Controle de Versão com Subversion

Para Subversion 1.4

(Compilado da revisão 129)

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

Controle de Versão com Subversion: Para Subversion 1.4: (Compilado da revisão 129)

por Ben Collins-Sussman, Brian W. Fitzpatrick, e C. Michael Pilato

Publicado (TBA)

Índice

Lista de Figuras

Lista de Tabelas

Lista de Exemplos

Apresentação

Karl Fogel
Chicago, 14 de Março de 2004

Uma base ruim de Perguntas Freqüentes (FAQ), é aquela que é composta não de perguntas que as pessoas realmente fizeram, mas de perguntas que o autor da FAQ *desejou* que as pessoas tivessem feito. Talvez você já tenha visto isto antes:

> P: De que forma posso utilizar o Glorbosoft XYZ para maximizar a produtividade da equipe?
>
> R: Muitos dos nossos clientes desejam saber como podem maximizar a produtividade através de nossas inovações patenteadas de groupware para escritórios. A resposta é simples: primeiro, clique no menu "`Arquivo`". Desça até a opção "`Aumentar Produtividade`", então…

O problema com estas bases de FAQ é que eles não são, propriamente ditas, FAQ. Ninguém nunca liga para o suporte técnico e pergunta "Como nós podemos maximizar a produtividade?". Em vez disso, as pessoas fazem perguntas muito mais específicas, como: "Como podemos alterar o sistema de calendário para enviar lembretes dois dias antes ao invés de um?", etc. Mas é muito mais fácil forjar Perguntas Freqüentes imaginárias do que descobrir as verdadeiras. Compilar uma verdadeira base de FAQ exige um esforço contínuo e organizado: através do ciclo de vida do software, as questões que chegam devem ser rastreadas, respostas monitoradas, e tudo deve ser reunido em um todo coerente, pesquisável que reflete a experiência coletiva dos usuários em seu mundo. Isto requer a paciência e a atitude observadora de um cientista. Nenhuma grande teoria ou pronunciamentos visionários aqui—olhos abertos e anotações precisas são o principal.

O que eu amo neste livro é que ele surgiu deste processo, e isto é evidenciado em cada página. Ele é o resultado direto dos encontros entre os autores e usuários. Ele começou quando Ben Collins-Sussman observou que as pessoas estavam fazendo as mesmas perguntas básicas diversas vezes nas listas de discussão do Subversion, como por exemplo: Quais são os procedimentos-padrão para se utilizar o Subversion? Branches e tags funcionam do mesmo modo como em outros sistemas de controle de versão? Como eu posso saber quem realizou uma alteração em particular?

Frustrado em ver as mesmas questões dia após dia, Ben trabalhou intensamente durante um mês no verão de 2002 para escrever *The Subversion Handbook*, um manual de sessenta páginas cobrindo todas as funcionalidades básicas do Subversion. O manual não tinha a pretensão de ser completo, mas ele foi distribuído com o Subversion e auxiliou os usuários a ultrapassarem as dificuldades iniciais na curva de aprendizado. Quando a O'Reilly and Associates decidiu publicar um livro completo sobre o Subversion, o caminho menos crítico estava óbvio: apenas estender o manual.

Para os três co-autores do novo livro então lhes foi dada uma oportunidade ímpar. Oficialmente, sua tarefa era escrever um livro top-down, starting from a table of contents and an initial draft. But they also had access to a steady stream—indeed, an uncontrollable geyser—of bottom-up source material. Subversion was already in the hands of thousands of early adopters, and those users were giving tons of feedback, not only about Subversion, but about its existing documentation.

Durante todo o tempo em que eles escreveram o livro, Ben, Mike, e Brian habitaram as listas de discussão e salas de bate-papo do Subversion, registrando cuidadosamente os problemas que os usuário estavam tendo em situações na vida-real. Monitorar este feedback, fazia parte da descrição de sua função na CollabNet, e isto lhes deu uma enorme vantagem quando começaram a documentar o Subversion. O livro que eles produziram é solidamente fundamentado na rocha da experiência, não nas areias mutáveis da ilusão; ele combina os melhores aspectos de um manual de usuário e uma base de FAQ. Esta dualidade talvez não seja perceptível numa primeira leitura. Lido na ordem, de frente para trás, o livro é uma descrição bem direta de uma peça de software. Existe a visão geral, o obrigatório tour, o capítulo sobre configuração

administrativa, alguns tópicos avançados, e é claro uma referência de comandos e um guia de resolução de problemas. Somente quando você o ler novamente mais tarde, procurando a solução para um problema específico, é que sua autenticidade reluzirá: the telling details that can only result from encounters with the unexpected, the examples honed from genuine use cases, and most of all the sensitivity to the user's needs and the user's point of view.

É claro, que ninguém pode prometer que este livro responderá todas as dúvidas que você tem sobre o Subversion. Certas vezes, a precisão com que ele antecipa suas perguntas parecerá assustadoramente telepática; ainda sim, ocasionalmente, você will stumble into a hole no conhecimento da comunidade, e sairá de mãos-vazias. Quando isto acontecer, a melhor coisa que você pode fazer é enviar um email para `<users@subversion.tigris.org>` e apresentar seu problema. Os autores ainda estão lá, continuam observando, e não somente os três listados na capa, mas muitos outros que contribuíram com correções e materiais originais. Do ponto de vista da comunidade, resolver o seu problema é meramente um agradável efeito de um projeto muito maior—namely, slowly adjusting this book, and ultimately Subversion itself, to more closely match the way people actually use it. They are eager to hear from you not merely because they can help you, but because you can help them. Com o Subversion, assim como em todo projeto ativo de software livre, *você não está sozinho*.

Que este livro seja seu primeiro companheiro.

Preface

> “It is important not to let the perfect become the enemy of the good, even when you can agree on what perfect is. Doubly so when you can't. As unpleasant as it is to be trapped by past mistakes, you can't make any progress by being afraid of your own shadow during design.”
>
> —Greg Hudson

In the world of open-source software, the Concurrent Versions System (CVS) was the tool of choice for version control for many years. And rightly so. CVS was open-source software itself, and its non-restrictive modus operandi and support for networked operation allowed dozens of geographically dispersed programmers to share their work. It fit the collaborative nature of the open-source world very well. CVS and its semi-chaotic development model have since become cornerstones of open-source culture.

But CVS was not without its flaws, and simply fixing those flaws promised to be an enormous effort. Enter Subversion. Designed to be a successor to CVS, Subversion's originators set out to win the hearts of CVS users in two ways—by creating an open-source system with a design (and “look and feel”) similar to CVS, and by attempting to avoid most of CVS's noticeable flaws. While the result isn't necessarily the next great evolution in version control design, Subversion *is* very powerful, very usable, and very flexible. And for the most part, almost all newly-started open-source projects now choose Subversion instead of CVS.

This book is written to document the 1.4 series of the Subversion version control system. We have made every attempt to be thorough in our coverage. However, Subversion has a thriving and energetic development community, so there are already a number of features and improvements planned for future versions of Subversion that may change some of the commands and specific notes in this book.

Público-Alvo

Este livro foi escrito para pessoas habituadas com computadores que desejam usar o Subversion para gerenciar seus dados. Ainda que o Subversion rode em vários sistemas operacionais diferentes, a sua interface de usuário primária é baseada em linha de comando. Essa ferramenta de linha de comando (**svn**), e alguns programas auxiliares, são o foco deste livro.

Por motivo de consistência, os exemplos neste livro presumem que o leitor está usando um sistema operacional baseado em Unix e se sente relativamente confortável com Unix e com interfaces de linha de comando. Dito isto, o programa **svn** também roda em plataformas não-Unix, como o Microsoft Windows. Com poucas exceções, como o uso de barras invertidas (\) em lugar de barras regulares (/) para separadores de caminho, a entrada e a saída desta ferramenta, quando executada no Windows, são idênticas ao seu companheiro Unix.

Muitos leitores são provavelmente programadores ou administradores de sistemas que necessitam rastrear alterações em código-fonte. Essa é a finalidade mais comum para o Subversion, e por isso é o cenário por trás de todos os exemplos deste livro. Entretanto, o Subversion pode ser usado para gerenciar qualquer tipo de informação—imagens, músicas, bancos de dados, documentação, etc. Para o Subversion, dados são apenas dados.

Enquanto que este livro presume que o leitor nunca usou um sistema de controle de versão, nós também tentamos facilitar para os usuários do CVS (e outros sistemas) a migração para o Subversion. Ocasionalmente, notas especiais poderão mencionar outros controles de versão. Há também um apêndice que resume muitas das diferenças entre CVS e Subversion.

Note também que os exemplos de código-fonte usados ao longo do livro são apenas exemplos. Ainda que eles compilem com os truques apropriados do compilador, seu propósito é ilustrar um cenário em particular, não necessariamente servir como exemplos de boas práticas de programação.

Como ler este livro

Livros técnicos sempre enfrentam um certo dilema: se os leitores devem fazer uma leitura *top-down* (do início ao fim) ou *bottom-up* (do fim ao começo). Um leitor top-down prefere ler ou folhear toda a documentação antes, ter uma visão geral de como o sistema funciona e apenas, então, é que que ele inicia o uso do software. Já um leitor bottom-up “aprende fazendo”, ele é alguém que mergulha no software e o esmiuça, voltando às seções do livro quando necessário. Muitos livros são escritos para um ou outro tipo de pessoa, e esse livro é, sem dúvida, indicado para o leitor top-down. (Se você está lendo esta seção, provavelmente você já é um leitor top-down nato). No entanto, se você é um leitor bottom-up, não se desespere. Mesmo que este livro pode ser definido como um apanhado geral sobre o Subversion, o conteúdo de cada seção tende a aprofundar com exemplos específicos nos quais você pode aprender fazendo. As pessoas impacientes que já querem ir fazendo, podem pular direto para o Apêndice A, *Guia Rápido de Subversion*.

Independente do seu estilo de aprendizado, este livro pretende ser útil para os mais diversos tipos de pessoas — os que não possuem nenhuma experiência com controle de versão até os administradores de sistema mais experientes. Dependendo do seu conhecimento, certos capítulos podem ser mais ou menos importantes para você. A lista abaixo é uma “recomendação de leitura” para os diversos tipos de leitores:

Administradores de Sistemas Experientes
: Supõem-se aqui que você, provavelmente, já tenha usado controle de versão anteriormente, e está morrendo de vontade de usar um servidor Subversion o quanto antes. O Capítulo 5, *Administração do Respositório* e o Capítulo 6, *Configuração do Servidor* irão mostrar como criar seu primeiro repositório e torná-lo disponível na rede. Depois disso, o Capítulo 2, *Uso Básico* e o Apêndice B, *Subversion para Usuários de CVS* vão mostrar o caminho mais rápido para se aprender a usar o cliente Subversion.

Novos usuário
: Seu administrador provavelmente já diponibilizou um servidor Subversion, e você precisa aprender a usar o cliente. Se você nunca utilizou um sistema de controle de versão, então o Capítulo 1, *Conceitos Fundamentais* será vital para introduzir e mostrar as idéias por trás do controle de versão. O Capítulo 2, *Uso Básico* é um guia do cliente do Subversion.

Usuários avançados
: Seja você é um usuário ou um administrador, eventualmente seu projeto irá crescer muito. Você irá querer aprender a fazer coisas avançadas com o Subversion, como usar branches e fazer merges (Capítulo 4, *Fundir e Ramificar*), como usar as propriedades de suporte do Subversion (Capítulo 3, *Tópicos Avançados*), como configurar as opções de runtime (Capítulo 7, *Customizando sua Experiência com Subversion*), entre outras coisas. Estes capítulos não são críticos no início, porém não deixe de lê-los quando estiver familiarizado com o básico.

Desenvolvedores
: Presumidamente, você já está familiarizado com o Subversion, e quer ou extendê-lo ou construir um novo software baseado nas suas diversas APIs. O Capítulo 8, *Incorporando o Subversion* foi feito justamente pra pra você.

O livro termina com um material de referência — o Capítulo 9, *Referência Completa do Subverion* é um guia de referência para todos os comandos do Subversion, e o apêndice cobre um número considerável de tópicos úteis. Estes capítulos serão os que você irá voltar com mais freqüência depois que terminar de ler o livro.

Convenções Usadas Neste Livro

Esta seção cobre as várias convenções usadas neste livro.

Convenções tipográficas

Largura constante
: Comandos usados, comando de saída, e opções

`Largura constante em itálico`
: Usado para substituir itens no código e texto

`Itálico`
: Usado para nomes de arquivo e diretório

Ícones

Este ícone representa uma nota relacionada ao texto em citado.

Este ícone representa uma dica útil relacionada ao texto citado.

Este ícone representa um aviso relacionado ao texto citado.

Organização Deste Livro

Abaixo estão listados os capítulos e seus conteúdos estão listados:

Preface
: Cobre a história do Subversion bem como suas características, arquitetura e compomentes.

Capítulo 1, *Conceitos Fundamentais*
: Explica o básico sobre controle de versão e os diferentes modelos de versionamento, o repositório do Subversion, cópias de trabalho e revisões.

Capítulo 2, *Uso Básico*
: Faz um tour em um dia na vida de um usuário do Subversion. Demonstra como usar o cliente do subversion para obter, modificar e submeter dados.

Capítulo 3, *Tópicos Avançados*
: Cobre as características mais complexas que os usuários regulares irão encontrar eventualmente, como metadados, travamento de arquivo(*locking*) e rotulagem de revisões.

Capítulo 4, *Fundir e Ramificar*
: Discute sobre ramos, fusões, e rotulagem, incluindo as melhores práticas para a criação de ramos e fusões, casos de uso comuns, como desfazer alterações, e como facilitar a troca de um ramo para o outro.

Capítulo 5, *Administração do Respositório*
: Descreve o básico de um repositório Subversion, como criar, configurar, e manter um repositório, e as ferramentas que podem ser usadas para isso.

Capítulo 6, *Configuração do Servidor*
: Explica como configurar um servidor Subversion e os diferentes caminhos para acessar seu repositório: `HTTP`, o protocolo`svn` e o acesso local pelo disco. Também cobre os detalhes de autenticação, autorização e acesso anônimo.

Capítulo 7, *Customizando sua Experiência com Subversion*
: Explora os arquivos de configuração do cliente Subversion, a internacionalização de texto, e como fazer com que ferramentas externas trabalhem com o subversion.

Capítulo 8, *Incorporando o Subversion*
: Descreve as partes internas do Subversion, o sistema de arquivos do Subversion, e a cópia de trabalho da área administrativa do ponto de vista de um programador. Demonstra como usar as APIs públicas para escrever um programa que usa o Subversion, e o mais importante, como contribuir para o desenvolvimento do Subversion.

Capítulo 9, *Referência Completa do Subverion*
: Explica em detalhes todos os subcomandos do **svn**, **svnadmin**, e **svnlook** com abundância de exemplos para toda a família!

Apêndice A, *Guia Rápido de Subversion*
: Para os impacientes, uma rápida explicação de como instalar o Subversion e iniciar seu uso imediatamente. Você foi avisado.

Apêndice B, *Subversion para Usuários de CVS*
: Cobre as similaridades e diferenças entre o Subversion e o CVS, com numerosas sugestões de como fazer para quebrar todos os maus hábitos que você adquiriu ao longo dos anos com o uso do CVS. Inclui a descrição de número de revisão do Subversion, versionamento de diretórios, operações offline, **update** vs. **status**, ramos, rótulos, resolução de conflitos e autenticação.

Apêndice C, *WebDAV and Autoversioning*
: Descreve os detalhes do WebDAV e DeltaV, e como você pode configurar e montar seu repositório Subversion em modo de leitura/escrita em um compartilhamento DAV.

Apêndice D, *Third Party Tools*
: Discute as ferramentas que suportam ou usam o Subversion, incluindo programas clientes alternativos, ferramentas para navegação no repositório, e muito mais.

This Book is Free

Este livro teve início com a documentação escrita pelos desenvolvedores do projeto Subversion, o qual foi reunido em um único trabalho e reescrito. Assim, ele sempre esteve sob uma licença livre. (Veja Apêndice E, *Copyright*.) De fato, o livro foi escrito sob uma visão pública, originalmente como parte do próprio projeto Subversion. Isto significa duas coisas:

- Você sempre irá encontrar a última versão deste livro no próprio repositório Subversion do livro.

- Você pode fazer alterações neste livro e redistribuí-lo, entretanto você deve fazê-lo—sob uma licença livre. Sua única obrigação é manter o créditos originais dos autores. É claro que, ao invés de distribuir sua própria versão deste livro, gostaríamos muito mais que você enviasse seu feedback e correções para a comunidade de desenvolvimento do Subversion.

O site deste livro está em desenvolvimento, e muitos dos tradutores voluntários estão se reunindo no site http://svnbook.red-bean.com. Lá você pode encontrar links para as últimas versões lançadas e versões

compiladas deste livro em diversos formatos, bem como as instruções de acesso ao repositório Subversion do livro(onde está o código-fonte em formato DocBook XML) Um feedback é bem vindo—e encorajado também. Por favor, envie todos os seus comentários, reclamações, e retificações dos fontes do livro para o e-mail <svnbook-dev@red-bean.com>.

Agradecimentos

Este livro não existiria (e nem seria útil) se o Subversion não existisse. Assim, os autores gostaria de agradecer a Brian Behlendorf e a CollabNet, pela visão em acreditar en um arriscado e ambicioso projeto de Código Aberto; Jim Blandy pelo nome e projeto original do Subversion—nós amamos você, Jim; Karl Fogel, por ser um excelente amigo e grande líder na comunidade, nesta ordem.[1]

Agradecimentos a O'Reilly e nossos editores, Linda Mui e Tatiana Diaz por sua paciente e apoio.

Finalmente, agrademos às inúmeras pessoas que contribuíram para este livro com suas revisões informais, sugestões e retificações. Certamente, esta não é uma lista completa, mas este livro estaria imcompleto e incorreto sem a ajuda de: David Anderson, Jani Averbach, Ryan Barrett, Francois Beausoleil, Jennifer Bevan, Matt Blais, Zack Brown, Martin Buchholz, Brane Cibej, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Nick Duffek, Ben Elliston, Justin Erenkrantz, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Dave Gilbert, Eric Gillespie, David Glasser, Matthew Gregan, Art Haas, Eric Hanchrow, Greg Hudson, Alexis Huxley, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Andreas J. Koenig, Nuutti Kotivuori, Matt Kraai, Scott Lamb, Vincent Lefevre, Morten Ludvigsen, Paul Lussier, Bruce A. Mah, Philip Martin, Feliciano Matias, Patrick Mayweg, Gareth McCaughan, Jon Middleton, Tim Moloney, Christopher Ness, Mats Nilsson, Joe Orton, Amy Lyn Pilato, Kevin Pilch-Bisson, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, Jack Repenning, Tobias Ringstrom, Garrett Rooney, Joel Rosdahl, Christian Sauer, Larry Shatzer, Russell Steicke, Sander Striker, Erik Sjoelund, Johan Sundstroem, John Szakmeister, Mason Thomas, Eric Wadsworth, Colin Watson, Alex Waugh, Chad Whitacre, Josef Wolf, Blair Zajac e a comunidade inteira do Subversion.

Agradecimentos de Ben Collins-Sussman

Agradeço a minha esposa Frances, quem, por vários meses, começou a ouvir, “Mas docinho, eu estou trabalhando no livro”, do que o habitual, “Mas docinho, eu estou escrevendo um e-mail.” Eu não sei onde ela arruma tanta paciência! Ela é meu equilíbrio perfeito.

Agradeço à minha extensa família e amigos por seus sinceros votos de encorajamento, apesar de não terem qualquer interesse no assunto. (Você sabe, tem uns que dizem: “Você escreveu um livro?”, e então quando você diz que é um livro de computador, eles te olham torto.)

Agradeço aos meus amigos mais próximos, que me fazem um rico, rico homem. Não me olham da mesma forma—vocês sabem quem são.

Agradeço aos meus pais pela minha perfeita formação básica, e pelos inacreditáveis conselhos. Agradeços ao meu filho pela oportunidade de passar isto adiante.

Agradecimentos de Brian W. Fitzpatrick

Um grande obrigado à minha esposa Marie por sua inacreditável compreensão, apoio, e acima de tudo, paciência. Obrigado ao meu irmão Eric, quem primeiro me apresentou a programação voltada para UNIX.

[1] Oh, e agradecemos ao Karl, por ter dedicado muito trabalho ao escrever este livro sozinho.

Agradeço à minha mãe e minha avó por seu apoio, sem falar nas longas férias de Natal, quando eu chegava em casa e mergulhava minha cabeça no laptop para trabalhar no livro.

Mike e Ben, foi um prezer trabalhar com você neste livro. Heck, é um prazer trabalhar com você nesta obra!

Para todos da comunidade Subversion e a Apache Software Foundation, agradeço por me receberem. Não há um dia onde eu não aprenda algo com pelo menos um de vocês.

Finalmente, agradeço ao meu avô que sempre me disse “Liberdade igual responsabilidade.” Eu não poderia estar mais de acordo.

Agradecimentos de C. Michael Pilato

Um obrigado especial a Amy, minha melhor amiga e esposa por inacreditáveis nove anos, por seu amor e apoio paciente, por me tolerar até tarde da noite, e por agüentar o duro processo de controle de versão que impus a ela. Não se precupe, querida—você será um assistente do TortoiseSVN logo!

Gavin, provavelmente não há muitas palavras neste livro que você possa, com sucesso, “pronunciar” nesta fase de sua vida, mas quando você, finalmente, aprender a forma escrita desta louca língua que falamos, espero que você esteja tão orgulhoso de seu pai quanto ele de você.

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

Mãe e Pai, agraço pelo apoio e entusiasmo constante. Sogra e Sogro, agradeço por tudo da mesma forma e *mais* um pouco por sua fabulosa filha.

Tiro o chapéu para Shep Kendall, foi através dele que o mundo dos computadores foi aberto pela primeira vez a mim; Ben Collins-Sussman, meu orientador pelo mundo do código-aberto; Karl Fogel— você *é* meu `.emacs`; Greg Stein, o difusor da programação prática como-fazer; Brian Fitzpatrick—por compartilhar esta experiência de escrever junto comigo. Às muitas pessoas com as quais eu estou constantemente aprendendo—e continuo aprendendo!

Finalmente, agradeço a alguém que demonstra ser perfeitamente criativo em sua excelência—você.

What is Subversion?

Subversion is a free/open-source version control system. That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data, or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of “time machine”.

Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incorrect change is made to the data, just undo that change.

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code, and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for

[2] Tradução: Papai te ama e espera que você goste de computadores assim como você irá gostar de basquete, basebol e futebol. (Isso seria óbvio?)

building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage *any* collection of files. For you, those files might be source code—for others, anything from grocery shopping lists to digital video mixdowns and beyond.

Subversion's History

In early 2000, CollabNet, Inc. (http://www.collab.net) began seeking developers to write a replacement for CVS. CollabNet offers a collaboration software suite called CollabNet Enterprise Edition (CEE) of which one component is version control. Although CEE used CVS as its initial version control system, CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better. Unfortunately, CVS had become the de facto standard in the open source world largely because there *wasn't* anything better, at least not under a free license. So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

In February 2000, they contacted Karl Fogel, the author of *Open Source Development with CVS* (Coriolis, 1999), and asked if he'd like to work on this new project. Coincidentally, at the time Karl was already discussing a design for a new version control system with his friend Jim Blandy. In 1995, the two had started Cyclic Software, a company providing CVS support contracts, and although they later sold the business, they still used CVS every day at their jobs. Their frustration with CVS had led Jim to think carefully about better ways to manage versioned data, and he'd already come up with not only the name “Subversion”, but also with the basic design of the Subversion data store. When CollabNet called, Karl immediately agreed to work on the project, and Jim got his employer, Red Hat Software, to essentially donate him to the project for an indefinite period of time. CollabNet hired Karl and Ben Collins-Sussman, and detailed design work began in May. With the help of some well-placed prods from Brian Behlendorf and Jason Robbins of CollabNet, and Greg Stein (at the time an independent developer active in the WebDAV/DeltaV specification process), Subversion quickly attracted a community of active developers. It turned out that many people had had the same frustrating experiences with CVS, and welcomed the chance to finally do something about it.

The original design team settled on some simple goals. They didn't want to break new ground in version control methodology, they just wanted to fix CVS. They decided that Subversion would match CVS's features, and preserve the same development model, but not duplicate CVS's most obvious flaws. And although it did not need to be a drop-in replacement for CVS, it should be similar enough that any CVS user could make the switch with little effort.

After fourteen months of coding, Subversion became “self-hosting” on August 31, 2001. That is, Subversion developers stopped using CVS to manage Subversion's own source code, and started using Subversion instead.

While CollabNet started the project, and still funds a large chunk of the work (it pays the salaries of a few full-time Subversion developers), Subversion is run like most open-source projects, governed by a loose, transparent set of rules that encourage meritocracy. CollabNet's copyright license is fully compliant with the Debian Free Software Guidelines. In other words, anyone is free to download, modify, and redistribute Subversion as he pleases; no permission from CollabNet or anyone else is required.

Subversion's Features

When discussing the features that Subversion brings to the version control table, it is often helpful to speak of them in terms of how they improve upon CVS's design. If you're not familiar with CVS, you may not understand all of these features. And if you're not familiar with version control at all, your eyes may glaze over unless you first read Capítulo 1, *Conceitos Fundamentais*, in which we provide a gentle introduction to version control.

Subversion provides:

Directory versioning
: CVS only tracks the history of individual files, but Subversion implements a “virtual” versioned filesystem that tracks changes to whole directory trees over time. Files *and* directories are versioned.

True version history
: Since CVS is limited to file versioning, operations such as copies and renames—which might happen to files, but which are really changes to the contents of some containing directory—aren't supported in CVS. Additionally, in CVS you cannot replace a versioned file with some new thing of the same name without the new item inheriting the history of the old—perhaps completely unrelated—file. With Subversion, you can add, delete, copy, and rename both files and directories. And every newly added file begins with a fresh, clean history all its own.

Atomic commits
: A collection of modifications either goes into the repository completely, or not at all. This allows developers to construct and commit changes as logical chunks, and prevents problems that can occur when only a portion of a set of changes is successfully sent to the repository.

Versioned metadata
: Each file and directory has a set of properties—keys and their values—associated with it. You can create and store any arbitrary key/value pairs you wish. Properties are versioned over time, just like file contents.

Choice of network layers
: Subversion has an abstracted notion of repository access, making it easy for people to implement new network mechanisms. Subversion can plug into the Apache HTTP Server as an extension module. This gives Subversion a big advantage in stability and interoperability, and instant access to existing features provided by that server—authentication, authorization, wire compression, and so on. A more lightweight, standalone Subversion server process is also available. This server speaks a custom protocol which can be easily tunneled over SSH.

Consistent data handling
: Subversion expresses file differences using a binary differencing algorithm, which works identically on both text (human-readable) and binary (human-unreadable) files. Both types of files are stored equally compressed in the repository, and differences are transmitted in both directions across the network.

