe the differences between two filesystem trees.
3. query for the commit log messages associated with all (or some) of the revisions in which a set of files was modified in the filesystem.
4. generate a human-readable "dump" of the filesystem, a complete representation of the revisions in the filesystem.
5. parse that dump format, loading the dumped revisions into a different Subversion repository.

As Subversion continues to evolve, the repository library will grow with the filesystem library to offer increased functionality and configurable option support.

Repository Access Layer

If the Subversion Repository Layer is at "the other end of the line", the Repository Access Layer is the line itself. Charged with marshalling data between the client libraries and the repository, this layer includes the libsvn_ra module loader library, the RA modules themselves (which currently includes libsvn_ra_dav, libsvn_ra_local, and libsvn_ra_svn), and any additional libraries needed by one or more of those RA modules, such as the mod_dav_svn Apache module with which libsvn_ra_dav communicates or libsvn_ra_svn's server, **svnserve**.

Since Subversion uses URLs to identify its repository resources, the protocol portion of the URL schema (usually `file:`, `http:`, `https:`, or `svn:`) is used to determine which RA module will handle the communications. Each module registers a list of the protocols it knows how to "speak" so that the RA loader can, at runtime, determine which module to use for the task at hand. You can determine which RA modules are available to the Subversion command-line client, and what protocols they claim to support, by running **svn --version**:

```
$ svn --version
svn, version 0.25.0 (dev build)
  compiled Jul 18 2003, 16:25:59
```

Copyright (C) 2000-2003 CollabNet.
Subversion is open source software, see <http://subversion.tigris.org/>

The following repository access (RA) modules are available:

```
* ra_dav : Module for accessing a repository via WebDAV (DeltaV)
protocol.
  - handles 'http' schema
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' schema
* ra_svn : Module for accessing a repository using the svn network
protocol.
  - handles 'svn' schema
```

RA-DAV (Repository Access Using HTTP/DAV)

The `libsvn_ra_dav` library is designed for use by clients that are being run on different machines than the servers with which they communicating, specifically machines reached using URLs that contain the `http:` or `https:` protocol portions. To understand how this module works, we should first mention a couple of other key components in this particular configuration of the Repository Access Layer—the powerful Apache HTTP Server, and the Neon HTTP/WebDAV client library.

Subversion's primary network server is the Apache HTTP Server. Apache is a time-tested, extensible open-source server process that is ready for serious use. It can sustain a high network load and runs on many platforms. The Apache server supports a number of different standard authentication protocols, and can be extended through the use of modules to support many others. It also supports optimizations like network pipelining and caching. By using Apache as a server, Subversion gets all of these features for free. And since most firewalls already allow HTTP traffic to pass through, sysadmins typically don't even have to change their firewall configurations to allow Subversion to work.

Subversion uses HTTP and WebDAV (with DeltaV) to communicate with an Apache server. You can read more about this in the WebDAV section of this chapter, but in short, WebDAV and DeltaV are extensions to the standard HTTP 1.1 protocol that enable sharing and versioning of files over the web. Apache 2.0 comes with `mod_dav`, an Apache module that understands the DAV extensions to HTTP. Subversion itself supplies `mod_dav_svn`, though, which is another Apache module that works in conjunction with (really, as a back-end to) `mod_dav` to provide Subversion's specific implementations of WebDAV and DeltaV.

When communicating with a repository over HTTP, the RA loader library chooses `libsvn_ra_dav` as the proper access module. The Subversion client makes calls into the generic RA interface, and `libsvn_ra_dav` maps those calls (which embody rather large-scale Subversion actions) to a set of HTTP/WebDAV requests. Using the Neon library, `libsvn_ra_dav` transmits those requests to the Apache server. Apache receives these requests (exactly as it does generic HTTP requests that your web browser might make), notices that the requests are directed at a URL that is configured as a DAV location (using the `Location` directive in `httpd.conf`), and hands the request off to its own `mod_dav` module. When properly configured, `mod_dav` knows to use Subversion's `mod_dav_svn` for any filesystem-related needs, as opposed to the generic `mod_dav_fs` that comes with Apache. So ultimately, the client is communicating with `mod_dav_svn`, which binds directly to the Subversion Repository Layer.

That was a simplified description of the actual exchanges taking place, though. For example, the Subversion repository might be protected by Apache's authorization directives. This could result in initial attempts to communicate with the repository being rejected by Apache on authorization grounds. At this point, `libsvn_ra_dav` gets back the notice from Apache that insufficient identification was supplied, and calls back into the Client Layer to get some updated authentication data. If the data is supplied correctly, and the user has the permissions that Apache seeks, `libsvn_ra_dav`'s next automatic attempt at performing the original operation will be granted, and all will be well. If sufficient authentication information cannot be supplied, the request will ultimately fail, and the client will report the failure to the user.

By using Neon and Apache, Subversion gets free functionality in several other complex areas, too. For example, if Neon finds the OpenSSL libraries, it allows the Subversion client to attempt to use SSL-encrypted communications with the Apache server (whose own `mod_ssl` can "speak the language"). Also, both Neon itself and Apache's `mod_deflate` can understand the "deflate" algorithm (the same used by the PKZIP and gzip programs), so requests can be sent in smaller, compressed chunks across the wire. Other complex features that Subversion hopes to support in the future include the ability to automatically handle server-specified redirects (for example, when a repository has been moved to a new canonical URL) and taking advantage of HTTP pipelining.

RA-SVN (Proprietary Protocol Repository Access)

In addition to the standard HTTP/WebDAV protocol, Subversion also provides an RA implementation that uses a proprietary protocol. The `libsvn_ra_svn` module implements its own network socket connectivity, and communicates with a stand-alone server—the `svnserve` program—on the machine that hosts the repository. Clients access the repository using the `svn://` schema.

This RA implementation lacks most of the advantages of Apache mentioned in the previous section; however, it may be appealing to some sysadmins nonetheless. It is dramatically easier to configure and run; setting up an `svnserve` process is nearly instantaneous. It is also much smaller (in terms of lines of code) than Apache, making it much easier to audit, for security reasons or otherwise. Furthermore, some sysadmins may already have an SSH security infrastructure in place, and want Subversion to use it. Clients using `ra_svn` can easily tunnel the protocol over SSH.

RA-Local (Direct Repository Access)

Not all communications with a Subversion repository require a powerhouse server process and a network layer. For users who simply wish to access the repositories on their local disk, they may do so using `file:` URLs and the functionality provided by `libsvn_ra_local`. This RA module binds directly with the repository and filesystem libraries, so no network communication is required at all.

Subversion requires the server name included as part of the `file:` URL be either `localhost` or empty, and that there be no port specification. In other words, your URLs should look like either `file://localhost/path/to/repos` or `file:///path/to/repos`.

Also, be aware that Subversion's `file:` URLs cannot be used in a regular web browser the way typical `file:` URLs can. When you attempt to view a `file:` URL in a regular web browser, it reads and displays the contents of the file at that location by examining the filesystem directly. However, Subversion's resources exist in a virtual filesystem (see [the section called “Repository Layer”](#)), and your browser will not understand how to read that filesystem.

Your RA Library Here

For those who wish to access a Subversion repository using still another protocol, that is precisely why the Repository Access Layer is modularized! Developers can simply write a new library that implements the RA interface on one side and communicates with the repository on the other. Your new library can use existing network protocols, or you can invent your own. You could use inter-process communication (IPC) calls, or—let's get crazy, shall we?—you could even implement an email-based protocol. Subversion supplies the APIs; you supply the creativity.

Client Layer

On the client side, the Subversion working copy is where all the action takes place. The bulk of functionality implemented by the client-side libraries exists for the sole purpose of managing working copies—directories full of files and other subdirectories which serve as a sort of local, editable "reflection" of one or more repository locations—and propagating changes to and from the Repository Access layer.

Subversion's working copy library, `libsvn_wc`, is directly responsible for managing the data in the working copies. To accomplish this, the library stores administrative information about each working copy directory within a special subdirectory. This subdirectory, named `.svn` is present in each working copy directory and contains various other files and directories which record state and provide a private workspace for administrative action. For those familiar with CVS, this `.svn` subdirectory is similar in purpose to the `CVS` administrative directories found in CVS working copies. For more information about the `.svn` administrative area, see [the section called “Inside the Working Copy Administration Area”](#) in this chapter.

The Subversion client library, `libsvn_client`, has the broadest responsibility; its job is to mingle the functionality of the working copy library with that of the Repository Access Layer, and then to provide the highest-level API to any application that wishes to perform general revision control actions. For example, the function `svn_client_checkout` takes a URL as an argument. It passes this URL to the RA layer and opens an authenticated session with a particular repository. It then asks the repository for a certain tree, and sends this tree into the working copy library, which then writes a full working copy to disk (`.svn` directories and all).

The client library is designed to be used by any application. While the Subversion source code includes a standard command-line client, it should be very easy to write any number of GUI clients on top of the client library. New GUIs (or any new client, really) for Subversion need not be clunky wrappers around the included command-line client—they have full access via the `libsvn_client` API to same functionality, data, and callback mechanisms that the command-line client uses.

Binding Directly—A Word About Correctness

Why should your GUI program bind directly with a `libsvn_client` instead of acting as a wrapper around a command-line program? Besides simply being more efficient, this can address potential correctness issues as well. A command-line program (like the one supplied with Subversion) that binds to the client library needs to effectively translate feedback and requested data bits from C types to some form of human-

readable output. This type of translation can be lossy. That is, the program may not display all of the information harvested from the API, or may combine bits of information for compact representation.

If you wrap such a command-line program with yet another program, the second program has access only to already-interpreted (and as we mentioned, likely incomplete) information, which it must *again* translate into *its* representation format. With each layer of wrapping, the integrity of the original data is potentially tainted more and more, much like the result of making a copy of a copy (of a copy ...) of a favorite audio or video cassette.

Using the APIs

Developing applications against the Subversion library APIs is fairly straightforward. All of the public header files live in the `subversion/include` directory of the source tree. These headers are copied into your system locations when you build and install Subversion itself from source. These headers represent the entirety of the functions and types meant to be accessible by users of the Subversion libraries.

The first thing you might notice is that Subversion's datatypes and functions are namespace protected. Every public Subversion symbol name begins with `"svn_"`, followed by a short code for the library in which the symbol is defined (such as `"wc"`, `"client"`, `"fs"`, etc.), followed by a single underscore (`"_"`) and then the rest of the symbol name. Semi-public functions (used among source files of a given library but not by code outside that library, and found inside the library directories themselves) differ from this naming scheme in that instead of a single underscore after the library code, they use a double underscore (`"__"`). Functions that are private to a given source file have no special prefixing, and are declared `static`. Of course, a compiler isn't interested in these naming conventions, but they definitely help to clarify the scope of a given function or datatype.

The Apache Portable Runtime Library

Along with Subversion's own datatype, you will see many references to datatypes that begin with `"apr_"`—symbols from the Apache Portable Runtime (APR) library. APR is Apache's portability library, originally carved out of its server code as an attempt to separate the OS-specific bits from the OS-independent portions of the code. The result was a library that provides a generic API for performing operations that differ mildly—or wildly—from OS to OS. While Apache HTTP Server was obviously the first user of the APR library, the Subversion developers immediately recognized the value of using APR as well. This means that there are practically no OS-specific code portions in Subversion itself. Also, it means that the Subversion client compiles and runs anywhere that the server does. Currently this list includes all flavors of Unix, Win32, BeOS, OS/2, and Mac OS X.

[In addition to providing consistent implementations of system calls that differ across operating systems,](#)^[30] APR gives Subversion immediate access to many custom datatypes, such as dynamic arrays and hash tables. Subversion uses these types

extensively throughout the codebase. But perhaps the most pervasive APR datatype, found in nearly every Subversion API prototype, is the `apr_pool_t`—the APR memory pool. [Subversion uses pools internally for all its memory allocation needs \(unless an external library requires a different memory management schema for data passed through its API\)](#),^[31] and while a person coding against the Subversion APIs is not required to do the same, they are required to provide pools to the API functions that need them. This means that users of the Subversion API must also link against APR, must call `apr_initialize()` to initialize the APR subsystem, and then must acquire a pool for use with Subversion API calls. See [the section called “Programming with Memory Pools”](#) for more information.

URL and Path Requirements

With remote version control operation as the whole point of Subversion's existence, it makes sense that some attention has been paid to internationalization (i18n) support. After all, while "remote" might mean "across the office", it could just as well mean "across the globe." To facilitate this, all of Subversion's public interfaces that accept path arguments expect those paths to be canonicalized, and encoded in UTF-8. This means, for example, that any new client binary that drives the `libsvn_client` interface needs to first convert paths from the locale-specific encoding to UTF-8 before passing those paths to the Subversion libraries, and then re-convert any resultant output paths from Subversion back into the locale's encoding before using those paths for non-Subversion purposes. Fortunately, Subversion provides a suite of functions (see `subversion/include/svn_utf.h`) that can be used by any program to do these conversions.

Also, Subversion APIs require all URL parameters to be properly URI-encoded. So, instead of passing `file:///home/username/My File.txt` as the URL of a file named `My File.txt`, you need to pass `file:///home/username/My%20File.txt`. Again, Subversion supplies helper functions that your application can use—`svn_path_uri_encode` and `svn_path_uri_decode`, for URI encoding and decoding, respectively.

Using Languages Other than C and C++

If you are interested in using the Subversion libraries in conjunction with something other than a C program—say a Python script or Java application—Subversion has some initial support for this via the Simplified Wrapper and Interface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig` and are slowly maturing into a usable state. These bindings allow you to call Subversion API functions indirectly, using wrappers that translate the datatypes native to your scripting language into the datatypes needed by Subversion's C libraries.

There is an obvious benefit to accessing the Subversion APIs via a language binding—simplicity. Generally speaking, languages such as Python and Perl are much more flexible and easy to use than C or C++. The sort of high-level datatypes and context-driven type checking provided by these languages are often better at handling information that comes from users. As you know, only a human can botch up the

input to a program as well as they do, and the scripting-type language simply handle that misinformation more gracefully. Of course, often that flexibility comes at the cost of performance. That is why using a tightly-optimized, C-based interface and library suite, combined with a powerful, flexible binding language is so appealing.

Let's look at an example that uses Subversion's Python SWIG bindings. Our example will do the same thing as our last example. Note the difference in size and complexity of the function this time!

Example 7.2. Using the Repository Layer with Python

```
from svn import fs
import os.path

def crawl_filesystem_dir (root, directory, pool):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and
    return
    a list of all the paths at or below DIRECTORY.  Use POOL for all
    allocations."""

    # Get the directory entries for DIRECTORY.
    entries = fs.dir_entries(root, directory, pool)

    # Initialize our returned list with the directory path itself.
    paths = [directory]

    # Loop over the entries
    names = entries.keys()
    for name in names:
        # Calculate the entry's full path.
        full_path = os.path.join(basepath, 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 fs.is_dir(fsroot, full_path, pool):
            subpaths = crawl_filesystem_dir(root, full_path, pool)
            paths.extend(subpaths)

        # Else, it is a file, so add the entry's full path to the FILES
    list.
        else:
            paths.append(full_path)

    return paths
```

An implementation in C of the previous example would stretch on quite a bit longer. The same routine in C would need to pay close attention to memory usage, and need to use custom datatypes for representing the hash of entries and the list of paths. Python has hashes and lists (called "dictionaries" and "sequences", respectively) as built-in datatypes, and provides a wonderful selection of methods for operating on those types. And since Python uses reference counting and garbage collection, users of the language don't have to bother themselves with allocating and freeing memory.

In the previous section of this chapter, we mentioned the `libsvn_client` interface, and how it exists for the sole purpose of simplifying the process of writing a Subversion client. The following is a brief example of how that library can be accessed via the SWIG bindings. In just a few lines of Python, you can check out a fully functional Subversion working copy!

Example 7.3. A Simple Script to Check Out a Working Copy.

```
#!/usr/bin/env python
import sys
from svn import util, _util, _client

def usage():
    print "Usage: " + sys.argv[0] + " URL PATH\n"
    sys.exit(0)

def run(url, path):
    # Initialize APR and get a POOL.
    _util.apr_initialize()
    pool = util.svn_pool_create(None)

    # Checkout the HEAD of URL into PATH (silently)
    _client.svn_client_checkout(None, None, url, path, -1, 1, None,
                                pool)

    # Cleanup our POOL, and shut down APR.
    util.svn_pool_destroy(pool)
    _util.apr_terminate()

if __name__ == '__main__':
    if len(sys.argv) != 3:
        usage()
    run(sys.argv[1], sys.argv[2])
```

Currently, it is Subversion's Python bindings that are the most complete. Some attention is also being given to the Java bindings. Once you have the SWIG interface files properly configured, generation of the specific wrappers for all the supported SWIG languages (which currently includes versions of C#, Guile, Java, Mzscheme, OCaml, Perl, PHP, Python, Ruby, and Tcl) should theoretically be trivial. Still, some extra programming is required to compensate for complex APIs that SWIG needs some help generalizing. For more information on SWIG itself, see the project's website at <http://www.swig.org>.

Inside the Working Copy Administration Area

As we mentioned earlier, each directory of a Subversion working copy contains a special subdirectory called `.svn` which houses administrative data about that working copy directory. Subversion uses the information in `.svn` to keep track of things like:

- Which repository location(s) are represented by the files and subdirectories in the working copy directory.
- What revision of each of those files and directories are currently present in the working copy.

- Any user-defined properties that might be attached to those files and directories.
- Pristine (un-edited) copies of the working copy files.

While there are several other bits of data stored in the `.svn` directory, we will examine only a couple of the most important items.

The Entries File

Perhaps the single most important file in the `.svn` directory is the `entries` file. The `entries` file is an XML document which contains the bulk of the administrative information about a versioned resource in a working copy directory. It is this one file which tracks the repository URLs, pristine revision, file checksums, pristine text and property timestamps, scheduling and conflict state information, last-known commit information (author, revision, timestamp), local copy history—practically everything that a Subversion client is interested in knowing about a versioned (or to-be-versioned) resource!