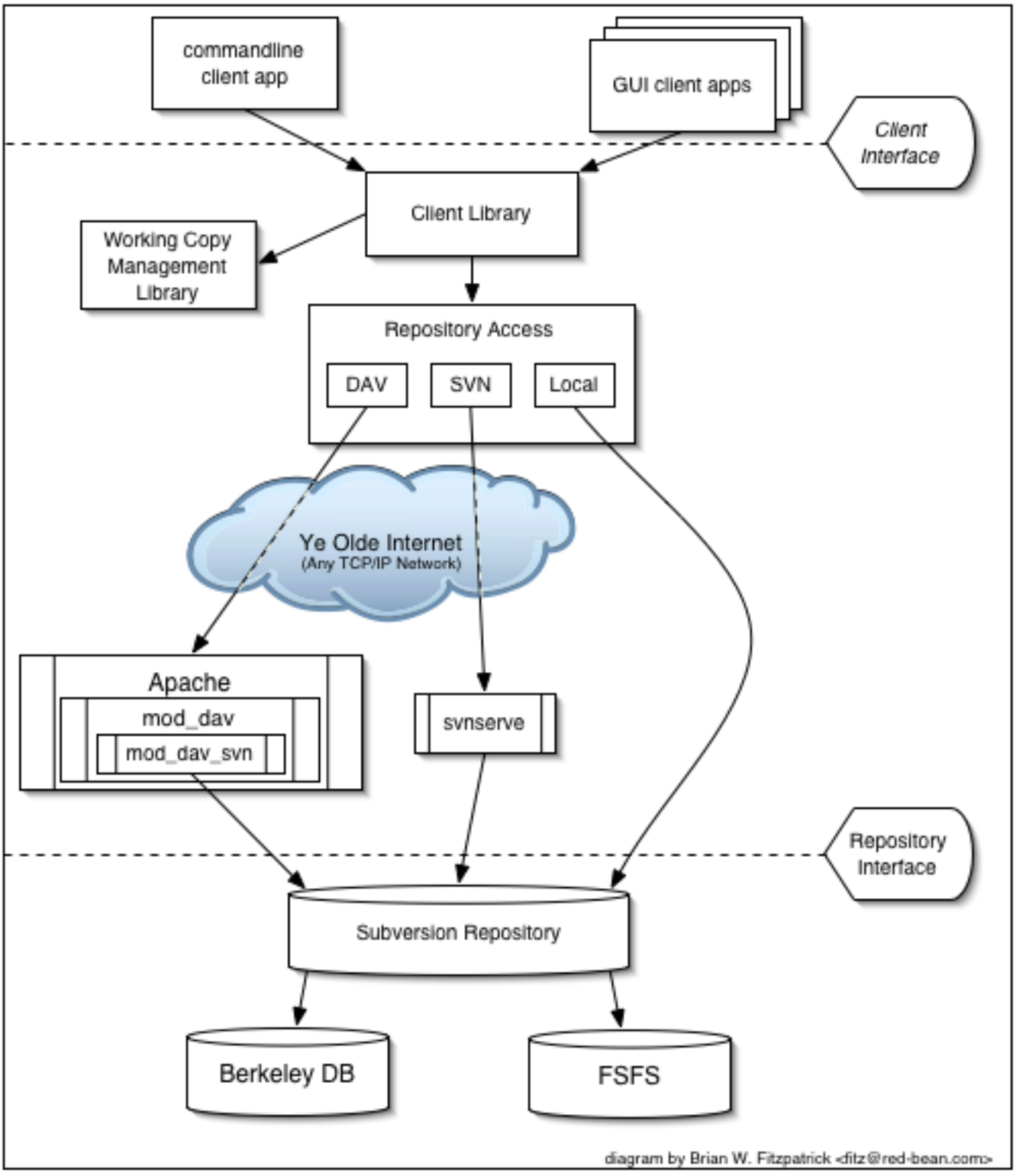
Efficient branching and tagging
: The cost of branching and tagging need not be proportional to the project size. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time.

Hackability
: Subversion has no historical baggage; it is implemented as a collection of shared C libraries with well-defined APIs. This makes Subversion extremely maintainable and usable by other applications and languages.

Subversion's Architecture

Figura 1, “Subversion's Architecture” illustrates a “mile-high” view of Subversion's design.

Figura 1. Subversion's Architecture



On one end is a Subversion repository that holds all of your versioned data. On the other end is your Subversion client program, which manages local reflections of portions of that versioned data (called

“working copies”). Between these extremes are multiple routes through various Repository Access (RA) layers. Some of these routes go across computer networks and through network servers which then access the repository. Others bypass the network altogether and access the repository directly.

Subversion's Components

Subversion, once installed, has a number of different pieces. The following is a quick overview of what you get. Don't be alarmed if the brief descriptions leave you scratching your head—there are *plenty* more pages in this book devoted to alleviating that confusion.

svn
: The command-line client program.

svnversion
: A program for reporting the state (in terms of revisions of the items present) of a working copy.

svnlook
: A tool for directly inspecting a Subversion repository.

svnadmin
: A tool for creating, tweaking or repairing a Subversion repository.

svndumpfilter
: A program for filtering Subversion repository dump streams.

mod_dav_svn
: A plug-in module for the Apache HTTP Server, used to make your repository available to others over a network.

svnserve
: A custom standalone server program, runnable as a daemon process or invokable by SSH; another way to make your repository available to others over a network.

svnsync
: A program for incrementally mirroring one repository to another over a network.

Assuming you have Subversion installed correctly, you should be ready to start. The next two chapters will walk you through the use of **svn**, Subversion's command-line client program.

Capítulo 1. Conceitos Fundamentais

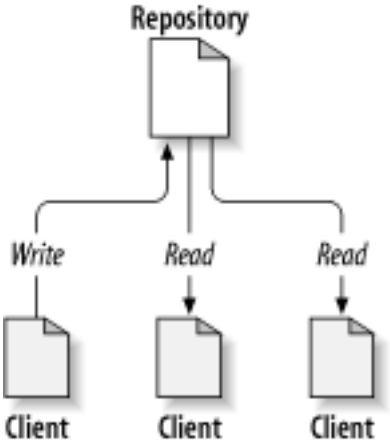
Este capítulo é uma breve e casual introdução ao Subversion. Se você é novo em controle de versão, este capítulo é definitivamente para você. Nós começaremos com uma discussão sobre os conceitos gerais de controle de versão, avançaremos para as idéias específicas por trás do Subversion, e mostraremos alguns exemplos simples do Subversion em uso.

Embora os exemplos neste capítulo mostrem pessoas compartilhando coleções de código fonte de programas, tenha em mente que o Subversion pode gerenciar qualquer tipo de coleção de arquivos - ele não está limitado a ajudar programadores.

O Repositório

O Subversion é um sistema centralizado de compartilhamento de informação. Em seu núcleo está um repositório, que é uma central de armazenamento de dados. O repositório armazena informação em forma de uma *árvore de arquivos* - uma hierarquia típica de arquivos e diretórios. Qualquer número de *clientes* se conecta ao repositório, e então lê ou escreve nestes arquivos. Ao gravar dados, um cliente torna a informação disponível para outros; ao ler os dados, o cliente recebe informação de outros. Figura 1.1, “Um típico sistema cliente/servidor” ilustra isso.

Figura 1.1. Um típico sistema cliente/servidor



Então, por que razão isto é interessante? Até ao momento, isto soa como a definição de um típico servidor de arquivos. E, na verdade, o repositório *é* uma espécie de servidor de arquivos, mas não de um tipo comum. O que torna o repositório do Subversion especial é que *ele se lembra de cada alteração* já ocorrida nele: de cada mudança em cada arquivo, e até mesmo alterações na árvore de diretórios em si, como a adição, eliminação, e reorganização de arquivos e diretórios.

Quando um cliente lê dados de um repositório, ele normalmente vê apenas a última versão da árvore de arquivos. Mas o cliente também tem a habilidade de ver os estados *anteriores* do sistema de arquivos. Por exemplo, um cliente pode perguntar questões de histórico como, “O que este diretório continha na última quarta-feira?” ou “Quem foi a última pessoa que alterou este aquivo, e que alterações ela fez?” Estes são os tipos de questões que estão no coração de qualquer *sistema de controle de versão*: sistemas que são projetados para monitorar alterações nos dados ao longo do tempo.

Modelos de Versionamento

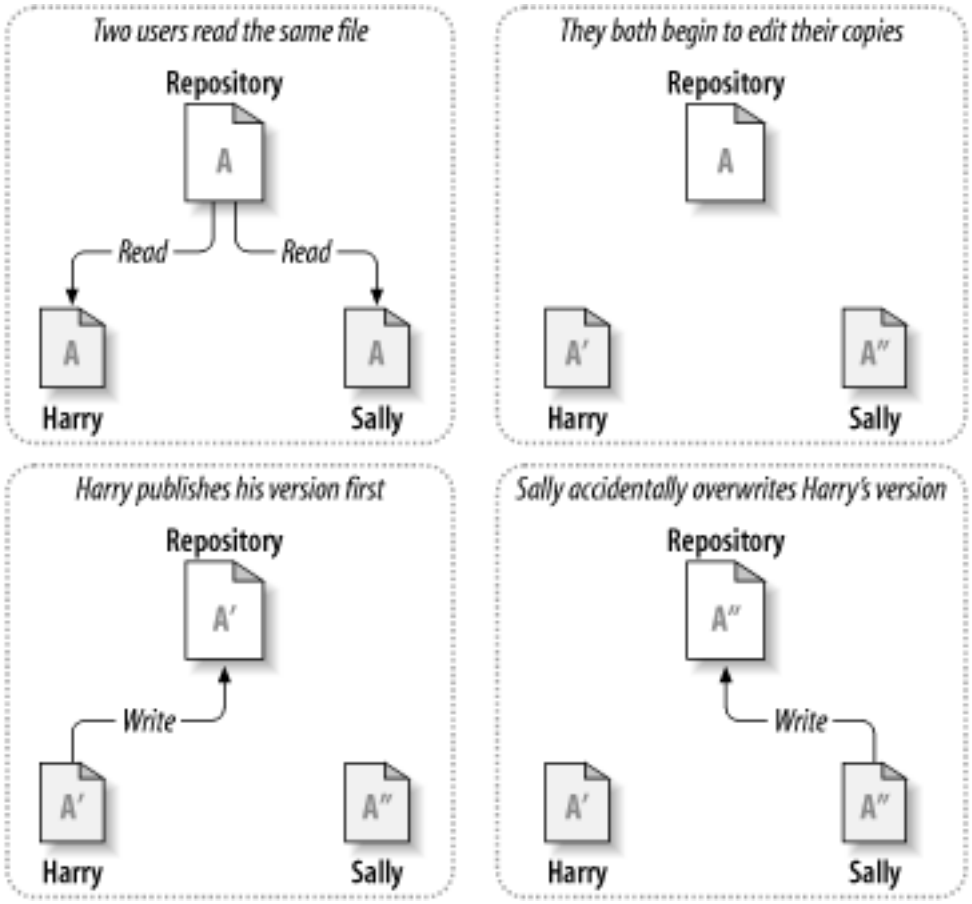
A missão principal de um sistema de controle de versão é permitir a edição colaborativa e o compartilhamento de dados. Mas diferentes sistemas usam diferentes estratégias para atingir este objetivo. É importante compreender essas diferentes estratégias por várias razões. Primeiro, irá ajudá-lo a comparar os sistemas de controle de versão existentes, no caso de você encontrar outros sistemas similares ao Subversion. Além disso, irá ajudá-lo ainda a tornar o uso do Subversion mais eficaz, visto que o Subversion por si só permite diversas formas diferentes de se trabalhar.

O Problema do Compartilhamento de Arquivos

Todos so sistemas de controle de versão têm de resolver o mesmo problema fundamental: como os sistema irá permitir que os usuários compartilhem informação, e como ele irá prevenir que eles acidentalmente tropecem uns nos pés dos outros? É muito fácil para os usuários acidentalmente sobrescrever as mudanças feitas pelos outros no repositório.

Considere o cenário mostrado em Figura 1.2, “O problema para evitar”. Suponhamos que nós temos dois colegas de trabalho, Harry and Sally. Cada um deles decide editar o mesmo arquivo no repositório ao mesmo tempo. Se Harry salvar suas alterações no repositório primeiro, então é possível que (poucos momentos depois) Sally possa acidentalmente sobrescrevê-lo com a sua própria nova versão do arquivo. Embora a versão de Harry não seja perdida para sempre (porque o sistema se lembra de cada mudança), todas as mudanças feitas por Harry *não* vão estar presentes na versão mais recente do arquivo de Sally, porque ela nunca viu as mudanças de Harry's para começar. O trabalho de Harry efetivamente se perdeu - ou pelo menos desapareceu da última versão do arquivo - e provavelmente por acidente. Trata-se definitivamente de uma situação que queremos evitar!

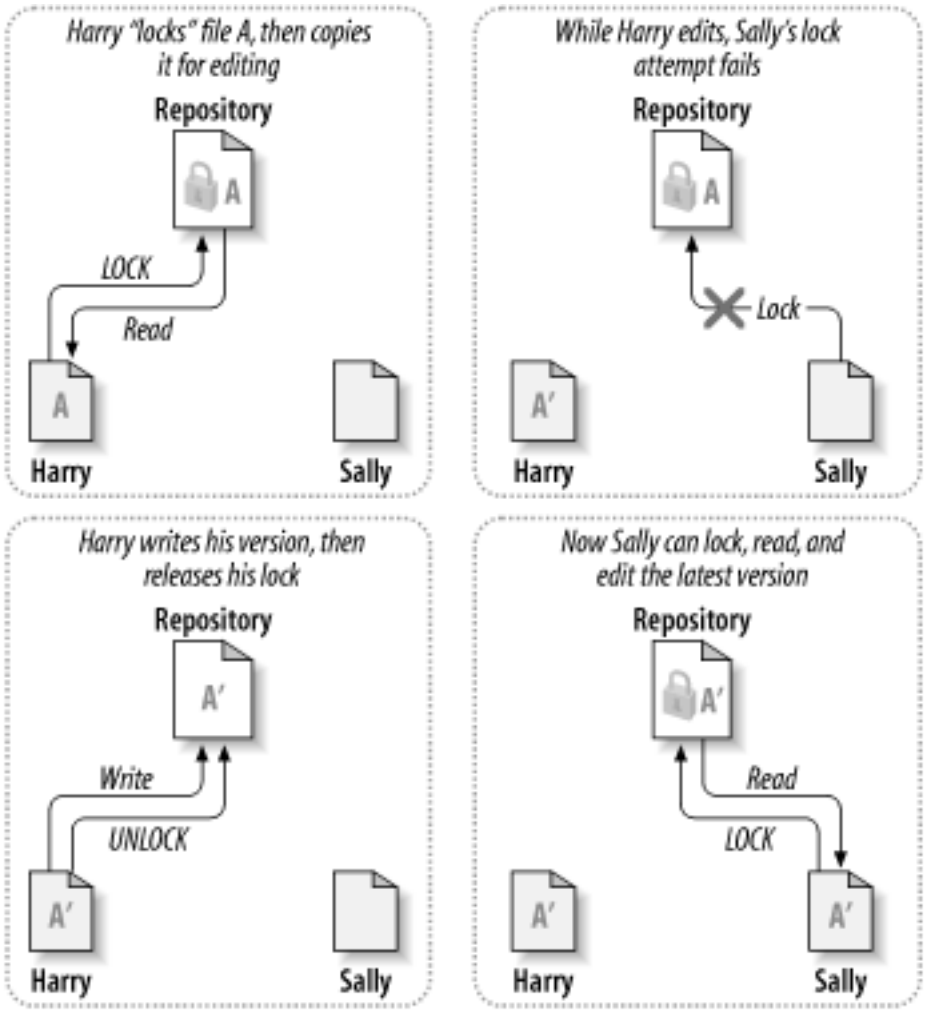
Figura 1.2. O problema para evitar



A Solução Lock-Modify-Unlock

Muitos sistemas de controle de versão usam o modelo *lock-modify-unlock* (travar-modificar-destravar) para resolver o problema de vários autores destruirem o trabalho uns dos outros. Neste modelo, o repositório permite que apenas uma pessoa de cada vez altere o arquivo. Esta política de exclusividade é gerenciada usando locks (travas). Harry precisa “travar ” (lock) um arquivo antes que possa fazer alterações nele. Se Harry tiver travado o arquivo, então Sally não poderá travá-lo também, e portanto, não poderá fazer nenhuma alteração nele. Tudo que ela pode fazer é ler o arquivo, e esperar que Harry termine suas alterações e destrave (unlock) o arquivo. Depois que Harry destravar o arquivo, Sally poderá ter a sua chance de travar e editar o arquivo. A figura Figura 1.3, “A solução lock-modify-unlock” demonstra essa solução simples.

Figura 1.3. A solução lock-modify-unlock



O problema com o modelo lock-modify-unlock é que ele é um pouco restritivo, muitas vezes se torna um obstáculo para os usuários:

- *Locks podem causar problemas administrativos.* Algumas vezes Harry irá travar o arquivo e se esquecer disso. Entretanto, devido a Sally ainda estar esperando para editar o arquivo, suas mãos estão atadas. E Harry então sai de férias. Agora Sally tem que pedir a um administrador para destravar o arquivo que Harry travou. Essa situação acaba causando uma série de atrasos desnecessários e perda de tempo.

- *Locking pode causar serialização desnecessária.* E se Harry está editando o começo de um arquivo de texto, e Sally simplesmente quer editar o final do mesmo arquivo? Essas mudanças não vão se sobrepor afinal. Eles podem facilmente editar o arquivo simultaneamente, sem grandes danos, assumindo que as alterações serão apropriadamente combinadas depois. Não há necessidade de se trabalhar em turnos nessa situação.

- *Locking pode criar falsa sensação de segurança.* Suponha que Harry trave e edite o arquivo A, enquanto Sally simultaneamente trava e edita o arquivo B. Mas e se A e B dependem um do outro, e se as mudanças feitas em cada são semanticamente incompatíveis? Subitamente A e B não funcionam juntos mais. O sistema de locking não foi suficientemente poderoso para prevenir o problema - ainda que de certa forma tenha proporcionado uma falsa sensação de segurança. É fácil para Harry e Sally imaginar que travando os arquivos, cada um está começando uma tarefa isolada segura, e assim não se preocupar

em discutir as incompatibilidades que virão com suas mudanças. Locking frequentemente se torna um substituto para a comunicação real.

A Solução Copy-Modify-Merge

O Subversion, CVS, e muitos outros sistemas de controle de vesão usam um modelo de *copy-modify-merge* (copiar-modificar-combinar) como uma alternativa ao locking. Neste modelo, cada usuário se conecta ao repositório do projeto e cria uma *cópia de trabalho* pessoal (personal working copy, ou cópia local) - um espelho local dos arquivos e diretórios no repositório. Os usuários então trabalham simultaneamente e independentemente, modificando suas cópias privadas. Finalmente as cópias privadas são combinadas (merged) numa nova versão final. O sistema de controle de versão, frequentemente ajuda com a combinação, mas no final, a intervenção humana é a única capaz de garantir que as mudanças foram realizadas de forma correta.

Aqui vai um exemplo. Digamos que Harry and Sally cada, criaram cópias de trabalho do mesmo projeto, copiadas do repositório. Eles trabalharam concorrentemente, e fizeram alterações no mesmo arquivo A em suas próprias cópias. Sally salva suas alteações no repositório primeiro. Quando Harry tentar salvar suas alterações mais tarde, o repositório vai informá-lo que seu arquivo A está *desatualizado* (out-of-date). Em outras palavras, este arquivo A no repositório foi de alguma forma alterado desde a última vez que ele foi copiado. Então Harry pede a seu programa cliente para ajudá-lo a *combinar* (merge) todas as alterações no repositório da sua cópia de trabalho de A. Provavelmente as mudanças de Sally não se sobrepõem com as suas próprias; então, uma vez que ele tiver ambos os conjuntos de alterações integradas, ele salva sua cópia de trabalho de volta no repositório. As figuras Figura 1.4, “A solução copy-modify-merge” e Figura 1.5, “A solução copy-modify-merge (continuando)” mostram este processo.

Figura 1.4. A solução copy-modify-merge

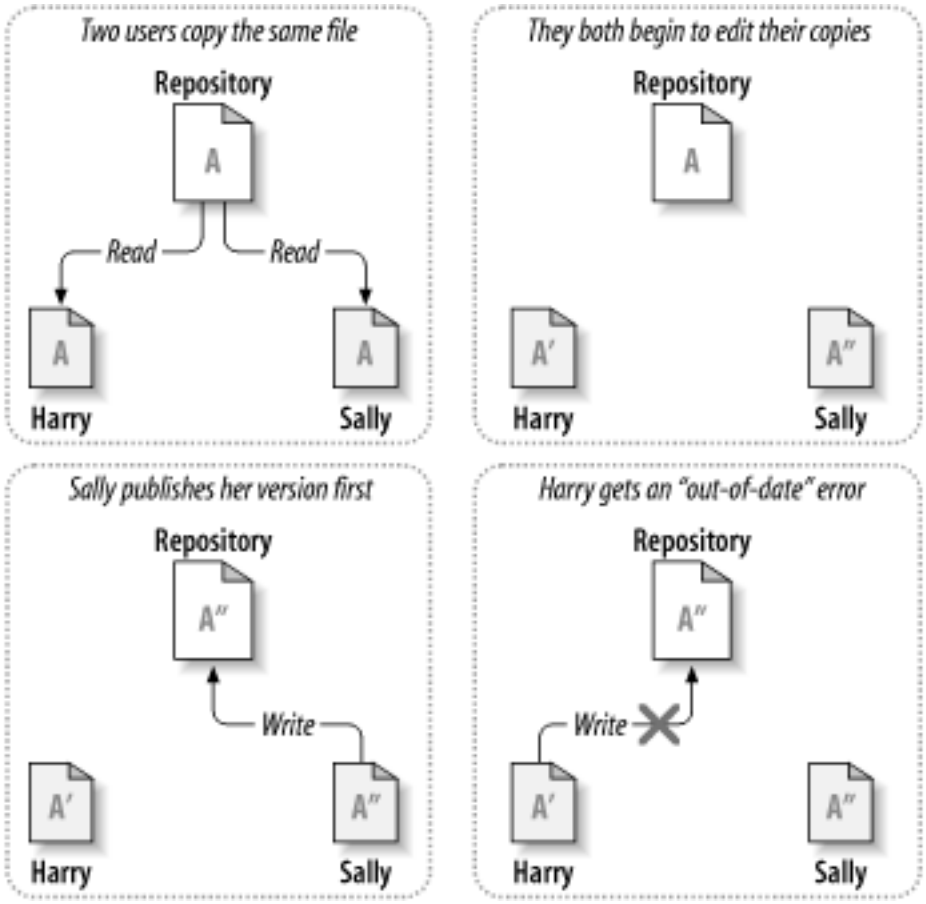
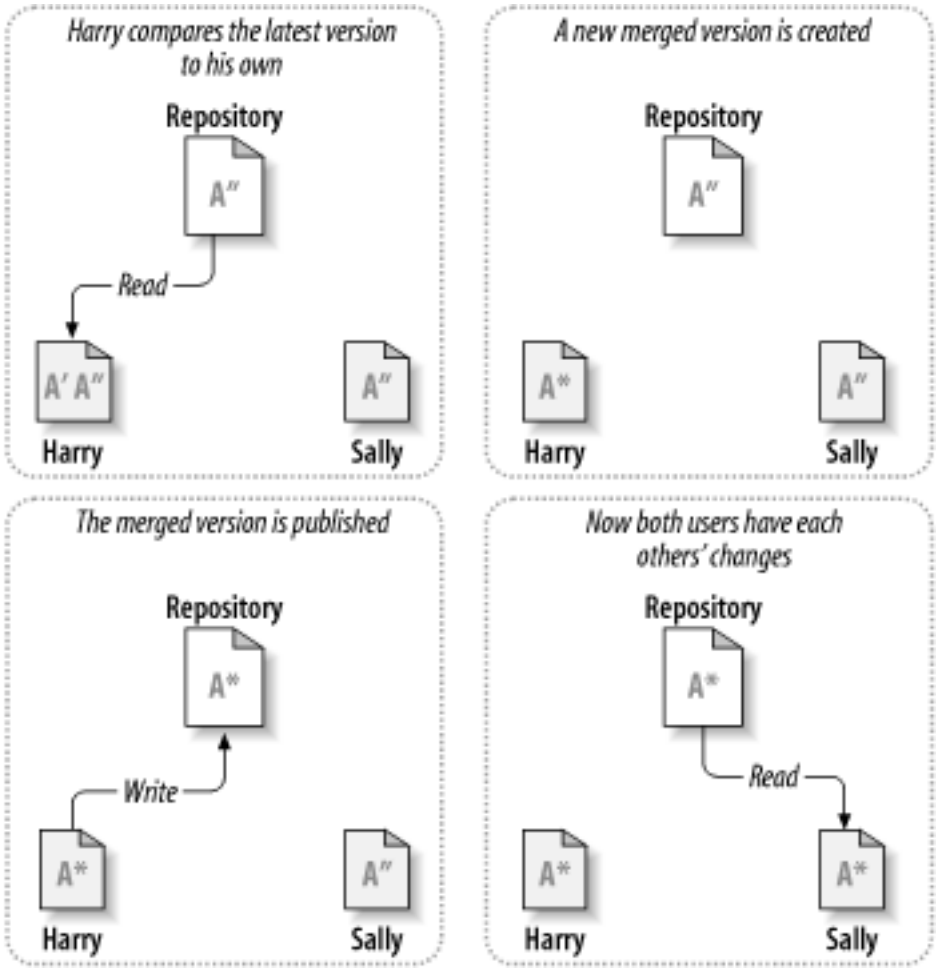


Figura 1.5. A solução copy-modify-merge (continuando)



Mas e se as alterações de Sally *sobrescreverem* as de Harry? E então? Esta situação é chamada de *conflito* , e usualmente não é um problema. Quando Harry pedir a seu cliente para combinar as últimas alterações do repositório em sua cópia de local, sua cópia do arquivo A estará de alguma forma sinalizada como estando numa situação de conflito: ele será capaz de ver ambos os conjuntos de alterações conflitantes, e manualmente escolher entre elas. Note que o software não tem como resolver os conflitos automaticamente; apenas pessoas são capazes de compreender e fazer as escolhas inteligentes. Uma vez que Harry tenha resolvido manualmente as alterações conflitantes - talvez depois de uma conversa com Sally - ele poderá tranquilamente salvar o arquivo combinado de volta no repositório.

O modelo copy-modify-merge pode soar um pouco caótico, mas na prática, ele funciona de forma bastante suave. Os usuários podem trabalhar em paralelo, nunca esperando uns pelos outros. Quando eles trabalham nos mesmos arquivos, verifica-se que a maioria de suas alterações concorrentes não se sobrepõe afinal; conflitos não são muito frequentes. E a quantidade de tempo que eles levam para resolver os conflitos é usualmente muito menor que o tempo perdido no sistema de locks.

No fim, tudo se reduz a um fator crítico: a comunicação entre os usuários. Quando os usuários se comunicam mal, tanto conflitos sintáticos quanto semânticos aumentam. Nenhum sistema pode forçar os usuários a se comunicarem perfeitamente, a nenhum sistema pode detectar conflitos semânticos. Portanto, não tem como confiar nesta falsa sensação de segurança de que o sistema de locking vai prevenir conflitos; na prática, o lock parece inibir a produtividade mais do que qualquer outra coisa.