Comparing the Administrative Areas of Subversion and CVS

A glance inside the typical `.svn` directory turns up a bit more than what CVS maintains in its `CVS` administrative directories. The `entries` file contains XML which describes the current state of the working copy directory, and basically serves the purposes of CVS's `Entries`, `Root`, and `Repository` files combined.

The following is an example of an actual `entries` file:

Example 7.4. Contents of a Typical `.svn/entries` File

```
<?xml version="1.0" encoding="utf-8"?>
<wc-entries
  xmlns="svn:">
  <entry
    committed-rev="1"
    name="svn:this_dir"
    committed-date="2002-09-24T17:12:44.064475Z"
    url="http://svn.red-bean.com/tests/.greek-repo/A/D"
    kind="dir"
    revision="1"/>
  <entry
    committed-rev="1"
    name="gamma"
    text-time="2002-09-26T21:09:02.000000Z"
    committed-date="2002-09-24T17:12:44.064475Z"
    checksum="QSE4vWd9ZM0cMvr7/+YkXQ=="
    kind="file"
    prop-time="2002-09-26T21:09:02.000000Z"/>
  <entry
    name="zeta"
    kind="file"
    schedule="add"
```

```

    revision="0"/>
<entry
  url="http://svn.red-bean.com/tests/.greek-repo/A/B/delta"
  name="delta"
  kind="file"
  schedule="add"
  revision="0"/>
<entry
  name="G"
  kind="dir"/>
<entry
  name="H"
  kind="dir"
  schedule="delete"/>
</wc-entries>

```

As you can see, the entries file is essentially a list of entries. Each entry tag represents one of three things: the working copy directory itself (noted by having its name attribute set to "svn:this-dir"), a file in that working copy directory (noted by having its kind attribute set to "file"), or a subdirectory in that working copy (kind here is set to "dir"). The files and subdirectories whose entries are stored in this file are either already under version control, or (as in the case of the file named zeta above) are scheduled to be added to version control when the user next commits this working copy directory's changes. Each entry has a unique name, and each entry has a node kind.

Developers should be aware of some special rules that Subversion uses when reading and writing its entries files. While each entry has a revision and URL associated with it, note that not every entry tag in the sample file has explicit revision or url attributes attached to it. [Subversion allows entries to not explicitly store those two attributes when their values are the same as \(in the](#) [revision](#) [case\)](#) or trivially calculable from ^[32] (in the url case) the data stored in the "svn:this-dir" entry. Note also that for subdirectory entries, Subversion stores only the crucial attributes—name, kind, url, revision, and schedule. In an effort to reduce duplicated information, Subversion dictates that the method for determining the full set of information about a subdirectory is to traverse down into that subdirectory, and read the "svn:this-dir" entry from its own .svn/entries file. However, a reference to the subdirectory is kept in its parent's entries file, with enough information to permit basic versioning operations in the event that the subdirectory itself is actually missing from disk.

Pristine Copies and Property Files

As mentioned before, the .svn directory also holds the pristine "text-base" versions of files. Those can be found in .svn/text-base. The benefits of these pristine copies are multiple—network-free checks for local modifications and "diff" reporting, network-free reversion of modified or missing files, smaller transmission of changes to the server—but comes at the cost of having each versioned file stored at least twice on disk. These days, this seems to be a negligible penalty for most files. However, the situation gets uglier as the size of your versioned files grows. Some attention is being given to making the presence of the "text-base" an option. Ironically though, it is as your versioned files' sizes get larger that the existence of the "text-base" becomes

more crucial—who wants to transmit a huge file across a network just because they want to commit a tiny change to it?

Similar in purpose to the "text-base" files are the property files and their pristine "prop-base" copies, located in `.svn/props` and `.svn/prop-base` respectively. Since directories can have properties, too, there are also `.svn/dir-props` and `.svn/dir-prop-base` files. Each of these property files ("working" and "base" versions) uses a simple "hash-on-disk" file format for storing the property names and values.

WebDAV

WebDAV ("Web-based Distributed Authoring and Versioning") is an extension of the standard HTTP protocol designed to make the web into a read/write medium, instead of the basically read-only medium that exists today. The theory is that directories and files can be shared—as both readable and writable objects—over the web. RFCs 2518 and 3253 describe the WebDAV/DeltaV extensions to HTTP, and are available (along with a lot of other useful information) at <http://www.webdav.org/>.

A number of operating system file browsers are already able to mount networked directories using WebDAV. On Win32, the Windows Explorer can browse what it calls "WebFolders" (which are just WebDAV-ready network locations) as if they were regular shared folders. Mac OS X also has this capability, as do the Nautilus and Konqueror browsers (under GNOME and KDE, respectively).

How does all of this apply to Subversion? The `mod_dav_svn` Apache module uses HTTP, extended by WebDAV and DeltaV, as its primary network protocol. Rather than implementing a new proprietary protocol, the original Subversion designers decided to simply map the versioning concepts and actions used by Subversion onto the concepts exposed by RFCs 2518 and 3253.^[33]

For a more thorough discussion of WebDAV, how it works, and how Subversion uses it, see [Appendix D, WebDAV 与自动版本](#). Among other things, that appendix discusses the degree to which Subversion adheres to the generic WebDAV specification, and how that affects interoperability with generic WebDAV clients.

Programming with Memory Pools

Almost every developer who has used the C programming language has at some point sighed at the daunting task of managing memory usage. Allocating enough memory to use, keeping track of those allocations, freeing the memory when you no longer need it—these tasks can be quite complex. And of course, failure to do those things properly can result in a program that crashes itself, or worse, crashes the computer. Fortunately, the APR library that Subversion depends on for portability provides the `apr_pool_t` type, which represents a "pool" of memory.

A memory pool is an abstract representation of a chunk of memory allocated for use by a program. Rather than requesting memory directly from the OS using the standard `malloc()` and friends, programs that link against APR can simply request that a pool of memory be created (using the `apr_pool_create()` function). APR will allocate a

moderately sized chunk of memory from the OS, and that memory will be instantly available for use by the program. Any time the program needs some of the pool memory, it uses one of the APR pool API functions, like `apr_palloc()`, which returns a generic memory location from the pool. The program can keep requesting bits and pieces of memory from the pool, and APR will keep granting the requests. Pools will automatically grow in size to accommodate programs that request more memory than the original pool contained, until of course there is no more memory available on the system.

Now, if this were the end of the pool story, it would hardly have merited special attention. Fortunately, that's not the case. Pools can not only be created; they can also be cleared and destroyed, using `apr_pool_clear()` and `apr_pool_destroy()` respectively. This gives developers the flexibility to allocate several—or several thousand—things from the pool, and then clean up all of that memory with a single function call! Further, pools have hierarchy. You can make "subpools" of any previously created pool. When you clear a pool, all of its subpools are destroyed; if you destroy a pool, it and its subpools are destroyed.

Before we go further, developers should be aware that they probably will not find many calls to the APR pool functions we just mentioned in the Subversion source code. APR pools offer some extensibility mechanisms, like the ability to have custom "user data" attached to the pool, and mechanisms for registering cleanup functions that get called when the pool is destroyed. Subversion makes use of these extensions in a somewhat non-trivial way. So, Subversion supplies (and most of its code uses) the wrapper functions `svn_pool_create()`, `svn_pool_clear()`, and `svn_pool_destroy()`.

While pools are helpful for basic memory management, the pool construct really shines in looping and recursive scenarios. Since loops are often unbounded in their iterations, and recursions in their depth, memory consumption in these areas of the code can become unpredictable. Fortunately, using nested memory pools can be a great way to easily manage these potentially hairy situations. The following example demonstrates the basic use of nested pools in a situation that is fairly common—recursively crawling a directory tree, doing some task to each thing in the tree.

Example 7.5. Effective Pool Usage

```
/* Recursively crawl over DIRECTORY, adding the paths of all its file
   children to the FILES array, and doing some task to each path
   encountered. Use POOL for the all temporary allocations, and
   store
   the hash paths in the same pool as the hash itself is allocated in.
*/
static apr_status_t
crawl_dir (apr_array_header_t *files,
          const char *directory,
          apr_pool_t *pool)
{
    apr_pool_t *hash_pool = files->pool; /* array pool */
    apr_pool_t *subpool = svn_pool_create (pool); /* iteration pool */
    apr_dir_t *dir;
    apr_finfo_t finfo;
```

```

apr_status_t apr_err;
apr_int32_t flags = APR_FINFO_TYPE | APR_FINFO_NAME;

apr_err = apr_dir_open (&dir, directory, pool);
if (apr_err)
    return apr_err;

/* Loop over the directory entries, clearing the subpool at the top
of
each iteration. */
for (apr_err = apr_dir_read (&finfo, flags, dir);
    apr_err == APR_SUCCESS;
    apr_err = apr_dir_read (&finfo, flags, dir))
{
    const char *child_path;

    /* Skip entries for "this dir" ('.') and its parent ('..'). */
    if (finfo.filetype == APR_DIR)
    {
        if (finfo.name[0] == '.'
            && (finfo.name[1] == '\0'
                || (finfo.name[1] == '.' && finfo.name[2] == '\0')))
            continue;
    }

    /* Build CHILD_PATH from DIRECTORY and FINFO.name. */
    child_path = svn_path_join (directory, finfo.name, subpool);

    /* Do some task to this encountered path. */
    do_some_task (child_path, subpool);

    /* Handle subdirectories by recursing into them, passing
SUBPOOL
    as the pool for temporary allocations. */
    if (finfo.filetype == APR_DIR)
    {
        apr_err = crawl_dir (files, child_path, subpool);
        if (apr_err)
            return apr_err;
    }

    /* Handle files by adding their paths to the FILES array. */
    else if (finfo.filetype == APR_REG)
    {
        /* Copy the file's path into the FILES array's pool. */
        child_path = apr_pstrdup (hash_pool, child_path);

        /* Add the path to the array. */
        (*((const char **) apr_array_push (files))) = child_path;
    }

    /* Clear the per-iteration SUBPOOL. */
    svn_pool_clear (subpool);
}

/* Check that the loop exited cleanly. */
if (apr_err)
    return apr_err;

/* Yes, it exited cleanly, so close the dir. */
apr_err = apr_dir_close (dir);

```

```

if (apr_err)
    return apr_err;

/* Destroy SUBPOOL. */
svn_pool_destroy (subpool);

return APR_SUCCESS;
}

```

The previous example demonstrates effective pool usage in *both* looping and recursive situations. Each recursion begins by making a subpool of the pool passed to the function. This subpool is used for the looping region, and cleared with each iteration. The result is memory usage is roughly proportional to the depth of the recursion, not to total number of file and directories present as children of the top-level directory. When the first call to this recursive function finally finishes, there is actually very little data stored in the pool that was passed to it. Now imagine the extra complexity that would be present if this function had to `alloc()` and `free()` every single piece of data used!

Pools might not be ideal for every application, but they are extremely useful in Subversion. As a Subversion developer, you'll need to grow comfortable with pools and how to wield them correctly. Memory usage bugs and bloating can be difficult to diagnose and fix regardless of the API, but the pool construct provided by APR has proven a tremendously convenient, time-saving bit of functionality.

Contributing to Subversion

The official source of information about the Subversion project is, of course, the project's website at <http://subversion.tigris.org>. There you can find information about getting access to the source code and participating on the discussion lists. The Subversion community always welcomes new members. If you are interested in participating in this community by contributing changes to the source code, here are some hints on how to get started.

Join the Community

The first step in community participation is to find a way to stay on top of the latest happenings. To do this most effectively, you will want to subscribe to the main developer discussion list ([<dev@subversion.tigris.org>](mailto:dev@subversion.tigris.org)) and commit mail list ([<svn@subversion.tigris.org>](mailto:svn@subversion.tigris.org)). By following these lists even loosely, you will have access to important design discussions, be able to see actual changes to Subversion source code as they occur, and be able to witness peer reviews of those changes and proposed changes. These email based discussion lists are the primary communication media for Subversion development. See the Mailing Lists section of the website for other Subversion-related lists you might be interested in.

But how do you know what needs to be done? It is quite common for a programmer to have the greatest intentions of helping out with the development, yet be unable to find a good starting point. After all, not many folks come to the community having already decided on a particular itch they would like to scratch. But by watching the developer discussion lists, you might see mentions of existing bugs or feature requests fly by

that particularly interest you. Also, a great place to look for outstanding, unclaimed tasks is the Issue Tracking database on the Subversion website. There you will find the current list of known bugs and feature requests. If you want to start with something small, look for issues marked as "bite-sized".

Get the Source Code

To edit the code, you need to have the code. This means you need to check out a working copy from the public Subversion source repository. As straightforward as that might sound, the task can be slightly tricky. Because Subversion's source code is versioned using Subversion itself, you actually need to "bootstrap" by getting a working Subversion client via some other method. The most common methods include downloading the latest binary distribution (if such is available for your platform), or downloading the latest source tarball and building your own Subversion client. If you build from source, make sure to read the INSTALL file in the top level of the source tree for instructions.

After you have a working Subversion client, you are now poised to checkout a working copy of the Subversion source repository from <http://svn.collab.net/repos/svn/trunk>:^[34]

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subversion
A HACKING
A INSTALL
A README
A autogen.sh
A build.conf
...
```

The above command will checkout the bleeding-edge, latest version of the Subversion source code into a subdirectory named `subversion` in your current working directory. Obviously, you can adjust that last argument as you see fit. Regardless of what you call the new working copy directory, though, after this operation completes, you will now have the Subversion source code. Of course, you will still need to fetch a few helper libraries (`apr`, `apr-util`, etc.)—see the `INSTALL` file in the top level of the working copy for details.

Become Familiar with Community Policies

Now that you have a working copy containing the latest Subversion source code, you will most certainly want to take a cruise through the `HACKING` file in that working copy's top-level directory. The `HACKING` file contains general instructions for contributing to Subversion, including how to properly format your source code for consistency with the rest of the codebase, how to describe your proposed changes with an effective change log message, how to test your changes, and so on. Commit privileges on the Subversion source repository are earned—a government by meritocracy.^[35] The `HACKING` file is an invaluable resource when it comes to making sure that your proposed changes earn the praises they deserve without being rejected on technicalities.

Make and Test Your Changes

With the code and community policy understanding in hand, you are ready to make your changes. It is best to try to make smaller but related sets of changes, even tackling larger tasks in stages, instead of making huge, sweeping modifications. Your proposed changes will be easier to understand (and therefore easier to review) if you disturb the fewest lines of code possible to accomplish your task properly. After making each set of proposed changes, your Subversion tree should be in a state in which the software compiles with no warnings.

[Subversion has a fairly thorough](#) ^[36] regression test suite, and your proposed changes are expected to not cause any of those tests to fail. By running **make check** (in Unix) from the top of the source tree, you can sanity-check your changes. The fastest way to get your code contributions rejected (other than failing to supply a good log message) is to submit changes that cause failure in the test suite.

In the best-case scenario, you will have actually added appropriate tests to that test suite which verify that your proposed changes actually work as expected. In fact, sometimes the best contribution a person can make is solely the addition of new tests. You can write regression tests for functionality that currently works in Subversion as a way to protect against future changes that might trigger failure in those areas. Also, you can write new tests that demonstrate known failures. For this purpose, the Subversion test suite allows you to specify that a given test is expected to fail (called an `XFAIL`), and so long as Subversion fails in the way that was expected, a test result of `XFAIL` itself is considered a success. Ultimately, the better the test suite, the less time wasted on diagnosing potentially obscure regression bugs.

Donate Your Changes

After making your modifications to the source code, compose a clear and concise log message to describe those changes and the reasons for them. Then, send an email to the developers list containing your log message and the output of **svn diff** (from the top of your Subversion working copy). If the community members consider your changes acceptable, someone who has commit privileges (permission to make new revisions in the Subversion source repository) will add your changes to the public source code tree. Recall that permission to directly commit changes to the repository is granted on merit—if you demonstrate comprehension of Subversion, programming competency, and a "team spirit", you will likely be awarded that permission.

^[28] The choice of Berkeley DB brought several automatic features that Subversion needed, such as data integrity, atomic writes, recoverability, and hot backups.

^[29] We understand that this may come as a shock to sci-fi fans who have long been under the impression that Time was actually the *fourth* dimension, and we apologize for any emotional trauma induced by our assertion of a different theory.

[30] Subversion uses ANSI system calls and datatypes as much as possible.

[31] Neon and Berkeley DB are examples of such libraries.

[32] That is, the URL for the entry is the same as the concatenation of the parent directory's URL and the entry's name.

[33] As it turns out, Subversion has more recently evolved a proprietary protocol anyway, implemented by the `libsvn_ra_svn` module.

[34] Note that the URL checked out in the example above ends not with `svn`, but with a subdirectory thereof called `trunk`. See our discussion of Subversion's branching and tagging model for the reasoning behind this.

[35] While this may superficially appear as some sort of elitism, this "earn your commit privileges" notion is about efficiency—whether it costs more in time and effort to review and apply someone else's changes that are likely to be safe and useful, versus the potential costs of undoing changes that are dangerous.

[36] You might want to grab some popcorn. "Thorough", in this instance, translates to somewhere around thirty minutes of non-interactive machine churn.

Chapter 8. 完整 Subversion 参考手册

Table of Contents

[Subversion 命令列客户端: svn](#)

[svn 选项](#)

[svn 子命令](#)

[svn add](#)

[svn cat](#)

[svn checkout](#)

[svn cleanup](#)

[svn commit](#)

[svn copy](#)

[svn delete](#)

[svn diff](#)

[svn export](#)

[svn help](#)

[svn import](#)

[svn info](#)

[svn list](#)

[svn log](#)

[svn merge](#)

[svn mkdir](#)

[svn move](#)

[svn propdel](#)

[svn propedit](#)

[svn propget](#)

- [svn proplist](#)
- [svn propset](#)
- [svn resolved](#)
- [svn revert](#)
- [svn status](#)
- [svn switch](#)
- [svn update](#)
- [svnadmin](#)
 - [svnadmin 选项](#)
 - [svnadmin 子命令](#)
 - [svnadmin list-unused-dblogs](#)
 - [svnadmin create](#)
 - [svnadmin dump](#)
 - [svnadmin help](#)
 - [svnadmin load](#)
 - [svnadmin lstxns](#)
 - [svnadmin recover](#)
 - [svnadmin rmtxns](#)
 - [svnadmin setlog](#)
- [svnlook](#)
 - [svnlook 选项](#)
 - [svnlook author](#)
 - [svnlook cat](#)
 - [svnlook changed](#)
 - [svnlook date](#)
 - [svnlook diff](#)
 - [svnlook dirs-changed](#)
 - [svnlook help](#)
 - [svnlook history](#)
 - [svnlook info](#)
 - [svnlook log](#)
 - [svnlook proplist](#)
 - [svnlook tree](#)
 - [svnlook youngest](#)

本章是用来作为 Subversion 的完整参考手册, 里面包含了命令列客户端 (**svn**) 与其所有的子命令, 以及档案库管理程序 (**svnadmin** 与 **svnlook**) 与其子命令.

Subversion 命令列客户端: svn

[要使用命令列客户端](#), 请输入 **svn**, 欲使用的子命令, ^[37] 然后再加上任何你想使用的选项或目标 — 子命令与选项并没有特定的次序. 举个例子, 以下都是 **svn status** 的有效用法:

```
$ svn -v status
$ svn status -v
$ svn status -v myfile
```

在 [Chapter 3, 导览](#) 里, 可以找到大部份客户端命令的范例, 在 [the section called “性质”](#) 里, 可以找到管理性质的命令范例.

svn 选项

虽然 Subversion 的子命令有许多不同的选项, 但是所有的选项都是全面通用的 — 也就是不管使用的子命令为何, 每个选项都保证表示同样的东西. 像是 `--verbose (-v)` 一定是“详细输出”, 不管你与哪个子命令使用.

`--diff-cmd CMD`

指定用以显示档案差异的外部程序. 当 **svn diff** 执行时, 它会使用 Subversion's 内部的差异引擎, 预设输出统一差异格式. 如果你想使用外部的差异程序, 请使用 `--diff-cmd`. 你可以透过 `--extensions` 选项 (本节稍后会提到) 来传递差异程序的选项.

`--diff3-cmd CMD`

指定用以合并档案的外部程序.

`--dry-run`

进行所有命令该作的动作, 但是实际上不进行更动 — 不管是对磁盘, 还是档案库的内容.

`--editor-cmd CMD`

指定用以编辑送交讯息或性质内容的外部程序.

`--encoding ENC`

让 Subversion 知道你的送交讯息是以指定字集编码的. 预设是你的操作系统的原生语系环境设定, 如果你的送交讯息使用不同的编码, 你应该要指定编码方式.

`--extensions (-x) "ARGS"`

指定 Subversion 在产生档案之间差异时, 要传给外部差异程序的一个或多个自变量. 如果你想要传递多个自变量的话, 你必须以引号将它们全都包起来 (举个例子, **svn diff --diff-cmd /usr/bin/diff -x "-b -E"**). 本选项 *只有在* `--diff-cmd` 选项也一并使用时, 才会发生效用.

`--file (-F) FILENAME`

对某个子命令, 使用本选项自变量指定的档案的内容.

`--force`

强迫执行某个特定的命令或是动作. 在正常的使用下, 有些动作会被 **Subversion** 阻止, 但是使用这个选项后, 就等于告诉 **Subversion** “我知道我在作什么, 可能会发生的后果我也知道, 所以让我作吧”. 这个选项就像你在总开关开着的时候, 修理电气线路 — 如果你不知道你在作什么, 你很有可能被电得很惨.

`--force-log`

强制将送给 `--message (-m)` 或 `--file (-F)` 选项的有问题参数视为有效的. 如果这些参数的选项看起来像是给子命令用的, **Subversion** 预设会发出错误讯息. 举个例子, 如果你将一个纳入版本控制的档案路径传给 `--file (-F)` 选项, **Subversion** 会假设你搞错了, 这个路径应该是该项作业的目标, 而你就是忘了指定其它 — 未纳入版本控制 — 档案作为记录讯息的来源. 要坚持你的立场, 藐视这类的错误检查, 请传递 `--force-log` 选项给接受记录讯息的命令.

`--help (-h 或 -?)`

如果与一个或以上的子命令一起使用, 显示每个子命令的内建求助讯息. 如果单独使用, 它会显示通用的客户端求助讯息.

`--notice-ancestry`

在计算差异时, 将其演进历程考虑进去.

`--incremental`

以适合附加的格式来输出讯息.

`--message (-m) MESSAGE`

表示你在这个选项后, 在命令列指定一个送交讯息. 例如:

```
$ svn commit -m "They don't make Sunday."
```

`--new ARG`

将 **ARG** 视为较新的目标.

`--no-auth-cache`

防止在 **Subversion** 的管理目录中暂存认证信息 (例如使用者名称与密码).

`--no-diff-deleted`

防止 **Subversion** 显示已删除档案的差异. 在删除档案后, **svn diff** 的预设行为会将差异显示出来, 但是假装档案还在, 而档案内容已悉数删除.

`--no-ignore`

在列出档案状态时, 显示一般因符合 `svn:ignore` 性质的样式而被忽略的档案. 请参考 [the section called “Config”](#) 以了解详细信息.

`--non-interactive`

如果认证失败, 或是没有足够的凭证, 也不会显示输入凭证的提示字 (也就是使用者名称与密码). 如果你是在自动执行的脚本档中使用 **Subversion** 就很有用, 让 **Subversion** 直接失败, 要比要求输入的提示要更恰当.

`--non-recursive (-N)`

阻止子命令递归至子目录里去. 大多数的子命令预设会使用递归, 但是有些子命令 — 通常是可能会移除, 或是反悔本地修改的 — 不会.

`--old ARG`

将 **ARG** 视为较旧的目标.

`--password PASS`

表示你在命令列提供了认证用的密码 — 不然的话, **Subversion** 会依需要提示你输入密码.

`--quiet (-q)`

要求客户端在进行作业时, 只显示必要的信息.

`--recursive (-R)`

让子命令会递归地处理子目录. 大多数的子命令预设行为皆如此.

`--relocate FROM TO [PATH...]`

与 **svn switch** 子命令一同使用, 用来变更工作复本所参考的档案库位置. 当你的档案库位置变更, 而你已经有一个正在使用的工作复本时, 这个选项就很有用. 请参考 **svn switch** 的范例.

`--revision (-r) REV`

表示你对某一个作业指定一个 (或一个范围的) 修订版. 你可以指定修订版号, 修订版关键词, 或是日期 (要包在大括号里) 作为本选项的自变量. 如果你想要指定一个范围的修订版, 请在中间以冒号隔开. 例如:

```
$ svn log -r 1729
$ svn log -r 1729:HEAD
$ svn log -r 1729:1744
```

```
$ svn log -r {12/04/01}:{2/17/02}
$ svn log -r 1729:{2/17/02}
```

参见 [the section called “修订版关键词”](#) 以取得更多的信息.

`--revprop`

针对一个目录或档案修订版性质, 而不是 **Subversion** 性质. 使用本选项时, 必须一并以 `--revision (-r)` 选项指定修订版. 请参考 [the section called “无版本控制的性质”](#), 以了解更多未纳入版本控制性质的细节.

`--show-updates (-u)`

让客户端显示哪些工作目录的档案已过时. 这不会真的更新你的档案 — 它只会显示如果真的执行 **svn update** 时, 哪些档案会被更新.

`--stop-on-copy`

如果 **Subversion** 的子命令会收集纳入版本控制的历史纪录, 那么这个选项会让它在收集历史纪录的过程中, 停在复制 — 也就是从档案库的一处复制至另一处的地方 — 的位置.

`--strict`

让 **Subversion** 使用严格的语意.

`--targets FILENAME`

让 **Subversion** 自指定的档案取得欲处理的档案列表, 而不是在命令列中列出所有的档案.

`--usernameNAME`

表示你在命令列提供了认证用的使用者名称 — 不然的话, **Subversion** 会依需要提示你输入使用者名称.

`--verbose`

要求客户端在执行任何子命令时, 尽量提供详细的信息. 这可能让 **Subversion** 列出额外的字段, 每个档案的详细信息, 或是更详细的行为信息.

`--version`

显示客户端版本信息. 不只客户端的版本号码, 还包括了所有可用来存取 **Subversion** 档案库的档案库存取模块.

`--xml`

以 XML 格式输出.

svn 子命令

Name

svn add — 新增目录或档案

摘要

svn add PATH...

描述

将目录与档案新增至工作复本中, 并且排定预备加入档案库. 它们会在你下一次送交时, 上载并新增到档案库中. 如果你新增了某个东西, 在送交前又反悔了, 你可以使用 **svn revert** 来取消新增的排程.

替代名称

无

更动

工作复本

选项

```
--targets FILENAME  
--non-recursive (-N)  
--quiet (-q)
```

范例

新增一个档案至工作复本中:

```
$ svn add foo.c  
A      foo.c
```

新增一个目录时, 预设的 **svn add** 行为会递归地进行:


```
$ svn add testdir
A          testdir
A          testdir/a
A          testdir/b
A          testdir/c
A          testdir/d
```

你也可以新增一个目录, 而不包含其内容:

```
$ svn add --non-recursive otherdir
A          otherdir
```

Name

`svn cat` — 输出指定档案或 URL 的内容.

摘要

```
svn cat TARGET...
```

描述

输出指定的档案或 URL 的内容. 如果要列出目录的内容, 请参见 **svn list**.

替代名称

无

更动

无

存取档案库

是

选项

```
--revision (-r) REV
--username USER
--password PASS
```

范例

如果你想要看档案库里的 `readme.txt` 的内容, 而又不想要取出它:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```

Tip

如果你的工作复本已过时的话 (或者已有了本地更动), 而你想看看工作复本里某个档案的 **HEAD** 修订版, 当你指定一个路径给 **svn cat** 时, 它可自动取出 **HEAD** 修订版:

```
$ cat foo.c
This file is in my local working copy
and has changes that I've made.

$ svn cat foo.c
Latest revision fresh from the repository!
```

Name

`svn checkout` — 自档案库取出一个工作复本.

摘要

```
svn checkout URL... [PATH]
```

描述

自档案库取出一个工作复本. 如果省略 `PATH` 的话, `URL` 的主档名 (`basename`) 会被视作目录名称. 如果指定多个 `URL` 的话, 每一个都会取出为 `PATH` 的子目录, 其名称为 `URL` 的主档名.

替代名称

`co`

更动

产生工作复本.

存取档案库

是

选项

```
--revision (-r) REV
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

范例

取出一个工作复本为 'mine' 的目录:

```
$ svn checkout file:///tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2.
$ ls
mine
```

取出两个不同的目录, 变成两个不同的工作复本:

```
$ svn checkout file:///tmp/repos/test file:///tmp/repos/quiz
A  test/a
A  test/b
Checked out revision 2.
A  quiz/l
A  quiz/m
Checked out revision 2.
$ ls
quiz  test
```

取出两个不同的目录, 变成两个不同的工作复本, 但是将它们置于名为 'working-copies' 的目录之内:

```
$ svn checkout file:///tmp/repos/test file:///tmp/repos/quiz
working-copies
A  working-copies/test/a
A  working-copies/test/b
Checked out revision 2.
A  working-copies/quiz/l
A  working-copies/quiz/m
```

```
Checked out revision 2.  
$ ls  
working-copies
```

如果你中断了取出的动作 (或是其它事情中断了正在进行的取出动作, 像是断线等等), 你可以再利用相同的 `checkout` 命令重新开始, 或是更新尚未完整的工作复本:

```
$ svn checkout file:///tmp/repos/test test  
A  test/a  
A  test/b  
^C  
svn: The operation was interrupted  
svn: caught SIGINT  
  
$ svn checkout file:///tmp/repos/test test  
A  test/c  
A  test/d  
^C  
svn: The operation was interrupted  
svn: caught SIGINT  
  
$ cd test  
$ svn update  
A  test/e  
A  test/f  
Updated to revision 3.
```

Name

`svn cleanup` — 递归式地清除工作复本.

摘要

`svn cleanup [PATH ...]`

描述

递归式地清除工作复本, 移除未完成动作的锁定. 如果你遇到了 “working copy locked” (工作复本已锁定) 的错误, 执行这个命令以移除遗留下来的锁定, 让工作复本再回复到可用的状态. 请参见 [Appendix C, 故障排除](#).

替代名称

无

更动 Changes

工作复本

存取档案库

否

选项:

无

范例

因为 **svn cleanup** 不会产生任何输出, 所以这里没什么范例. 如果你没有指定 **PATH**, 就使用 **!.**

```
$ svn cleanup
```

```
$ svn cleanup /path/to/working-copy
```

Name

svn commit — 从工作复本传送更动至档案库.

摘要

```
svn commit [PATH ... ]
```

描述

从工作复本传送更动至档案库. 如果你没有以 **--file** 或 **--message** 选项来指定纪录讯息, **svn** 就会执行你的文字编辑器, 让你撰写送交讯息. 请参考 [the section called “Config”](#) 的 **editor-cmd** 一节.

Tip

如果你开始送交的动作, **Subversion** 也已经执行编辑器, 让你撰写送交讯息, 你还是可以中断送交更动的动作. 如果你想要取消你的送交, 只要结束编辑器, 不要储存送交讯息, 那么 **Subversion** 就会给

你提示讯息, 是想要中断送交, 以无讯息而继续, 还是再编辑讯息一次.

替代名称

ci (“check in” 的简称, 而不是 “co”, 那是 “checkout” 的简称)

更动

工作复本, 档案库

存取档案库

是

选项

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--non-recursive (-N)
--targets FILENAME
--force-log
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
```

范例

送交一个简单的档案更动, 在命令列指定送交讯息, 并以目前的工作目录 (“.”) 作为隐含的目标:

```
$ svn commit -m "added howto section."
Sending          a
Transmitting file data .
Committed revision 3.
```

送交档案 foo.c (在命令列明确地指定) 的更动, 并以档案 msg 的内容为送交讯息:

```
$ svn commit -F msg foo.c
Sending          foo.c
Transmitting file data .
Committed revision 5.
```

如果你想要让 `--file` 使用一个纳入版本控制的档案来指定送交讯息, 你必须加上 `--force-log` 选项:

```
$ svn commit --file file_under_vc.txt foo.c
svn: The log message file is under version control
svn: Log message file is a versioned file; use '--force-log' to
override.

$ svn commit --force-log --file file_under_vc.txt foo.c
Sending          foo.c
Transmitting file data .
Committed revision 6.
```

要送交一个预定被删除的档案:

```
svn commit -m "removed file 'c'."
Deleting        c

Committed revision 7.
```

Name

`svn copy` — 复制一个工作复本或档案库的目录或档案.

摘要

```
svn copy SRC DST
```

描述

复制一个工作复本或档案库里的档案. **SRC** 与 **DST** 可以是工作复本 (WC), 或是 URL:

WC -> WC

复制一个项目, 并排定新增它 (连同历史纪录).

WC -> URL

立即送交一个 WC 的复本至 URL.

URL -> WC

将 URL 取出至 WC, 并排定新增它.

URL -> URL

完成服务器端的复制. 这通常是用来进行分支与标记.

Warning

你只能复制同一个档案库里的档案. Subversion 不支持跨档案库的复制.

替代名称

cp

更动

如果目标是 URL, 则为档案库.

如果目标是工作复本, 则为工作复本.

存取档案库

如果来源或目标在档案库内, 或是需要查看来源的修订版号时.

选项

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--encoding ENC
```

范例

在工作复本里复制一个项目 (只是排程复制而已 — 在你进行送交之前, 不会有东西送入档案库):

```
$ svn copy foo.txt bar.txt
A      bar.txt
```



```
$ svn status
A +   bar.txt
```

将一工作复本的项目复制到档案库的 URL (这是立即送交, 所以你必须提供送交讯息):

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m "Remote copy."
```

Committed revision 8.

将一个档案库的项目复制到工作复本 (只是排程复制而已 — 在你进行送交之前, 不会有东西送入档案库):

Tip

这是重新取回已于档案库消失的档案的建议用法!

```
$ svn copy file:///tmp/repos/test/far-away near-here
A      near-here
```

最后, 从一个 URL 复制到另一个 URL:

```
$ svn copy file:///tmp/repos/test/far-away
file:///tmp/repos/test/over-there -m "remote copy."
```

Committed revision 9.

Tip

这是在档案库中, '标记' 修订版的最简易方法 — 只要以 **svn copy** 将该修订版 (通常是 HEAD) 复制到你的 tags 目录即可.

```
$ svn copy file:///tmp/repos/test/trunk
file:///tmp/repos/test/tags/0.6.32-prerelease -m "tag tree"
```

Committed revision 12.

如果你忘了作标记, 不必太担心 — 你可以在任何时间, 在标记时指定旧的修订版:

```
$ svn copy -r 11 file:///tmp/repos/test/trunk
file:///tmp/repos/test/tags/0.6.32-prerelease -m "Forgot to tag at
rev 11"
```

Name

svn delete — 删除一个工作复本或档案库的项目.

摘要

```
svn delete PATH...  
svn delete URL...
```

描述

以 **PATH** 指定的项目, 会被排定在下一次送交时删除. 档案 (还有尚未送交的目录) 会马上自工作复本中删除. 本命令不会删除任何未受版本控制, 或是被修改过的档案; 请使用 `--force` 选项来强制取消这个限制.

以 **URL** 指定的项目, 会透过立即送交自档案库中删除. 多个 **URL** 会自动地以不可分割的方式送交.

替代名称

del, remove, rm

更动

若针对档案则为工作复本, 若针对 **URL** 则为档案库.

存取档案库

仅在针对 **URL** 时

选项

```
--force  
--force-log  
--message (-m) TEXT  
--file (-F) FILE  
--quiet (-q)  
--targets FILENAME  
--username USER  
--password PASS  
--no-auth-cache  
--non-interactive
```

```
--encoding ENC
```

范例

使用 **svn** 来删除工作复本的档案, 只是预定被删除. 当你送交更动时, 该档案才会自档案库删除.