Quando Lock é Necessário

Enquanto o modelo lock-modify-unlock é geralmente considerado prejudicial à colaboração, ainda há momentos em que ele é apropriado.

O modelo copy-modify-merge é baseado no pressuposto de que os arquivos são contextualmente combináveis: isto é, que os arquivos no repositório sejam majoritariamente texto plano (como código fonte). Mas para arquivos com formatos binários, como os imagens ou som, frequentemente é impossível combinar as mudanças conflitantes. Nessas situações, é realmente necessário que o arquivo seja alterado por um usuário de cada vez. Sem um acesso serializado, alguém acabará perdendo tempo em mudanças que no final serão descartadas.

Enquanto o Subversion é primariamente um sistema copy-modify-merge, ele ainda reconhece a necessidade ocasional de lock em algum arquivo e assim fornece mecanismos para isso. Este recurso será discutido mais tarde neste livro, em “Travamento”.

Subversion em Ação

Chegou a hora de passar do abstrato para o concreto. Nesta seção, nós mostraremos exemplos reais do Subversion sendo usado.

URLs do Repositório Subversion

Ao longo de todo este livro, o Subversion utiliza URLs para identificar arquivos e diretórios versionados nos repositórios. Na maior parte, esses URLs usam a sintaxe padrão, permitindo nomes do servidores e números de portas serem especificados como parte da URL:

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

Mas existem algumas nuances no manuseio de URLs pelo Subversion que são notáveis. Por exemplo, URLs contendo o método de acesso `file://` (usado para repositórios locais) precisam, de acordo com a convenção, ter como nome do servidor `localhost` ou nenhum nome de servidor:

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

Além disso, usuários do esquema `file://` em plataformas Windows precisarão utilizar um padrão de sintaxe “não-oficial” para acessar repositórios que estão na mesma máquina, mas em um drive diferente do atual drive de trabalho. Qualquer uma das seguintes sintaxes de URLs funcionarão, sendo `X` o drive onde o repositório reside:

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

Na segunda sintaxe, você precisa colocar a URL entre aspas de modo que o caracter de barra vertical não seja interpretado como um pipe. Além disso, note que a URL utiliza barras normais, enquanto no Windows os paths (não URLs) utilizam contrabarras.

URLs `file://` do Subversion não podem ser utilizadas em um browser comum da mesma forma que URLs `file://` típicas podem. Quando você tenta ver uma URL `file://` num web browser comum, ele lê e mostra o conteúdo do local examinando o sitema de arquivos diretamente. Entretanto, os recursos do Subversion existem em um sistema de arquivos virtual (veja “Camado do respositório”), e o seu browser não vai saber como interagir com este sistema de arquivos.

Por último, comvém notar que o cliente Subversion vai automaticamente codificar as URLs conforme necessário, de forma semelhante a um browser. Por exemplo, se a URL contiver espaços ou algum caracter não ASCII:

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

...então o Subversion irá aplicar "escape" aos caracteres inseguros e se comportar como se você tivesse digitado:

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

Se a URL contiver espaços, certifique-se de colocá-la entre aspas, de forma que o seu shell trate-a inteiramente como um único argumento do programa **svn**.

Cópias de Trabalho, ou Cópias Locais

Você já leu sobre as cópias de trabalho; agora vamos demonstrar como o cliente do Subversion as cria e usa.

Uma cópia de trabalho do Subversion é uma árvore de diretórios comum no seu sistema de arquivos local, contendo uma coleção de arquivos. Você pode editar esses arquivos conforme desejar, e se eles são arquivos de código fonte, você pode compilar o seu programa a partir deles da maneira usual. Sua cópia de local é sua área de trabalho privada: O Subversion jamais incorporará as mudanças de terceiros ou tornará as suas próprias alterações disponíveis para os outros, até que você explicitamente o diga para fazer isso. Você pode ter múltiplas cópias de trabalho do o mesmo projeto.

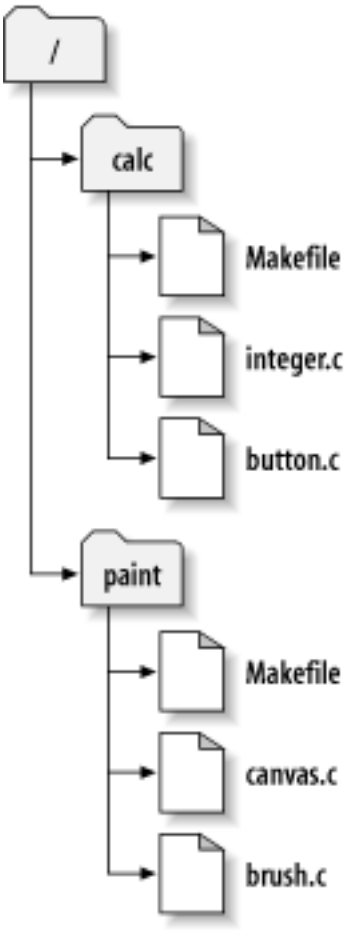
Após você ter feito algumas alterações nos arquivos de sua cópia de trabalho e verificado que elas funcionam corretamente, o Subversion lhe disponibiliza comandos para “publicar” (commit) suas alterações para as outras pessoas que estão trabalhando com você no mesmo projeto (gravando no repositório). Se outras pessoas publicarem alterações, o Subversion lhe disponibiliza comandos para combinar (merge) essas alterações em sua cópia de trabalho (lendo do repositório).

Uma cópia de trabalho também contém alguns arquivos extras, criados e mantidos pelo Subversion, para ajudá-lo a executar esse comandos. Em particular, cada diretório em sua cópia local contém um subdiretório chamado `.svn`, também conhecido como o *diretório administrativo* da cópia de local. Os arquivos em cada diretório administrativo ajudam o Subversion a reconhecer quais arquivos possuem alterções não-publicadas, e quais estão desatualizados em relação ao trabalho dos outros.

Um típico repositório Subversion frequentemente detém os arquivos (ou código fonte) para vários projetos, geralmente, cada projeto é um subdiretório na árvore de arquivos do repositório. Desse modo, uma cópia de trabalho de um normalmente corresponderá a uma sub-árvore particular do repositório.

Por exemplo, suponha que você tenha um repositório que contenha dois projetos de software, `paint` e `calc`. Cada projeto reside em seu próprio subdiretório, como é mostrado em Figura 1.6, “O Sistema de Arquivos do Repositório”.

Figura 1.6. O Sistema de Arquivos do Repositório



Para obter uma cópia local, você deve fazer *check out* de alguma sub-árvore do repositório. (O termo “check out” pode soar como algo que tem a ver com locking ou com reserva de recursos, o que não é verdade; ele simplesmente cria uma cópia privada do projeto para você.) Por exemplo, se você fizer check out de `/calc`, você receberá uma cópia de trabalho como esta:

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

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

A lista de letras A na margem esquerda indica que o Subversion está adicionando um certo número de itens à sua cópia de trabalho. Você tem agora uma cópia pessoal do diretório `/calc` do repositório, com uma entrada adicional - `.svn` - a qual detém as informações extras que o Subversion precisa, conforme mencionado anteriormente.

Suponha que você faça alterações no arquivo `button.c`. Visto que o diretório `.svn` se lembra da data de modificação e conteúdo do arquivo original, o Subversion tem como saber que você modificou o arquivo.

Entretanto o Subversion não torna as suas alterações públicas até você explicitamente lhe dizer para fazer isto. O ato de publicar as suas alterações é conhecido como *committing* (ou *checking in*) no repositório.

Para publicar as suas alterações para os outros, você deve usar o comando **commit** do Subversion.

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

Agora as suas alterações no arquivo `button.c` foram “comitadas” no repositório, com uma nota descrevendo as suas alterações (especificamente você corrigiu um erro de digitação). Se outros usuários fizerem check out de `/calc`, eles verão suas alterações na última versão do arquivo.

Suponha que você tenha um colaborador, Sally, que tenha feito check out de `/calc` ao mesmo tempo que você. Quando você publicar suas alterções em `button.c`, a cópia de trabalho de Sally será deixada intacta; o Subversion somente modifica as cópias locais quando o usuário requisita.

Para atualizar o seu projeto, Sally pede ao Subversion para realizar um *update* na cópia de trabalho dela, usando o comando **update** do Subversion. Isto irá incorporar as suas alterações na cópia local dela, bem como as alterações de todos que tenham feito um commit desde que ela fez check out.

```
$ pwd
/home/sally/calc

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

$ svn update
U    button.c
Updated to revision 57.
```

A saída do comando **svn update** indica que o Subversion atualizou o conteúdo de `button.c`. Note que Sally não precisou especificar quais arquivos seriam atualizados; o Subversion usou as informações no diretório `.svn`, e mais algumas no repositório, para decidir quais arquivos precisariam ser atualizados.

URLs do Repositório

Os repositórios do Subversion podem ser acessados através de diversos métodos - em um disco local, através de vários protocolos de rede, dependendo de como o adminstrador configurou as coisas para você. Qualquer local no repositório, entretanto, é sempre uma URL. A Tabela Tabela 1.1, “URLs de Acesso ao Repositório” descreve como diferentes esquemas de URLs mapeiam para os métodos de acesso disponíveis.

Tabela 1.1. URLs de Acesso ao Repositório

Esquema	Método de Acesso
`file:///`	acesso direto ao repositório (em um disco local).
`http://`	acesso via protocolo WebDAV em um servidor Apache especialmente configurado.
`https://`	mesmo que `http://`, mas com encriptação SSL.
`svn://`	acesso via protocolo próprio em um servidor `svnserve`.
`svn+ssh://`	mesmo que `svn://`, mas através de um túnel SSH.

Para obter mais informações sobre como o Subversion analisa as URLs, veja “URLs do Repositório Subversion”. Para obter mais informações sobre os diferentes tipos de servidores de rede disponíveis para Subversion, veja Capítulo 6, *Configuração do Servidor*.

Revisões

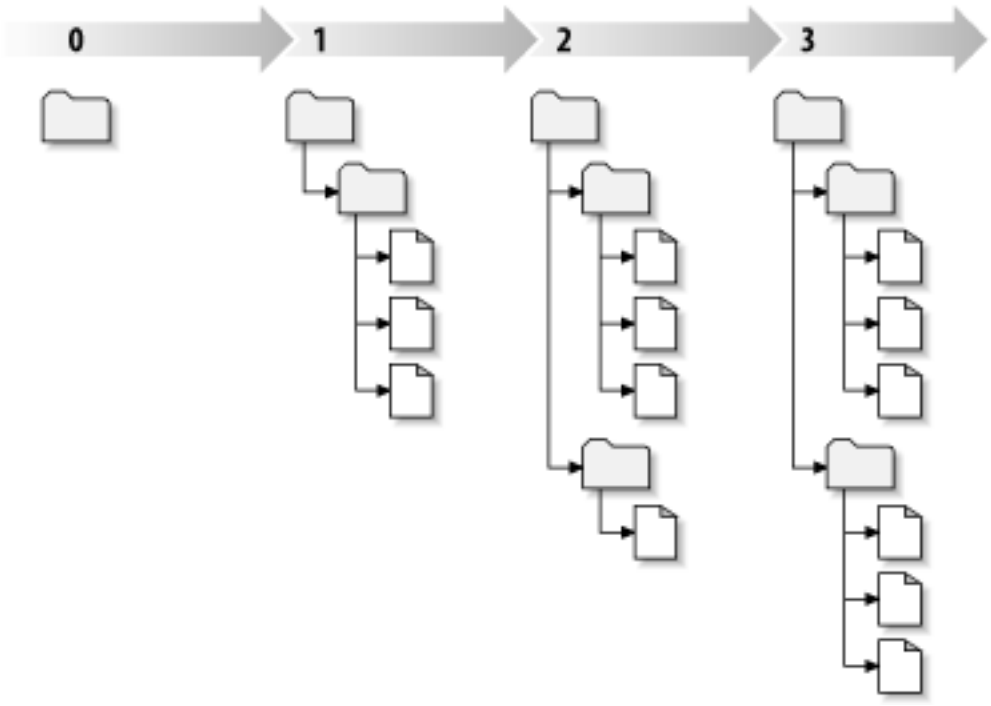
Uma operação **svn commit** publica as alterações feitas em qualquer número de arquivos o diretórios como uma única transação atômica. Em sua cópia de trabalho, você pode alterar o conteúdo de arquivos; criar, deletar, renomear e copiar arquivos e diretórios; e então comitar um conjunto completo de alterações em uma transação atômica.

Por “transação atômica”, nos entendemos simplesmente isto: Ou são efetivadas todas as alterações no repositório, ou nenhuma delas. O Subversion tenta manter esta atomicidade em face de crashes do programa ou do sistema, problemas de rede ou outras ações de usuários.

Cada vez que o repositório aceita um commit, isto cria um novo estado na árvore de arquivos, chamado *revisão*. Cada revisão é assinalada com um único número natural, incrementado de um em relação à revisão anterior. A revisão inicial de um repositório recém criado é numerada com zero, e consiste em nada além de um diretório raiz vazio.

A figura Figura 1.7, “O Repositório” ilustra uma forma simples para visualizar o repositório. Imagine um array de números de revisões, iniciando em zero, alongando-se da esquerda para a direita. Cada número de revisão tem uma árvore de arquivos pendurada abaixo dela, e cada árvore é um “snapshot” da forma como o repositório podia ser visto após um commit.

Figura 1.7. O Repositório



Números de Revisão Globais

Ao contrários de outros sistemas de controle de versão, os números de revisão do Subversion se aplicam à *árvore inteira*, não a arquivos individuais. Cada número de revisão refere-se a uma árvore inteira, um estado particular do repositório após determinadas alterações serem comitadas. Uma outra forma de pensar a respeito é imaginar que a revisão N representa o estado do sistema de arquivos do repositório após o N-ésimo commit. Quando os usuários do Subversion falam sobre a "revisão número 5 do arquivo `foo.c`", eles realmente entendem o "`foo.c` que aparece na revisão 5." Note que em geral, revisões N e M de um arquivo podem *não* ser necessariamente diferentes! Muitos outros sistemas de controle de versão usam número de revisão por arquivo, então este conceito pode parecer não usual à primeira vista. (Usuários do CVS podem querer ver Apêndice B, *Subversion para Usuários de CVS* para mais detalhes.)

É importante notar que nem sempre as cópias de trabalho correspondem a uma única revisão do repositório; elas podem conter arquivos de várias revisões diferentes. Por exemplo, suponha que você faça checkout de uma cópia de trabalho cuja revisão mais recente seja 4:

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

Neste momento, este diretório de trabalho corresponde exatamente à revisão número 4 no repositório. Contudo, suponha que você faça uma alteração no arquivo `button.c`, e publique essa alteração. Assumindo que nenhum outro commit tenha sido feito, o seu commit irá criar a revisão 5 no repositório, e sua cópia de trabalho agora irá parecer com isto:

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

Suponha que neste ponto, Sally publique uma alteração no arquivo `integer.c`, crianado a revisão 6. Se você usar o comando **svn update** para atualizar a sua cópia de trabalho, então ela irá parecer com isto:

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

A alteração de Sally no arquivo `integer.c` irá aparecer em sua cópia de trabalho, e a sua alteração no arquivo `button.c` ainda estará presente. Neste exemplo, o texto do arquivo `Makefile` é idêntico nas revisões 4, 5, e 6, mas o Subversion irá marcar a sua cópia do arquivo `Makefile` com a revisão 6 para indicar que a mesma é a corrente. Então, depois de você fazer uma atualização completa na sua cópia de trabalho, ela geralmente corresponderá exatamente a uma revisão do repositório.

Como as Cópias de Trabalho Acompanham o Repositório

Para cada arquivo em um diretório de trabalho, o Subversion registra duas peças de informações essenciais na área administrativa `.svn/`:

- em qual revisão o seu arquivo local é baseado (isto é chamado de *revisão local* do arquivo), e
- a data e a hora da última vez que a cópia local foi atualizada a partir do repositório.

Dadas estas inmformações, conversando com o repositório, o Subversion pode dizer em qual dos seguintes quatro estados um arquivo local está:

Não-Modificado, e corrente
: O arquivo não foi modificado no diretório local, e nenhuma alteração foi publicada no repositório desde a revisão corrente. O comando **svn commit** no arquivo não fará nada, e um comando **svn update** também não..

Localmente alterado, e corrente
: O arquivo foi alterado no diretório local, mas nenhuma alteração foi publicada no repositório desde o último update. existem alterações locais que ainda não foram publicadas no repositório, assim o comando **svn commit** no arquivo resultará na publicação dessas alterações, e um comando **svn update** não fará nada.

Não-Modificado, e desatualizado
: O arquivo não foi alterado no diretório local, mas foi alterado no repositório. O arquivo pode ser eventualmente atualizado, para sincronizá-lo com a última revisão pública. O comando **svn commit** no arquivo não irá fazer nada, mas o comando **svn update** irá trazer as últimas alterações para a sua cópia local.

Localmente Modificado, e desatualizado
: O arquivo foi alterado tanto no diretório loca quanto no repositório. O comando **svn commit** no arquivo irá falhar com o erro “out-of-date” (desatualizado). O arquivo deve ser atualizado primeiro; o comando **svn update** vai tentar combinar as alterações do repositório com as locais. Se o Subversion não conseguir completar a combinação de uma forma plausível automaticamente, ele deixará para o usuário resolver o conflito.

Isto pode soar como muito para acompanhar, mas o comando **svn status** mostrará para você o estado de qualquer item em seu diretório local. Para maiores informações sobre este comando, veja “See an overview of your changes”.

Revisões Locais Mistas

Como um princípio geral, o subversion tenta ser tão flexível quanto possível. Um tipo especial de flexibilidade é a capacidade de ter uma cópia local contendo arquivos e diretórios com uma mistura de diferentes revisões. Infelizmente esta flexibilidade tende a confundir inúmeros novos usuários. Se o exemplo anterior mostrando revisões mistas deixou você perplexo, aqui está um exemplo mostrando tanto a razão pela qual o fucncionalidade existe, quanto como fazer para usá-la.

Updates e Commits são Separados

Uma das regras fundamentais do Subversion é que uma ação de “push” não causa um “pull”, e vice versa. Só porque você está pronto para publicar novas alterações no repositório não significa que você está pronto para receber as alterações de outras pessoas. E se você tiver novas alterações em curso, então o comando **svn update** deveria graciosamente combinar as alterações no repositório com as suas próprias, ao invés de forçar você a publicá-las.

O principal efeito colateral dessa regra significa que uma cópia local tem que fazer uma escrituração extra para acompanhar revisões mistas, bem como ser tolerante a misturas. Isso fica mais complicado pelo fato de os diretórios também serem versionados.

Por exemplo, suponha que você tenha uma cópia local inteiramente na revisão 10. Você edita o arquivo `foo.html` e então realiza um comando **svn commit**, o qual cria a revisão 15 no repositório. Após o commit acontecer, muitos novos usuários poderiam esperar que a cópia local estivesse na revisão 15, mas este não é o caso! Qualquer número de alterações poderia ter acontecido no repositório entre as revisões 10 e 15. O cliente nada sabe sobre essas alterações no repositório, pois você ainda não executou o comando **svn update** , e o comando **svn commit** não baixou as novas alterações no repositório. Se por outro lado, o comando **svn commit** *tivesse* feito o download das novas alterações automaticamente, então seria possível que a cópia local inteira estivesse na revisão 15 - mas então nós teríamos quebrado a regra fundamental onde “push” e “pull” permanecem como ações separadas. Portanto a única coisa segura que o cliente Subversion pode fazer é marcar o arquivo - `foo.html` com a revisão 15. O restante da cópia local permanece na revisão 10. Somente executando o comando **svn update** as alterações mais recentes no repositório serão baixadas, o a cópia local inteira será marcada com a revisão 15.

Revisões misturadas são normais

O fato é, *cada vez* que você executar um comando **svn commit**, sua cópia local acabará tendo uma mistura de revisões. As coisas que você acabou de publicar são marcadas com um número de revisão maior que todo o resto. Após vários commits (sem updates entre eles) sua cópia local irá conter uma completa mistura de revisões. Mesmo que você seja a única pessoa utilizando o repositório, você ainda verá este fenômeno. Para analisar a sua mistura de revisões use o comando **svn status --verbose** (veja “See an overview of your changes” para maiores informações.)

Frequentemente, os novos usuários nem tomam consciência de que suas cópias locais contêm revisões mistas. Isso pode ser confuso, pois muitos comandos no cliente são sensíveis às revisões que eles estão examinando. Por exemplo, o comando **svn log** é usado para mostrar o histórico de alterações em um arquivo ou diretório (veja “Generating a list of historical changes”). Quando o usuário invoca este comando em um objeto da cópia local, ele espera ver o histórico inteiro do objeto. Mas se a revisão local do objeto é muito velha (muitas vezes porque o comando **svn update** não foi executado por um longo tempo), então o histórico da versão *antiga* do objeto é que será mostrado.

Revisões mistas são úteis

Se o seu projeto for suficientemente complexo, você irá descobrir que algumas vezes é interessante forçar um *backdate* (ou, atualizar para uma revisão mais antiga que a que você tem) de partes de sua cópia local para revisões anteriores; vocÇe irá aprender como fazer isso em Capítulo 2, *Uso Básico*. Talvez você queira testar uma versão anterior de um sub-módulo contido em um subdiretório, ou talvez queira descobrir quando um bug apareceu pela primeira vez eu arquivo específico. Este é o aspecto de “máquina do tempo” de um sitema de controle de versão - a funcionalidade que te permite mover qualquer parte de sua cópia local para frente ou para trás na história.

Revisões mistas têm limitações

Apesar de você poder fazer uso de revisõesmistas em seu ambiente local, esta flexibilidade tem limitações.

Primeiramente, você não pode publicar a deleção de um arquivo ou diretório que não esteja completamente atualizado. Se uma versão mais nova do item existe no repositório, sua tentativa de deleção será rejeitada, para prevenir que você acidentalmente destrua alterações que você ainda não viu.

Em segundo lugar, você não pode publicar alterações em meta-dados de diretórios a menos que ele esteja completamente atualizado. Você irá aprender a anexar “propriedades” aos itens em Capítulo 3, *Tópicos Avançados*. Uma revisão em um dretório local define um conjunto específico de entradas e propriedades, e assim, publicar alterações em propriedades de um diretório desatualizado pode destruie propriedades que você ainda não viu.

Sumário

Nós abordamos uma série de conceitos fundamentais do Subversion neste capítulo:

- Nós introduzimos as noções de repositório central, cópia local do cliente, e o array de árvores de revisões.
- Vimos alguns exemplos simples de como dois colaboradores podem utilizar o Subversion para publicar e receber as alterações um do outro, utilizando o modelo “ copy-modify-merge”.
- Nós falamos um pouco sobre a maneira como o Subversion acompanha e gerencia as informações de uma cópia local do repositório.

Neste ponto, você deve ter uma boa idéia de como o Subversion funciona no sentido mais geral. Com este conhecimento, você já deve estar pronto para avançar para o próximo capítulo, que é um relato detalhado dos comandos e recursos do Subversion.

Capítulo 2. Uso Básico

Agora entraremos em detalhes do uso do Subversion. Quando chegar ao final deste capítulo, você será capaz de realizar todas as tarefas necessárias para usar Subversion em um dia normal de trabalho. Iniciará acessando seus arquivos que estão no Subversion, após um checkout inicial de seu código. Guiaremos você pelo processo de fazer modificações e examinar estas modificações. Também verá como trazer mudanças feitas por outros para sua cópia de trabalho, examiná-las, e resolver quaisquer conflitos que possam surgir.

Note que este capítulo não pretende ser uma lista exaustiva de todos os comandos do Subversion—antes, é uma introdução conversacional às tarefas mais comuns que você encontrará no Subversion. Este capítulo assume que você leu e entendeu o Capítulo 1, *Conceitos Fundamentais* e está familiarizado com o modelo geral do Subversion. Para uma referência completa de todos os comandos, veja Capítulo 9, *Referência Completa do Subverion*.

Help!

Antes de continuar a leitura, aqui está o comando mais importante que você precisará quando usar o Subversion: **svn help**. A linha de comando do cliente Subversion é auto-documentada—a qualquer momento, um rápido **svn help *SUBCOMANDO*** descreverá a sintaxe, opções, e comportamento do subcomando.

```
$ svn help import
import: Faz commit de um arquivo não versionado ou árvore no repositório.
uso: import [CAMINHO] URL

  Recursivamente faz commit de uma cópia de CAMINHO para URL.
  Se CAMINHO é omitido '.' é assumido.
  Diretórios pais são criados conforme necessário no repositório.
  Se CAMINHO é um diretório, seu conteúdo será adicionado diretamente
  abaixo de URL.

Opções válidas:
  -q [--quiet]             : imprime o mínimo possível
  -N [--non-recursive]     : opera somente em um diretório
…
```

Colocando dados em seu Repositório

Há dois modos de colocar novos arquivos em seu repositório Subversion: **svn import** e **svn add**. Discutiremos **svn import** aqui e **svn add** mais adiante neste capítulo quando analisarmos um dia típico com o Subversion.

svn import

O comando **svn import** é um modo rápido para copiar uma árvore de arquivos não versionada em um repositório, criando diretórios intermediários quando necessário. **svn import** não requer uma cópia de trabalho, e seus arquivos são imediatamente submetidos ao repositório. Este é tipicamente usado quando você tem uma árvore de arquivos existente que você quer monitorar em seu repositório Subversion. Por exemplo:

```
$ svnadmin create /usr/local/svn/newrepos
$ svn import mytree file:///usr/local/svn/newrepos/some/project \
             -m "Importação inicial"
Adicionando    mytree/foo.c
Adicionando    mytree/bar.c
Adicionando    mytree/subdir
Adicionando    mytree/subdir/quux.h

Commit da revisão 1.
```

O exemplo anterior copiou o conteúdo do diretório `mytree` no diretório `some/project` no repositório:

```
$ svn list file:///usr/local/svn/newrepos/some/project
bar.c
foo.c
subdir/
```

Note que após a importação finalizar, a árvore inicial *não* está convertida em uma cópia de trabalho. Para começar a trabalhar, você ainda precisa obter(**svn checkout**) uma nova cópia de trabalho da árvore.

Layout de repositório recomendado

Enquanto a flexibilidade do Subversion permite que você organize seu repositório da forma que você escolher, nós recomendamos que você crie um diretório `trunk` para armazenar a “linha principal” de desenvolvimento, um diretório `branches` para conter cópias ramificadas, e um diretório `tags` para conter cópias rotuladas, por exemplo:

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

Você aprenderá mais sobre tags e branches no Capítulo 4, *Fundir e Ramificar*. Para detalhes e como configurar multiplos projetos, veja “Repository Layout” e “Planning Your Repository Organization” para ler mais sobre “raizes dos projetos”.