```
$ svn delete myfile
D          myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting          myfile
Transmitting file data .
Committed revision 14.
```

但是删除 URL 的效果是立即的, 所以你必须提供纪录讯息:

```
$ svn delete -m "Deleting file 'yourfile'"
file:///tmp/repos/test/yourfile

Committed revision 15.
```

这里的例子, 是如何强制进行有本地更动的档案:

```
$ svn delete over-there
svn: Attempting restricted operation for modified resource
svn: Use --force to override this restriction
svn: 'over-there' has local modifications

$ svn delete --force over-there
D          over-there
```

Name

svn diff — 显示两个路径之间的差异.

摘要

```
svn diff [-r N[:M]] [TARGET...]
svn diff URL1[@N] URL2[@M]
```

描述

显示两个路径之间的差异. 每一个 **TARGET** 可以是工作复本, 或是 **URL**. 如果没有指定 **TARGET** 的话, 会使用 '!'.

如果 **TARGET** 是 **URL** 的话, 那么就必须透过 `--revision` 指定修订版 **N** 与 **M**.

如果 **TARGET** 是工作复本路径, 那么 `--revision` 选项表示:

--revision N:M

服务器会比较 **TARGET@N** 与 **TARGET@M**.

--revision N

客户端会将 **TARGET@N** 与工作复本作比较.

(没有 --revision)

客户端会比较 **TARGET** 的 **BASE** 修订版与工作复本.

如果使用另一个语法的话, 服务器会比较 **URL1** 的 **N** 修订版与 **URL2** 的 **M** 修订版. 如果 **N** 与 **M** 都省略的话, 就假设是 **HEAD**.

替代名称

di

更动

无

存取档案库

只有在取得工作复本 **BASE** 以外修订版的差异时

选项

```
--revision (-r) REV
--extensions (-x) "ARGS"
--non-recursive (-N)
--diff-cmd CMD
--username USER
--password PASS
--no-auth-cache
--non-interactive
--no-diff-deleted
```

范例

比较 BASE 修订版与工作复本 (**svn diff** 最常见的用法之一):

```
$ svn diff COMMITTERS
Index: COMMITTERS
=====
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
```

查看工作复本的修改与旧修订版之间的差异:

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
=====
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

利用 '@' 语法, 比较修订版 3000 与修订版 3500 之间的差异:

```
$ 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)
...
```

透过指定范围的语法, 比较修订版 3000 与 3500 之间的差异 (此时, 你只要指定一个 URL 即可):

```
$ svn diff -r 3000:3500
http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
=====
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

如果你有工作复本的话, 你可以不必输入冗长的 URL, 就可以取得差异:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
=====
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

利用 `--diff-cmd CMD -x`, 直接传递参数给外部的差异程序:

```
svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
=====
0a1,2
> This is a test
>
```

Name

`svn export` — 输出一个干净的目录树.

摘要

```
svn export [-r REV] URL [PATH]
svn export PATH1 PATH2
```

描述

第一种形式会从 **URL** 指定的档案库, 汇出一个干净的目录树到 **PATH**. 如果有指定 **REV** 的话, 内容即为该修订版的, 否则就是 **HEAD** 修订版. 如果 **PATH** 被省略的话, **URL** 的最后部份会被用来当成本地的目录名称.

第二种形式会在工作复本中, 从指定的 **PATH1** 汇出一个干净的目录树到 **PATH2**. 所有的本地修改都还会保持着, 但是未纳入版本控制的档案不会被复制.

替代名称

无

修改

本地磁盘

存取档案库

只有从 **URL** 汇出时

选项

`--revision (-r) REV`

```
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

范例

自你的工作副本汇出 (不会显示每一个目录与档案):

```
$ svn export a-wc my-export
pantheon: /tmp>
```

直接从档案库汇出 (显示每一个目录与档案):

```
$ svn export file:///tmp/repos my-export
A my-export/test
A my-export/quiz
...
Exported revision 15.
```

Name

svn help — 救助!

摘要

svn help [SUBCOMMAND...]

描述

当你正在使用 Subversion, 而这本书又不在手边, 这是你最好的朋友!

替代名称

?, h

更动

无

存取档案库

无

选项

```
--version  
--quiet (-q)
```

Name

svn import — 递归式地送交 PATH 的复本至 URL.

摘要

```
svn import [PATH] URL
```

描述

递归式地送交 PATH 的复本至 URL. 如果省略 PATH, 预设为 '!'. 父目录会依需要, 在档案库内建立.

替代名称

否

更动

档案库

存取档案库

是

选项

```
--message (-m) TEXT  
--file (-F) FILE  
--quiet (-q)  
--non-recursive (-N)  
--username USER  
--password PASS  
--no-auth-cache  
--non-interactive
```



```
--force-log
--encoding ENC
```

范例

这会直接从本地目录 'myproj' 汇入至档案库的根目录中.

```
$ svn import -m "New import" myproj http://svn.red-
bean.com/repos/test
Adding          myproj/sample.txt
...
Transmitting file data .....
Committed revision 16.
```

这会汇入本地目录 'myproj' 至档案库的 'trunk/vendors'. 'trunk/vendors' 目录不需在汇入之前就存在 — **svn import** 会帮你递归地建立目录:

```
$ svn import -m "New import" myproj \
    http://svn.red-bean.com/repos/test/trunk/vendors/myproj
Adding          myproj/sample.txt
...
Transmitting file data .....
Committed revision 19.
```

Name

svn info — 显示 PATH 的信息.

摘要

```
svn info [PATH...]
```

描述

显示工作目录里的路径信息, 包括了:

- 路径 (Path)
- 名称 (Name)
- 网址 (Url)
- 修订版 (Revision)
- 节点类型 (Node Kind)
- 前次更动作者 (Last Changed Author)
- 前次更动修订版 (Last Changed Revision)

- 前次更动日期 (Last Changed Date)
- 前次本文更新日期 (Text Last Updated)
- 前次性质更新日期 (Properties Last Updated)
- 总和检查码 (Checksum)

替代名称

无

更动

无

存取档案库

否

选项

```
--targets FILENAME  
--recursive (-R)
```

范例

svn info 会提供给你所有关于工作复本的项目的有用信息. 它会显示档案的信息:

```
$ svn info foo.c  
Path: foo.c  
Name: foo.c  
Url: http://svn.red-bean.com/repos/test/foo.c  
Revision: 4417  
Node Kind: file  
Schedule: normal  
Last Changed Author: sally  
Last Changed Rev: 20  
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)  
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)  
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)  
Checksum: /3L38YwzhT93BWvgpdF6Zw==
```

它也会显示目录的信息:

```
$ svn info vendors  
Path: trunk
```

Url: <http://svn.red-bean.com/repos/test/vendors>
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)

Name

svn list — 列出档案库中的目录项目.

摘要

svn list [TARGET...]

描述

列出每个 TARGET 档案与每个档案库里的 TARGET 目录. 如果 TARGET 是工作复本的路径, 那么相对应的档案库 URL 就会被拿来使用.

预设的 TARGET 是 '.', 表示目前工作复本目录的档案库 URL.

加上 --verbose 的话, 以下的字段会显示出项目的状态:

- 前次送交的修订版(Revision number of the last commit)
- 前次送交的作者(Author of the last commit)
- 大小 (Size) (以字节为单位)
- 前次送交的日期与时间(Date and time of the last commit)

替代名称

ls

更动

无

存取档案库

是

选项

```
--revision (-r) REV
--verbose (-v)
--recursive (-R)
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

范例

如果你想知道 repository 中有什么档案, 又不想下载工作复本的话, **svn list** 就很方便.

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
...
```

就像 UNIX 的 **ls**, 你也可以传递 `--verbose` 选项, 以取得更详细的信息:

```
svn list --verbose file:///tmp/repos
  16 sally          28361 Jan 16 23:18 README.txt
  27 sally           0 Jan 18 15:27 INSTALL
  24 harry          Jan 18 11:27 examples/
```

欲知详情, 请参考 [the section called “svn list”](#).

Name

svn log — 显示送交纪录讯息.

摘要

```
svn log [PATH]
svn log URL [PATH...]
```

描述

预设的目标, 是你目前工作目录的路径. 如果没有提供任何自变量的话, **svn log** 会显示目前工作复本的工作目录里, 所有档案的纪录讯息. 你可以透过指定一个

路径, 一个或多个修订版, 或两者的任何组合来微调产生的结果. 预设对本地路径的修订版范围为 **BASE:1**.

如果你只有指定一个 **URL**, 那么它会显示该 **URL** 所有的每一个项目的纪录讯息. 如果你在 **URL** 之后指定路径, 那么只会显示该 **URL** 下的这些指定的路径的讯息. 预设对 **URL** 的修订版范围为 **HEAD:1**.

使用 `--verbose` 的话, 每一个纪录讯息的影响路径也会一并显示. 使用 `--quiet` 的话, **svn log** 不会显示纪录讯息 (这与 `--verbose` 兼容).

每一个纪录讯息只会显示一次而已, 就算明确地指定多个被它所影响的路径亦同. 预设的显示讯息会跟着复制历史; 使用 `--stop-on-copy` 可以关闭这个功能, 在找出分支点的很方便.

替代名称

无

更动

无

存取档案库

是

选项

```
--revision (-r) REV
--quiet (-q)
--verbose (-v)
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--stop-on-copy
--incremental
--xml
```

范例

你可以在工作目录的最上层目录执行 **svn log**, 以得知所有更动过的路径的纪录讯息:

```
$ svn log
-----
---
r20:  harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line
Tweak.
-----
---
r17:  sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
...
```

检视特定工作复本档案的所有纪录讯息.

```
$ svn log foo.c
-----
---
r32:  sally | 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003) | 1 line
Added defines.
-----
---
r28:  sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
...
```

如果你没有工作复本的话, 也可以对 URL 执行本命令:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c
-----
---
r32:  sally | 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003) | 1 line
Added defines.
-----
---
r28:  sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
...
```

如果你要看在同一个 URL 下的数个不同的路径, 你可以使用 URL [PATH...] 语法.

```
$ svn log http://svn.red-bean.com/repos/test/ foo.c bar.c
-----
---
r32:  sally | 2003-01-13 16:43:13 -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  
...
```

它与明确地在命令列指定两个 URL 的效果相同:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c \  
          http://svn.red-bean.com/repos/test/foo.c  
...
```

当你要把数个 log 命令的结果合并起来时,你可能会想要使用 `--incremental` 选项. 正常来讲, **svn log** 会在纪录讯息开头, 在后续的纪录讯息之间, 在最后一个纪录讯息之后加上虚线列. 如果你对两个修订版的范围执行 **svn log** 的话, 你会得到以下的结果:

```
$ svn log -r 14:15  
-----  
---  
r14:  ...  
  
-----  
---  
r15:  ...  
  
-----  
---
```

不过如果你想要把两个不连续的纪录讯息放在同一个档案中, 你应该会作像下列的事:

```
$ svn log -r 14 > mylog  
$ svn log -r 19 >> mylog  
$ svn log -r 27 >> mylog  
$ cat mylog  
-----  
---  
r14:  ...  
  
-----  
---  
-----  
---  
r19:  ...  
  
-----  
---  
-----  
---  
r27:  ...
```


你可以藉由使用 `--incremental` 选项, 避免在你的输出出现连续两个虚线列:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
```

```
-----
---
r14: ...
```

```
-----
---
r19: ...
```

```
-----
---
r27: ...
```

当使用 `--xml` 选项时, `--incremental` 选项也提供了类似的输出控制.

Tip

如果你对特定的路径执行 **svn log**, 提供了特定的修订版, 但是没有得到任何的输出:

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
```

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

这只是表示那个路径并没有在该修订版中被修改. 如果你从档案库最上层执行本命令的话, 或是知道在该修订版变动的档案, 你可以明确地指定它:

```
$ svn log -r 20 touched.txt
```

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

```
r20: sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003)
| 1 line
```

```
Made a change.
```

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

Name

`svn merge` — 将两个来源之间的差异套用到工作复本路径.

摘要

```
svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]
svn merge -r N:M SOURCE [PATH]
```

描述

第一种形式中, 来源 **URL** 各被指定到修订版 **N** 与 **M**. 这两个就是作为比较的来源. 如果没有指定修订版的话, 预设值为 **HEAD**.

第二种形式中, **SOURCE** 可为 **URL** 或工作复本项目, 后者会使用对应的 **URL**. 在修订版 **N** 与 **M** 的 **URL**, 定义出两个比较的来源.

WCPATH 是接收更动的工作复本路径. 如果省略 **WCPATH** 的话, 默认值为 **!**, 除非有一个 **!** 中的档案与来源中的主档名相同; 此时, 产生的差异会套用到该档案.

替代名称

无

更动

工作复本

存取档案库

仅在有关 **URL** 的情况下

选项

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--force
--dry-run
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

范例

将分支合并回 **trunk** (假设你有一份 **trunk** 的工作复本):

```
$ svn merge http://svn.red-bean.com/repos/trunk/vendors \
    http://svn.red-bean.com/repos/branches/vendors-with-fix
U   myproj/tiny.txt
U   myproj/thhgttg.txt
U   myproj/win.txt
U   myproj/flo.txt
```

如果你在修订版 23 建立分支, 然后想要把 **trunk** 的更动合并至分支中, 你可以从分支的工作复本中执行下列命令:

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors
U   myproj/thhgttg.txt
...
```

将更动合并到一个档案中:

```
$ cd myproj
$ svn merge -r 30:31 thhgttg.txt
U   thhgttg.txt
```

Name

svn mkdir — 在版本控制之下建立一个新目录.

摘要

```
svn mkdir PATH...
svn mkdir URL...
```

描述

建立一个与 **PATH** 或 **URL** 的最后部份同名的目录. 以工作复本 **PATH** 指定的目录会预订加入至工作复本里. 以 **URL** 指定的目录, 会透过立即送交, 于档案库内建立. 在两种情况下, 中间的目录都必须事先建立.

替代名称

无

更动

工作复本, 若针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force-log
```

范例

在工作复本中建立一个目录:

```
$ svn mkdir newdir
A          newdir
```

在档案库建立目录 (这是立即送交, 需要提供纪录讯息):

```
$ svn mkdir -m "Making a new dir." http://svn.red-
bean.com/repos/newdir
```

```
Committed revision 26.
```

Name

svn move — 移动目录或档案.

摘要

```
svn move SRC DST
```

描述

本命令可移动工作复本或档案库的目录或档案.

Tip

本命令等同于 **svn copy**, 紧接着 **svn delete**.

Warning

Subversion 不支持工作复本与 URL 之间的移动动作. 此外, 你只能于相同的档案库内移动档案 — Subversion 不支持跨档案库的移动.

WC -> WC

移动并排定新增目录或档案 (连同历史纪录一起).

URL -> URL

完成伺服端的更名动作.

替代名称

mv, rename, ren

更动

工作复本, 若针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force-log
```

范例

在工作复本中移动档案:

```
$ svn move foo.c bar.c
A      bar.c
D      foo.c
```

在档案库中移动档案 (这是立即送交, 需要提供纪录讯息):

```
$ 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.

Name

svn propdel — 移除一个项目的性质.

摘要

```
svn propdel PROPNAME [PATH...]
svn propdel PROPNAME --revprop -r REV [URL]
```

描述

本命令会移除目录, 档案, 以及修订版的性质. 第一种形式会移除纳入版本控制的工作复本性质, 而第二种形式会移除无版本控制的档案库远程修订版性质.

替代名称

pdel

更动

工作复本, 如果针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
```

范例

删除工作副本档案的性质

```
$ svn propdel svn:mime-type some-script
property `svn:mime-type' deleted from 'some-script'.
```

删除一个修订版性质:

```
$ svn propdel --revprop -r 26 release-date
property `release-date' deleted from repository revision '26'
```

Name

svn propedit — 编辑纳入版本控制的一个或多个项目的性质.

摘要

```
svn propedit PROPNAME PATH...
svn propedit PROPNAME --revprop -r REV [URL]
```

描述

使用你偏爱的文字编辑器, 编辑一个或多个性质. 第一种形式可编辑纳入版本控制的工作复本性质, 而第二种形式可编辑无版本控制的档案库远程修订版性质更动.

替代名称

pedit, pe

更动

工作复本, 若针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--revision (-r) REV
--revprop
--encoding ENC
```

范例

svn propedit 让修改拥有多个数值的性质相当地容易:

```
$ svn propedit svn:keywords foo.c
<svn 在这里会执行你偏好的文字编辑器, 开启一个含有目前 svn:keywords
数值的编辑内容. 你只要在这里每一列输入一个数值, 就可以很容易地对一个
性质新增多个数值.>
Set new value for property 'svn:keywords' on 'foo.c'
```

Name

svn propget — 显示性质的数值.

摘要

```
svn propget PROPNAME [PATH...]
svn propget PROPNAME --revprop -r REV [URL]
```

描述

显示目录, 档案, 或修订版的性质数值. 第一个形式会显示纳入版本控制的工作复本性质, 而第二个形式会显示无版本控制的档案库修订版性质. 请参考 [the section called “性质”](#) 以取得更多有关性质的信息.

替代名称

pget, pg

更动

工作复本, 若针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--recursive (-R)  
--revision (-r) REV  
--revprop
```

范例

检视工作复本档案的性质:

```
$ svn propget svn:keywords foo.c  
Author  
Date  
Rev
```

一样, 但是目标是修订版性质:

```
$ svn propget svn:log --revprop -r 20  
Began journal.
```

Name

svn proplist — 列出所有的性质.

摘要

```
svn proplist [PATH...]  
svn proplist --revprop -r REV [URL]
```

描述

列出所有档案, 目录, 或是修订版的性质. 第一种形式会显示纳入版本管理的工作复本性质, 第二种形式会显示无版本控制的档案库远程修订版性质更动.

替代名称

plist, pl

更动

工作复本, 若针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--verbose (-v)
--recursive (-R)
--revision (-r) REV
--revprop
```

范例

你可以使用本命令, 察看工作复本中某一个项目的性质:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

如果使用 `--verbose` 选项的话, `svn proplist` 就变得更加地方便, 因为它会连性质的数值也一并显示出来:

```
$ svn proplist --verbose foo.c
Properties on 'foo.c':
  svn:mime-type : text/plain
  svn:keywords  : Author Date Rev
  owner         : sally
```

Name

svn propset — 设定目录, 档案, 或修订版的 **PROPNAME** 内容为 **PROPVAL**.

摘要

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [URL]
```

描述

设定目录, 档案, 或修订版的 **PROPNAME** 内容为 **PROPVAL**. 第一个例子会在工作复本中建立一个纳入版本控制的本地性质更动, 第二个例子对档案库修订版建立一个无版本控制的远程性质更动.

Tip

Subversion 的行为会被几个“特殊”性质影响. 请参考 [the section called “特殊性质”](#) 以了解更多这类的性质.

替代名称

pset, ps

更动

工作复本, 若针对 URL 则为档案库

存取档案库

仅在 URL 的情况下

选项

```
--file (-F) FILE
--quiet (-q)
--revision (-r) REV
--targets FILENAME
--recursive (-R)
--revprop
--encoding ENC
```

范例

设定一个档案的 mime 型别:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property `svn:mime-type' set on 'foo.jpg'
```

在 UNIX 系统上, 如果你想要设定一个档案的执行权限:

```
$ svn propset svn:executable ON somescript
property `svn:executable' set on 'somescript'
```

也许你们有个内部原则, 为了同事们的方便, 要设定某些性质:

```
$ svn propset owner sally foo.c
property `owner' set on 'foo.c'
```

如果你写错了某个修订版的纪录讯息, 想要修正它, 使用 `--revprop` 并设定 `svn:log` 为新的纪录讯息:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New
York."
property `svn:log' set on repository revision '25'
```

或者没有工作复本的话, 你可以提供一个 URL.

```
$ svn propset --revprop -r 26 svn:log "Document nap." http://svn.red-
bean.com/repos
property `svn:log' set on repository revision '25'
```

最后, 你可以让 `propset` 从档案取得它的输入内容. 你甚至可以透过这个功能, 将某个性质设定成二进制的内容:

```
$ svn propset owner-pic -F sally.jpg moo.c
property `owner-pic' set on 'moo.c'
```

Warning

预设情况下, 你无法设定 Subversion 的档案库的修订版性质. 你的档案库管理员必须先建立名为 `'pre-revprop-change'` 的挂勾程序, 明确地开启更动修订版性质的功能. 请参考 [the section called “Hook scripts”](#), 以了解更多关于挂勾程序的信息.

Name

svn resolved — 移除工作复本的目录或档案的“冲突”状态.

摘要

svn resolve PATH...

描述

移除工作复本的目录或档案的“冲突”状态. 这个动作并不会依语法来解决冲突标记; 它只是移除冲突的相关档案, 然后让 **PATH** 可以再度送交; 也就是说, 它告诉 Subversion 冲突已经“解决”了. 请参考 [the section called “解决冲突 \(合并他人的更动\)”](#), 以便对解决冲突有更进一步的了解.

替代名称

无

更动

工作复本

存取档案库

否

选项

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
```

范例

如果你在更新时得到一个冲突, 你的工作复本会出现三个新档案:

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
```

```
foo.c.r30  
foo.c.r31
```

当冲突解决了之后, `foo.c` 就可以进行送交. 执行 **svn resolved**, 让工作复本知道你已经处理妥当了.

Warning

你 可以 直接删除冲突档案, 然后送行送交, 但是 **svn resolved** 除了删除冲突档案之外, 还会在工作复本的管理区域进行一些状态更新的工作, 所以我们建议你使用本命令.

Name

svn revert — 回复所有的本地编辑.

摘要

```
svn revert PATH...
```

描述

回复所有目录或档案的本地修改, 并且解除冲突的状况. **svn revert** 不只会回复档案的内容而已, 它连性质更动也会回复. 最后, 你可以用它来解除你已进行的预定排程动作 (像是预定新增或删除的档案, 可以“取消预定排程”).

替代名称

无

更动

工作复本

存取档案库

否

选项

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
```

范例

放弃档案的更动:

```
$ svn revert foo.c
Reverted foo.c
```

如果你想要回复一整个目录的档案, 请使用 `--recursive` 旗标:

```
$svn revert --recursive .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

最后, 你可以取消任何预定的排程:

```
$ 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
```

Warning

如果你没有指定任何目录给 **svn revert** 的话, 它什么都不会作 — 为了预防你不小心失去所有工作复本的更动, **svn revert** 要求你必须提供至少一个目标.

Name

svn status — 显示工作复本目录与档案的状态.

摘要

```
svn status [PATH...]
```

描述

显示工作复本目录与档案的状态. 未指定自变量时, 它只会显示本地修改的项目 (没有档案库的存取动作). 使用 `--show-updates` 时, 会增加工作修订版与服务器过时的信息. 使用 `--verbose` 时, 显示每一个项目的完整修订版信息.

输出的前五个字段各为一个字符宽, 每一个字段提供你每一个工作复本项目不同部份的信息.

第一个字段表示一个项目是新增, 删除, 不然就是修改过的.

' '

无修改.

'A'

该项目预计要新增.

'D'

该项目预计要删除.

'M'

该项目已被修改.

'C'

该项目与来自档案库的更新有所冲突.

'I'

该项目已被忽略 (例如透过 `svn:ignore` 性质).

'?'

该项目未纳入版本控制.

'!'

该项目已遗失 (例, 未使用 **svn** 而删除它), 或是不完整 (取出或更新时被中断).

'~'

该项目纳入版本控制为目录, 但是已经被取代成档案, 或是相反的情况.

第二栏显示目录或档案的性质状态:

''

无修改.

'M'

本项目的性质已被更动.

'C'

本项目的性质与来自档案库的性质更新有所冲突.

第三栏只在工作复本目录被锁定时才会出现.

''

本项目未被锁定.

'L'

本项目已被锁定.

第四栏只在本项目已预定要连同历史纪录被新增时, 才会出现.

''

预定的送交并不包含历史纪录.

'+'

预定的送交包含历史纪录.

相对其父目录, 该项目已被切换 (参见 [the section called “切换工作复本”](#)) 时, 第五栏才会出现.

''

该项目是其父目录的子项目.

'S'

该项目已被切换.

是否过时的信息, 出现的位置是第八栏 (仅于使用 `--show-updates` 选项时).

''

工作复本的项目是最新版的.

'*'

服务器上有该项目的更新修订版.

剩余的字段皆为变动宽度, 并以空白隔开. 如果使用 `--show-updates` 或 `--verbose` 的话, 工作修订版就是下一个字段.

如果使用 `--verbose` 选项的话, 跟着会显示最后送交的修订版, 以及最后送交的作者.

工作复本路径一定是最后一栏, 所以它可以包含空格符.

替代名称

stat, st

更动

无

存取档案库

仅在使用 `--show-updates` 的情况下

选项

```
--show-updates (-u)
--verbose (-v)
--non-recursive (-N)
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--no-ignore
```

范例

这是更容易找出你在工作复本中作了哪些更动的方法:

```
$ svn status wc
M      wc/bar.c
A +    wc/qax.c
```

如果你想要找出哪些工作复本的档案是过时的话, 请使用 `--show-updates` 选项 (这 不会对你的工作复本产生任何更动). 在这里, 你可以看出 `wc/foo.c` 在我们上次更新工作复本后, 已经在档案库中有所变动了:

```
$ svn status --show-updates wc
M      965      wc/bar.c
      *      965      wc/foo.c
A +     965      wc/qax.c
Head revision: 981
```

Warning

`--show-updates` 只会在过时的项目 (也就是执行 **svn update** 时, 会自档案库更新的项目) 旁边加上星号. `--show-updates` 不会让列出来的状态, 显示出该项目位于档案库的版本.

最后, 以下是你从 `status` 子命令所能取得最详细的信息:

```
$ svn status --show-updates --verbose wc
M      965      938 sally      wc/bar.c
      *      965      922 harry      wc/foo.c
A +     965      687 harry      wc/qax.c
      965      687 harry      wc/zip.c
Head revision: 981
```

欲得知更多有关 **svn status** 的例子, 请参考 [the section called “svn status”](#).

Name

`svn switch` — 将工作复本更新至不同的 URL.

摘要

```
svn switch URL [PATH]
```

描述

本子命令会更新你的工作复本, 映像到一个新的 URL — 通常是一个与工作复本有共通起源的 URL, 不过这不是必要的. 这是 Subversion 移动工作复本到一个新的分支的方法. (请参考 [the section called “切换工作复本”](#)), 以了解更多有关于切换的细节.

替代名称

sw

更动

工作复本

存取档案库

是

选项

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--relocate
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

范例

如果你现在在目录 'vendors' 之中, 它有一个分支 'vendors-with-fix', 而你想要将工作复本切换到该分支去:

```
$ 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.
```

要切换回来, 只要使用原来用以取出工作复本的档案库位置的 URL 即可:

```
$ 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.
```

Tip

如果你不需要将整个工作复本都切换过去的话, 你也可以只切换部份工作复本到某个分支.

如果你的档案库位置变动了, 但是现有的工作复本仍想继续使用, 不想变更时, 你可以使用 **svn switch --relocate**, 将工作复本的 URL 从这一个换到另外一个:

```
$ svn checkout file:///tmp/repos test
A test/a
A test/b
...

$ mv repos newlocation
$ cd test/

$ svn update
svn: Couldn't open a repository.
svn: Unable to open an ra_local session to URL
svn: Unable to open repository 'file:///tmp/repos'

$ svn switch --relocate file:///tmp/repos file:///tmp/newlocation .
$ svn update
At revision 3.
```

Name

svn update — 更新工作复本.

摘要

svn update [PATH...]

描述

svn update 会将档案库的更动带入至工作复本. 如果没有提供修订版的话, 它会将你的工作复本更新至 **HEAD** 修订版. 不然的话, 同步至 **--revision** 选项所指定的修订版.

对每一个更新的项目, 每一列开头会以一个字符表示所采取的行为. 这些字符代表如下:

A

新增

D

删除

U

更新

C

冲突

M

合并

第一栏的字符表示实际档案的更新, 而档案性质显示在第二栏.

替代名称

up

更动

工作复本

存取档案库

是

选项

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

范例

取得上次更新后, 档案库里产生的更动:

```
$ svn update
A newdir/toggle.c
A newdir/disclose.c
A newdir/launch.c
D newdir/README
Updated to revision 32.
```

你也可以将工作复本更新至旧的修订版 (Subversion 没有 CVS 的 “sticky” 概念. 请参见 [Appendix A, 给 CVS 使用者的 Subversion 指引](#)):

```
svn update -r30
A newdir/README
D newdir/toggle.c
D newdir/disclose.c
D newdir/launch.c
U foo.c
Updated to revision 30.
```

Tip

如果你想要检视单一档案的旧修订版内容, 你可能会想用 **svn cat**.

svnadmin

svnadmin 是用来监控与修复 Subversion 档案库的管理工具. 详情请参考 [the section called “svnadmin”](#).

由于 **svnadmin** 是以直接存取档案库的方式工作 (所以只能在存放有档案库的机器上使用), 因此它是以 *路径* 来指定档案库, 而非 *URL*.

svnadmin 选项

--bypass-hooks

略过档案库挂勾系统.

--copies

检视路径时, 跟随复制历史纪录.

--in-repos-template ARG

在建立新的档案库时, 指定一个作为档案库结构的模板.

“in-repository” 范本可指定档案库本身的配置 (存在于 db/ 目录的 Berkeley DB 档案中), 像是 /trunk, /branches 等等. 这些模板可被管理员或应用程序作为设定档案库初始加载之用 (不需要执行挂勾程序). 这没有默认值; 档案库一开始是 “空的”, 除非你作另外的指定.

`--incremental`

将修订版内容以对前一版的差异倾印出来, 而非一般使用的完整文字.

`--on-disk-template ARG`

指定一个模板, 作为你想要建立的档案库在磁盘中的目录结构 (也就是 `conf/`, `hooks/` 等等).

“磁盘” 模板描述档案库目录. 每一个模板都有一个名称, 而 “预设” 的磁盘模板包含了:

- default/
- README.txt
- dav/
- format
- hooks/
- post-commit.tmpl
- post-revprop-change.tmpl
- pre-commit.tmpl
- pre-revprop-change.tmpl
- start-commit.tmpl
- locks/
- db.lock

磁盘结构一般用来预先定义要建立的挂勾命令稿. 举例来说, 你可以预先建立 `post-commit` 命令稿, 里面使用邮件与备份命令稿. 接下来, 每一次管理员建立一个新的档案库时, 她就可以使用这个新的模板, 自动地将所有的挂勾包含进来.

`--revision (-r) ARG`

指定运作的修订版.

svnadmin 子命令

Name

`svnadmin list-unused-dblogs` — 询问 Berkeley DB, 哪些纪录文件可安全地删除

摘要

```
svnadmin list-unused-dblogs REPOS_PATH
```

描述

Berkeley 会对所有档案库的更动建立纪录, 以便在灾难事件发生时, 得以进行重建. 随着使用时间的增加, 纪录文件会逐渐累积, 但是大部份都不再使用, 可以将之删除, 以释放磁盘空间. 请参照 [the section called “Berkeley DB 工具”](#), 以取得更多的信息.

范例

自档案库移除所有不再使用的纪录文件:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## 释放磁盘空间!
```

Name

svnadmin create — 在 REPOS_PATH 建立一个新的, 空的档案库.

摘要

```
svnadmin create REPOS_PATH
```

描述

在指定的路径建立一个新的, 空的档案库. 如果提供的目录不存在, 它会自动被建立出来.

选项

```
--on-disk-template arg
--in-repos-template arg
```

范例

建立一个新的档案库就是这么简单:


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

Name

svnadmin dump — 将档案系统的内容倾印到标准输出.

摘要

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

描述

将档案系统的内容, 以一种可携式 '倾印档' 格式输出到标准输出, 并将讯息回报输出到标准错误. 将 **LOWER** 与 **UPPER** 之间修订版内容倾印出来. 如果没有指定修订版的话, 倾印所有的修订版树. 如果只有指定 **LOWER** 的话, 只倾印一个修订版树. 请参考 [the section called “汇入档案库”](#) 看看实际的使用.

选项

```
--revision (-r)  
--incremental
```

范例

倾印整个档案库:

```
$ svnadmin dump /usr/local/svn/repos  
SVN-fs-dump-format-version: 1  
Revision-number: 0  
* Dumped revision 0.  
Prop-content-length: 56  
Content-length: 56  
...
```

以递增的方式, 从档案库倾印单一异动 (transaction):

```
$ svnadmin dump /usr/local/svn/repos -r 21 --incremental  
* Dumped revision 21.  
SVN-fs-dump-format-version: 1  
Revision-number: 21  
Prop-content-length: 101  
Content-length: 101
```

...

Name

svnadmin help

摘要

svn help [SUBCOMMAND...]

描述

当你困在荒岛上, 没有网络, 没有这本书的时候, 这个子命令可提供很大的帮助.

Name

svnadmin load — 自标准输入读取“倾印档格式”的串流.

摘要

svnadmin load REPOS_PATH

描述

从标准输入读取“倾印档”格式的串流, 将新的修订版送交至档案库的档案系统中. 将进度回报送至标准输出.

范例

这里示范开始从备份档 (当然啰, 由 **svn dump** 产生的) 载入至档案库中:

```
$ svnadmin load /usr/local/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
    * adding path : test ... done.
    * adding path : test/a ... done.
...
```

Name

svnadmin lstxns — 显示所有未处理异动的名称.

摘要

```
svnadmin lstxns REPOS_PATH
```

描述

显示所有未处理异动的名称. 请参考 [the section called “档案库善后”](#), 以得知更多有关于未送交的异动是如何产生的, 以及你该怎么处理.

范例

列出所有档案库中未处理的异动:

```
$ svnadmin lstxns /usr/local/svn/repos/  
lw  
lx
```

Name

svnadmin recover — 修复档案库失去的状态.

摘要

```
svnadmin recover REPOS_PATH
```

描述

如果你遇到要求修复档案库的错误讯息, 请执行本命令.

Warning

只有在你 *绝对确定* 你是唯一存取档案库的人时, 才执行本命令 — 本命令必须有独占的存取权. 请参考 [the section called “档案库回复”](#), 以取得修复档案库更详细的说明.

范例

修复一个有问题的档案库:

```
$ svnadmin recover /usr/local/svn/repos/  
Acquiring exclusive lock on repository db.  
Recovery is running, please stand by...  
Recovery completed.  
The latest repos revision is 34.
```

Name

svnadmin rmtxns — 自档案库删除异动.

摘要

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

描述

从档案库删除所有未处理的异动. 细节请参照在 [the section called “档案库善后”](#).

范例

移除一个具名异动:

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

很幸运的, `svn lstxns` 的输出可以直接作为 `rmtxns` 的输入:

```
$ svnadmin rmtxns /usr/local/svn/repos/ `svnadmin lstxns  
/usr/local/svn/repos/`
```

这样子, 所有尚未处理的异动就会从档案库中移除.

Name

svnadmin setlog — 设定某个修订版的纪录讯息.

摘要

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

描述

将 **FILE** 的内容, 设定为修订版 **REVISION** 的纪录讯息.

这跟使用 **svn propset --revprop** 来设定 修订版的 `svn:log` 性质相当类似, 但是你也可以使用 `--bypass-hooks` 选项, 不执行任何 **pre-commit** 或 **post-commit** 挂勾. 如果 **pre-revprop-change** 挂勾中并不允许修改修订版性质的话, 此时就很方便了.

Warning

修订版性质并未纳入版本控管, 所以这个命令会永远盖写掉先前的纪录讯息.

选项

`--revision (-r) ARG`
`--bypass-hooks`

范例

将修订版 19 的纪录讯息设定成档案 'msg' 的内容:

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

svnlook

svnlook 是用来检视 Subversion 档案库不同方面的命令列工具程序. 它不会对档案库作任何的更动 — 它只是用来“偷看”而已. **svnlook** 通常被档案库的挂勾程序所使用, 但是档案库管理员也可以用它来进行检测系统.

由于 **svnlook** 是透过直接存取档案库来运作 (因为也只能在拥有档案库的机器上使用), 所以它都是以路径作为目标, 而非 **URL**.

如果没有指定修订版或交易的话, **svnlook** 预设会以档案库最年轻的 (最近的) 修订版.

svnlook 选项

svnlook 的选项是全面通用的, 就像 **svn** 与 **svnadmin** 一样. 但是大多数的选项只能用在子命令上, 因为 **svnlook** 的功能都限定在单一的范围上.

`--no-diff-deleted`

不让 **svnlook** 显示已删除档案的差异. 如果有一个档案在异动/修订版中被删除, 预设行为会将差异显示出来, 就好像你还留着档案, 但是档案内容已悉数删除.

`--revision (-r)`

指定你想要检视的修订版号.

`--transaction (-t)`

指定你想要检视的异动编号.

`--show-ids`

显示档案系统树中, 每一个路径的档案系统节点修订版编号.

Name

`svnlook author` — 显示作者.

摘要

`svnlook author REPOS_PATH`

描述

显示档案库中, 某个修订版或异动的作者.

选项

`--revision (-r)`
`--transaction (-t)`

范例

svnlook author 很方便, 但是没什么好玩的:

```
$ svnlook author -r 40 /usr/local/svn/repos
sally
```

Name

svnlook cat — 显示档案的内容.

摘要

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

描述

显示档案的内容.

选项

```
--revision (-r)
--transaction (-t)
```

范例

这会显示异动 ax8 中, 位于 /trunk/README 的档案内容:

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README

      Subversion, a version control system.
      =====

$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

    I. A FEW POINTERS
    II. DOCUMENTATION
    III. PARTICIPATING IN THE SUBVERSION COMMUNITY
...
```

Name

svnlook changed — 显示已更动的路径.