Checkout Inicial

Na maioria das vezes, você começa a usar um repositório Subversion por efetuar um *checkout* de seu projeto. Fazer um checkout de um repositório cria uma “cópia de trabalho” em sua máquina local. Esta cópia contém o `HEAD` (revisão mais recente) do repositório Subversion que você especificou na linha de comando:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A    trunk/Makefile.in
A    trunk/ac-helpers
A    trunk/ac-helpers/install.sh
A    trunk/ac-helpers/install-sh
A    trunk/build.conf
…
Gerado cópia de trabalho para revisão 8810.
```

O que há em um Nome?

Subversion tenta arduamente não limitar o tipo de dado que você pode colocar sob controle de versão. O conteúdo dos arquivos e valores de propriedades são armazenados e transmitidos como dados binários, e “Tipo de Conteúdo do Arquivo” diz-lhe como dar ao Subversion uma dica de que operações “textuais” não têm sentido para um arquivo em particular. Há umas poucas ocasiões, porém, onde o Subversion coloca restrições sobre as informações nele armazenadas.

O Subversion manipula internamente determinados fragmentos de dados—por exemplo, nomes de propriedades, nomes de caminhos, e mensagens de log—como Unicode codificado em UTF-8. Porém, isto não quer dizer que todas suas interações com o Subversion devam envolver UTF-8. Como uma regra geral, clientes Subversion graciosa e transparentemente manipulará conversões entre UTF-8 e o sistema de codificação em uso em seu computador, caso tal conversão possa ser feita de forma significativa (o que é o caso das codificações mais comuns em uso hoje).

Adicionalmente, nomes de caminhos são usados como valores de atributos XML nas trocas WebDAV, bem como em alguns arquivos internamente mantidos pelo Subversion. Isto significa que nomes de caminhos podem somente conter caracteres aceitos no XML (1.0). Subversion também proibe os caracteres TAB, CR, e LF em nomes de caminhos para prevenir que caminhos sejam quebrados nos diffs, ou em saídas de comandos como svn log ou svn status.

Embora pareça que há muito o que recordar, na pratica estas limitações raramente são um problema. Enquanto suas configurações regionais são compatíveis com UTF-8, e você não usar caracteres de controle nos nomes dos caminhos, você não terá problemas na comunicação com o Subversion. O cliente de linha de comando dá um pouco de ajuda extra—ele automaticamente adiciona informações de escape para os caracteres ilegais nos caminhos em URLs que você digita para criar versões “legalmente corretas” para uso interno quando necessário.

Embora os exemplos acima efetuem o checkout do diretório trunk, você pode facilmente efetuar o checkout em qualquer nível de subdiretórios de um repositório por especificar o subdiretório na URL do checkout:

```
$ svn checkout \
      http://svn.collab.net/repos/svn/trunk/subversion/tests/cmdline/
A    cmdline/revert_tests.py
A    cmdline/diff_tests.py
A    cmdline/autoprop_tests.py
A    cmdline/xmltests
A    cmdline/xmltests/svn-test.sh
…
Gerado cópia de trabalho para revisão 8810.
```

Uma vez que o Subversion usa um modelo “copiar-modificar-fundir” ao invés de “travar-modificar-destravar” (veja “Modelos de Versionamento”), você pode iniciar por fazer alterações nos arquivos e diretórios em sua cópia de trabalho. Sua cópia de trabalho é igual a qualquer outra coleção de arquivos e diretórios em seu sistema. Você pode editá-los e alterá-los, movê-los, você pode até mesmos apagar toda sua cópia de trabalho e esquecê-la.

Apesar de sua cópia de trabalho ser “igual a qualquer outra coleção de arquivos e diretórios em seu sistema”, você pode editar os arquivos a vontade, mas tem que informar o Subversion sobre *tudo o mais* que você fizer. Por exemplo, se você quiser copiar ou mover um item em uma cópia de trabalho, você deve usar os comandos **svn copy** or **svn move** em vez dos comandos copiar e mover fornecidos por ser sistema operacional. Nós falaremos mais sobre eles posteriormente neste capítulo.

A menos que você esteja pronto para submeter a adição de novos arquivos ou diretórios, ou modificações nos já existentes, não há necessidade de continuar a notificar o servidor Subversion que você tenha feito algo.

O que há no diretório `.svn`?

Cada diretório em uma cópia de trabalho contém uma área administrativa, um subdiretório nomeado `.svn`. Normalmente, comandos de listagem de diretórios não mostrarão este subdiretório, mas este é um diretório importante. Faça o que fizer, não apague ou modifique nada nesta área administrativa! O Subversion depende dela para gerenciar sua cópia de trabalho.

Se você remover o subdiretório `.svn` acidentalmente, o modo mais fácil de resolver o problema é remover todo o conteúdo do diretório (uma exclusão normal pelo sistema, não **svn delete**), então executar **svn update** apartir do diretório pai. O cliente Subversion fará novamente o download do diretório que você excluiu, bem como uma nova área `.svn`.

Além de você certamente poder obter uma cópia de trabalho com a URL do repositório como único argumento, você também pode especificar um diretório após a URL do repositório. Isto coloca sua cópia de trabalho no novo diretório que você informou. Por exemplo:

```
$  svn checkout http://svn.collab.net/repos/svn/trunk subv
A    subv/Makefile.in
A    subv/ac-helpers
A    subv/ac-helpers/install.sh
A    subv/ac-helpers/install-sh
A    subv/build.conf
…
Gerado cópia de trabalho para revisão 8810.
```

Isto colocará sua cópia de trabalho em um diretório chamado `subv` em vez de um diretório chamado `trunk` como fizemos anteriormente. O diretório `subv` será criado se ele não existir.

Desabilitando o Cache de Senhas

Quando você realiza uma operação no Subversion que requer autenticação, por padrão o Subversion mantém suas credenciais de autenticação num cache em disco. Isto é feito por conveniencia, para que você não precise continuamente ficar re-digitando sua senha em operações futuras. Se você estiver preocupado com o fato de o Subversion manter um cache de suas senhas, [1] você pode desabilitar o cache de forma permanente ou analisando caso a caso.

Para desabilitar o cache de senhas para um comando específico uma vez, passe a opção `--no-auth-cache` na linha de comando. Para desabilitar permanentemente o cache, você pode adicionar a linha `store-passwords = no` no arquivo de configuração local do seu Subversion. Veja “Client Credentials Caching” para maiores detalhes.

Autenticando como um Usuário Diferente

Uma vez que por default o Subversion mantém um cache com as credenciais de autenticação (tanto usuário quanto senha), ele convenientemente se lembra que era você estava ali na última vez que você modificou sua cópia de trabalho. Mas algumas vezes isto não é útil — particularmente se você estava trabalhando

[1] É claro, você não está terrivelmente preocupado — primeiro porque você sabe que você não pode *realmente* deletar nada do Subversion e, em segundo lugar, porque sua senha do Subversion não é a mesma que as outras três milhões de senhas que você tem, certo? Certo?

numa cópia de trabalho compartilhada, como um diretório de configuração do sistema ou o documento raiz de um servidor web. Neste caso, apenas passe a opção `--username` na linha de comando e o Subversion tentará autenticar como aquele usuário, pedindo uma senha se necessário.

Basic Work Cycle

Subversion has numerous features, options, bells and whistles, but on a day-to-day basis, odds are that you will only use a few of them. In this section we'll run through the most common things that you might find yourself doing with Subversion in the course of a day's work.

The typical work cycle looks like this:

- Update your working copy
  - **svn update**
- Make changes
  - **svn add**
  - **svn delete**
  - **svn copy**
  - **svn move**
- Examine your changes
  - **svn status**
  - **svn diff**
- Possibly undo some changes
  - **svn revert**
- Resolve Conflicts (Merge Others' Changes)
  - **svn update**
  - **svn resolved**
- Commit your changes
  - **svn commit**

Update Your Working Copy

When working on a project with a team, you'll want to update your working copy to receive any changes made since your last update by other developers on the project. Use **svn update** to bring your working copy into sync with the latest revision in the repository.

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

In this case, someone else checked in modifications to both `foo.c` and `bar.c` since the last time you updated, and Subversion has updated your working copy to include those changes.

When the server sends changes to your working copy via **svn update**, a letter code is displayed next to each item to let you know what actions Subversion performed to bring your working copy up-to-date. To find out what these letters mean, see svn update.

Make Changes to Your Working Copy

Now you can get to work and make changes in your working copy. It's usually most convenient to decide on a discrete change (or set of changes) to make, such as writing a new feature, fixing a bug, etc. The Subversion commands that you will use here are **svn add**, **svn delete**, **svn copy**, **svn move**, and **svn mkdir**. However, if you are merely editing files that are already in Subversion, you may not need to use any of these commands until you commit.

There are two kinds of changes you can make to your working copy: file changes and tree changes. You don't need to tell Subversion that you intend to change a file; just make your changes using your text editor, word processor, graphics program, or whatever tool you would normally use. Subversion automatically detects which files have been changed, and in addition handles binary files just as easily as it handles text files—and just as efficiently too. For tree changes, you can ask Subversion to “mark” files and directories for scheduled removal, addition, copying, or moving. These changes may take place immediately in your working copy, but no additions or removals will happen in the repository until you commit them.

Here is an overview of the five Subversion subcommands that you'll use most often to make tree changes.

Versioning symbolic links

On non-Windows platforms, Subversion is able to version files of the special type *symbolic link* (or, “symlink”). A symlink is a file which acts as a sort of transparent reference to some other object in the filesystem, allowing programs to read and write to those objects indirectly by way of performing operations on the symlink itself.

When a symlink is committed into a Subversion repository, Subversion remembers that the file was in fact a symlink, as well as the object to which the symlink “points”. When that symlink is checked out to another working copy on a non-Windows system, Subversion reconstructs a real filesystem-level symbolic link from the versioned symlink. But that doesn't in any way limit the usability of working copies on systems such as Windows which do not support symlinks. On such systems, Subversion simply creates a regular text file whose contents are the path to which to the original symlink pointed. While that file can't be used as a symlink on a Windows system, it also won't prevent Windows users from performing their other Subversion-related activities.

svn add foo

Schedule file, directory, or symbolic link `foo` to be added to the repository. When you next commit, `foo` will become a child of its parent directory. Note that if `foo` is a directory, everything underneath `foo` will be scheduled for addition. If you only want to add `foo` itself, pass the `--non-recursive (-N)` option.

svn delete foo

Schedule file, directory, or symbolic link `foo` to be deleted from the repository. If `foo` is a file or link, it is immediately deleted from your working copy. If `foo` is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, `foo` will be entirely removed from your working copy and the repository. [2]

[2] Of course, nothing is ever totally deleted from the repository—just from the HEAD of the repository. You can get back anything you delete by checking out (or updating your working copy to) a revision earlier than the one in which you deleted it. Also see “Resurrecting Deleted Items”.

svn copy foo bar
Create a new item `bar` as a duplicate of `foo` and automatically schedule `bar` for addition. When `bar` is added to the repository on the next commit, its copy history is recorded (as having originally come from `foo`). **svn copy** does not create intermediate directories.

svn move foo bar
This command is exactly the same as running **svn copy foo bar; svn delete foo**. That is, `bar` is scheduled for addition as a copy of `foo`, and `foo` is scheduled for removal. **svn move** does not create intermediate directories.

svn mkdir blort
This command is exactly the same as running **mkdir blort; svn add blort**. That is, a new directory named `blort` is created and scheduled for addition.

Changing the Repository Without a Working Copy

There *are* some use cases that immediately commit tree changes to the repository. This only happens when a subcommand is operating directly on a URL, rather than on a working-copy path. In particular, specific uses of **svn mkdir**, **svn copy**, **svn move**, and **svn delete** can work with URLs (And don't forget that **svn import** always makes changes to a URL).

URL operations behave in this manner because commands that operate on a working copy can use the working copy as a sort of “staging area” to set up your changes before committing them to the repository. Commands that operate on URLs don't have this luxury, so when you operate directly on a URL, any of the above actions represent an immediate commit.

Examine Your Changes

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can make a more accurate log message. You may also discover that you've inadvertently changed a file, and this gives you a chance to revert those changes before committing. Additionally, this is a good opportunity to review and scrutinize changes before publishing them. You can see an overview of the changes you've made by using **svn status**, and dig into the details of those changes by using **svn diff**.

Look Ma! No Network!

The commands **svn status**, **svn diff**, and **svn revert** can be used without any network access even if your repository *is* across the network. This makes it easy to manage your changes-in-progress when you are somewhere without a network connection, such as travelling on an airplane, riding a commuter train or hacking on the beach.[3]

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

Subversion has been optimized to help you with this task, and is able to do many things without communicating with the repository. In particular, your working copy contains a hidden cached “pristine” copy

[3] And also that you don't have a WAN card. Thought you got us, huh?

of each version controlled file within the `.svn` area. Because of this, Subversion can quickly show you how your working files have changed, or even allow you to undo your changes without contacting the repository.

See an overview of your changes

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

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

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

```
A       stuff/loot/bloo.h   # file is scheduled for addition
C       stuff/loot/lump.c   # file has textual conflicts from an update
D       stuff/fish.c        # file is scheduled for deletion
M       bar.c               # the content in bar.c has local modifications
```

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

`A item`
: The file, directory, or symbolic link `item` has been scheduled for addition into the repository.

`C item`
: The file `item` is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy. You must resolve this conflict before committing your changes to the repository.

`D item`
: The file, directory, or symbolic link `item` has been scheduled for deletion from the repository.

`M item`
: The contents of the file `item` have been modified.

If you pass a specific path to **svn status**, you get information about that item alone:

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

svn status also has a `--verbose (-v)` option, which will show you the status of *every* item in your working copy, even if it has not been changed:

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

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

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

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

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

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

Examine the details of your local modifications

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

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

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

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

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

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

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

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

```
$ svn diff > patchfile
```

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

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

Undoing Working Changes

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

This is a perfect opportunity to use **svn revert**:

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

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

```
$ svn status foo
?      foo

$ svn add foo
A         foo

$ svn revert foo
Reverted 'foo'

$ svn status foo
?      foo
```

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

Or perhaps you mistakenly removed a file from version control:

```
$ svn status README
       README

$ svn delete README
D         README

$ svn revert README
Reverted 'README'

$ svn status README
       README
```

Resolve Conflicts (Merging Others' Changes)

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

```
$ svn update
U  INSTALL
G  README
C  bar.c
Updated to revision 46.
```

The `U` and `G` codes are no cause for concern; those files cleanly absorbed changes from the repository. The files marked with `U` contained no local changes but were `U`pdated with changes from the repository. The `G`

stands for merGed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap with the local changes.

But the C stands for conflict. This means that the changes from the server overlapped with your own, and now you have to manually choose between them.

Whenever a conflict occurs, three things typically occur to assist you in noticing and resolving that conflict:

- Subversion prints a C during the update, and remembers that the file is in a state of conflict.
- If Subversion considers the file to be mergeable, it places *conflict markers*—special strings of text which delimit the “sides” of the conflict—into the file to visibly demonstrate the overlapping areas. (Subversion uses the `svn:mime-type` property to decide if a file is capable of contextual, line-based merging. See “Tipo de Conteúdo do Arquivo” to learn more.)
- For every conflicted file, Subversion places three extra unversioned files in your working copy:

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

  `filename.rOLDREV`
  : This is the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.

  `filename.rNEWREV`
  : This is the file that your Subversion client just received from the server when you updated your working copy. This file corresponds to the HEAD revision of the repository.

  Here OLDREV is the revision number of the file in your `.svn` directory and NEWREV is the revision number of the repository HEAD.

For example, Sally makes changes to the file `sandwich.txt` in the repository. Harry has just changed the file in his working copy and checked it in. Sally updates her working copy before checking in and she gets a conflict:

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

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

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

If you get a conflict, you need to do one of three things:

- Merge the conflicted text “by hand” (by examining and editing the conflict markers within the file).
- Copy one of the temporary files on top of your working file.
- Run **svn revert <filename>** to throw away all of your local changes.

Once you've resolved the conflict, you need to let Subversion know by running **svn resolved**. This removes the three temporary files and Subversion no longer considers the file to be in a state of conflict.[4]

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

Merging Conflicts by Hand

Merging conflicts by hand can be quite intimidating the first time you attempt it, but with a little practice, it can become as easy as falling off a bike.

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

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

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

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

[4] You can always remove the temporary files yourself, but would you really want to do that when Subversion can do it for you? We didn't think so.

The text between the second and third sets of conflict markers is the text from Sally's commit:

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

Usually you won't want to just delete the conflict markers and Sally's changes—she's going to be awfully surprised when the sandwich arrives and it's not what she wanted. So this is where you pick up the phone or walk across the office and explain to Sally that you can't get sauerkraut from an Italian deli.[5] Once you've agreed on the changes you will check in, edit your file and remove the conflict markers.

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

Now run **svn resolved**, and you're ready to commit your changes:

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

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

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

Copying a File Onto Your Working File

If you get a conflict and decide that you want to throw out your changes, you can merely copy one of the temporary files created by Subversion over the file in your working copy:

```
$ 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
```

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

```
$ svn resolved sandwich.txt
```

Punting: Using svn revert

If you get a conflict, and upon examination decide that you want to throw out your changes and start your edits again, just revert your changes:

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

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

Commit Your Changes

Finally! Your edits are finished, you've merged all changes from the server, and you're ready to commit your changes to the repository.

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

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

However, if you've been composing your log message as you work, you may want to tell Subversion to get the message from a file by passing the filename with the `--file (-F)` option:

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

If you fail to specify either the `--message` or `--file` option, then Subversion will automatically launch your favorite editor (see the `editor-cmd` section in “Config”) for composing a log message.

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

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

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

```
$
```

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

```
$ svn commit -m "Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: Your file or directory 'sandwich.txt' is probably out-of-date
…
```

(The exact wording of this error message depends on the network protocol and server you're using, but the idea is the same in all cases.)

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

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

Examining History

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

There are several commands that can provide you with historical data from the repository:

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

svn diff
: Shows line-level details of a particular change.

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

svn list
: Displays the files in a directory for any given revision.

Generating a list of historical changes

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

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

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

Added main() methods.
------------------------------------------------------------------------
r1 | sally | Mon, 15 Jul 2002 17:40:08 -0500 | 1 line

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

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

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

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

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

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

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

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

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

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

Frozzled the sub-space winch.

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

svn log also takes a `--quiet` (`-q`) option, which suppresses the body of the log message. When combined with `--verbose`, it gives just the names of the changed files.

Why Does svn log Give Me an Empty Response?

After working with Subversion for a bit, most users will come across something like this:

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

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

Examining the details of historical changes

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

In fact, it turns out that there are *three* distinct uses of **svn diff**:

- Examining local changes
- Comparing your working copy to the repository
- Comparing repository to repository

Examining Local Changes

As we've seen, invoking **svn diff** with no options will compare your working files to the cached “pristine” copies in the `.svn` area:

```
$ 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
$
```

Comparing Working Copy to Repository

If a single `--revision` (`-r`) number is passed, then your working copy is compared to the specified revision in the repository.

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

Comparing Repository to Repository

If two revision numbers, separated by a colon, are passed via `--revision (-r)`, then the two revisions are directly compared.

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

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

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

Lastly, you can compare repository revisions even when you don't have a working copy on your local machine, just by including the appropriate URL on the command line:

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

```
…
$
```

Browsing the repository

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

svn cat

If you want to examine an earlier version of a file and not necessarily the differences between two files, you can use **svn cat**:

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

You can also redirect the output directly into a file:

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

svn list

The **svn list** command shows you what files are in a repository directory without actually downloading the files to your local machine:

```
$ svn list http://svn.collab.net/repos/svn
README
branches/
clients/
tags/
trunk/
```

If you want a more detailed listing, pass the `--verbose (-v)` flag to get output like this:

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

The columns tell you the revision at which the file or directory was last modified, the user who modified it, the size if it is a file, the date it was last modified, and the item's name.

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

Fetching older repository snapshots

In addition to all of the above commands, you can use **svn update** and **svn checkout** with the `--revision` option to take an entire working copy “back in time” [6]:

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

Many Subversion newcomers attempt to use the above **svn update** example to “undo” committed changes, but this won't work as you can't commit changes that you obtain from backdating a working copy if the changed files have newer revisions. See “Resurrecting Deleted Items” for a description of how to “undo” a commit.

Lastly, if you're building a release and wish to bundle up your files from Subversion but don't want those pesky `.svn` directories in the way, then you can use **svn export** to create a local copy of all or part of your repository sans `.svn` directories. As with **svn update** and **svn checkout**, you can also pass the `--revision` option to **svn export**:

```
$ svn export http://svn.example.com/svn/repos1 # Exports latest revision
…
$ svn export http://svn.example.com/svn/repos1 -r 1729
# Exports revision r1729
…
```

Sometimes You Just Need to Clean Up

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

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

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

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

[6] See? We told you that Subversion was a time machine.

Don't confuse these working copy locks with the ordinary locks that Subversion users create when using the “lock-modify-unlock” model of concurrent version control; see Os três significados de “trava” for clarification.

Summary

Now we've covered most of the Subversion client commands. Notable exceptions are those dealing with branching and merging (see Capítulo 4, *Fundir e Ramificar*) and properties (see “Properties”). However, you may want to take a moment to skim through Capítulo 9, *Referência Completa do Subverion* to get an idea of all the many different commands that Subversion has—and how you can use them to make your work easier.

Capítulo 3. Tópicos Avançados

Se você está lendo este livro capítulo por capítulo, do inicio ao fim, você deve agora ter adquirido conhecimentos suficientes para usar o cliente Subversion para executar a operações de controle de verão mais comuns. Você entendeu como obter uma cópia de trabalho de um repositório Subversion. Você sente-se confortável para submeter e receber mudanças usando as funções **svn commit** e **svn update** . Você provavelmente desenvolveu um reflexo que lhe impele a executar o comando **svn status** quase inconsientemente. Para todos os intentos e propósitos, você está pronto para usar o Subversion em um ambiente típico.

Mas o conjunto de recursos do Subversion não para nas “operações de controle de versão comuns”. Ele tem outras pequenas funcionalidades além de cumunicar mudanças de arquivos e diretórios para e a partir de um repositório central.

Este capítulo destaca alguns dos recursos do Subversion que, apesar de importantes, não fazem pare da rotina diária de um usuário típico. Ele assume que você está familiarizado com capacidades básicas de controle de versão sobre arquivos e diretórios. Se não está, você vai querer ler primeiro o Capítulo 1, *Conceitos Fundamentais* e Capítulo 2, *Uso Básico*. Uma vez tenha dominado estes fundamentos e terminado este capítulo, você será um usuário avançado do Subversion!

Revision Specifiers

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

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

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

Revision Keywords

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

HEAD
: The latest (or “youngest”) revision in the repository.

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

COMMITTED
: The most recent revision prior to, or equal to, BASE, in which an item changed.

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

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

Here are some examples of revision keywords in action:

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

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

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

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

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

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

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

Revision Dates

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

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

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

When you specify a date, Subversion resolves that date to the most recent revision of the repository as of that date, and then continues to operate against that resolved revision number:

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

Is Subversion a Day Early?

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

If you want to include the 27th in your search, you can either specify the 27th with the time (`{"2006-11-27 23:59"}`), or just specify the next day (`{2006-11-28}`).

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

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

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

Properties

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

But it doesn't stop there.

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

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

Properties show up elsewhere in Subversion, too. Just as files and directories may have arbitrary property names and values attached to them, each revision as a whole may have arbitrary properties attached to it. The same constraints apply—human-readable names and anything-you-want binary values. The main difference is that revision properties are not versioned. In other words, if you change the value of, or delete, a revision property, there's no way within the scope of Subversion's functionality to recover the previous value.

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

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

Why Properties?

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

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

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

Now consider the same website deployed in a way that makes use of Subversion's file properties. Imagine having a single image file, `image123.jpg`, and then properties set on that file named `caption`,

`datestamp`, and even `thumbnail`. Now your working copy directory looks much more manageable—in fact, it looks to the casual browser like there are nothing but image files in it. But your automation scripts know better. They know that they can use **svn** (or better yet, they can use the Subversion language bindings—see “Using the APIs”) to dig out the extra information that your site needs to display without having to read an index file or play path manipulation games.

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

Searchability (or, Why *Not* Properties)

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

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

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

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

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

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

Manipulating Properties

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

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

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

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

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

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

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

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

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

All of this property adding and editing isn't really very useful if you can't easily get the stored property value. So the **svn** program supplies two subcommands for displaying the names and values of properties stored on files and directories. The **svn proplist** command will list the names of properties that exist on a path. Once you know the names of the properties on the node, you can request their values individually using **svn propget**. This command will, given a property name and a path (or set of paths), print the value of the property to the standard output stream.

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

[1] If you're familiar with XML, this is pretty much the ASCII subset of the syntax for XML "Name".

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

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

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

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

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

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

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

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

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

[2] Fixing spelling errors, grammatical gotchas, and “just-plain-wrongness” in commit log messages is perhaps the most common use case for the `--revprop` option.

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

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

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11 --revprop \
              http://svn.example.com/repos/project
property 'svn:log' set on repository revision '11'
$
```

Note that the ability to modify these unversioned properties must be explicitly added by the repository administrator (see “Commit Log Message Correction”). That's because the properties aren't versioned, so you run the risk of losing information if you aren't careful with your edits. The repository administrator can set up methods to protect against this loss, and by default, modification of unversioned properties is disabled.

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

Properties and the Subversion Workflow

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

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

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

$
```

Notice how the **status** subcommand displays `M` in the second column instead of the first. That is because we have modified the properties on `calc/button.c`, but not its textual contents. Had we changed both, we would have seen `M` in the first column, too (see “See an overview of your changes”).

Property Conflicts

As with file contents, local property modifications can conflict with changes committed by someone else. If you update your working copy directory and receive property changes on a versioned object that clash with your own, Subversion will report that the object is in a conflicted state.

```
% svn update calc
M  calc/Makefile.in
 C calc/button.c
Updated to revision 143.
$
```

Subversion will also create, in the same directory as the conflicted object, a file with a `.prej` extension which contains the details of the conflict. You should examine the contents of this file so you can decide how to resolve the conflict. Until the conflict is resolved, you will see a `C` in the second column of **svn status** output for that object, and attempts to commit your local modifications will fail.