摘要

```
svnlook changed REPOS_PATH
```

描述

显示某个修订版或异动中更动的路径,并在第一栏显示“svn 更新样式”的状态码:
A 表示新增, D 为删除, U 为更新 (更动).

选项

```
--revision (-r)  
--transaction (-t)
```

范例

以下会显示某个测试档案库的修订版 39 中,所有更动的档案:

```
$ svnlook changed -r 39 /usr/local/svn/repos  
A   trunk/vendors/deli/  
A   trunk/vendors/deli/chips.txt  
A   trunk/vendors/deli/sandwich.txt  
A   trunk/vendors/deli/pickle.txt
```

Name

svnlook date — 显示日期戳记.

摘要

```
svnlook date REPOS_PATH
```

描述

显示档案库某个修订版或异动的日期戳记.

选项

```
--revision (-r)  
--transaction (-t)
```

范例

这会显示某个测试档案库的修订版 40 的日期.

```
$ svnlook date -r 40 /tmp/repos/
```


Name

svnlook diff — 显示更动档案与性质的差异.

摘要

svnlook diff REPOS_PATH

描述

以 GNU 样式, 显示档案库中更动的档案与性质差异.

选项

```
--revision (-r)
--transaction (-t)
--no-diff-deleted
```

范例

这会显示一个新增的 (空的) 档案, 一个被删除档案, 以及一个复制的档案:

```
$ svnlook diff -r 40 /usr/local/svn/repos/
Copied: egg.txt (from rev 39, trunk/vendors/deli/pickle.txt)

Added: trunk/vendors/deli/soda.txt
=====

Modified: trunk/vendors/deli/sandwich.txt
=====
--- trunk/vendors/deli/sandwich.txt      (original)
+++ trunk/vendors/deli/sandwich.txt      2003-02-22 17:45:04.000000000 -
0600
@@ -0,0 +1 @@
+Don't forget the mayo!

Deleted: trunk/vendors/deli/chips.txt
=====

Deleted: trunk/vendors/deli/pickle.txt
=====
```

Name

`svnlook dirs-changed` — 显示本身曾更动过的目录.

摘要

```
svnlook dirs-changed REPOS_PATH
```

描述

显示本身曾更动过的目录 (性质编辑), 或是其下的档案更曾动过目录.

选项

```
--revision (-r)  
--transaction (-t)
```

范例

这会显示我们范例档案库中, 在修订版 40 中曾更动过的目录:

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos  
trunk/vendors/deli/
```

Name

`svnlook help`

摘要

亦为 `svnlook -h` 与 `svnlook -?.`

描述

显示 `svnlook` 的求助讯息. 这个命令就像它的兄弟 **`svn help`** 一样, 就算你不再与它连络, 连上次聚会都忘了邀请它, 它还是你忠实的盟友.

Name

`svnlook history` — 显示档案库中, 某个路径 (如果没有指定, 则为根目录) 的历史纪录信息

摘要

```
svnlook history REPOS_PATH
                [PATH_IN_REPOS]
```

描述

显示档案库中, 某个路径 (如果没有指定, 则为根目录) 的历史纪录信息

选项

```
--revision (-r)
--show-ids
```

范例

以下显示我们的范例档案库中, 修订版 15 的 `/tags/1.0` 路径的历史信息输出.

```
$ svnlook history -r 20 /usr/local/svn/repos /tags/1.0 --show-ids
REVISION  PATH <ID>
-----  -
      19  /tags/1.0 <1.2.12>
      17  /branches/1.0-rc2 <1.1.10>
      16  /branches/1.0-rc2 <1.1.x>
      14  /trunk <1.0.q>
      13  /trunk <1.0.o>
      11  /trunk <1.0.k>
       9  /trunk <1.0.g>
       8  /trunk <1.0.e>
       7  /trunk <1.0.b>
       6  /trunk <1.0.9>
       5  /trunk <1.0.7>
       4  /trunk <1.0.6>
       2  /trunk <1.0.3>
       1  /trunk <1.0.2>
```

Name

`svnlook info` — 显示作者, 日期戳记, 纪录讯息大小, 以及纪录讯息.

摘要

```
svnlook info REPOS_PATH
```

描述

显示作者, 日期戳记, 纪录讯息大小, 以及纪录讯息

选项

```
--revision (-r)  
--transaction (-t)
```

范例

这会显示我们范例档案库中, 修订版 40 的信息.

```
$ svnlook info -r 40 /usr/local/svn/repos  
sally  
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)  
15  
Rearrange lunch.
```

Name

svnlook log — 显示纪录讯息.

摘要

```
svnlook log REPOS_PATH
```

描述

显示纪录讯息

选项

```
--revision (-r)  
--transaction (-t)
```

范例

这会显示我们的范例档案库中, 修订版 40 的纪录讯息:

```
$ svnlook log /tmp/repos/  
Rearrange lunch.
```

Name

svnlook proplist — 显示纳入版本控制的档案与目录的性质名称与内容.

摘要

```
svnlook proplist REPOS_PATH PATH_IN_REPOS
```

描述

列出档案库中, 某个路径的性质. 加上 -v 的话, 一并显示性质内容.

选项

```
--revision (-r)  
--transaction (-t)  
--verbose (-v)
```

范例

这会显示 HEAD 修订版中, 设定给档案 /trunk/README 的性质名称:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README  
original-author  
svn:mime-type
```

与前一个例子的命令相同, 但是这次一并显示性质的内容:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README  
original-author : fitz  
svn:mime-type : text/plain
```

Name

svnlook tree — 显示档案树

摘要

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

描述

显示档案树, 自 `PATH_IN_REPOS` 开始 (如果有提供的话, 不然就从根目录开始), 外加选择性的节点修订版编号.

选项

```
--revision (-r)
--transaction (-t)
--show-ids
```

范例

这会显示我们范例档案库中, 修订版 40 的档案树 (连同节点编号一起):

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids
/ <0.0.2j>
trunk/ <p.0.2j>
  vendors/ <q.0.2j>
    deli/ <lg.0.2j>
      egg.txt <li.e.2j>
      soda.txt <lk.0.2j>
      sandwich.txt <lj.0.2j>
```

Name

svnlook youngest — 显示最年轻的修订版号.

摘要

```
svnlook youngest REPOS_PATH
```

描述

显示档案库的最年轻修订版号.

范例

这会显示我们示范档案库的最年轻修订版号:

```
$ svnlook youngest /tmp/repos/  
42
```

[\[37\]](#) 是, 是, 使用 `--version` 选项不必使用任何子命令, 不过等一下我们就会提到了.

Appendix A. 给 CVS 使用者的 Subversion 指引

Table of Contents

- [不同的修订版号](#)
- [目录版本](#)
- [更多不需网络的动作](#)
- [区分状态与更新](#)
- [分支与标记](#)
- [中介资料性质](#)
- [冲突排解](#)
- [二进制档案与转换](#)
- [Versioned Modules](#)

本附录是给刚开始接触 Subversion 的 CVS 使用者的指引. 基本上, 就是“从 10,000 呎来看”两个系统不同之处的列表. 如果可以的话, 我们会在每个章节放上相关章节的参照.

虽然 Subversion 的目标, 是接收现在与未来的 CVS 使用者基础, 但是某些新的功能与设计变更还是需要的, 以修正 CVS 特有的“不正确”行为. 也就是说, 身为一个 CVS 使用者, 你可能需要戒除一些习惯—一些你已经忘了, 在刚开始觉得很奇怪的事情.

不同的修订版号

在 CVS 中, 修订版号是每个档案不同的. 这是因为 CVS 是根基在 RCS 之上; 每一个档案在档案库都有对应的 RCS 档案, 而档案库的结构, 大致上就是依计划的目录结构展开.

在 Subversion 中,档案库看起来就像一个档案系统. 每一个送交动作, 会产生一个全新的档案系统; 本质上来说,档案库就是一个档案树的数组. 每一个这样的档案树, 都以单一修订版本号标示出来. 当某个人说“54 号修订版”时, 所指的就是某一个特定的档案树 (间接地来说, 就是第 54 次送交之后的档案系统).

技术上来说, 讲 `foo.c` 的“5 号修订版”是不正确的, 应该说“`foo.c` 在 5 号修订版的状态”. 另外, 在认定档案的演进状态时也要注意. 在 CVS 中, `foo.c` 的 5 号与 6 号修订版是一定不同的. 在 Subversion 中, `foo.c` 在 5 号与 6 号修订版, 非常有可能是没有任何更动的.

欲更深入了解这个主题, 请参阅 [the section called “修订版本”](#).

目录版本

Subversion 也会追踪档案树结构, 而不光只是档案内容而已. 这也是 Subversion 是用来取代 CVS 的最大原因.

以下是要让身为前 CVS 使用者的你, 了解这代表什么意义:

- **svn add** 与 **svn rm** 命令可使用在目录上, 就像使用在档案一样. **svn copy** 与 **svn move** 亦同. 但是这些目录 不会 马上让档案库有任何的变化. 相反地, 工作项目只是“预定”要被新增或删除. 除非你执行 **svn commit**, 不然档案库不会有任何变动.
- 目录不再只是单纯的收容物而已; 他们像档案一样, 也有修订版本号. (更适当地说, 讲“5 号修订版里的目录 `foo/` 是正确的”).

让我们对最后一点再多作说明. 目录版本控管是个困难的问题; 因为我们允许混合修订版的工作复本, 所以会有一些限制, 以防止我们破坏这样的模型.

就理论观点, 我们定义“目录 `foo` 的 5 号修订版”, 表示特定的目录项目与性质. 现在假设我们开始自 `foo` 中新增与删除档案, 然后送交更动. 说我们仍然有 5 号修订版的 `foo` 是个谎言. 但是, 如果我们在送交后, 就直接增加 `foo` 的修订版本号, 这还是个谎言; 可能还有其它我们尚未取得的 `foo` 更动, 因为我们还没进行更新.

Subversion 解决这个问题的方法, 就是闷不吭声地在 `.svn` 区域里新增与删除项目. 当你终于执行 **svn update** 后, 所有需注意的东西, 都会在档案库里尘埃落定, 而目录的修订版本号也会被正确地设定. 因此, 只有在作过更新之后, 我们才能说你有一个“正确无误”的目录修订版. 绝大多部份的时间, 你的工作复本会有“不怎么正确”的目录修订版.

类似的情形, 如果你想要送交目录的性质更动的话, 也会有问题产生. 正常的情况下, 送交动作会增加工作目录的本地修订版本号. 但是这还是个谎言, 因为你还是可能会有这个目录还没有的新增与删除, 因为你还没有作过更新. 因此, 除非目录是最新版的话, 你还是不能送交性质的更动.

欲了解有关于目录版本控管的限制, 请参见 [the section called “混合修订版的限制”](#).

更多不需网络的动作

近几年来, 磁盘空间愈来愈便宜, 愈来愈大, 但是网络频宽并非如此. 因此, Subversion 的工作复本是针对这项珍贵的资源作最佳化.

.svn 与 cvs 目录一样, 都是管理用的目录, 但是他还多存放了档案的“原始未更动”复本. 这让你能够离线进行许多事:

svn status

显示你产生的本地更动 (请参见 [the section called “svn status”](#))

svn diff

显示详细的更动细节 (请参见 [the section called “svn diff”](#))

svn revert

移除你的本地更动 (请参见 [the section called “svn revert”](#))

另外, 快取的未更动档案, 可让 Subversion 客户端在送交时仅仅传送差异即可, 这点是 CVS 作不到的.

列表最后的子命令是新的; 它不只是移除本地更动, 它还能取消像新增与删除的预定动作. 这是个较适宜的复原档案的方法; 执行 **rm file; svn update** 还是有用, 但是它会模糊更新动作的目的. And, while we're on this subject...

区分状态与更新

在 Subversion 中, 我们试着要洗刷 **cvs status** 与 **cvs update** 命令之间的混淆不清.

cvs status 命令有两个目的: 第一, 显示使用者在工作复本中的本地更动, 第二, 显示使用者的过时档案. 很不幸地, 由于 CVS 显示的状态不易理解, 许多 CVS 的使用者完全无法善用这个命令. 取而代之地, 他们发展出一个习惯, 就是执行 **cvs up** 来看他们的更动. 当然啰, 它有副作用, 就是合并你尚未准备要处理的档案库的更动.

就 Subversion 来说, 我们试着让 **svn status** 输出的数据易于让人与剖析器理解, 来解决这个不清楚的地方. 另外, **svn update** 只会显示被更新的档案信息, 而不会显示本地的更动.

以下是 **svn status** 的快速指引. 我们鼓励所有的 Subversion 使用者尽早使用, 多多使用:

svn status 显示所有有本地更动的档案; 预设不会使用到网络.

-u

增加取自档案库的是否过时的数据.

-v

显示 *所有* 纳入版本控制的项目.

-N

非递归式的.

status 命令有两种输出格式. 预设是“简短”格式, 本地更动看起来像这样:

```
% svn status
M      ./foo.c
M      ./bar/baz.c
```

如果你指定了 **--show-update (-u)** 选项, 会使用较长的输出格式:

```
% svn status -u
M      1047      ./foo.c
      *      1045      ./faces.html
      *      -      ./bloo.png
M      1050      ./bar/baz.c
Head revision: 1066
```

在这个例子中, 出现了两个新的字段. 如果档案或目录是过时的话, 第二栏会有星号. 第三个字段显示该项目在工作复本的修订版号. 在上面的例子中, 星号表示更新会让 `faces.html` 取得更动修补, 而 `bloo.png` 是一个档案库新增的档案. (在 `bloo.png` 旁的 `-`, 表示它尚未出现在工作复本中.)

最后, 以下是一份简短的摘要, 说明你可能最常会看到的状态码:

A	新增
D	删除
R	取代 (删除, 然后重新加入)
M	本地更动
U	已更新
G	被合并
C	冲突
X	外部

- A 预计要新增的资源
- D 预计要删除的资源
- M 有本地更动的资源
- C 有冲突的资源（档案库与工作复本之间的更动无法完全地合并）
- X 本工作复本的外部资源（来自其它的档案库，请参见
[the section called "svn:externals"](#)）
- ? 未纳入版本控制的资源
- ! 遗失或不完整的资源（被 Subversion 以外的工作移除）

Subversion 将 CVS 的 **P** 与 **U** 结合成一个 **U**. 当合并或冲突发生时, Subversion 只会显示 **G** 或 **C**, 而不是冗长的一句话.

欲了解更多有关 **svn status** 的详细讨论, 请参照 [the section called "svn status"](#).

分支与标记

Subversion 不会区分档案系统空间与“分支”空间的差别; 分支与标记都只是档案系统里的普通目录而已. 这也许是 CVS 使用者所需跨越的最大心理障碍. 请参阅 [Chapter 4, 分支与合并](#) 以了解详情.

中介资料性质

Subversion 的一个新功能, 就是你可以将任何的中介资料, 附加到档案与目录上. 我们偏好称此资料为 *性质*, 而它们可以想成是附加到工作复本中, 随意的名称/数值对的集合.

要设定或取得性质的名称, 可使用 **svn propset** 与 **svn propget** 子命令. 要列出一个对象上所有的性质, 可使用 **svn proplist**.

欲取得更多的信息, 请参见 [the section called "性质"](#).

冲突排解

CVS 会在档案内放置“冲突标记”, 将冲突地方标示出来, 在更新时显示一个 **C** 代码. 就以前的记录来看, 这导致很多的问题, 因为 CVS 作的并不够. 许多使用者忘了 (或没看到) 在终端机上快速闪掉的 **C** 代码. 使用者也常常忘了有冲突标记的存在, 然后就不小心将含有冲突标记的档案送交回去.

Subversion 解决这个问题的方法, 是让冲突更明确地表示出来. 它会记得档案是处于冲突的状态中, 除非你执行了 **svn resolved**, 它不会允许你送交更动. 请参见 [the section called "解决冲突 \(合并他人的更动\)"](#) 以了解更多细节.

二进制档案与转换

一般而言, Subversion 比 CVS 更能优雅地处理二进制档案. 因为 CVS 使用 RCS 的关系, 对于一个变动中的二进档案, 它只能将每个更动的复本都储存下来. 但是 Subversion 不管档案是文字或是二进制类型, 在内部都是以二进制差异比较算法来表示档案的差异. 这表示所有的档案在档案库中, 都是以 (压缩的) 差异来储存的, 而且在网络上传输的, 都是较小的档案差异而已.

CVS 使用者必须以 `-kb` 标记二进制档案, 以避免数据被搞烂 (因为关键词展开, 以及列尾符号转换的关系). 他们有时候会忘了作这件事.

Subversion 实行比较疯狂的行径: 首先, 它绝不进行任何的关键词或列尾符号转换, 除非你要求它这么作 (请参见 [the section called “svn:keywords”](#) 与 [the section called “svn:eol-style”](#)). Subversion 预设会将所有的数据视为字面字节字符串, 而档案都会以未转换的状态, 储存在档案库中.

再者, Subversion 内部会维护记录档案是否为“文字”或“二进制”数据的记录, 但是这项记录 只会存在于工作复本中. 在执行 **svn update** 的过程中, Subversion 会对本地的文本文件进行内容合并, 但是不会对二进制档案作这样的事.

要决定内容合并是否可行, Subversion 会检视 `svn:mime-type` 性质. 如果档案没有 `svn:mime-type` 性质, 或是有一个文字的 `mime` 类型 (例, `text/*`), Subversion 就会假设它是文字. 不然的话, Subversion 会假设它是二进制的. 在 **svn import** 与 **svn add** 中, Subversion 会进行二进制侦测算法来帮助使用者. 这些命令会产生一个不错的猜测, 然后 (可能) 对要加入的档案设定二进制的 `svn:mime-type` 性质. (如果 Subversion 猜错了, 使用可以自行移除该性质, 或是手动移除它.)

Versioned Modules

Unlike CVS, a Subversion working copy is aware that it has checked out a module. That means that if somebody changes the definition of a module, then a call to **svn update** will update the working copy appropriately.

Subversion defines modules as a list of directories within a directory property: see [the section called “外部定义”](#).

Appendix B. 汇入 CVS 档案库

Table of Contents

[需求](#)
[执行 cvs2svn.py](#)

由于 Subversion 是设计成为 CVS 的继任者, 只有提供汇入的工具才有意义. Subversion 有命令稿可以将 CVS 档案库汇入至 Subversion 档案库. 是的, 你 可以带着 CVS 的历史一起进入美丽新世界.

需求

这工具称为 `cvs2svn.py`, 它是个 Python 命令稿, 位于 Subversion 源码树的 `tools` 子目录里. 要执行这个程序, 你需要一些外部的东西:

python 2.0

确定你有安装 python 2.0 或更新版本. 你可以从 <http://www.python.org/> 取得最新版.

rcsparse.py

这是个用来剖析 RCS 档案的 python 模块, 也是 ViewCVS 项目的一部份. 我们需要它来读取你的 CVS 模块. 为了方便起见, 一份复本置于 `cvs2svn.py` 相同目录中, 不过更实时的版本可以从 ViewCVS 项目取得 — <http://viewcvs.sf.net/> .. 只要把这个模块放在 python 可以找得到的地方即可, 像是 `/usr/local/lib/python2.2/`.

执行 `cvs2svn.py`

由于 CVS 没有不可分割的送交, `cvs2svn.py` 很难将其推衍出来. 它是藉由检视 RCS 档案, 然后找出每一个档案修订版的相同送交讯息. 如果两个 RCS 修订版有着相同的送交讯息, 又差不多在相同的时间送交 (差不多彼此相差几分钟的时间), `cvs2svn.py` 会将它们置于一个共同的更动“群组”, 然后将送交群组以单一修订版送交至新的 Subversion 档案库.

前面的解释有点简化; 它要记录的东西比那还要多得多. 事实上, `cvs2svn.py` 以许多不同的“阶段”在磁盘上建立大量的暂存数据, 来进行它的工作. 如果你中断该命令稿, 你可以稍后传递 `<options>-p</options>` 选项给命令稿, 表示它应该从哪一个阶段继续下去.

执行该命令稿很简单:

```
$ svnadmin create /new/svn/repos
$ cvs2svn.py -s /new/svn/repos /cvs/repos
...
```

转换可能要花数分钟, 到十几个小时不等, 完全视你的 CVS 档案库大小而定. 对 Subversion 的 CVS 档案库执行时 (Subversion 第一年的历史, 当时还无法以自己来管理), 花了 30 分钟, 送交了大概 3000 个修订版到 Subversion 档案库.

Appendix C. 故障排除

Table of Contents

常见问题

使用 Subversion 的问题

每当我想要存取档案库时, 我的 Subversion 客户端会停在那里.

当我想要执行 `svn` 时, 它就说我的工作复本被锁定了.

寻找或开启档案库时有错误发生, 但是我确定我的档案库 URL 是正确的.

我要如何在 `file://` URL 中指定 Windows 的磁盘驱动器代号?

我没有办法经由网络写入数据至 Subversion 档案库.

在 Windows XP 中, Subversion 服务器有时会送出损坏的数据.

要在 Subversion 客户端与服务器进行网络传输的检查, 最好的方法是什么?

编译 Subversion 的问题

我把执行档编辑好了, 但是当我想要取出 Subversion 时, 我得到

Unrecognized URL scheme. 的错误.

当我执行 `configure`, 我得到像 `subs-1.sed line 38: Unterminated `s' command` 的错误.

我无法在 Windows 以 MSVC++ 6.0 来编译 Subversion.

常见问题

在安装与使用 Subversion 时, 你可能会遇到一些问题. 在你熟悉 Subversion 如何运作之后, 有些问题就会解决, 而其它问题发生的原因, 是因为你习于别种版本控制系统运作的关系. 还有因为某些 Subversion 所执行的操作系统的关系, 有一些问题是无法解决的 (思考一下 Subversion 可以执行的操作系统有多少, 有趣的是, 有许多是我们没用过的).

以下的列表, 是使用 Subversion 数年经验中所编纂的, 它们涵盖的范围, 就是你在 Subversion 经常会遇到的问题 — 编译, 安装, 使用. 如果你无法在这里找到你遇到的问题, 或是已经试过了所有我们给的建议, 但是徒劳无功的话, 请寄电子邮件至 [<users@subversion.tigris.org>](mailto:users@subversion.tigris.org), 并且详加描述你的问题 ^[38]

使用 Subversion 的问题

- 每当我想要存取档案库的时候, `svn` 就会停在那里.
- 当我想要执行 `svn` 时, 它就说我的工作复本被锁定了.
- 寻找或开启档案库时有错误发生, 但是我确定我的档案库 URL 是正确的.
- 我要如何在 `file://` URL 中指定 Windows 的磁盘驱动器代号?
- 我没有办法经由网络写入数据至 Subversion 档案库.
- 在 Windows XP 中, Subversion 服务器有时会送出损坏的数据.
- 要在 Subversion 客户端与服务器进行网络传输的检查, 最好的方法是什么?

编译 Subversion 的问题

- 我把执行档编译好了, 但是当我想要取出 Subversion 时, 我得到“Unrecognized URL scheme”的错误.
- 当我执行 `configure`, 我得到像 `subs-1.sed line 38: Unterminated `s' command` 的错误.

- 我无法在 Windows 以 MSVC++ 6.0 编译 Subversion.

使用 Subversion 的问题

每当我想要存取档案库时, 我的 Subversion 客户端会停在那里.

其它想要存取档案库的客户端就会直接卡住, 等锁定档消失. 要叫醒你的档案库, 你需要告诉 Berkeley DB 结束该项工作, 或是把数据库回复到先前已知正确的状态. 要达到这个目的, 从含有 Subversion 档案库的机器上执行以下的命令:

```
$ svnadmin recover /path/to/repos
```

在作这件事之前, 请确定你把所有档案库的存取方式都关掉 (关掉 httpd 或 svnserve). 确定你以拥有与管理该数据库的使用者来执行这个命令, 而且也不要以 root 来进行, 不然这样会在 db/ 目录里留下 root 拥有的档案, 这样会让非 root 的使用者无法开启, 这表示是你, 或是 Apache 的 httpd 行程.

当我想要执行 svn 时, 它就说我的工作复本被锁定了.

Subversion 的工作复本就像 Berkeley DB 一样, 使用日志机制来进行所有的工作. 也就是说, 它会在进行工作之前, 先把它纪录下来. 如果 svn 在进行工作时被中断, 那会留下一个或多个锁定档案, 以及描述尚未结束的档案. (svn status 会在被锁定的目录旁边显示 L).

其它想要存取工作复本的行为在遇到锁定时, 就会失败. 要叫醒你的工作复本, 你需要叫 svn 客户端来结束该项工作. 要修正这个问题, 请从工作复本最上层的目录执行这个命令:

```
$ svn cleanup working-copy
```

寻找或开启档案库时有错误发生, 但是我确定我的档案库 URL 是正确的.

See [the section called “每当我想要存取档案库时, 我的 Subversion 客户端会停在那里.”](#).

请参考 [the section called “每当我想要存取档案库时, 我的 Subversion 客户端会停在那里.”](#).

我要如何在 file:// URL 中指定 Windows 的磁盘驱动器代号?

请参见 [档案库 URL](#).

我没有办法经由网络写入数据至 Subversion 档案库.

如果以本地存取进行汇入是没有问题的话:

```
$ mkdir test
$ touch test/testfile
$ svn import test file:///var/svn/test -m "Initial import"
Adding          test/testfile
Transmitting file data .
Committed revision 1.
```

但是无法经由远端主机:

```
$ svn import test http://svn.red-bean.com/test -m "Initial import"
harry's password: xxxxxxxx

svn_error: #21110 :

The specified activity does not exist.
```

我们看过这个问题, 它发生在 `REPOS/dav/` 目录无法被 `httpd` 行程写入时. 请检查档案权限, 确定 Apache `httpd` 可以写入 `dav/` 目录 (当然还有对应的 `db/` 目录).

在 Windows XP 中, Subversion 服务器有时会送出损坏的数据.

你需要安装 Window XP Service Pack 1. 你可以在 <http://support.microsoft.com/default.aspx?scid=kb;EN-US;q317949> 取得所有有关该 Service Pack 的信息.

要在 Subversion 客户端与服务器进行网络传输的检查, 最好的方法是什么?

Use Ethereal to eavesdrop on the conversation:

使用 Ethereal 来偷听对话:

Note

以下的指令是特别针对 Ethereal 的图形接口, 可能无法套用在命令列版本上 (其二进制程序通常名为 **tethereal**).

- 拉下 Capture 目录, 选择 Start.
- 在 Filter 输入 port 80, 然后关掉 promiscuous 模式.
- 执行你的 Subversion 客户端.
- 按下 Stop. 现在你就有了封包撷取. 它看起来是一大串的文字列.
- 按下 Protocol 字段, 对其排序.
- 然后, 点选第一个 TCP 列.
- 按右键, 然后选择 Follow TCP Stream. 你会得到 Subversion 客户端的 HTTP 对话的要求响应对.

另外,你也可以在服务器设定档中设定 `http-debug` 参数,让你执行 `svn` 客户端时,会显示 `neon` 的除虫讯息. `neon-debug` 的数值是 `ne_utils.h` 标头文件的 `NE_DBG_*` 数值的组合. 设定 `http-debug` 为 130 (意即 `NE_DEBUG_HTTP + NE_DBG_HTTPBODY`) 会让 `HTTP` 数据也显示出来.

你可能在进行网络传输检查时,需要关闭压缩的功能. 请参考 `config` 设定档中的 `compression` 参数.

编译 Subversion 的问题

我把执行档编辑好了,但是当我想要取出 `Subversion` 时,我得到 “`Unrecognized URL scheme.`” 的错误.

`Subversion` 使用外挂系统来存取档案库. 目前有三个这样的外挂: `ra_local` 可以存取本地档案库, `ra_dav` 可以透过 `WebDAV` 存取档案库, 而 `ra_svn` 可以透过 `svnserve` 服务器来进行本地或远程的存取. 当你想要在 `Subversion` 进行一个作业时, 客户端会试着依 `URL schema` 动态加载一个外挂. `file:// URL` 会试着载入 `ra_local`, 而 `http:// URL` 会试着输入 `ra_dav`, 以此类推.

你看到的这个错误, 表示动态连结器/加载器无法找到要载入的外挂. 这个发生的原因, 通常是因为你以共享链接库的方式编译 `Subversion`, 但是还没有执行 `make install` 就要执行它. 另一个可能就是你执行了 `make install`, 但是链接库把它存在动态连结器/加载器不认得的地方. 在 `Linux` 下, 你可以把那个链接库目录加进 `/etc/ld.so.conf`, 然后执行 `ldconfig`, 让连结器/加载器可以找到链接库. 如果你不想这么作, 或是你没有 `root` 存取权限, 你可以在 `LD_LIBRARY_PATH` 环境变量指定该链接库目录.

当我执行 `configure`, 我得到像 `subs-1.sed line 38: Unterminated `s' command` 的错误.

你可能在系统上有 `/usr/local/bin/apr-config` 与 `/usr/local/bin/apu-config` 旧的复本. 移除它们, 确定你正在编译的 `apr/` 与 `apr-util/` 是最新的版本, 然后再试一次.

我无法在 `Windows` 以 `MSVC++ 6.0` 来编译 `Subversion`.

你必须取得最新的操作系统 SDK. 随 `VC++ 6.0` 附的并不够新.

[38] 请记住, 你所提供的设定与问题的详细程度, 与从邮件论坛得到回答的机率成正比. 我们鼓励你附上所有的信息, 除了早餐吃什么和令堂闺名.

Appendix D. WebDAV 与自动版本

Table of Contents

[基本 WebDAV 概念](#)

[简易 WebDAV](#)

[DeltaV 扩充](#)

[Subversion 与 DeltaV](#)

[将 Subversion 对映至 DeltaV](#)

[自动版本支持](#)

[mod_dav_lock 的替代品](#)

[自动版本互通性](#)

[Win32 网络数据夹](#)

[Mac OS X](#)

[Unix: Nautilus 2](#)

[Linux davfs2](#)

WebDAV 是 HTTP 的扩充语法, 愈来愈多人把它拿来当作档案分享的标准. 今日的操作系统愈来愈注重在 Web 功能, 很多现在都有内建支持挂载 WebDAV 服务器汇出的“分享点”.

如果你使用 Apache/mod_dav_svn 作为 Subversion 的网络服务器, 就某种程度上来说, 你就是在执行 WebDAV 服务器. 这篇附录对本通讯协议提供了一些基本背景数据, Subversion 如何使用它, 以及 Subversion 如何与其它使用 WebDAV 软件一起协同工作.

基本 WebDAV 概念

本节对 WebDAV 的背景提供了很简单的基本概念. 它对了解 WebDAV 客户端与服务器之间的兼容课题提供了基础知识.

简易 WebDAV

RFC 2518 定义了对 HTTP 1.1 的概念与扩充功能, 让网站能够变成更统一的读写媒介. 背后的基本概念, 是一个了解 WebDAV 的网站服务器可以当作一个通用的档案服务器; 客户端可以挂载 WebDAV “分享”, 就像 NFS 或 SMB 分享一样.

但是我们必须注意, 在 RFC 2518 之中, 除了 DAV 里的“V”以外, 它并未提供任何形式的版本控制模型. 基本 WebDAV 客户端与服务器都假设每个目录或档案都只有一个版本存在而已, 但是可以一直被盖写过去.^[39]

以下为基本 WebDAV 所提出的概念与方法:

新的写入方法

除了标准的 HTTP PUT 方法 (它会建立或覆写一个网页资源), WebDAV 还定义了新的 COPY 与 MOVE 方法, 用以复制与重新安排资源.

集合

这只是 WebDAV 的术语, 用来将资源 (URI) 群组在一起. 在大多数的情况下, 这就类似像“档案目录”一样. 你可以藉由结尾的 “/”, 来辨别它是不是一个群组. 档案资源可以藉由 PUT 方法对其盖写或建立, 不过集合资源则以 MKCOL 方法来建立.

性质

这与 Subversion 中的作法是一样的 — 连系至集合与档案的描述数据. 客户端可以利用新的 PROPFIND 方法列出或取得连系至某个资源的性质, 也可以利用 PROPPATCH 方法来变更. 某些性质完全是被使用者控制的 (像是被称为 “color” 的性质), 而有些则是完全被 WebDAV 服务器建立与控制的 (像是存有最近一次更动档案的时间). 前者被称为 “dead” 性质, 而后者被称为 “live” 性质.

锁定

一个 WebDAV 服务器可能会提供锁定功能供客户端使用 — 这个部份的规格是选用的, 不过大部份的 WebDAV 服务器都会提供这样的功能. 如果有的话, 客户端可以使用新的 LOCK 与 UNLOCK 方法, 来协调对某资源的存取. 在大多数的情况下, 这些方法都是用来建立独占式写入锁定 (像是 [the section called “锁定-修改-解锁的解决方案”](#) 所讨论的情况), 不过也有可能是分享式写入锁定.

DeltaV 扩充

由于 RFC 2518 并未提及版本控制的概念, 另一个有能力的团体就写出了 RFC 3256, 它对 WebDAV 加上了版本控制. WebDAV/DeltaV 客户端与服务器通常只被称为 “DeltaV” 客户端与服务器, 这是因为 DeltaV 就已隐含了基本 WebDAV 的存在.

DeltaV 又引进了一狗票的缩写, 不过别害怕, 它们都满简单的. 以下是 DeltaV 引进的新概念与方法:

各资源的版本控制

如同 CVS 与其它的版本控制系统一样, DeltaV 假设每一项资源都有可能无限数量的状态. 客户端一开始以新的 VERSION-CONTROL 方法, 将一个资源置于版本控制下. 这会建立一个新的版本控制资源 (Version Controlled Resource, VCR) 每一次你更动 VCR (透过 PUT, PROPPATCH 等等), 该资源就会建立一个新状态, 称为版本资源 (Version Resource, VR). VCR 与 VR 仍然是一般的网页资源, 藉由 URL 来使用. 某些特定的 VR 也可以拥有人类易用的名称.