```
$ svn status calc
 C     calc/button.c
?      calc/button.c.prej
$ cat calc/button.c.prej
prop 'linecount': user set to '1256', but update set to '1301'.
$
```

To resolve property conflicts, simply ensure that the conflicting properties contain the values that they should, and then use the **svn resolved** command to alert Subversion that you have manually resolved the problem.

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

Automatic Property Setting

Properties are a powerful feature of Subversion, acting as key components of many Subversion features discussed elsewhere in this and other chapters—textual diff and merge support, keyword substitution, newline translation, etc. But to get the full benefit of properties, they must be set on the right files and directories. Unfortunately, that step can be easily forgotten in the routine of things, especially since failing to set a property doesn't usually result in an obvious error (at least compared to, say, failing to add a file to version control). To help your properties get applied to the places that need them, Subversion provides a couple of simple but useful features.

Whenever you introduce a file to version control using the **svn add** or **svn import** commands, Subversion tries to assist by setting some common file properties automatically. First, on operating systems whose filesystems support an execute permission bit, Subversion will automatically set the `svn:executable` property on newly added or imported files whose execute bit is enabled. (See “Executabilidade de Arquivo” for more about this property.) Secondly, it runs a very basic heuristic to determine if that file contains human-readable content. If not, Subversion will automatically set the `svn:mime-type` property on that file to `application/octet-stream` (the generic “this is a collection of bytes” MIME type). Of course, if Subversion guesses incorrectly, or if you wish to set the `svn:mime-type` property to something more

precise—perhaps `image/png` or `application/x-shockwave-flash`—you can always remove or edit that property. (For more on Subversion's use of MIME types, see “Tipo de Conteúdo do Arquivo”.)

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

Portabilidade de Arquivo

Felizmente, para os usuários do Subversion que rotineiramente se encontram em diferentes computadores, com diferentes sistemas operacionais, o programa de linha de comando do Subversion comporta-se quase que da mesma forma em todos os sistemas. Se você sabe como usar o **svn** em uma plataforma, você saberá como manuseá-lo em qualquer outra.

Entretanto, o mesmo nem sempre é verdade em outras classes de software em geral, ou nos atuais arquivos que você mantém no Subversion. Por exemplo, em uma máquina Windows, a definição de um “arquivo de texto” seria similar à usada em uma máquina Linux, porém com uma diferença chave—os caracteres usados para marcar o fim das linhas destes arquivos. Existem outras diferenças também. As plataformas Unix têm (e o Subversion suporta) links simbólicos; Windows não. As plataformas Unix usam as permissões do sistema de arquivos para determinar a executabilidade; Windows usa as extensões no nome do arquivo.

Pela razão de que o Subversion não está em condição de unir o mundo inteiro em definições comuns e implementações de todas estas coisas, o melhor que podemos fazer é tentar ajudar a tornar sua vida mais simples quando você precisar trabalhar com seus arquivos e diretórios versionados em múltiplos computadores e sistemas operacionais. Esta seção descreve alguns dos meios de como o Subversion faz isto.

Tipo de Conteúdo do Arquivo

O Subversion combina a qualidade das muitas aplicações que reconhecem e fazem uso dos tipos de conteúdo do *Multipurpose Internet Mail Extensions* (MIME). Além de ser um local de armazenamento de propósito geral para um tipo de conteúdo do arquivo, o valor da propriedade de arquivo `svn:mime-type` determina algumas características comportamentais do próprio Subversion.

Identificando Tipos de Arquivo

Vários programas nos sistemas operacionais mais modernos fazem suposições sobre o tipo e formato do conteúdo de um arquivo pelo nome do arquivo, especificamente por sua extensão. Por exemplo, arquivos cujos nomes terminam em `.txt` são, geralmente, supostos ser legíveis por humanos, passíveis de serem compreendidos por simples leitura, em vez dos que requerem processamento complexo para os decifrar. Por outro lado, arquivos cujos nomes terminam em `.png` assume-se serem do tipo *Portable Network Graphics*— que não são legíveis por humanos, sendo perceptíveis apenas quando interpretados pelo software que entende o formato PNG, e pode tornar a informação neste formato como uma imagem desenhada por linhas.

Infelizmente, algumas destas extensões têm seus significados modificados ao longo do tempo. Quando os computadores pessoais apareceram pela primeira vez, um arquivo chamado `README.DOC` certamente era um arquivo de texto simples, como são hoje os arquivos `.txt`. Porém, no meio dos anos de 1990, você poderia apostar que um arquivo com este nome não seria mais um arquivo de texto simples, mas sim um documento do Microsoft Word em um formato proprietário e humanamente ilegível. Mas esta mudança não ocorreu da noite para o dia—houve certamente um período de confusão para os usuário de computador sobre o que exatamente eles tinham em mãos quando viam um arquivo `.DOC`. [3]

A popularidade das redes de computadores lançou ainda mais dúvidas sobre o mapeamento entre um nome de arquivo e seu conteúdo. Com informações sendo servidas através das redes e geradas dinamicamente por scripts no servidor, freqüentemente, observava-se arquivos não reais e, portanto, sem nome. Os servidores Web, por exemplo, precisavam de algum outro modo para dizer aos navegadores que eles estavam baixando um arquivo, assim o navegador poderia fazer algo inteligente com esta informação, quer seja para exibir os dados usando um programa registrado para lidar com este tipo de dados, quer seja para solicitar ao usário onde armazenar os dados baixados.

Finalmente, um padrão surgiu para, entre outras coisas, descrever o conteúdo de um fluxo de dados. Em 1996, a RFC2045 foi publicada, a primeira de cinco RFC's descrevendo o MIME. Esta RFC descreve o conceito de tipos e subtipos de mídia, e recomenda uma sintaxe para a representação destes tipos. Hoje, os tipos de mídia MIME—ou “tipos MIME”—são usados quase que universalmente em todas as aplicações de e-mail, servidores Web e outros softwares como o mecanismo de fato para esclarecer a confusão do conteúdo de arquivo.

Por exemplo, um dos benefícios que o Subversion tipicamente fornece é a fusão contextual, baseada nas linhas, das mudanças recebidas do servidor durante uma atualização em seu arquivo de trabalho. Mas, para arquivos contendo dados não-textuais, muitas vezes não existe o conceito de “linha”. Assim, para os arquivos versionados cuja propriedade `svn:mime-type` é definida com um tipo MIME não-textual (geralmente, algo que não inicie com `text/`, embora existam exceções), o Subversion não tenta executar fusões contextuais durante as atualizações. Em vez disso, quando você modifica localmente um arquivo binário em sua cópia de trabalho, na momento da atualização, seu arquivo não é mexido, pois o Subversion cria dois novos arquivos. Um deles tem a extensão `.oldrev` e contém a revisão BASE do arquivo. O outro arquivo tem uma extensão `.newrev` e contém o conteúdo da revisão atualizada do arquivo. Este comportamento serve de proteção ao usuário contra falhas na tentativa de executar fusões contextuais nos arquivos que simplesmente não podem ser contextualmente fundidos.

Além disso, se a propriedade `svn:mime-type` estiver definida, então o módulo Apache do Subversion usará seu valor para preencher o cabeçalho HTTP `Content-type:` quando responder a solicitações GET. Isto oferece ao navegador web uma dica crucial sobre como exibir um arquivo quando você o utiliza para examinar o conteúdo de seu repositório Subversion.

[3] Você acha que foi complicado? Durante este mesmo período, o WordPerfect também usou `.DOC` como extensão para seu formato de arquivo proprietário!

Executabilidade de Arquivo

Em muitos sistemas operacionais, a capacidade de executar um arquivo como um comando é comandada pela presença de um bit de permissão para execução. Este bit, usualmente, vem desabilitado por padrão, e deve ser explicitamente habilitado pelo usuário em cada arquivo que seja necessário. Mas seria um grande incômodo ter que lembrar, exatamente, quais arquivos de uma cópia de trabalho verificada recentemente estavam com seus bits de execução habilitados, e, então, ter que trocá-los. Por esta razão, o Subversion oferece a propriedade `svn:executable`, que é um modo de especificar que o bit de execução para o arquivo no qual esta propriedade está definida deve ser habilitado, e o Subversion honra esta solicitação ao popular cópias de trabalho com tais arquivos.

Esta propriedade não tem efeito em sistemas de arquivo que não possuem o conceito de bit de permissão para executável, como, por exemplo, FAT32 e NTFS. [4] Além disso, quando não houver valor definido, o Subversion forçará o valor * ao definir esta propriedade. Por fim, esta propriedade só é válido em arquivos, não em diretórios.

Seqüência de Caracteres de Fim-de-Linha

A não você ser que esteja usando a propriedade `svn:mime-type` em um arquivo sob controle de versão, o Subversion assume que o arquivo contém dados humanamente legíveis. De uma forma geral, o Subversion somente usa esse conhecimento para determinar se os relatórios de diferenças contextuais para este arquivo são possíveis. Ao contrário, para o Subversion, bytes são bytes.

Isto significa que, por padrão, o Subversion não presta qualquer atenção para o tipo de *marcadores de fim-de-linha, ou end-of-line (EOL)* usados em seus arquivos. Infelizmente, diferentes sistemas operacionais possuem diferentes convenções sobre qual seqüência de caracteres representa o fim de uma linha de texto em um arquivo. Por exemplo, a marca usual de término de linha usada por softwares na plataforma Windows é um par de caracteres de controle ASCII—um retorno de carro (`CR`) seguido por um avanço de linha (`LF`). Os softwares em Unix, entretanto, utilizam apenas o caracter `LF` para definir o término de uma linha.

Nem todas as ferramentas nestes sistemas operacionais compreendem arquivos que contêm terminações de linha em um formato que difere do *estilo nativo de terminação de linha* do sistema operacional no qual estão executando. Assim, normalmente, programas Unix tratam o caracter `CR`, presente em arquivos Windows, como um caracter normal (usualmente representado como `^M`), e programas Windows juntam todas as linhas de um arquivo Unix dentro de uma linha enorme, porque nenhuma combinação dos caracteres de retorno de carro e avanço de linha (ou `CRLF`) foi encontrada para determinar os términos das linhas.

Esta sensibilidade quanto aos marcadores EOL pode ser frustrante para pessoas que compartilham um arquivo em diferentes sistemas operacionais. Por exemplo, considere um arquivo de código-fonte, onde desenvolvedores que editam este arquivo em ambos os sistemas, Windows e Unix. Se todos os desenvolvedores sempre usarem ferramentas que preservem o estilo de término de linha do arquivo, nenhum problema ocorrerá.

Mas na prática, muitas ferramentas comuns, ou falham ao ler um arquivo com marcadores EOL externos, ou convertem as terminações de linha do arquivo para o estilo nativo quando o arquivo é salvo. Se o precedente é verdadeiro para um desenvolvedor, ele deve usar um utilitário de conversão externo (tal como **dos2unix** ou seu similar, **unix2dos**) para preparar o arquivo para edição. O caso posterior não requer nenhuma preparação extra. Mas ambos os casos resultam em um arquivo que difere do original literalmente em cada uma das linhas! Antes de submeter suas alterações, o usuário tem duas opções. Ou ele pode utilizar um utilitário de conversão para restaurar o arquivo modificado para o mesmo estilo de término de

[4] Os sitemas de arquivos do Windows usam extensões de arquivo (tais como `.EXE`, `.BAT`, e `.COM`) para indicar arquivos executáveis.

linha utilizado antes de suas edições serem feitas. Ou ele pode simplesmente submeter o arquivo—as novas marcas EOL e tudo mais.

O resultado de cenários como estes incluem perda de tempo e modificções desnecessárias aos arquivos submetidos. A perda de tempo é suficientemente dolorosa. Mas quando submissões mudam cada uma das linhas em um arquivo, isso dificulta o trabalho de determinar quais dessas linhas foram modificadas de uma forma não trivial. Onde o bug foi realmente corrigido? Em qual linha estava o erro de sintaxe introduzido?

A solução para este problema é a propriedade `svn:eol-style`. Quando esta propriedade é definida com um valor válido, o Subversion a utiliza para determinar que tratamento especial realizar sobre o arquivo de modo que o estilo de término de linha do arquivo não fique alternando a cada submissão vinda de um sistema operacional diferente. Os valores válidos são:

`native`
: Isso faz com que o arquivo contenha as marcas EOL que são nativas ao sistema operacional no qual o Subversion foi executado. Em outras palavras, se um usuário em um computador Windows adquire uma cópia de trabalho que contém um arquivo com a propriedade `svn:eol-style` atribuída para `native`, este arquivo conterá `CRLF` como marcador EOL. Um usuário Unix adquirindo uma cópia de trabalho que contém o mesmo arquivo verá `LF` como marcador EOL em sua cópia do arquivo.

  Note que o Subversion na verdade armazenará o arquivo no repositório usando marcadores normalizados como `LF` independentemente do sistema operacional. Isto, no entanto, será essencialmente transparente para o usuário.

`CRLF`
: Isso faz com que o arquivo contenha seqüências `CRLF` como marcadores EOL, independentemente do sistema operacional em uso.

`LF`
: Isso faz com que o arquivo contenha caracteres `LF` como marcadores EOL, independentemente do sistema operacional em uso.

`CR`
: Isso faz com que o arquivo contenha caracteres `CR` como marcadores EOL, independentemente do sistema operacional em uso. Este estilo de término de linha não é muito comum. Ele foi utilizado em antigas plataformas Macintosh (nas quais o Subversion não executa regularmente).

Ignorando Itens Não-Versionados

Em qualquer cópia de trabalho obtida, há uma boa chance que juntamente com todos os arquivos e diretórios versionados estão outros arquivos e diretórios que não são versionados e nem pretendem ser. Editores de texto deixam diretórios com arquivos de backup. Compiladores de software produzem arquivos intermediários—ou mesmo definitivos—que você normalmente não faria controle de versão. E os próprios usuários deixam vários outros arquivos e diretórios, sempre que acharem adequado, muitas vezes em cópias de trabalho com controle de versão.

É ridículo esperar que cópias de trabalho do Subversion sejam de algum modo impenetráveis a este tipo de resíduo e impureza. De fato, o Subversion os considera como um *recurso* que suas cópias de trabalho estão apenas com diretórios normais, como árvores não-versionadas. Mas estes arquivos e diretórios que não deveriam ser versionados podem causar algum incômodo aos usuários do Subversion. Por exemplo, pelo fato dos comandos **svn add** e **svn import** agirem recursivamente por padrão, e não saberem quais arquivos em uma dada árvore você deseja ou não versionar, é acidentalmente fácil adicionar coisas ao controle de versão que você não pretendia. E pelo fato do comando **svn status** reportar, por padrão, cada item de interesse em uma cópia de trabalho—incluindo arquivos e diretórios não versionados—sua saída pode ficar muito poluída, onde grande número destas coisas aparecem.

Portanto, o Subversion oferece dois meios para dizer quais arquivos você preferiria que ele simplesmente desconsiderasse. Um dos meios envolve o uso do sistema de configuração do ambiente de execução do Subversion (veja “Runtime Configuration Area”), e conseqüentemente aplica-se a todas operações do Subversion que fazem uso desta configuração do ambiente de execução, geralmente aquelas executadas em um computador específico, ou por um usuário específico de um computador. O outro meio faz uso do suporte de propriedade de diretório do Subversion, é mais fortemente vinculado à própria árvore versionada e, conseqüentemente, afeta todos aqueles que têm uma cópia de trabalho desta árvore. Os dois mecanismos usam filtros de arquivo.

O sistema de configuração *runtime* do Subversion oferece uma opção, `global-ignores`, cujo valor é uma coleção de filtros de arquivo delimitados por espaços em branco (também conhecida com *globs*). O cliente do Subversion verifica esses filtros em comparação com os nomes dos arquivos que são candidatos para adição ao controle de versão, bem como os arquivos não versionados os quais o comando **svn status** notifica. Se algum nome de arquivo coincidir com um dos filtros, basicamente, o Subversion atuará como se o arquivo não existisse. Isto é realmente útil para os tipos de arquivos que você raramente precisará controlar versão, tal como cópias de arquivos feitas por editores como os arquivos `*~` e `.*~` do *Emacs*.

Quando encontrada em um diretório versionado, a propriedade `svn:ignore` espera que contenha uma lista de filtros de arquivo delimitadas por quebras de linha que o Subversion deve usar para determinar objetos ignoráveis neste mesmo diretório. Estes filtros não anulam os encontrados na opção `global-ignores` da configuração *runtime*, porém, são apenas anexados a esta lista. E é importante notar mais uma vez que, ao contrário da opção `global-ignores`, os filtros encontrados na propriedade `svn:ignore` aplicam-se somente ao diretório no qual esta propriedade está definida, e em nenhum de seus subdiretórios. A propriedade `svn:ignore` é uma boa meneira para dizer ao Subversion ignorar arquivos que estão susceptíveis a estarem presentes em todas as cópias de trabalho de usuário deste diretório, assim como as saídas de compilador ou—para usar um exemplo mais apropriado para este livro—os arquivos HTML, PDF, ou PostScript produzidos como o resultado de uma conversão de alguns arquivos XML do fonte DocBook para um formato de saída mais legível.

O suporte do Subversion para filtros de arquivos ignoráveis extende somente até o processo de adicionar arquivos e diretórios não versionados ao controle de versão. Desde que um objeto está sob o controle do Subversion, os mecanismos de filtro de ignoração já não são mais aplicáveis a ele. Em outras palavras, não espere que o Subversion deixe de realizar a submissão de mudanças que você efetuou em arquivos versionados simplesmente porque estes nomes de arquivo coincidem com um filtro de ignoração—o Subversion *sempre* avisa quais objetos foram versionados.

Filtros de Rejeição para Usuários CVS

A propriedade `svn:ignore` do Subversion é muito similar em sintaxe e função ao arquivo `.cvsignore` do CVS. De fato, se você está migrando de uma cópia de trabalho CVS para Subversion, você pode migrar os filtros de rejeição, diretamente, pelo uso do arquivo `.cvsignore` como arquivo de entrada para o comando **svn propset**:

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

Existem, entretanto, algumas diferenças nos meios que CVS e Subversion manipulam filtros de rejeição. Os dois sistemas usam os filtros de rejeição em tempos um pouco diferentes, e existem legeiras discrepâncias na aplicação dos filtros de rejeição. Além disso, o Subversion não reconhece o uso do filtro `!` como uma redefinição que torna os filtros seguintes como não-ignorados.

A lista global de filtros de rejeição tende ser mais uma questão de gosto pessoal, e vinculada mais estreitamente a uma série de ferramentas específicas do usuário do que aos detalhes de qualquer cópia de trabalho particular necessita. Assim, o resto desta seção focará na propriedade `svn:ignore` e seus usos.

Digamos que você tenha a seguinte saída do **svn status**:

```
$ 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
```

Neste exemplo, você realizou algumas modificações no arquivo `button.c`, mas em sua cópia de trabalho você também possui alguns arquivos não-versionados: o mais recente programa `calculator` que você compilou a partir do seu código fonte, um arquivo fonte nomeado `data.c`, e uma série de arquivos de registro da saída de depuração. Agora, você sabe que seu sistema de construção sempre resulta no programa `calculator` como produto. [5] E você sabe que sua ferramenta de testes sempre deixa aqueles arquivos de registro de depuração alojando ao redor. Estes fatos são verdadeiros para todas cópias de trabalho deste projeto, não para apenas sua própria. E você também não está interessado em ver aquelas coisas toda vez que você executa **svn status**, e bastante seguro que ninguém mais está interessado em nenhuma delas. Sendo assim, você executa **svn propedit svn:ignore calc** para adicionar alguns filtros de rejeição para o diretório `calc`. Por exemplo, você pode adicionar os filtros abaixo como o novo valor da propriedade `svn:ignore`:

```
calculator
debug_log*
```

Depois de você adicionar esta propriedade, você terá agora uma modificação de propriedade local no diretório `calc`. Mas note que o restante da saída é diferente para o comando **svn status**:

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

Agora, todas aqueles resíduos não são apresentados nos resultados! Certamente, seu programa compidado `calculator` e todos aqueles arquivos de registro estão ainda em sua cópia de trabalho. O Subversion está simplesmente não lembrando você que eles estão presentes e não-versionados. E agora com todos os arquivos desinteressantes removidos dos resultados, você visualizará somente os itens mais interessantes—assim como o arquivo de código fonte `data.c` que você provavelmente esqueceu de adicionar ao controle de versão.

Evidentemente, este relatório menos prolixo da situação de sua cópia de trabalho não é a única disponível. Se você realmente quiser ver os arquivos ignorados como parte do relatório de situação, você pode passar a opção `--no-ignore` para o Subversion:

```
$ svn status --no-ignore
```

[5] Não é isso o resultado completo de um sistema de construção?

```
 M     calc
 M     calc/button.c
I      calc/calculator
?      calc/data.c
I      calc/debug_log
I      calc/debug_log.1
I      calc/debug_log.2.gz
I      calc/debug_log.3.gz
```

Como mencionado anteriormente, a lista de filtros de arquivos a ignorar também é usada pelos comandos **svn add** e **svn import**. Estas duas operações implicam solicitar ao Subversion iniciar o gerenciamento de algum conjunto de arquivos e diretórios. Ao invés de forçar o usuário a escolher quais arquivos em uma árvore ele deseja iniciar o versionamento, o Subversion usa os filtros de rejeição—tanto a lista global quanto a por diretório (`svn-ignore`)—para determinar quais arquivos não devem ser varridos para o sistema de controle de versão como parte de uma operação recursiva de adição ou importação. E da mesma forma, você pode usar a opção `--no-ignore` para indicar ao Subversion desconsiderar suas listas de rejeição e operar em todos os arquivos e diretórios presentes.

Substituição de Palavra-Chave

O Subversion possui a capacidade de substituir *palavras-chave*—pedaços de informação úteis e dinâmicos sobre um arquivo versionado—dentro do conteúdo do próprio arquivo. As palavras-chave geralmente fornece informação sobre a última modificação realizada no arquivo. Pelo fato desta informação modificar toda vez que o arquivo é modificado, e mais importante, apenas *depois* que o arquivo é modificado, isto é um aborrecimento para qualquer processo a não ser que o sistema de controle de versão mantenha os dados completamente atualizados. Se deixada para os autores humanos, a informação se tornaria inevitavelmente obsoleta.

Por exemplo, digamos que você tem um documento no qual gostaria de mostrar a última data em que ele foi modificado. Você poderia obrigar que cada autor deste documento que, pouco antes de submeter suas alterações, também ajustasse a parte do documento que descreve quando ele fez a última alteração. Porém, mais cedo ou mais tarde, alguém esqueceria de fazer isto. Em vez disso, basta solicitar ao Subversion que efetue a substituição da palavra-chave `LastChangedDate` pelo valor adequado. Você controla onde a palavra-chave é inserida em seu documento colocando uma *âncora de palavra-chave* no local desejado dentro do arquivo. Esta âncora é apenas uma sequência de texto formatada como `$`*`NomeDaPalavraChave`*`$`.

Todas as palavras-chave são sensíveis a minúsculas e maiúsculas onde aparecem como âncoras em arquivos: você deve usar a capitalização correta para que a palavra-chave seja expandida. Você deve considerar que o valor da propriedade `svn:keywords` esteja ciente da capitalização também—certos nomes de palavras-chave serão reconhecidos, independentemente do caso, mas este comportamento está desaprovado.

O Subversion define a lista de palavras-chave disponíveis para substituição. Esta lista contém as seguintes cinco palavras-chave, algumas das quais possuem apelidos que você pode também utilizar:

`Date`
: Esta palavra-chave descreve a última vez conhecida em que o arquivo foi modificado no repositório, e está na forma `$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $`. Ela também pode ser especificada como `LastChangedDate`.

`Revision`
: Esta palavra-chave descreve a última revisão conhecida em que este arquivo foi modificado no repositório, e é apresentada na forma `$Revision: 144 $`. Ela também pode ser especificada como `LastChangedRevision` ou `Rev`.

`Author`
: Esta palavra-chave descreve o último usuário conhecido que modificou este arquivo no repositório, e é apresentada na forma `$Author: harry $`. Ela também pode ser especificada como `LastChangedBy`.

`HeadURL`
: Esta palavra-chave descreve a URL completa para a versão mais recente do arquivo no repositório, e é apresentada na forma `$HeadURL: http://svn.collab.net/repos/trunk/README $`. Ela também pode ser abreviada como `URL`.

`Id`
: Esta palavra-chave é uma combinação comprimida das outras palavras-chave. Sua substituição apresenta-se como `$Id: calc.c 148 2006-07-28 21:30:43Z sally $`, e é interpretada no sentido de que o arquivo `calc.c` foi modificado pela última vez na revisão 148 na noite de 28 de julho de 2006 pelo usuário `sally`.

Muitas das descrições anteriores usam a frase “último valor conhecido” ou algo parecido. Tenha em mente que a expansão da palavra-chave é uma operação no lado do cliente, e seu cliente somente “conhece” sobre mudanças que tenham ocoridas no repositório quando você atualiza sua cópia de trabalho para incluir essas mudanças. Se você nunca atualizar sua cópia de trabalho, suas palavras-chave nunca expandirão para valores diferentes, mesmo que esses arquivos versionados estejam sendo modificados regularmente no repositório.

Simplesmente adicionar texto da âncora de uma palavra-chave em seu arquivo faz nada de especial. O Subversion nunca tentará executar substituições textuais no conteúdo de seu arquivo a não ser que seja explicitamente solicitado. Afinal, você pode estar escrevendo um documento [6] sobre como usar palavras-chave, e você não quer que o Subversion substitua seus belos exemplos de âncoras de palavra-chave, permanecendo não-substituídas!

Para dizer ao Subversion se substitui ou não as palavras-chave em um arquivo particular, voltamos novamente aos subcomandos relacionados a propriedades. A propriedade `svn:keywords`, quando definida em um arquivo versionado, controla quais palavras-chave serão substituídas naquele arquivo. O valor é uma lista delimitada por espaços dos nomes ou apelidos de palavra-chave encontradas na tabela anterior.

Por exemplo, digamos que você tenha um arquivo versionado nomeado `weather.txt` que possui esta aparência:

```
Aqui está o mais recente relatório das linhas iniciais.
$LastChangedDate$
$Rev$
Acúmulos de núvens estão aparecendo com mais freqüência quando o verão se aproxima.
```

Sem definir a propriedade `svn:keywords` neste arquivo, o Subversion fará nada especial. Agora, vamos permitir a substituição da palavra-chave `LastChangedDate`.

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

Agora você fez uma modificação local da propriedade no arquivo `weather.txt`. Você verá nenhuma mudança no conteúdo do arquivo (ao menos que você tenha feito alguma definição na propriedade anteriormente). Note que o arquivo continha uma âncora de palavra-chave para a palavra-chave `Rev`, no

[6] … ou até mesmo uma seção de um livro …

entanto, não incluímos esta palavra-chave no valor da propriedade que definimos. Felizmente, o Subversion ignorará pedidos para substituir palavras-chave que não estão presentes no arquivo, e não substituirá palavras-chave que não estão presentes no valor da propriedade `svn:keywords`.

Imediatamente depois de você submeter esta mudança de propriedade, o Subversion atualizará seu arquivo de trabalho com o novo texto substituto. Em vez de ver a sua âncora da palavra-chave `$LastChangedDate$`, você verá como resultado seu valor substituído. Este resultado também contém o nome da palavra-chave, que continua sendo limitada pelos caracteres de sinal de moeda ($). E como previámos, a palavra-chave `Rev` não foi substituída porque não solicitamos que isto fosse realizado.

Note também, que definimos a propriedade `svn:keywords` para “Date Author” e, no entanto, a âncora da palavra-chave usou o apelido `$LastChangedDate$` e ainda sim expandiu corretamente.

```
Aqui está o mais recente relatório das linhas iniciais.
$LastChangedDate: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $
$Rev$
Acúmulos de núvens estão aparecendo com mais freqüência quando o verão se aproxima.
```

Se agora alguém submeter uma mudança para `weather.txt`, sua cópia deste arquivo continuará a mostrar o mesmo valor para a palavra-chave substituída como antes—até que você atualize sua cópia de trabalho. Neste momento as palavras-chave em seu arquivo `weather.txt` serão re-substituídas com a informação que reflete a mais recente submissão conhecida para este arquivo.

Onde está $GlobalRev$?

Novos usuários são freqüentemente confundidos pela forma que a palavra-chave `$Rev$` trabalha. Como o repositório possui um número de revisão único, globalmente incrementado, muitas pessoas assumem que este número está refletido no valor da palavra-chave `$Rev$`. Porém, `$Rev$` reflete a última revisão na qual o arquivo foi *modificado*, não a última revisão para qual ele foi atualizado. Compreender isto esclarece a confusão, mas a frustração muitas vezes permanece—sem o suporte de uma palavra-chave do Subversion para isso, como podemos obter automaticamente o número de revisão global em seus arquivos?

Para fazer isto, você precisa de processamento externo. O Subversion vem com uma ferramenta chamada **svnversion** que foi projetada apenas para este propósito. O comando **svnversion** rastreia sua cópia de trabalho e produz como saída as revisões que encontra. Você pode usar este programa, mais algumas outras ferramentas, para embutir esta informação sobre as revisões globais em seus arquivos. Para mais informações sobre **svnversion**, veja “**svnversion**”.

O Subversion 1.2 introduziu uma nova variante da sintaxe de palavra-chave que trouxe funcionalidade adicional e útil—embora talvez atípica. Agora você pode dizer ao Subversion para manter um tamanho fixo (em termos do número de bytes consumidos) para a palavra-chave substituída. Pelo uso de um duplo dois pontos (`::`) após o nome da palavra-chave, seguido por um número de caracteres de espaço, você define esta largura fixa. Quando o Subversion for substituir sua palavra-chave para a palavra-chave e seu valor, ele substituirá essencialmente apenas aqueles caracteres de espaço, deixando a largura total do campo da palavra-chave inalterada. Se o valor substituído for menor que a largura definida para o campo, haverá caracteres de enchimento extras (espaços) no final do campo substituído; se for mais longo, será truncado com um caractere de contenção especial (#) logo antes do sinal de moeda delimitador de fim.

Por exemplo, digamos que você possui um documento em que temos alguma seção com dados tabulares refletindo as palavras-chave do Subversion sobre o documento. Usando a sintaxe de substituição de palavra-chave original do Subversion, seu arquivo pode parecer com alguma coisa como:

```
$Rev$:     Revisão da última submissão
```

```
$Author$:  Autor da última submissão
$Date$:    Data da última submissão
```

Neste momento, vemos tudo de forma agradável e tabular. Mas quando você em seguida submete este arquivo (com a substituição de palavra-chave habilitada, certamente), vemos:

```
$Rev: 12 $:     Revisão da última submissão
$Author: harry $:  Autor da última submissão
$Date: 2006-03-15 02:33:03 -0500 (Wed, 15 Mar 2006) $:    Data da última submissão
```

O resultado não é tão elegante. E você pode ser tentado a então ajustar o arquivo depois da substituição para que pareça tabular novamente. Mas isto apenas funciona quando os valores da palavra-chave são da mesma largura. Se a última revisão submetida aumentar em uma casa decimal (ou seja, de 99 para 100), ou se uma outra pessoa com um nome de usuário maior submete o arquivo, teremos tudo bagunçado novamente. No entanto, se você está usando o Subversion 1.2 ou superior, você pode usar a nova sintaxe para palavra-chave com tamanho fixo, definir algumas larguras de campo que sejam razoáveis, e agora seu arquivo pode ter esta parência:

```
$Rev::               $:  Revisão da última submissão
$Author::            $:  Autor da última submissão
$Date::              $:  Data da última submissão
```

Você submete esta mudança ao seu arquivo. Desta vez, o Subversion nota a nova sintaxe para palavra-chave com tamanho fixo, e mantém a largura dos campos como definida pelo espaçamento que você colocou entre o duplo dois pontos e o sinal de moeda final. Depois da substituição, a largura dos campos está completamente inalterada—os curtos valores de `Rev` e `Author` são preenchidos com espaços, e o longo campo `Date` é truncado com um caractere de contenção:

```
$Rev:: 13            $:  Revisão da última submissão
$Author:: harry      $:  Autor da última submissão
$Date:: 2006-03-15 0#$:  Data da última submissão
```

O uso de palavras-chave de comprimento fixo é especialmente útil quando executamos substituições em formatos de arquivo complexos que por si mesmo usam campos de comprimento fixo nos dados, ou que o tamanho armazenado de um determinado campo de dados é predominantemente difícil de modificar fora da aplicação original do formato (assim como para documentos do Microsoft Office).

Esteja ciente que pelo fato da largura do campo de uma palavra-chave é medida em bytes, o potencial de corrupção de valores de multi-byte existe. Por exemplo, um nome de usuário que contém alguns caracteres multi-byte em UTF-8 pode sofrer truncamento no meio da seqüência de bytes que compõem um desses caracteres. O resultado será um mero truncamento quando visualizado à nível de byte, mas provavelmente aparecerá como uma cadeia com um caractere adicional incorreto ou ilegível quando exibido como texto em UTF-8. É concebível que certas aplicações, quando solicitadas a carregar o arquivo, notariam o texto em UTF-8 quebrado e ainda considerem todo o arquivo como corrompido, recusando-se a operar sobre o arquivo de um modo geral. Portanto, ao limitar palavras-chave para um tamanho fixo, escollha um tamanho que permita este tipo de expansão ciente dos bytes.

Travamento

O modelo de controle de versão copiar-modificar-fundir do Subversion ganha e perde sua utilidade em seus algoritmos de fusão de dados, especificamente sobre quão bem esses algoritmos executam ao tentar resolver conflitos causados por múltiplos usuários modificando o mesmo arquivo simultaneamente.

O próprio Subversion oferece somente um algoritmo, um algoritmo de diferenciação de três meios, que é inteligente o suficiente para manipular dados até uma granularidade de uma única linha de texto. O Subversion também permite que você complemente o processamento de fusão de conteúdo com utilitários de diferenciação externos (como descrito em “External diff3”), alguns dos quais podem fazer um trabalho ainda melhor, talvez oferecendo granularidade em nível de palavra ou em nível de caractere de texto. Mas o comum entre esses algoritmos é que eles geralmente trabalham apenas sobre arquivos de texto. O cenário começa a parecer consideravelmente rígido quando você começa a discursar sobre fusões de conteúdo em formatos de arquivo não-textual. E quando você não pode encontrar uma ferramenta que possa manipular este tipo de fusão, você começa a verificar os problemas com o modelo copiar-modificar-fundir.

Vejamos um exemplo da vida real onde este modelo não trabalha adequadamente. Harry e Sally são ambos desenhistas gráficos trabalhando no mesmo projeto, que faz parte do marketing paralelo para um automóvel mecânico. O núcleo da concepção de um determinado cartaz é uma imagem de um carro que necessita de alguns reparos, armazenada em um arquivo usando o formato de imagen PNG. O leiaute do cartaz está quase pronto, e tanto Harry quanto Sally estão satisfeitos com a foto que eles escolheram para o carro danificado—um Ford Mustang 1967 azul bebê com uma parte infelizmente amassada no para-lama dianteiro esquerdo.

Agora, como é comum em trabalhos de desenho gráfico, existe uma mudança de planos que faz da cor do carro uma preocupação. Então, Sally atualiza sua cópia de trabalho para a revisão HEAD, inicializa seu software de edição de fotos, e realiza alguns ajustes na imagem de modo que o carro está agora vermelho cereja. Enquanto isso, Harry, sentindo-se particularmente inspirado neste dia, decide que a imagem teria mais impacto se o carro também apresentasse ter sofrido um maior impacto. Ele, também, atualiza para a revisão HEAD, e então, desenha algumas rachaduras no pára-brisa do veículo. Ele conduz de forma a concluir seu trabalho antes de Sally terminar o dela, e depois, admirando o fruto de seu inegável talento, submete a imagem modificada. Pouco tempo depois, Sally finaliza sua nova versão do carro, e tenta submeter suas mudanças. Porém, como esperado, o Subversion falha na submissão, informando Sally que agora sua versão da imagem está desatualizada.

Vejamos onde a dificuldade ocorre. Se Harry e Sally estivessem realizando mudanças em um arquivo de texto, Sally iria simplesmente atualizar sua cópia de trabalho, recebendo as mudanças que Harry realizou. No pior caso possível, eles teriam modificado a mesma região do arquivo, e Sally teria que realizar uma adequada resolução do conflito. Mas estes não são arquivos de texto—são imagens binárias. E enquanto seja uma simples questão de descrever o que seria esperado como resultado desta fusão de conteúdos, existe uma pequena chance preciosa de que qualquer software existente seja inteligente o suficiente para examinar a imagem que cada um dos artistas gráficos se basearam para realizarem seu trabalho, as mudanças que Harry fez e as mudanças que Sally faz, e produzir uma imagem de um Mustang vermelho degradado com um pára-brisa trincado!

Obviamente, as coias teriam sido mais simples se Harry e Sally tivessem seqüenciado suas modificações na imagem—se, digamos, Harry aguardasse para desenhar seus trincados no pára-brisa no novo carro vermelho de Sally, ou se Sally trocasse a cor de um carro cujo pára-brisa já estivesse trincado. Como é discutido em “A Solução Copy-Modify-Merge”, a maioria destes tipos de problemas desaparecerão totalmente quando existir uma perfeita comunicação entre Harry e Sally. [7] Porém, como um sistema de controle de versão é de fato uma forma de comunicação, ter um software que facilita a a serialização de esforços não passíveis de paralelamento não é ruim. É neste cenário que a implementação do Subversion do modelo travar-modificar-destravar ganha maior destaque. Este é o momento que falamos sobre a característica de *travamento* do Subversion, a qual é similar aos mecanismos de “obter cópias reservadas” de outros sistemas de controle de versão.

A funcionalidade de travamento do Subversion serve dois propósitos principais:

- *Serializar o acesso a um objeto versionado*. Ao permitir que um usuário requeira programaticamente o direito exclusivo de modificar um arquivo no repositório, este usuário pode estar razoavelmente seguro

[7] A comunicação não teria sido algo tão ruim para os homônimos de Harry e Sally em Hollywood, ainda que seja para nosso caso.

de que os esforços investidos nas mudanças não-mescláveis não serão desperdiçados—a submissão de suas alterações será bem sucedida.

- *Ajudar a comunicação*. Ao alertar outros usuários que a serialização está em vigor para um determinado objeto versionado, estes outros usuários podem razoavelmente esperar que o objeto está prestes de ser modificado por outra pessoa, e eles, também, podem evitar o desperdício de seu tempo e energia em mudanças não-mescláveis que não serão submetidas adequadamente e ocasionando possível perda de dados.

Quando nos referimos à funcionalidade de travamento do Subversion, estaremos também falando sobre uma coleção de comportamentos bastante diversificada que incluem a capacidade de travar um arquivo [8] versionado (requerendo o direito exclusivo de modificar o arquivo), de destravar este arquivo (cedendo este direito exclusivo de modificar), de ver relatórios sobre quais arquivos estão travados e por quem, de marcar arquivos para os quais o travamento antes da edição é fortemente aconselhado, e assim por diante. Nesta seção, cobriremos todas destas facetas da ampla funcionalidade de travamento.

Os três significados de “trava”

Nesta seção, e em quase todas neste livro, as palavras “trava” e “travamento” representam um mecanismo para exclusão mútua entre os usuários para evitar submissões conflitantes. Infelizmente, existem dois outros tipos de “trava” com os quais o Subversion, e portanto este livro, algumas vezes precisam se preocupar.

O primeiro tipo são as *travas da cópia de trabalho*, usadas internamente pelo Subversion para prevenir conflitos entre múltiplos clientes Subversion operando na mesma cópia de trabalho. Este é o tipo de trava indicada por um L na terceira coluna da saída produzida por **svn status**, e removida pelo comando **svn cleanup**, como especificado em “Sometimes You Just Need to Clean Up”.

Em segundo lugar, existem as *travas do banco de dados*, usadas internamente pelo sistema Berkeley DB para prevenir conflitos entre múltiplos programas tentando acessar o banco de dados. Este é o tipo de trava cuja indesejável persistência após um erro pode fazer com que um repositório seja “corrompido”, como descrito em “Berkeley DB Recovery”.

Você pode geralmente esquecer destes outros tipos de travas até que algo de errado ocorra e requeira seus cuidados sobre eles. Neste livro, “trava” possui o significado do primeiro tipo ao menos que o contrário esteja claro pelo contexto ou explicitamente indicado.

Criando travas

No repositório Subversion, uma *trava* é um pedaço de metadados que concede acesso exclusivo para um usuário modificar um arquivo. Este usuário é chamado de *proprietário da trava*. Cada trava também tem um identificador único, tipicamente uma longa cadeia de caracteres, conhecida como o *sinal de trava*. O repositoório gerencia as travas, basicamente manipulando sua criação, aplicação e remoção. Se qualquer transação de submissão tenta modificar ou excluir um arquivo travado (ou excluir um dos diretórios pais do arquivo), o repositório exigirá dois pedaços de informação—que o cliente executante da submissão esteja autenticado como o proprietário da trava, e que o sinal de trava tenha sido fornecido como parte do processo de submissão como um tipo de prova que o cliente conhece qual trava ele está usando.

Para demonstrar a criação de uma trava, vamos voltar ao nosso exemplo de múltiplos desenhistas gráficos trabalhando sobre os mesmos arquivos binários de imagem. Harry decidiu modificar uma imagem JPEG.

[8] Atualmente o Subversion does não permite travas em diretórios.

Para prevenir que outras pessoas submetessem mudanças no arquivo enquanto ele está modificando-o (bem como alertando-os que ele está prestes a mudá-lo), ele trava o arquivo no repositório usando o comando **svn lock**.

```
$ svn lock banana.jpg -m "Editando arquivo para a liberação de amanhã."
'banana.jpg' locked by user 'harry'.
$
```

Existe uma série de novas coisas demonstradas no exemplo anterior. Primeiro, note que Harry passou a opção `--message (-m)` para o comando **svn lock**. Similar ao **svn commit**, o comando **svn lock** pode receber comentários (seja via `--message (-m)` ou `--file (-F)`) para descrever a razão do travamento do arquivo. Ao contrário do **svn commit**, entretanto, o **svn lock** não exigirá uma mensagem executando seu editor de texto preferido. Os comentários de trava são opcionais, mas ainda recomendados para ajudar na comunicação.

Em segundo lugar, a trava foi bem sucedida. Isto significa que o arquivo não estava travado, e que Harry tinha a mais recente versão do arquivo. Se o arquivo da cópia de trabalho de Harry estivesse desatualizado, o repositório teria rejeitado a requisição, forçando Harry a executar **svn update** e tentar o comando de travamento novamente. O comando de travamento também teria falhado se o arquivo já estivesse travado por outro usuário.

Como você pode ver, o comando **svn lock** imprime a confirmação do sucesso no travamento. A partir deste ponto, o fato de que o arquivo está travado torna-se aparente na saída dos relatórios dos subcomandos **svn status** e **svn info**.

```
$ svn status
     K banana.jpg

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

$
```

O comando **svn info**, o qual não consulta o repositório quando executa sobre caminhos de uma cópia de trabalho, pode mostrar o sinal de trava e revela um importante fato sobre o sinal de trava—que eles são

colocados em cache na cópia de trabalho. A presença do sinal de trava é crítica. Ele dá à cópia de trabalho a autorização para fazer uso da trava mais tarde. Além disso, o comando **svn status** mostra um K próximo ao arquivo (abreviação para *locKed*), indicando que o sinal de trava está presente.

Regarding lock tokens

A lock token isn't an authentication token, so much as an *authorization* token. The token isn't a protected secret. In fact, a lock's unique token is discoverable by anyone who runs **svn info URL**. A lock token is special only when it lives inside a working copy. It's proof that the lock was created in that particular working copy, and not somewhere else by some other client. Merely authenticating as the lock owner isn't enough to prevent accidents.

For example, suppose you lock a file using a computer at your office, but leave work for the day before you finish your changes to that file. It should not be possible to accidentally commit changes to that same file from your home computer later that evening simply because you've authenticated as the lock's owner. In other words, the lock token prevents one piece of Subversion-related software from undermining the work of another. (In our example, if you really need to change the file from an alternate working copy, you would need to *break* the lock and re-lock the file.)

Now that Harry has locked `banana.jpg`, Sally is unable to change or delete that file:

```
$ svn delete banana.jpg
D         banana.jpg
$ svn commit -m "Delete useless file."
Deleting       banana.jpg
svn: Commit failed (details follow):
svn: DELETE of
'/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-df4224ddc35d/banana.jpg':
423 Locked (http://svn.example.com)
$
```

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

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

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

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

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

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

Discovering locks

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

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

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

```
$ svn info http://svn.example.com/repos/project/raisin.jpg
Path: raisin.jpg
Name: raisin.jpg
URL: http://svn.example.com/repos/project/raisin.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 105
Node Kind: file
Last Changed Author: sally
Last Changed Rev: 32
Last Changed Date: 2006-01-25 12:43:04 -0600 (Sun, 25 Jan 2006)
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Lock Comment (1 line):
Need to make a quick tweak to this image.
$
```

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

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

Breaking and stealing locks

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

From the administrator's chair, it's simple to break locks. The **svnlook** and **svnadmin** programs have the ability to display and remove locks directly from the repository. (For more information about these tools, see “An Administrator's Toolkit”.)

```
$ svnadmin lslocks /usr/local/svn/repos
Path: /project2/images/banana.jpg
UUID Token: opaquelocktoken:c32b4d88-e8fb-2310-abb3-153ff1236923
Owner: frank
Created: 2006-06-15 13:29:18 -0500 (Thu, 15 Jun 2006)
Expires:
Comment (1 line):
Still improving the yellow color.

Path: /project/raisin.jpg
UUID Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Owner: harry
Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Expires:
Comment (1 line):
Need to make a quick tweak to this image.

$ svnadmin rmlocks /usr/local/svn/repos /project/raisin.jpg
Removed lock on '/project/raisin.jpg'.
$
```

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

```
$ svn status -u
M              23   bar.c
M    O         32   raisin.jpg
       *       72   foo.h
Status against revision:     105
$ svn unlock raisin.jpg
svn: 'raisin.jpg' is not locked in this working copy
$ svn info raisin.jpg | grep URL
URL: http://svn.example.com/repos/project/raisin.jpg
$ svn unlock http://svn.example.com/repos/project/raisin.jpg
svn: Unlock request failed: 403 Forbidden (http://svn.example.com)
$ svn unlock --force http://svn.example.com/repos/project/raisin.jpg
'raisin.jpg' unlocked.
$
```

Now, Sally's initial attempt to unlock failed because she ran **svn unlock** directly on her working copy of the file, and no lock token was present. To remove the lock directly from the repository, she needs to pass a URL to **svn unlock**. Her first attempt to unlock the URL fails, because she can't authenticate as the lock owner (nor does she have the lock token). But when she passes `--force`, the authentication and authorization requirements are ignored, and the remote lock is broken.

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

```
$ svn lock raisin.jpg
svn: Lock request failed: 423 Locked (http://svn.example.com)
$ svn lock --force raisin.jpg
'raisin.jpg' locked by user 'sally'.
$
```

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

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

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

Locking Policies

Different systems have different notions of how strict a lock should be. Some folks argue that locks must be strictly enforced at all costs, releasable only by the original creator or administrator. They argue that if anyone can break a lock, then chaos runs rampant and the whole point of locking is defeated. The other side argues that locks are first and foremost a communication tool. If users are constantly breaking each others' locks, then it represents a cultural failure within the team and the problem falls outside the scope of software enforcement.

Subversion defaults to the “softer” approach, but still allows administrators to create stricter enforcement policies through the use of hook scripts. In particular, the `pre-lock` and `pre-unlock` hooks allow administrators to decide when lock creation and lock releases are allowed to happen. Depending on whether or not a lock already exists, these two hooks can decide whether or not to allow a certain user to break or steal a lock. The `post-lock` and `post-unlock` hooks are also available, and can be used to send email after locking actions. To learn more about repository hooks, see “Implementing Repository Hooks”.

Lock Communication

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

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

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

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

```
$ /usr/local/bin/gimp raisin.jpg
gimp: error: file is read-only!
$ ls -l raisin.jpg
-r--r--r--   1 sally   sally   215589 Jun  8 19:23 raisin.jpg
$ svn lock raisin.jpg
svn: Lock request failed: 423 Locked (http://svn.example.com)
$ svn info http://svn.example.com/repos/project/raisin.jpg | grep Lock
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-06-08 07:29:18 -0500 (Thu, 08 June 2006)
Lock Comment (1 line):
Making some tweaks.  Locking for the next two hours.
$
```

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

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

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

[9] Except, perhaps, a classic Vulcan mind-meld.

Definições Externas

Às vezes, é útil construir uma cópia de trabalho que é composta por diferentes *checkouts*. Por exemplo, talvez você queira que diferentes subdiretórios venham de diferentes locais em um repositório, ou até mesmo de diferentes repositórios. Você poderia configurar tal cenário manualmente— usando **svn checkout** para criar o tipo de estrutura aninhada de cópia de trabalho que você está tentando construir. Mas, se essa estrutura é importante para todos os que usam seu repositório, todos os outros usuários precisarão realizar as mesmas operações de *checkout* que você fez.

Felizmente, o Subversion provê suporte para *definições externas*. Uma definição externa é um mapeamento de um diretório local para a URL—e, idealmente, uma determinada revisão—de um diretório sob controle de versão. No Subversion, você declara definições externas em conjunto usando a propriedade `svn:externals`. Você pode criar ou modificar essa propriedade usando **svn propset** ou **svn propedit** (veja “Manipulating Properties”). Essa propriedade pode ser configurada em qualquer diretório sob controle de versão, e seu valor é uma tabela multi-linha de subdiretórios (relativos ao diretório sob controle de versão no qual a propriedade está configurada), opções de revisão, e URLs absolutas (totalmente qualificadas) de repositórios Subversion.

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

A conveniência da propriedade `svn:externals` é que, uma vez configurada em um diretório sob controle de versão, qualquer pessoa que obtém uma cópia de trabalho desse diretório também é beneficiada pelas definições externas. Em outras palavras, uma vez que alguém investiu tempo e esforço para definir essa cópia de trabalho feita de *checkouts* aninhados, ninguém mais precisa se incomodar—o Subversion, através do *checkout* da cópia de trabalho original, também obterá as cópias de trabalho externas.

Os subdiretórios alvos relativos das definições externas *não podem* existir no seu sistema de arquivos nem no de outros usuários—o Subversion irá criá-los quando obter a cópia de trabalho externa.

Note o exemplo anterior de definições externas. Quando alguém obtém uma cópia de trabalho do diretório `calc`, o Subversion também obtém os itens encontrados nas suas definições externas.

```
$ 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
…
```

If you need to change the externals definition, you can do so using the regular property modification subcommands. When you commit a change to the `svn:externals` property, Subversion will synchronize the checked-out items against the changed externals definition when you next run **svn update**. The same thing will happen when others update their working copies and receive your changes to the externals definition.

Because the `svn:externals` property has a multiline value, we strongly recommend that you use **svn propedit** instead of **svn propset**.

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

The **svn status** command also recognizes externals definitions, displaying a status code of X for the disjoint subdirectories into which externals are checked out, and then recursing into those subdirectories to display the status of the external items themselves.

The support that exists for externals definitions in Subversion is less than ideal, though. First, an externals definition can only point to directories, not files. Second, the externals definition cannot point to relative paths (paths like `../../skins/myskin`). Third, the working copies created via the externals definition support are still disconnected from the primary working copy (on whose versioned directories the `svn:externals` property was actually set). And Subversion still only truly operates on non-disjoint working copies. So, for example, if you want to commit changes that you've made in one or more of those external working copies, you must run **svn commit** explicitly on those working copies—committing on the primary working copy will not recurse into any external ones.

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

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

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

$
```

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

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

Committed revision 12.
$ svn update

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

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

Finally, there might be times when you would prefer that **svn** subcommands would not recognize, or otherwise operate upon, the external working copies. In those instances, you can pass the `--ignore-externals` option to the subcommand.

Peg and Operative Revisions

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

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

The simplest example of this occurs when a directory or file is deleted from version control, and then a new directory or file is created with the same name and added to version control. Clearly the thing you deleted

and the thing you later added aren't the same thing. They merely happen to have had the same path, `/trunk/object` for example. What, then, does it mean to ask Subversion about the history of `/trunk/object`? Are you asking about the thing currently at that location, or the old thing you deleted from that location? Are you asking about the operations that have happened to *all* the objects that have ever lived at that path? Clearly, Subversion needs a hint about what you really want.

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

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

In scenarios like these, attempting to instruct Subversion to work with these re-used paths can be a little like instructing a motorist in Chicago's West Suburbs to drive east down Roosevelt Road and turn left onto Main Street. In a mere twenty minutes, you can cross “Main Street” in Wheaton, Glen Ellyn, and Lombard. And no, they aren't the same street. Our motorist—and our Subversion—need a little more detail in order to do the right thing.

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

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

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

[11] 606 N. Main Street, Wheaton, Illinois, is the home of the Wheaton History Center. Get it—“History Center”? It seemed appropriate….

The peg revision algorithm

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

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

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

- Locate *item* in the revision identified by *PEG-REV*. There can be only one such object.
- Trace the object's history backwards (through any possible renames) to its ancestor in the revision *OPERATIVE-REV*.
- Perform the requested action on that ancestor, wherever it is located, or whatever its name might be or have been at that time.

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

- Locate *item* in the revision identified by *OPERATIVE-REV*. There can be only one such object.
- Trace the object's history backwards (through any possible renames) to its ancestor in the revision *PEG-REV*.
- Verify that the object's location (path-wise) in *PEG-REV* is the same as it is in *OPERATIVE-REV*. If that's the case, then at least the two locations are known to be directly related, so perform the requested action on the location in *OPERATIVE-REV*. Otherwise, relatedness was not established, so error out with a loud complaint that no viable location was found. (Someday, we expect that Subversion will be able to handle this usage scenario with more flexibility and grace.)

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

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

Now, years later, we wonder what the `IDEA` file looked like back in revision 1. But Subversion needs to know if we are asking about how the *current* file looked back in revision 1, or if we are asking for the contents of whatever file lived at `concepts/IDEA` in revision 1. Certainly those questions have different answers, and because of peg revisions, you can ask either of them. To find out how the current `IDEA` file looked in that old revision, you run:

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

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

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

And when executed, it has the expected results.

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

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

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

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

As you can see, the output from our operation appears to be correct. The text even mentions frabbing naggily worts, so this is almost certainly the file which describes the software now called Frabnaggilywort. In fact, we can verify this using the combination of an explicit peg revision and explicit operative revision. We know that in HEAD, the Frabnaggilywort project is located in the frabnaggilywort directory. So we specify that we want to see how the line of history identified in HEAD as the path frabnaggilywort/IDEA looked in revision 1.

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

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

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

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

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

Network Model

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

Requests and Responses

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

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

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

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

Client Credentials Caching

Many servers are configured to require authentication on every request. This would be a big annoyance to users, if they were forced to type their passwords over and over again. Fortunately, the Subversion client has a remedy for this—a built-in system for caching authentication credentials on disk. By default, whenever the command-line client successfully responds to a server's authentication challenge, it saves the credentials in the user's private runtime configuration area (`~/.subversion/auth/` on Unix-like systems or `%APPDATA%/Subversion/auth/` on Windows; see “Runtime Configuration Area” for more details about the runtime configuration system). Successful credentials are cached on disk, keyed on a combination of the server's hostname, port, and authentication realm.

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

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

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

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

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

- For other Unix-like operating systems, no standard “keychain” services exist. However, the `auth/` caching area is still permission-protected so that only the user (owner) can read data from it, not the world at large. The operating system's own file permissions protect the passwords.