伺服端的工作复本模型

有些 DeltaV 服务器允许在服务器上建立一个虚拟“工作空间”，你的所有工作都可以在那里进行。客户端使用 MKWORKSPACE 建立一个私有区域，藉由“取出特定 VCR”至工作空间，表示想要更改它们，对其编辑，然后再“存入它们”。以 HTTP 的术语来说，这些方法的顺序为 CHECKOUT, PUT, CHECKIN。在每一个 CHECKIN 之后，新的 VR 会被建立，被编辑的 VCR 的内容会“指向”最新的 VR。每一个 VR 也会有一个“历史”资源，它是用来追踪并记录各个 VR 状态。

客户端工作复本模型

有些 DeltaV 服务器也支持客户端可对特定的 VR 拥有自己的工作复本。（这就是 CVS 与 Subversion 运作的方法。）当客户端想要送交更动至服务器时，它一开始会先以 MKACTIVITY 方法，建立一个暂时的服务器异动（称为 activity）。接着，客户端会对每一个打算更动的 VR 进行 CHECKOUT，它会在 activity 中建立数个暂时的“工作资源”，然后可以利用 PUT 与 PROPPATCH 方法进行修改。最后，客户端会对每一个工作资源进行 CHECKIN，这会在每一个 VCR 中建立一个新的 VR，然后整个 activity 就会被删除。

配置

DeltaV 允许你定义包含 VCR 的弹性配置，称为“配置”，它并不一定必须对应到某些特定的目录。每一个 VCR 的内容可以透过 UPDATE 方法，对应到特定的 VR。只要配置没有问题的话，客户端可以建立整个配置的“快照”，称为“baseline”客户端使用 CHECKOUT 与 CHECKIN 方法以取得某项配置的状态，与使用这些方法来建立特定 VCR 的 VR 状态是一样的。

扩充性

DeltaV 定义了一个新的方法 REPORT，它可以让客户端与服务器进行自订的资料交换。客户端送出 REPORT 要求，以及包含许多自订数据的 properly-labeled XML 讯息；假设服务器能够了解该特定的报告类型，它会以相同的自订 XML 讯息响应。这个技巧与 XML-RPC 是非常相似的。

自动版本

对许多人而言，这是 DeltaV 的“必杀”功能。如果 DeltaV 服务器支持这个功能的话，那个基本的 WebDAV 客户端（也就是没有版本控制）还是可以对服务器写入数据，服务器就会安安静静地进行版本控制。以最简单的例子来说，由基本 WebDAV 客户端送出的普通 PUT 命令，可能会被这样的服务器解释为 CHECKOUT, PUT, CHECKIN。

Subversion 与 DeltaV

那么, Subversion 与其它的 DeltaV 软件有多“兼容”呢? 简单地说, 不很高. 至少 Subversion 1.0 如此.

虽然 `libsvn_ra_dav` 会对服务器送出 DeltaV 要求, Subversion 客户端并不是通用的 DeltaV 客户端. 事实上, 它预期服务器会有某些自订的功能 (尤其是自订的 `REPORT` 要求). 另外, `mod_dav_svn` 并不是一个通用的 DeltaV 服务器. 它只有实作 DeltaV 规格的一小部份而已. 更通用的 WebDAV 或 DeltaV 客户端可能更能够与它工作, 但是仅限于该客户端是在服务器的有限实作功能之内运作的. Subversion 发展团队计划在未来的 Subversion 版本中, 会特别着重在一般 WebDAV 的通用性.

将 Subversion 对映至 DeltaV

以下是 Subversion 客户端的动作如何使用 DeltaV 的“高阶”描述. 在许多情况下, 这些解释只是过份简化的粗略说明. 它们不应该被视为阅读 Subversion 源码, 或是与发展人员交换意见的替代品.

`svn checkout/list`

对指定集合进行深度为 1 的 `PROPFIND` 动作, 以取得其下的子项. 对每一个子项进行 `GET` (也许还有 `PROPFIND`). 会递归进入集合再继续前述动作.

`svn commit`

以 `MKACTIVITY` 建立 `activity`, 再对每一个更动的项目进行 `CHECKOUT`, 接着对新数据进行 `PUT`. 最后, 一个 `MERGE` 要求, 会导致对所有工作资源作一个隐含的 `CHECKIN`.

`svn update/switch/status/merge/diff`

送出一个自订的 `REPORT` 要求, 它会描述工作复本的混合修订版 (还有混合 URL) 的状态. 服务器会送出自订响应, 说明哪些项目应该要更新. 客户端会检查每一个响应, 依需要进行 `GET` 与 `PROPFIND`. 对 `update` 与 `switch` 动作, 会将新数据安装于工作复本中. 对 `diff` 与 `merge` 命令, 比较数据与工作复本, 也许会套用更动为本地更动.

自动版本支持

在写作之时, 实际上并没有多少 DeltaV 客户端存在; RFC 3256 还是很新的. 不过使用者还是可以取得“通用”的客户端, 因为几乎所有现代操作系统都有一个整合的基本 WebDAV 客户端. 了解到这一点, Subversion 发展人员发现, 如果 Subversion 1.0 要有任何互通功能的话, 支持 DeltaV 的自动版本功能是最好的策略.

要开启 `mod_dav_svn` 的自动版本功能, 请在 `httpd.conf` `Location` 区块里使用 `SVNAutoversioning`, 像这样:

```
<Location /repos>
  DAV svn
  SVNPath /absolute/path/to/repository
  SVNAutoversioning on
</Location>
```

由于许多操作系统已经整合了 **WebDAV** 功能, 这个功能的使用情况, 感觉就像天方夜谭一样: 想象一间办公室, 其中的有使用微软 **Windows** 或 **Mac OS** 的一般使用者. 每一台计算机都“挂载” **Subversion** 的档案库, 它会变成一般的网络分享. 然后就像平常一样使用这台服务器: 从服务器开启档案, 进行编辑, 然后把它们存回服务器. 但是在这个幻想中, 服务器会自动地对任何东西进行版本控制. 然后系统管理员就可以使用 **Subversion** 客户端, 寻找并取得所有旧的版本.

这个幻想是真的吗? 并不完全是. 最大的阻碍是 **Subversion 1.0** 并不支持 **WebDAV** 的 `LOCK` 或 `UNLOCK` 方法. 大多数操作系统的 **DAV** 客户端, 都会试着对一个直接从 **DAV** 挂载网络分享开启的资源进行 `LOCK`. 自此, 使用者可能得从 **DAV** 分享复制档案至本地磁盘, 编辑档案, 然后再把它复制回去. 不是个很理想的自动版本, 不过还是可行的.

mod_dav_lock 的替代品

`mod_dav` **Apache** 模块是很复杂的怪兽: 它能够了解, 并且剖析所有 **WebDAV** 与 **DeltaV** 的方法, 但是它还是要依靠后端的“**provider**”来存取资源.

举个最简单的例子, 使用者可以使用 `mod_dav_fs` 作为 `mod_dav` 的 **provider**. `mod_dav_fs` 使用一般的档案系统来储存目录与档案, 而且也只认得普通的 **WebDAV**, 而非 **DeltaV**.

但是 **Subversion** 则是使用 `mod_dav_svn` 作为 `mod_dav` 的 **provider**. `mod_dav_svn` 了解 `LOCK` 以外的方法, 而且也了解大半的 **DeltaV** 方法. 它会存取 **Subversion** 档案库中的数据, 而不是直接位于真实档案系统的数据. **Subversion 1.0** 不支持锁定, 这是因为 **Subversion** 使用复制-修改-合并模式, 实作上相当困难.^[40]

在 **Apache httpd-2.0** 中, `mod_dav` 支持 `LOCK` 的方法, 是藉由在私有数据库中追踪锁定, 它假设 **provider** 可以接受这样的要求. 但是在 **Apache httpd-2.1** 或其后的版本, 这个锁定支持就分离至一个独立的 `mod_dav_lock` 模块中. 它允许任何的 `mod_dav` **provider** 善用锁定数据库, 包括 `mod_dav_svn` 在内, 即使 `mod_dav_svn` 并不真的了解锁定.

搞糊涂了没?

简单地讲, 你可以在 **Aapche httpd-2.1** (或其后的版本) 使用 `mod_dav_lock`, 制造 `mod_dav_svn` 能够处理 `LOCK` 要求的假象. 请确定 `mod_dav_lock` 被编译在 `httpd`,

或是在 httpd.conf 载入. 那么只要在你的 Location 中加入 DAVGenericLockDB 指令, 像这样:

```
<Location /repos>
  DAV svn
  SVNPath /absolute/path/to/repository
  SVNAutoversioning on
  DavGenericLockDB /path/to/store/locks
</Location>
```

这个技巧是危险的交易: 某种方面来说, mod_dav_svn 是在对 WebDAV 客户端扯谎. 它宣称可以接受 LOCK 要求, 但是实际上, 并不是所有的阶段都有锁定. 如果有第二个 WebDAV 客户端试着对要同一个资源进行 LOCK 操作, 那么 mod_dav_lock 会发现, 然后正确地否决该要求. 但是即使如此, 还是挡不住以普通的 Subversion 客户端执行正常的 **svn commit** 而进行的档案更动! 如果你使用这样的技巧, 使用者就很有可能互相盖写彼此的更动. 特别的是 WebDAV 客户端很有可能不小心盖写普通 svn 客户端所送交的更动.

另一方面, 如果你很小心地设定你的环境, 这样的风险是能够降低的. 举个例子, 如果所有的用户都透过基本的 WebDAV 客户端 (而不是 svn 客户端), 那么就不会有什么问题.

自动版本互通性

在本节中, 我们会描述 (写作此时) 最常见的通用 WebDAV 客户端, 还有它们与使用 SVNAutoversioning 指令的 mod_dav_svn 服务器的互通性. RFC 2518 有点大, 也许也有点太过于弹性了. 每一个 WebDAV 客户端的行为都有点不一样, 也有各自不同的问题.

Win32 网络数据夹

Windows 98, 2000 与 XP 都有整合的 WebDAV 客户端, 称为“网络数据夹”. 在 Windows 98 上, 这个功能需要另外安装; 如果有的话, 在我的计算机中, 会有一个“网络数据夹”目录出现. 在 Windows 2000 与 XP 上, 只要开启网络芳邻, 然后执行新增网络位置图标即可. 在提示输入时, 键入 WebDAV 的网址. 分享出去的数据夹会出现在网络芳邻中.

对自动版本的 mod_dav_svn 服务器进行写入可以运作地很好, 不过有几个问题:

- 如果计算机是 NT 网域的一员, 那么就似乎无法连上 WebDAV 分享. 它要不停地要求使用者名称与密码, 就算 Apache 服务器并未送出认证要求! 有些人指出, 这个问题发生的原因, 可能是因为网络数据夹是设计来与 Microsoft 的 SharePoint DAV 服务器一起运作的. 如果机器不是 NT 网域的一部份, 那么挂载该分享就不会有任何的问题. 这个神秘事件尚未解决.

- 一个档案无法直接由分享进行直接编辑; 它都会以只读的方式开启。
`mod_dav_lock` 的技巧无法提供任何帮助, 因为网络数据夹完全不使用 `LOCK` 方法. 不过原先提到的“复制, 编辑, 重新复制”方法倒是可以使用. 分享上的档案, 可以被本地编辑复本盖写.

Mac OS X

Apple 的 OS X 操作系统有个整合的 WebDAV 客户端. 从 Finder 中, 从 Go 选单选取 “Connect to Server” 项目. 输入 WebDAV 网址, 然后它会以磁盘出现在桌面上, 就跟其它的档案服务器一样.^[41]

很不幸地, 这个客户端无法与自动版本 `mod_dav_svn` 工作, 因为它缺少 `LOCK` 的支持. Mac OS X 会在一开始的 `HTTP OPTIONS` 功能交换时, 发现没有 `LOCK` 功能, 因此决定将 Subversion 档案库以只读分享挂载起来. 之后, 就完全无法进行写入作业. 要将档案库以读写分享挂载的话, 你 *必须* 使用前面讨厌的 `mod_dav_lock` 技巧. 当锁定能够工作之后, 该分享就能正常地工作: 档案可以直接以读写模式开启, 只是每一次写入的作业, 会让客户端以 `PUT` 移到暂存位置, `DELETE` 删除原来的档案, 然后 `MOVE` 将暂存资源移回原来的文件名称. 每一次存档, 就是三个新的 Subversion 修订版!

还有一件事要注意: OS X 的 WebDAV 客户端对 HTTP 的重导向太过敏感. 如果你无法挂载档案库, 你可能需要在你的 `httpd.conf` 中, 加上 `BrowserMatch` 指令:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Unix: Nautilus 2

Nautilus 是 GNOME 桌面环境的官方档案管理员/浏览器, 其网页位于 <http://www.gnome.org/projects/nautilus/>. 只要将 WebDAV 网址输入 Nautilus 窗口, DAV 分享就会以本地档案系统出现.

一般来说, Nautilus 2 与自动版本 `mod_dav_svn` 可以合作无间, 不过有以下的例外:

- 任何直接从该分享开启的档案, 都会被视为只读. 即使是 `mod_dav_lock` 技巧也发挥不了作用. 感觉上, 似乎 Nautilus 完全不会送出 `LOCK` 方法. 不过“本地复制, 编辑, 复制回去”技巧还是可以工作. 很不幸地, Nautilus 会先送出一个 `DELETE` 方法, 这样会产生一个额外的修订版.
- 在覆写或建立一个档案时, Nautilus 首先会对一个空档案作 `PUT`, 然后再以第二个 `PUT` 盖写它. 这会建立两个 Subversion 档案系统修订版, 而不是一个.
- 在删除一个集合时, 它会对每一个子项目发出 `HTTP DELETE`, 而不是对该集合本身. 这样会产生一大堆的新修订版.

Linux davfs2

Linux davfs2 是供 Linux 核心使用的档案系统模块, 其发展位于 <http://dav.sourceforge.net/>. 安装之后, 就可以使用 Linux 的 **mount** 命令来挂载 WebDAV 网络分享.

有谣言说这个 DAV 客户端完全无法与 `mod_dav_svn` 的自动版本工作. 在每一次尝试对服务器写入之前, 都会先发出 LOCK 要求, 而这是 `mod_dav_svn` 不支持的. 到目前为止, 还没有任何数据显示使用 `mod_dav_lock` 能够解决这个问题.

^[39] 因为这个理由, 有些人戏称普通的 WebDAV 客户端为 “WebDA” 客户端!

^[40] 也许哪天 Subversion 会发展出可以与复制-修改-合并共存的 reserved-checkout 锁定模型, 但是不会马上成真.

^[41] Unix 使用者也可以执行 `mount -t webdav URL /mountpoint`.

Appendix E. 其它 Subversion 客户端

Table of Contents

[Out of One, Many](#)

Out of One, Many

[Subversion](#) 命令列客户端 `svn` 是正式 ^[42] 的 Subversion 客户端实作程序. 很幸运地, 为了那些有兴趣发展 Subversion 客户端的人们, Subversion 实作成一系列的链接库. 这些链接库除可过透过 C API 存取外, 还可被其它语言使用 (参见 [the section called “Using Languages Other than C and C++”](#)).

这样的组件设计, 表示它很容易 (呃, 至少 *比较容易*) 利用这些链接库来写出客户端. 所得到的结果, 就是在 1.0 之前, 有了许多为 Subversion 所用的 GUI 客户端, 每一个目前都在不同的发展阶段.

Table E.1. Subversion 的图形客户端

名称	语言	可移植性	授权	网址
RapidSVN	C++	是, 原生 widget	Apache-style	http://rapidsvn.tigris.org/
gsvn	Python	仅 Unix, 非原	GPL	http://gsvn.tigris.org/

名称	语言	可移植性	授权	网址
		生 widget		
TortoiseSVN	C++	仅 Win32	GPL	http://tortoisesvn.tigris.org/
svnup	Java, JNI bindings	Yes	Apache- style	http://svnup.tigris.org/
jsvn	Java, 包 装 svn 命令列 客户端	Yes	Academic Free License	http://jsvn.alternatecomputing.com

^[42] 我们说了算。

Appendix F. 协力厂商工具

Table of Contents

[ViewCVS](#)
[SubWiki](#)

Subversion 的模块化设计 (涵盖于 [the section called “Layered Library Design”](#) 中) 与程序语言系结功能 (描述于 [the section called “Using Languages Other than C and C++”](#)), 让 Subversion 适合扩充其它程序的功能, 或是成为其它软件的后端. 在本附录中, 我们会介绍几个使用 Subversion 的协力厂商开发的工具. 我们不会涵盖真正的 Subversion 客户端 — 请另行参考 [Appendix E, 其它 Subversion 客户端](#).

ViewCVS

也许第一个 — 而且绝对也是最受欢迎的 — 充份使用到 Subversion 的公用 API 的工具, 就是 ViewCVS. ViewCVS 其实是一个 CGI 命令稿, 允许浏览一个版本控制系统的目录与档案. 原先是设计为以 Python 写成, 用来取代 cvsweb 的工具.^[43] ViewCVS 对 CVS 档案库提供了完整功能的网页界面, 允许别人可以查看这些档案库内的档案的版本控制历史, 也可以进行一些精巧的功能, 像是产生这些档案于不同修订版之间的差异.

在 2002 年早期, ViewCVS 的档案库存取部份就已经进行模块化, 成为一个半通用的界面, 另外又有两个模块对 CVS 档案库提供这样的功能. 该年稍后, Subversion 的 Python 语言系结已经够成熟, 于是就为 ViewCVS 界面写出了

Subversion 档案库模块. 现在, ViewCVS 可以浏览 Subversion 的档案库, 并且为这些档案库提供历史与差异的机制, 就像为 CVS 所提供的一样.

想要得知更多有关 ViewCVS 的信息, 请参考该项目位于 <http://viewcvs.sf.net/> 的网站.

SubWiki

SubWiki 是一个以 Subversion 为后端的 Wiki 程序. Wiki 是由 World Wide Web 中窜起的特殊出版与社群工具 — 基本上, 就是以网页界面来编辑网页. SubWiki 吸收了 Wiki 的概念, 并且向前延伸, 使用版本控制系统作为其后端储存机制. 结果就是一个 CGI 程序, 让你可以就地编辑网页, 但是又不会失去这些网页的旧版数据.

想要得知更多有关 SubWiki 的信息, 请参考该项目位于 <http://subwiki.tigris.org/> 的网站.

^[43]CVSWeb 是以 Perl 写成的.

Glossary

Colophon

Etc.