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

To disable caching for a single command, pass the `--no-auth-cache` option:

```
$ svn commit -F log_msg.txt --no-auth-cache
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
Password for 'joe':

Adding         newfile
Transmitting file data .
Committed revision 2324.

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

```
$ svn delete newfile
$ svn commit -F new_msg.txt
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
…
```

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

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

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

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

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

K 8
username
V 3
joe
K 8
password
V 4
blah
K 15
svn:realmstring
V 45
<https://svn.domain.com:443> Joe's repository
END
```

Once you have located the proper cache file, just delete it.

One last word about **svn**'s authentication behavior, specifically regarding the `--username` and `--password` options. Many client subcommands accept these options, but it is important to understand using these options does *not* automatically send credentials to the server. As discussed earlier, the server “pulls” credentials from the client when it deems necessary; the client cannot “push” them at will. If a username and/or password are passed as options, they will only be presented to the server if the server requests them. [12] These options are typically used to authenticate as a different user than Subversion would have

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

chosen by default (such as your system login name), or when trying to avoid interactive prompting (such as when calling **svn** from a script).

Here is a final summary that describes how a Subversion client behaves when it receives an authentication challenge.

1. First, the client checks whether the user specified any credentials as command-line options (`--username` and/or `--password`). If not, or if these options fail to authenticate successfully, then

2. the client looks up the server's hostname, port, and realm in the runtime `auth/` area, to see if the user already has the appropriate credentials cached. If not, or if the cached credentials fail to authenticate, then

3. finally, the client resorts to prompting the user (unless instructed not to do so via the `--non-interactive` option or its client-specific equivalents).

If the client successfully authenticates by any of the methods listed above, it will attempt to cache the credentials on disk (unless the user has disabled this behavior, as mentioned earlier).

Capítulo 4. Fundir e Ramificar

> “ (É sobre o 'Tronco' que trabalha um cavalheiro.)”
>
> —Confucio

Criar Ramos, Rótulo, e Fundir são conceitor comuns a quase todos os sistemas de controle de Versão. Caso você não esteja familiarizado com estes conceitos, nos oferecemos uma boa introdução a estes nesse capitulo. Se você ja conhece estes conceitos, então você vai achar interessante conhecer a maneira como o Subversion implementa esses conceitos.

Criar Ramos é um item fundamental para Controle de Versão. Se você vai usar o Subversion para gerenciar seus dados, então essa é uma funcionalidade da qual você vai acaber dependendo. Este capitulo assume que você já esta familiarizado com os conceitos básicos do Subversion(Capítulo 1, *Conceitos Fundamentais*).

O que é um Ramo?

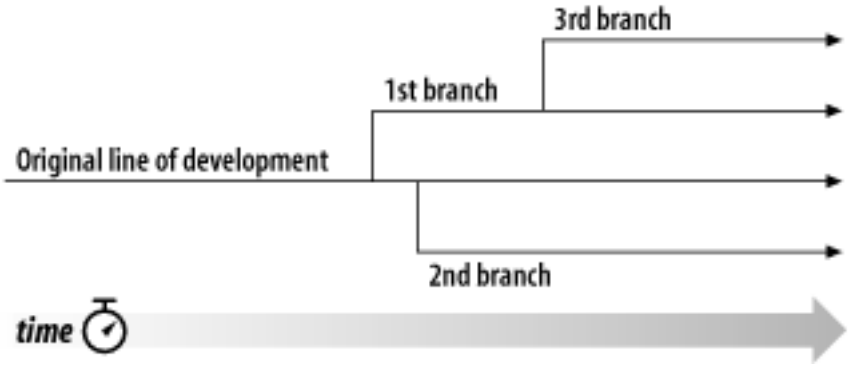
Suponha que é o seu trabalho manter um documento de uma divisão de sua empresa, um livro de anotações por exemplo. Um dia, uma outra divisão lhe pede este mesmo livro, mas com alguns “ajustes” para eles, uma vez que eles trabalham de uma forma um pouco diferente.

O que você faz nessa situação? Você faz o obvio: faz uma segunda cópia do seu documento, e começa a controlar as duas cópias separadamente. Quando cada departamento lhe requisitar por alterações, você as realiza em um copia, ou na outra.

Em raros casos você vai precisar fazer alterações nos dois documentos. Um exemplo, se você encontrar um erro em um dos arquivos, é muito provavel que este erro exista na segunda cópia. A final, os dois documentos são quase identicos, eles apenas tem pequenas diferenças, em locais especificos.

Este é o conceito basico de *Ramo*—namely, uma linha de desenvolvimento que existe independente de outra linha, e ainda, partilham um historico em comum, se você olha para tras na linha tempo. Um Ramo sempre se inicia como cópia de outra coisa, e segue rumo proprio a partir desse ponto, gerando seu proprio historico. (veja Figura 4.1, “Ramos no desenvolvimento”).

Figura 4.1. Ramos no desenvolvimento



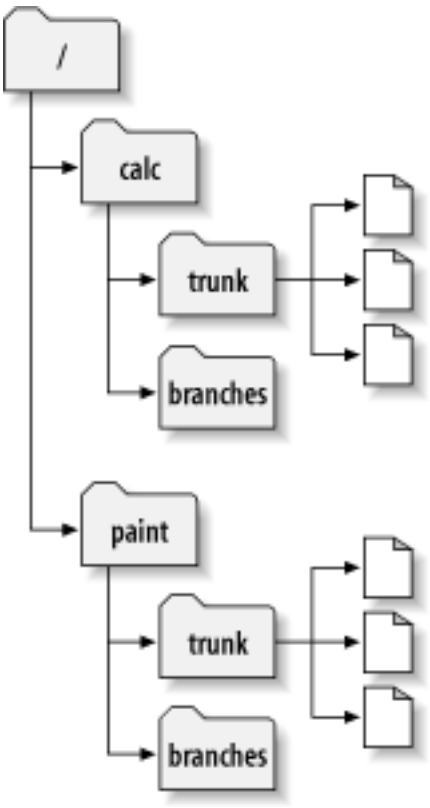
O Subversion tem comandos para ajudar a controlar Ramos paralelos de um arquivo ou diretório. Ele permite você criar ramos copiando seus dados, e ainda lembra que as copias tem relação entre si. Ainda é possivel duplicar copias de um ramo para outro. Finalmente, ele pode fazer com que partes de sua copia de trabalho reflitam ramos diferentes, assim você pode “misturar e combinar” diferentes linhas de desenvolvimento no seu trabalho de dia-a-dia.

Usando Ramos

Até aqui, você ja deve saber como cada commit cria uma nova árvore de arquivos (chamada de “revisão”) no repositório. Caso não saiba, volte e leia sobre revisões em “Revisões”.

Neste capítulo, vamos usar o mesmo exemplo de antes: Capítulo 1, *Conceitos Fundamentais*. Lembre-se que você e Sally estão compartilhando um repositório que contem dois projetos, `paint` e `calc`. Note que em Figura 4.2, “Layout Inical do Repositorio”, entretanto, cada diretório de projeto contem subdiretorios chamados `trunk` e `branches`. O motivo para isso logo ficará mais claro.

Figura 4.2. Layout Inical do Repositorio



Como antes, assuma que você e Sally possuem copias de trabalho do projeto “calc”. Especificamente, cada um de vocês tem uma copia de trabalho de `/calc/trunk`. Todos os arquivos deste projeto estão nesse diretório ao invés de estarem no `/calc`, porque a sua equipe decidiu que `/calc/trunk` é onde a “Linha Principal” de desenvolvimento vai ficar.

Digamos que você recebeu a tarefa de implementar uma grande funcionalidade nova no projeto. Isso vai requerer muito tempo para escrever, e vai afetar todos os arquivos do projeto. O problema aqui é que você não quer interferir no trabalho de Sally, que esta corrigindo pequenos bugs aqui e ali. Ela depende de que a ultima versão do projeto (em `/calc/trunk`) esteja sempre disponível. Se você começar a fazer commits de suas modificações bit-a-bit, com certeza você vai dificultar o trabalho de Sally.

Um estratégia é "se isolar": você e Sally podem para de compartilhar informações por uma semana ou duas.Isto é, começar cortar e reorganizar todos os arquivos da sua cópia de trabalho, mas não realizar commit ou update antes de ter terminado todo o trabalho. Existem alguns problemas aqui. Primeiro, não é seguro. A maioria das pessoas gostam de salvar seu trabalho no repositorio com frequencia, caso

algo ruim aconteça por acidente à cópia de trabalho. Segundo, não é nada flexível. Se você faz seu trabalho em computadores diferentes (talves você tenha uma cópia de trabalho de `/calc/trunk` em duas máquinas diferentes), você terá que, manualmente, copiar suas alterações de uma máquina para outra, ou simplesmente, realizar todo o trabalho em um único computador. Por esse mesmo método, é dificil compartilhar suas constantes modificações com qualquer pessoa. Uma “boa prática” comum em desenvolvimento de software é permitir que outros envolvidos revisem seu trabalho enquanto sendo realizado. Se ninguém verificar seus commits intermediários, você perde um potêncial feedback. E por fim, quando você terminar todas as modificações, você pode achar muito dificil de fundir seu trabalho com o resto da linha principal de desenvolvimento da empresa. Sally (ou outros) podem ter realizado muitas outras mudanças no repositório que podem ser difíceis de incorporar com sua cópia de trabalho— especialmente se você rodar um **svn update** depois de semanas trabalhando sozinho.

A melhor solução é criar seu próprio ramo, ou linha de desenvolvimento, no repositório. Isso lhe permite salvar seu trabalho ainda incompleto, sem interferir com outros, e ainda você pode escolher que informações compartilhar com seus colaboradores. Você verá exatamente como isso funciona mais a frente.

Criando um Ramo

Criar um ramo é realmente simples— você faz uma cópia do projeto no repositório usando o comando**svn copy**. O Subversion copia não somente arquivos mas também diretórios completos. Neste caso, você quer fazer a cópia do diretório `/calc/trunk`. Onde deve ficar a nova cópia? Onde você quiser— isso depende da "política" do projeto. Digamos que sua equipe tem a política de criar novos ramos na área `/calc/branches` do repositório, e você quer chamar o seu ramo de `my-calc-branch`. Você vai querer criar um novo diretório, `/calc/branches/my-calc-branch`, que inicia sua vida como cópia de `/calc/trunk`.

Há duas maneiras diferentes de fazer uma cópia. Vamos mostrar primeiro a maneira complicada, apenas para deixar claro o conceito. Para começar, faça um checkout do diretório raiz do projeto, `/calc`:

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

Agora para fazer uma cópia basta passar dois caminhos de cópia de trabalho ao comando **svn copy**:

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

Neste caso, o comando **svn copy** faz uma cópia recursiva do diretório `trunk` para um novo diretório de trabalho, `branches/my-calc-branch`. Como você pode ver pelo comando **svn status**, o novo diretório está agendado para ser adicionado ao repositório. Note também o sinal “+” próximo à letra A. Isso indica o item adicionado é uma *cópia* de algo e não um item novo. Quando você realizar o Commit das modificações, o Subversion vai criar o diretorio `/calc/branches/my-calc-branch` no repositóio copiando `/calc/trunk`, ao invés de reenviar todos os dados da cópia de trabalho pela rede:

```
$ svn commit -m "Criando um ramo do diretório /calc/trunk."
Adding         branches/my-calc-branch
Committed revision 341.
```

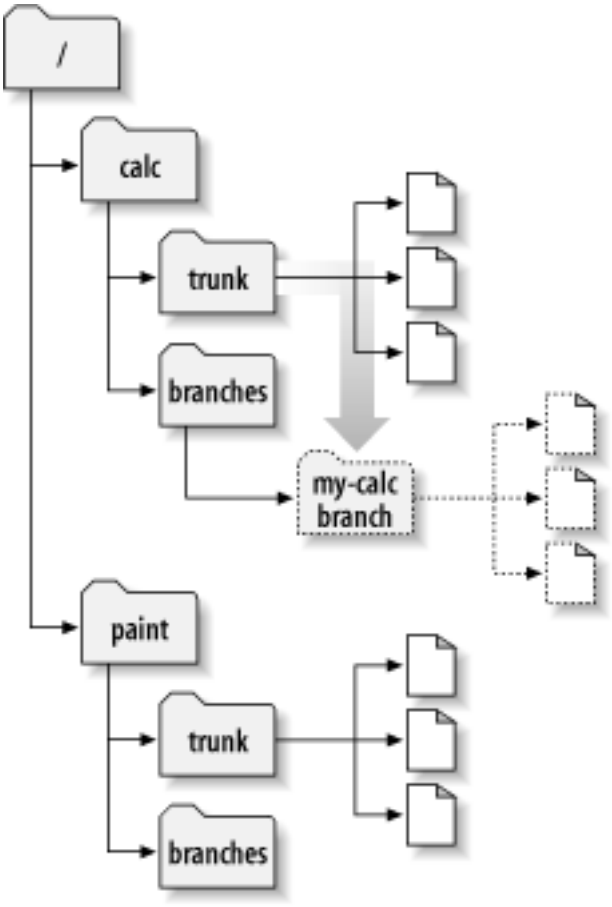
E aqui está o método mais fácil de criar um ramo, o qual nós deveriamos ter lhe mostrado desde o início: o comando **svn copy** é capaz de copiar diretamente duas URLs.

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/my-calc-branch \
      -m "Criando um ramo do diretório /calc/trunk."

Committed revision 341.
```

Do ponto de vista do diretório, não há diferença entre estes dois métodos. Ambos os processos criam um novo diretório na revisão 341, e o novo diretório é uma cópia de `/calc/trunk`. Isso é mostrado em Figura 4.3, “Repositório com uma nova cópia”. Note que o segundo método, entretanto, faz um commit *imediato* em tempo constante. [1] Este é um procedimento mais fácil, uma vez que você não precisa fazer o checkout de uma grande parte do repositório. Na verdade, para usar esta técnica você não precisa se quer ter uma cópia de trabalho. Esta é a maneira que a maioria dos usuários criam ramos.

Figura 4.3. Repositório com uma nova cópia



[1] O Subversion não suporta a cópia entre repositórios distintos. Quando usando URLs com os comandos **svn copy** ou **svn move**, você pode apenas copiar itens dentro de um mesmo repositório.

Cópias Leves

O repositório do Subversion tem um design especial. Quando você copia um diretório, você não precisa se preocupar com o repositório ficando gigante—O Subversion, na realidade, não duplica dados. Ao invés disso, ele cria uma nova entrada de diretório que aponta para uma outra árvore de diretório *já existente*. Caso você seja um usuário Unix, esse é o mesmo conceito do hard-link. Enquanto as modificações são feitas em pastas e arquivos no diretório copiado, o Subversion continua aplicando esse conceito de hard-link enquanto for possivel. Os dados somente serão duplicados quando for necessário tira ambiguidade de diferentes versões de um objeto.

É por isso que você quase não vai ouvir os usuários do Subversion reclamando de “Cópias Leves” (*cheap copies*). Não importa o quão grande é o diretório— a cópia sempre sera feita em um pequeno e constante espaço de tempo. Na verdade, essa funcionalidade é a base do funcionamento do commit no Subversion: cada revisão é uma “cópia leve” da revisão anterior, com algumas ligeiras modificações em alguns itens.(para ler mais sobre esse assunto, visite o website do Subversion e leia o método “bubble up” nos documentos de design do Subversion.)

Claro que estes mecanismos internos de copiar e compartilhar dados estão escondidos do usuário, que vê apenas copias das arvores de arquivos. O ponto principal aqui é que as cópias são leves, tanto em tempo quanto em tamanho. Se você criar um ramo inteiro dentro do repositório (usando o comando **svn copy URL1 URL2**), será uma operação rápida, e de tempo constante. Crie ramos sempre que quiser.

Trabalhando com o seu Ramo

Agora que você criou um ramo do projeto, você pode fazer um Checkout para uma nova cópia de trabalho e usa-la.

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

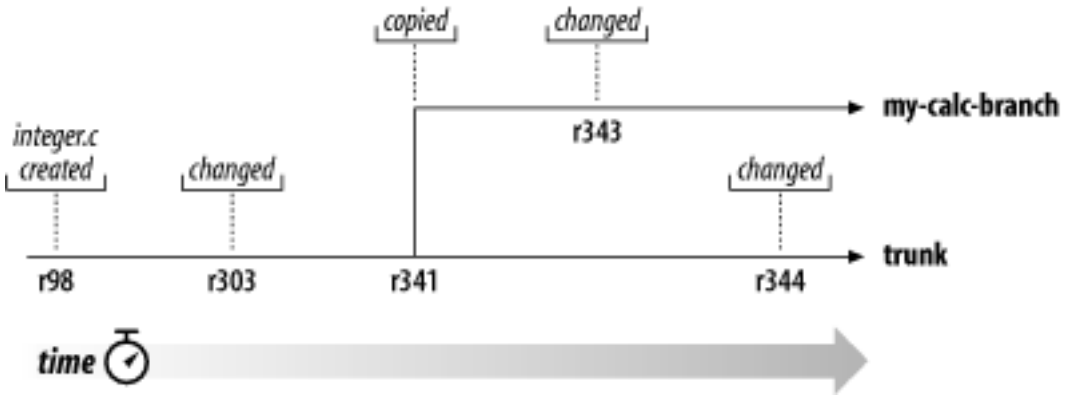
Não tem nada de especial nessa cópia de trabalho; ela simplesmente aponta para um diretório diferente no repositorio. Entretanto, quando você faz o commit de modificações, essas não ficarão visiveis para Sally quando ela fizer Update, porque a cópia de trabalho dela aponta paar `/calc/trunk`. (Leia “Traversing Branches” logo a frente neste capítulo: o comando **svn switch** é uma forma alternativa de se criar uma cópia de trabalho de um ramo.)

Vamo imaginar que tenha se passado uma semana, e o seguinte commit é realizado:

- Você faz uma modificação em `/calc/branches/my-calc-branch/button.c`, o que cria a revisão 342.
- Você faz uma modificação em `/calc/branches/my-calc-branch/integer.c`, o que cria a revisão 343.
- Sally faz uma modificação em `/calc/trunk/integer.c`, o que cria a revisão 344.

Exitem agora duas linha independentes de desenvolvimento, mostrando em Figura 4.4, “Ramificação do histórico de um arquivo”, afetando `integer.c`.

Figura 4.4. Ramificação do histórico de um arquivo



As coisas ficam interessantes quando você olha o histórico das alterações feitas na sua cópia de `integer.c`:

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

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

* integer.c:  frozzled the wazjub.

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

Creating a private branch of /calc/trunk.

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

* integer.c:  changed a docstring.

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

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

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

Note que o Subversion esta traçando o histórico do seu ramo de `integer.c` pelo tempo, até o momento em que ele foi copiado. Isso mostra o momento em que o ramo foi criado como uma evento no histórico, ja que `integer.c` foi copiado implicitamente quando `/calc/trunk/` foi copiado. Agora veja o que ocorre quando Sally executa o mesmo comando em sua cópia do arquivo:

```
$ pwd
/home/sally/calc

$ svn log -v integer.c
------------------------------------------------------------------------
r344 | sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

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

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

* integer.c:  changed a docstring.

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

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

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

Sally vê suas proprias modificações na revisão 344, e não as modificações que você fez na revisão 343. Até onde o Subversion sabe, esses dois commits afetaram arquivos diferentes em locais distintos no repositório. Entretanto o Subversion *mostra* que os dois arquivos tem um histórico em comum. Antes de ser feita a cópia/ramo na revisão 341, eles eram o mesmo arquivo. É por isso que você e Sally podem ver as alterações feitas nas revisões 303 e 98.

O conceito chame por trás de ramos

Há duas lições importantes que você deve se lembrar desta seção. Primeiro, o Subversion não tem um conceito interno de ramosmdash;ele apenas sabe fazer cópias. Quando você copia um diretório, o diretório resultante somente é um “ramo” porque *você* atribui esse significado a ele. Você pode pensar de forma diferente sobre esse diretório, ou trata-lo de forma diferente, mas para o subversion é apenas um diretório comum que carrega uma informação extra de histórico. Segundo, devido a este mecânismo de cópia, os ramos no Subversion existem como *diretórios com sistema de arquivo normal* no repositório. Isso é de outros sistemas de controle de versão, onde ramos são criados ao adicionar “rótulos” extra-dimensionais aos arquivos.

Copiando modificações entre ramos

Agora você e Sally estão trabalhando em ramos paralelos do projeto: você está trabalhando no seu próprio ramo, e Sally está trabalhando no *tronco*, ou linha principal de desenvolvimento.

Para projetos que tenham um grande numero de colaboradores, é comum que cada um tenha sua cópia de trabalho do tronco. Sempre que alguem precise fazer uma longa modificação que possa corromper o tronco, o procedimento padrão é criar um ramo privado e fazer os commits neste ramo até que todo o trabaho esteja concluido.

Então, a boa notícia é que você não está interferindo no trabalho de Sally, e vice-versa. A má notícia, é que é muito fácil se *distânciar* do projeto. Lembre-se que um dos problemas com a estratégia do “se isolar” é que quando você terminar de trabalhar no seu ramo, pode ser bem perto de impossivel de fundir suas modificações novamente com o tronco do projeto sem um grande numero de conflitos.

Ao invés disso, você e Sally devem continuamente compartilhar as modificações ao longo do seu trabalho. Depende de você para decidir quais modificações devem ser compartilhadas; O subversion lhe da a capacidade para selecionar o que “copiar” entre os ramos. E quando você terminar de trabalhar no seu ramo, todas as modificações realizadas no seu ramo podem ser copiadas novamente para o tronco.

Copiando modificações específicas

Na seção anterior, nos comentamos que tanto você quanto Sally fizeram alterações em `integer.c` em ramos distintos.Se você olhar a mensagem de log de Sally na revisão 344, você verá que ela corrigiu alguns erros de escrita. Sem duvida alguma, a sua cópia deste arquivo tem os mesmo erros de escrita. É provavel que suas futuras modificações a este arquivo vão afetar as mesmas áreas onde foram feitas as correções de escrita, então você tem grandes chances de ter varios conflitos quando for fundir o seu ramo, eventualmente. Portanto, é melhor receber as modificações de Sally agora, *antes* de você começar a trabalhar de forma massiva nessas áreas.

É hora de usar o comando **svn merge**. Esse comando é um primo muito próximo do comando **svn diff** (que você viu em Capítulo 2, *Uso Básico*). Os dois comando comparam dois objetos no repositório e mostram as diferenças. Por exemplo, você pode pedir com o comando **svn diff** para ver com exatidão as mudanças feitas por Sally na revisão 344:

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

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

     info->extra_header = (unsigned char *) my_malloc(total);
```

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

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

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

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

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

O comando **svn merge** é quase que o mesmo. Ao invés de imprimir as diferenças no terminal, ele as aplica diretamente à cópia de trabalho classificando como *local modifications*:

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

$ svn status
M  integer.c
```

A saida do comando **svn merge** mostra a sua cópia de `integer.c` sofreu uma correção. Agora ele contém as modificações feitas por Sally— essas modificações foram “copiadas” do tronco do repositório para a cópia de trabalho do seu ramo privado, e agora existe como uma modificação local. A esta altura, depende de você revisar essa modificação local e ter certeza de funciona.

Em outra simulação, é possivel que as coisas não tenham ocorrido tão bem assim, e o arquivo `integer.c` tenha entrado em estado de conflito. Pode ser que você precise resolver o conflito usando procedimentos padrão (see Capítulo 2, *Uso Básico*), ou se você decidir que fazer a fusão dos arquivos tenha sido uma má idéia, desista e rode o comando **svn revert** para retirar as modificações locais.

Partindo do pré-suposto que você revisou as modificações do processo de fusão , então você pode fazer o **svn commit** como de costume. A este ponto, a mudança foi fusionada ao seu ramo no repositório. Em tecnologias de controle de versão, esse ato de copiar mudanças entre ramos recebe o nome de *portar* mudanças.

Quando você fizer o commit das modificações locais, não esqueça de colocar na mensagem de log que você está portando uma modificação especifica de um ramo para outro.Por exemplo:

```
$ svn commit -m "integer.c: ported r344 (spelling fixes) from trunk."
Sending        integer.c
Transmitting file data .
Committed revision 360.
```

Como você verá nas próximas seções, essa é uma “boa pratica” importantíssima a ser seguida.

Porque não usar Patches?

Essa questão pode estar em sua mente, especialmente se você for um usuário de Unix: porque usar o comando **svn merge**? Porque não simplismente usar o comando do sistema **patch** para realizar esta tarefa? Por exemplo:

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

Neste caso em particular, sim, realmente não há diferença. Mas o comando **svn merge** tem habilidades especiais que superam o comando **patch**. O formato do arquivo usado pelo **patch** é bem limitado; é apenas capaz de mexer o conteúdo dos arquivos. Não há forma de representar mudanças em *arvores*, como o criar, remover e renomear arquivos e diretórios. Tão pouco pode o comando **patch** ver mudanças de propriedades. Se nas modificações de Sally, um diretório tivesse sido criado, a saida do comando **svn diff** não iria fazer menção disso. **svn diff** somente mostra forma limitada do patch, então existem coisa que ele simplismente não irá mostrar. O comando **svn merge**, por sua vez, pode mostrar modificações em estrutura de árvores e propriedades aplicando estes diretamente em sua cópia de trabalho.

Um aviso: enquanto o comando **svn diff** e o **svn merge** tem conceitos similare, eles apresentam sintaxe diferente em vários casos. Leia sobre isso em Capítulo 9, *Referência Completa do Subverion* para mais detalhes, ou peça ajuda ao comando **svn help**. Por exemplo, o comando **svn merge** precisa de uma cópia de trabalho com destino, isto é, um local onde aplicar as modificações. Se um destino não for especificado, ele assume que você está tentando uma dessas operações:

1. Você quer fundir modificações de diretorio no seu diretório de trabalho atual.

2. Você quer fundir as modificações de um arquivo em específico, em outro arquivo de mesmo nome que existe no seu diretorio atual de trabalho.

Se você esta fundindo um diretório e não especificou um destino, **svn merge** assume o primeiro caso acima e tenta aplicar as modificações no seu diretório atual. Se você está fundindo um arquivo, e este arquivo (ou arquivo de mesmo nome) existe no diretório atual, o **svn merge** assume o segundo caso, e tenta aplicar as modificações no arquivo local de mesmo nome.

Se você quer que as modificações seja aplicadas em outro local, você vai precisar avisar. Por exemplo, se você está no diretório pai de sua cópia de trabalho, você vai precisar especificar o diretório de destino a receber as modificações:

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

O conceito chave sobre fusão

Agora você viu um exemplo do comando **svn merge**, e você está prestes a ver vários outros. Se você está se sentindo confuso sobre como a fusão funciona, saiba que você não está sozinho. Vários usuários

(especial os novos em controle de versão) ficam perplexos com a sintaxe do comando, e sobre como e quando deve ser usado. Mas não temas, esse comando é muito mais simples do que você imagina! Existe uma técnica muito simples para entender exatamente o comportamento do comando **svn merge**.

O principal motivo de confusão é o *nome* do comando. O termo “fundir” de alguma forma denota que se junta ramos, ou que existe uma mistura misteriosa de código ocorrendo. Este não é o caso. O nome mais apropriado para o comando deveria ter sido **svn diff-and-apply** , porque isso é o que acontece: duas árvores de repositório são comparadas, e a diferença é aplicada a uma cópia de trabalho.

O comando recebe três argumentos:

1. Uma árvore de repositório inicial (geralmente chamada de *lado esquerdo* da comparação),

2. Uma árvore de repositório final (geralmente chamada de *lado direito* da comparação),

3. Uma cópia de trabalho para receber as diferenças como modificação local (geralmente chamada de *destino* da fusão).

Uma vez especificados estes três argumentos, as duas árvores são comparadas, e o resultado das diferenças são aplicadas sobre a cópia de trabalho de destino, como modificações locais. Uma vez executado o comando, o resultado não é diferente do que se você tivesse editado manualmente os arquivos, ou rodados vários comandos **svn add** ou **svn delete**. Se você gostar do resultado você pode fazer o commit dele. Se você não gostar do resultado, você pode simplismente reverter as mudanças com o comando **svn revert**.

A sintaxe do comando **svn merge** lhe permite especificar os três argumentos necessários de forma flexivel. Veja aqui alguns exemplos:

```
$ svn merge http://svn.example.com/repos/branch1@150 \
            http://svn.example.com/repos/branch2@212 \
            my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
```

A primeira sintaxe usa explicitamente os três argumentos, nomeando cada árvore na forma *URL@REV* e nomeando a cópia de trabalho de destino. A segunda sintaxe pode ser usada como um atalho em situações onde você esteja comparando duas revisões distintas de uma mesma URL. A ultima sintaxe mostra como o argumento da cópia de trabalho de destino é opcional; se omitido, assume como padrão o diretório atual.

Melhores práticas sobre Fusão

Rastreando Fusões manualmente

Fundir modificações parece simples, mas na prática pode se tornar uma dor de cabeça. O problema é que se você repetidamente fundir as modificações de uma ramo com outro, você pode acidentalmente fundir a mesma modificação *duas vezes*. Quando isso ocorre, algumas vezes as coisas vão funcionar corretamente. Quando aplicando um patch em um arquivo, Subversion verifica se o arquivo já possui aquelas modificações e se tiver não faz nada. Mas se a modificaçõa ja existente tiver sido de alguma forma modificada, você terá um conflito.

O ideal seria se o seu sistema de controle de versão prevenisse o aplicar-duas-vezes modificações a um ramo. Ele deveria lembrar automaticamente quais modificações um ramo ja recebeu, e ser capaz de lista-los para você. Essa informação deveria ser usada para ajudar a automatizar a Fusão o máximo possivel.

Infelizmentem, o Subversion não é esse sistema; ele ainda não grava informações sobre as fusões realizadas. [2] Quando você faz o commit das modificações locais, o repositório não faz a menor idéia se as alterações vieram de um comando **svn merge**, ou de uma edição manual no arquivo.

O que isso significa para você, o usuário? Significa que até que o Subversion tenha essa funcionalidade, você terá que rastrear as informações de Fusão pessoalmente. A melhor maneira de fazer isso é com as mensagens de log do commit. Como mostrado nos exemplos anteriores, é recomendavel que sua mensagem de log informe especificamente o número da revisão (ou números das revisões) que serão fundidas ao ramo. Depois, você pode rodar o comando **svn log** para verificar quais modificações o seu ramo ja recebeu. Isso vai lhe ajudar a construir um próximo comando **svn merge** que não será redundante com as modificações ja aplicadas.

Na próxima seção, vamos mostrar alguns exeplos dessa técnica na prática.

Visualizando Fusões

Primeiro, lembre-se de fundir seus arquivos para a cópia de trabalho quando esta *não* tiver alterações locais e tenha sido atualizada recentemente. Se a sua cópia de trabalho não estiver “limpa”, você pode ter alguns problemas.

Assumindo que a sua cópia de trabalho está no ponto, fazer a fusão não será uma operação de alto risco. Se você não fizer a primeira fusão de forma correta, rode o comando **svn revert** nas modificações e tente novamente.

Se você fez a fusão para uma cópia de trabalho que já possui modificações locais, a mudanças aplicadas pela fusão serão misturadas as pré existentes, e rodar o comando **svn revert** não é mais uma opção. Pode ser impossivel de separar os dois grupos de modificações.

Em casos como este, as pessoas se tranquilizam em poder prever e examinar as fusões antes de ocorrerem. Uma maneira simples de fazer isso é rodar o comando **svn diff** com os mesmos argumentos que você quer passar para o comando **svn merge**, como mostramos no primeiro exemplo de fusão. Outro método de prever os impactos é passar a opção `--dry-run` para o comando de fusão:

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

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

A opção `--dry-run` não aplica qualquer mudança para a copia de trabalho. Essa opção apenas exibe os codigos que *seriam* escritos em uma situação real de fusão. É útil poder ter uma previsão de “auto nível” da potencial fusão, para aqueles momentos em que o comando **svn diff** dá detalhes até demais.

Fundir conflitos

Assim como no comando **svn update**, o comando **svn merge** aplica modificações à sua cópia de trabalho. E portanto também é capaz de criar conflitos. Entretanto, os conflitos criados pelo comando **svn merge** são um tanto diferentes, e essa seção explica essas diferenças.

Para começar, assuma que sua cópia de trabalho não teve modificações locais. Quando você faz a atualização com o comando **svn update** para um revisão específica, as modificações enviadas pelo

[2] Entretanto, neste exato momento, essa funcionalidade está sendo preparada!

servidor vão ser sempre aplicadas à sua cópia de trabalho “sem erros”. O servidor produz o delta a partir da comparação de duas árvores: uma imagem virtual de sua cópia de trabalho, e a árvore da revisão na qual está interessado. Como o lado esquerdo da comparação é exatamente igual ao que você já possui, é garantido que o delta converterá corretamente sua cópia de trabalho, para a revisão escolhida no lado direito da compração.

Entretanto, o comando **svn merge** não possui essa garantia e pode ser bem mais caótico: o usuário pode pedir ao servidor para comparar *qualquer* árvore, até mesmo árvores que não tenham relação com a sua cópia de trabalho! Isso significa que existem uma grande marger para erro humano. Usuário vão acabar por compara duas árvores erradas, criando um delta que não se aplica sem conflitos. O comando **svn merge** vai fazer o melhor possivel para aplicar o delta o máximo possivel, mas em algumas partes isso pode ser impossivel. Assim como no comando Unix **patch** que as vezes reclama sobre “failed hunks”, o **svn merge** vai reclamar sobre “alvos perdidos”:

```
$ svn merge -r 1288:1351 http://svn.example.com/repos/branch
U  foo.c
U  bar.c
Skipped missing target: 'baz.c'
U  glub.c
C  glorb.h

$
```

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

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

Another small difference between **svn update** and **svn merge** are the names of the full-text files created when a conflict happens. In “Resolve Conflicts (Merging Others' Changes)”, we saw that an update produces files named `filename.mine`, `filename.rOLDREV`, and `filename.rNEWREV`. When **svn merge** produces a conflict, though, it creates three files named `filename.working`, `filename.left`, and `filename.right`. In this case, the terms “left” and “right” are describing which side of the double-tree comparison the file came from. In any case, these differing names will help you distinguish between conflicts that happened as a result of an update versus ones that happened as a result of a merge.

Noticing or Ignoring Ancestry

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

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

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

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

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

Merges and Moves

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

Alas, this scenario doesn't work so well right now, and is considered one of Subversion's current weak spots. The problem is that Subversion's **update** command isn't as robust as it should be, particularly when dealing with copy and move operations.

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

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

```
$ cd calc/trunk

$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
D   integer.c
A   whole.c
```

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

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

Common Use-Cases

There are many different uses for branching and **svn merge**, and this section describes the most common ones you're likely to run into.

Merging a Whole Branch to Another

To complete our running example, we'll move forward in time. Suppose several days have passed, and many changes have happened on both the trunk and your private branch. Suppose that you've finished working on your private branch; the feature or bug fix is finally complete, and now you want to merge all of your branch changes back into the trunk for others to enjoy.

So how do we use **svn merge** in this scenario? Remember that this command compares two trees, and applies the differences to a working copy. So to receive the changes, you need to have a working copy of the trunk. We'll assume that either you still have your original one lying around (fully updated), or that you recently checked out a fresh working copy of `/calc/trunk`.

But which two trees should be compared? At first glance, the answer may seem obvious: just compare the latest trunk tree with your latest branch tree. But beware—this assumption is *wrong*, and has burned many a new user! Since **svn merge** operates like **svn diff**, comparing the latest trunk and branch trees will *not* merely describe the set of changes you made to your branch. Such a comparison shows too many changes: it would not only show the addition of your branch changes, but also the *removal* of trunk changes that never happened on your branch.

To express only the changes that happened on your branch, you need to compare the initial state of your branch to its final state. Using **svn log** on your branch, you can see that your branch was created in revision 341. And the final state of your branch is simply a matter of using the HEAD revision. That means you want to compare revisions 341 and HEAD of your branch directory, and apply those differences to a working copy of the trunk.

A nice way of finding the revision in which a branch was created (the “base” of the branch) is to use the `--stop-on-copy` option to **svn log**. The log subcommand will normally show every change ever made to the branch, including tracing back through the copy which created the branch. So normally, you'll see history from the trunk as well. The `--stop-on-copy` will halt log output as soon as **svn log** detects that its target was copied or renamed.

So in our continuing example,

```
$ svn log -v --stop-on-copy \
          http://svn.example.com/repos/calc/branches/my-calc-branch
…
------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

$
```

As expected, the final revision printed by this command is the revision in which `my-calc-branch` was created by copying.

Here's the final merging procedure, then:

```
$ cd calc/trunk
$ svn update
At revision 405.

$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

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

# ...examine the diffs, compile, test, etc...

$ svn commit -m "Merged my-calc-branch changes r341:405 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 406.
```

Again, notice that the commit log message very specifically mentions the range of changes that was merged into the trunk. Always remember to do this, because it's critical information you'll need later on.

For example, suppose you decide to keep working on your branch for another week, in order to complete an enhancement to your original feature or bug fix. The repository's HEAD revision is now 480, and you're ready to do another merge from your private branch to the trunk. But as discussed in “Melhores práticas sobre Fusão”, you don't want to merge the changes you've already merged before; you only want to merge everything “new” on your branch since the last time you merged. The trick is to figure out what's new.

The first step is to run **svn log** on the trunk, and look for a log message about the last time you merged from the branch:

```
$ cd calc/trunk
$ svn log
…
------------------------------------------------------------------------
r406 | user | 2004-02-08 11:17:26 -0600 (Sun, 08 Feb 2004) | 1 line

Merged my-calc-branch changes r341:405 into the trunk.
------------------------------------------------------------------------
…
```

Aha! Since all branch-changes that happened between revisions 341 and 405 were previously merged to the trunk as revision 406, you now know that you want to merge only the branch changes after that—by comparing revisions 406 and HEAD.

```
$ cd calc/trunk
$ svn update
At revision 480.

# We notice that HEAD is currently 480, so we use it to do the merge:

$ svn merge -r 406:480 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn commit -m "Merged my-calc-branch changes r406:480 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 481.
```

Now the trunk contains the complete second wave of changes made to the branch. At this point, you can either delete your branch (we'll discuss this later on), or continue working on your branch and repeat this procedure for subsequent merges.

Undoing Changes

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

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

$ svn status
M  integer.c

$ svn diff
…
# verify that the change is removed
…

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

One way to think about a repository revision is as a specific group of changes (some version control systems call these *changesets*). By using the `-r` option, you can ask **svn merge** to apply a changeset, or whole range of changesets, to your working copy. In our case of undoing a change, we're asking **svn merge** to apply changeset #303 to our working copy *backwards*.

Subversion and Changesets

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

In Subversion, a global revision number N names a tree in the repository: it's the way the repository looked after the Nth commit. It's also the name of an implicit changeset: if you compare tree N with tree N-1, you can derive the exact patch that was committed. For this reason, it's easy to think of “revision N” as not just a tree, but a changeset as well. If you use an issue tracker to manage bugs, you can use the revision numbers to refer to particular patches that fix bugs—for example, “this issue was fixed by revision 9238.”. Somebody can then run **svn log -r9238** to read about the exact changeset which fixed the bug, and run **svn diff -c 9238** to see the patch itself. And Subversion's `merge` command also uses revision numbers. You can merge specific changesets from one branch to another by naming them in the merge arguments: **svn merge -r9237:9238** would merge changeset #9238 into your working copy.

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

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

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

Resurrecting Deleted Items

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

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

First, you might need to use **svn log** to discover the exact coordinate pair you wish to resurrect. A good strategy is to run **svn log --verbose** in a directory which used to contain your deleted item. The `--verbose`

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

(`-v`) option shows a list of all changed items in each revision; all you need to do is find the revision in which you deleted the file or directory. You can do this visually, or by using another tool to examine the log output (via **grep**, or perhaps via an incremental search in an editor).

```
$ cd parent-dir
$ svn log -v
…
------------------------------------------------------------------------
r808 | joe | 2003-12-26 14:29:40 -0600 (Fri, 26 Dec 2003) | 3 lines
Changed paths:
   D /calc/trunk/real.c
   M /calc/trunk/integer.c

Added fast fourier transform functions to integer.c.
Removed real.c because code now in double.c.
…
```

In the example, we're assuming that you're looking for a deleted file `real.c`. By looking through the logs of a parent directory, you've spotted that this file was deleted in revision 808. Therefore, the last version of the file to exist was in the revision right before that. Conclusion: you want to resurrect the path `/calc/trunk/real.c` from revision 807.

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

One option is to use **svn merge** to apply revision 808 “in reverse”. (We've already discussed how to undo changes, see “Undoing Changes”.) This would have the effect of re-adding `real.c` as a local modification. The file would be scheduled for addition, and after a commit, the file would again exist in `HEAD`.

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

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

```
$ svn copy -r 807 \
           http://svn.example.com/repos/calc/trunk/real.c ./real.c

$ svn status
A  +   real.c

$ svn commit -m "Resurrected real.c from revision 807, /calc/trunk/real.c."
Adding         real.c
Transmitting file data .
Committed revision 1390.
```

The plus sign in the status output indicates that the item isn't merely scheduled for addition, but scheduled for addition “with history”. Subversion remembers where it was copied from. In the future, running **svn log** on this file will traverse back through the file's resurrection and through all the history it had prior to revision 807. In other words, this new `real.c` isn't really new; it's a direct descendant of the original, deleted file.

Although our example shows us resurrecting a file, note that these same techniques work just as well for resurrecting deleted directories.

Common Branching Patterns

Version control is most often used for software development, so here's a quick peek at two of the most common branching/merging patterns used by teams of programmers. If you're not using Subversion for software development, feel free to skip this section. If you're a software developer using version control for the first time, pay close attention, as these patterns are often considered best practices by experienced folk. These processes aren't specific to Subversion; they're applicable to any version control system. Still, it may help to see them described in Subversion terms.

Release Branches

Most software has a typical lifecycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality-assurance teams take time to test supposedly-stable versions of the software. New work cannot halt while the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bugfix without having to wait for a major new release.

Here's where version control can help. The typical procedure looks like this:

- *Developers commit all new work to the trunk.* Day-to-day changes are committed to `/trunk`: new features, bugfixes, and so on.
- *The trunk is copied to a “release” branch.* When the team thinks the software is ready for release (say, a 1.0 release), then `/trunk` might be copied to `/branches/1.0`.
- *Teams continue to work in parallel.* One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on `/trunk`. If bugs are discovered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is “frozen” for final testing right before a release.
- *The branch is tagged and released.* When testing is complete, `/branches/1.0` is copied to `/tags/1.0.0` as a reference snapshot. The tag is packaged and released to customers.
- *The branch is maintained over time.* While work continues on `/trunk` for version 2.0, bugfixes continue to be ported from `/trunk` to `/branches/1.0`. When enough bugfixes have accumulated, management may decide to do a 1.0.1 release: `/branches/1.0` is copied to `/tags/1.0.1`, and the tag is packaged and released.

This entire process repeats as the software matures: when the 2.0 work is complete, a new 2.0 release branch is created, tested, tagged, and eventually released. After some years, the repository ends up with a number of release branches in “maintenance” mode, and a number of tags representing final shipped versions.

Feature Branches

A *feature branch* is the sort of branch that's been the dominant example in this chapter, the one you've been working on while Sally continues to work on `/trunk`. It's a temporary branch created to work on a complex change without interfering with the stability of `/trunk`. Unlike release branches (which may need to be supported forever), feature branches are born, used for a while, merged back to the trunk, then ultimately deleted. They have a finite span of usefulness.

Again, project policies vary widely concerning exactly when it's appropriate to create a feature branch. Some projects never use feature branches at all: commits to `/trunk` are a free-for-all. The advantage to this system is that it's simple—nobody needs to learn about branching or merging. The disadvantage is that the trunk code is often unstable or unusable. Other projects use branches to an extreme: no change is *ever* committed to the trunk directly. Even the most trivial changes are created on a short-lived branch, carefully reviewed and merged to the trunk. Then the branch is deleted. This system guarantees an exceptionally stable and usable trunk at all times, but at the cost of tremendous process overhead.

Most projects take a middle-of-the-road approach. They commonly insist that `/trunk` compile and pass regression tests at all times. A feature branch is only required when a change requires a large number of destabilizing commits. A good rule of thumb is to ask this question: if the developer worked for days in isolation and then committed the large change all at once (so that `/trunk` were never destabilized), would it be too large a change to review? If the answer to that question is “yes”, then the change should be developed on a feature branch. As the developer commits incremental changes to the branch, they can be easily reviewed by peers.

Finally, there's the issue of how to best keep a feature branch in “sync” with the trunk as work progresses. As we mentioned earlier, there's a great risk to working on a branch for weeks or months; trunk changes may continue to pour in, to the point where the two lines of development differ so greatly that it may become a nightmare trying to merge the branch back to the trunk.

This situation is best avoided by regularly merging trunk changes to the branch. Make up a policy: once a week, merge the last week's worth of trunk changes to the branch. Take care when doing this; the merging needs to be hand-tracked to avoid the problem of repeated merges (as described in “Rastreando Fusões manualmente”). You'll need to write careful log messages detailing exactly which revision ranges have been merged already (as demonstrated in “Merging a Whole Branch to Another”). It may sound intimidating, but it's actually pretty easy to do.

At some point, you'll be ready to merge the “synchronized” feature branch back to the trunk. To do this, begin by doing a final merge of the latest trunk changes to the branch. When that's done, the latest versions of branch and trunk will be absolutely identical except for your branch changes. So in this special case, you would merge by comparing the branch with the trunk:

```
$ cd trunk-working-copy

$ svn update
At revision 1910.

$ svn merge http://svn.example.com/repos/calc/trunk@1910 \
            http://svn.example.com/repos/calc/branches/mybranch@1910
U  real.c
U  integer.c
A  newdirectory
A  newdirectory/newfile
…
```

By comparing the `HEAD` revision of the trunk with the `HEAD` revision of the branch, you're defining a delta that describes only the changes you made to the branch; both lines of development already have all of the trunk changes.

Another way of thinking about this pattern is that your weekly sync of trunk to branch is analogous to running **svn update** in a working copy, while the final merge step is analogous to running **svn commit** from a working copy. After all, what else *is* a working copy but a very shallow private branch? It's a branch that's only capable of storing one change at a time.

Traversing Branches

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

```
$ cd calc

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

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

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

After “switching” to the branch, your working copy is no different than what you would get from doing a fresh checkout of the directory. And it's usually more efficient to use this command, because often branches only differ by a small degree. The server sends only the minimal set of changes necessary to make your working copy reflect the branch directory.

The **svn switch** command also takes a `--revision` (`-r`) option, so you need not always move your working copy to the HEAD of the branch.

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

1. Copy the project's entire “trunk” to a new branch directory.

2. Switch only *part* of the trunk working copy to mirror the branch.

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

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

Note that while it's okay for your working copy to reflect a mixture of repository locations, these locations must all be within the *same* repository. Subversion repositories aren't yet able to communicate with one another; that's a feature planned for the future. [4]

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

Switches and Updates

Have you noticed that the output of **svn switch** and **svn update** look the same? The `switch` command is actually a superset of the update command.

When you run **svn update**, you're asking the repository to compare two trees. The repository does so, and then sends a description of the differences back to the client. The only difference between **svn switch** and **svn update** is that the `update` command always compares two identical paths.

That is, if your working copy is a mirror of `/calc/trunk`, then **svn update** will automatically compare your working copy of `/calc/trunk` to `/calc/trunk` in the `HEAD` revision. If you're switching your working copy to a branch, then **svn switch** will compare your working copy of `/calc/trunk` to some *other* branch-directory in the `HEAD` revision.

In other words, an update moves your working copy through time. A switch moves your working copy through time *and* space.

Because **svn switch** is essentially a variant of **svn update**, it shares the same behaviors; any local modifications in your working copy are preserved when new data arrives from the repository. This allows you to perform all sorts of clever tricks.

For example, suppose you have a working copy of `/calc/trunk` and make a number of changes to it. Then you suddenly realize that you meant to make the changes to a branch instead. No problem! When you **svn switch** your working copy to the branch, the local changes will remain. You can then test and commit them to the branch.

Rótulos

Outro conceito comum do controle de versão é *ramo*. Um ramo é apenas uma “foto” do projeto no momento. No Subversion, essa idéia parece estar em todo lugar. Cada revisão do repositório é exatamente isso—uma foto da estrutura depois de cada commit.

Entretando, pessoas normalmente querem dar rótulos mais amigáveis como nomes de tags, como `versão-1.0`. E querem fazer “fotos” de pequenos sub-diretórios da estrutura. Além do mais, não é fácil lembrar que versão-1.0 de um pedaço do software é um particular sub-diretório da revisão 4822.

Criando um rótulo simples

Mais uma vez, **snv copy** vem para nos socorrer. Se você quer criar uma foto do `/calc/trunk` exatamente como ele está na revisão `HEAD`, fazendo uma copia dela:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/tags/release-1.0 \
      -m "Rótulando a versão 1.0 do projeto 'calc'."

Committed revision 351.
```

Este exemplo assume que o diretório `/calc/tags` já existe. (Se ele não existir, você pode criá-lo usando **svn mkdir**.) Depois da copia completar, o novo diretório `versão-1.0` será para sempre uma foto de como o projeto estava na revisão `HEAD` no momento que a copia foi feita. Claro que você pode querer mais precisão em saber qual revisão a copia foi feita, em caso de alguém ter feito commit no projeto quando

você não estava vendo. Então se você sabe que a revisão 350 do `/calc/trunk` é exatamente a foto que você quer, você pode especificar isso passando `-r 350` para o comando **svn copy**.

Mas espere um pouco: não é essa criação do rótulo o mesmo procedimento para criar um ramo? Sim, de fato, é. No Subversion, não há diferença entre um rótulo e um ramo. Assim como com ramos, a única razão uma cópia é um “rótulo” é porque *humanos* decidiram tratar isso desse jeito: desde que ninguém nunca faça commit para esse diretório, ele permanecerá para sempre uma foto. Se as pessoas começarem a fazer commit para ele, ele se transoforma num ramo.

Se você está administrando um repositório, existe duas maneiras para gerenciar rótulos. A primeira é “não toque”: como uma política do projeto, decida onde os rótulos vão morar, e garanta que todos os usuários saibam como tratar os diretórios que eles vão copiar para lá. (Isso quer dizer, garanta que eles saibam que não devem fazer neles.) A segunda é mais paranóica: você pode usar um dos scripts de controle de acesso providos com o Subversion para previnir que alguém faça algo além de apenas criar novas copias na área de rótulos (Veja Capítulo 6, *Configuração do Servidor*.) A maneira paranoica, entrentanto, não é necessária. Se algum usuário acidentalmente fizer commit de alguam mudança para o diretório de rótulo, você pode simplesmente desfazer a mudança como discutido na revisção anterior. É um controle de versão apesar de tudo.

Criando um rótulo complexo

Algumas vezes você que sua “foto” seja mais complicada que um simples diretório de uma única revisão.

Por exemplo, pense que seu projeto é muito maior que nosso exemplo `calc`: suponha que contém um número de sub-diretórios e muitos outros arquivos. No curso do seu trabalho, você pode decidir que você precisa criar um cópia de trabalho que é destinado para novos recursos e correções de erros. Você pode conseguir isso selecionando arquivos e diretórios com datas anteriores em uma revisão particular (usando **svn update -r** livremente), ou mudando arquivos e diretórios para um ramo em particular (fazendo uso do **svn switch**). Quando estiver pronto, sua cópia de trabalho será uma mistura de diferentes revisões. Mas depois de testes, você saberá que é exatamente a combinação que você precisa.

Hora de fazer a foto. Copiar uma URL para outra não vai funcionar aqui. Nesse caso, você quer fazer uma foto exata da cópia de trabalho que você organizou e armazenar no repositório. Felizmente, **svn copy** na verdade tem quatro diferentes maneiras de ser usado (você pode ler sobre em Capítulo 9, *Referência Completa do Subverion*), incluindo a habilidade de copiar uma árvore de cópia de trablho para o respositório:

```
$ ls
my-working-copy/

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

Committed revision 352.
```

Agora existe um novo diretório no respositório `/calc/tags/mytag`, que é uma foto exata da sua cópia de trabalho—combinado revisões, URLs, e tudo mais.

Outros usuários tem encontrado usos interessantes para esse recurso. Algumas vezes existe situações onde você tem um monte de mudanças locais na sua cópia de trabalho, e você gostaria que um colega de trabalho as visse. Ao invés de usar **svn diff** e enviar o arquivo patch (que não irá ter as informações de mudança na árvore de diretórios, em symlink e mudanças nas propriedades), você pode usar **svn copy** para “subir” sua cópia local para uma área privada no repositório. Seu colega pode verificar o nome de cópia da sua cópia de trabalho, ou usar **svn merge** para receber as exatas mudanças.

Sendo isso um método legal para subir uma rápida foto do seu trabalho local, note que isso *não* é uma boa maneira de iniciar um ramo. A criação de um ramo deve ser um evento solitário, e esse método exige a criação de um ramo com mudanças extras em arquivos, tudo em uma única revisão. Isso dificulta muito (mais tarde) a identificar um número de uma revisão como um ponto de um ramo.

Já se encontrou fazendo edições complexas (no sua cópia de trabalho `/trunk`) e de repente percebe, “Ei, estas mudanças deviam estar num ramo próprio?” Uma ótima técnica para fazer isso pode ser resumir em dois passos:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/newbranch
Committed revision 353.

$ svn switch http://svn.example.com/repos/calc/branches/newbranch
At revision 353.
```

O comando **svn switch**, como **svn update**, preserva suas edições locais. Nesse ponto, sua cópia de trabablho é um reflexo do novo ramo criado, e seu próximo **svn commit** ira enviar suas mudanças para lá.

Branch Maintenance

You may have noticed by now that Subversion is extremely flexible. Because it implements branches and tags with the same underlying mechanism (directory copies), and because branches and tags appear in normal filesystem space, many people find Subversion intimidating. It's almost *too* flexible. In this section, we'll offer some suggestions for arranging and managing your data over time.

Repository Layout

There are some standard, recommended ways to organize a repository. Most people create a `trunk` directory to hold the “main line” of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, then often people create these top-level directories:

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

If a repository contains multiple projects, admins typically index their layout by project (see “Planning Your Repository Organization” to read more about “project roots”):

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

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, the **svn**

move command can move or rename them however you wish. Switching from one layout to another is just a matter of issuing a series of server-side moves; if you don't like the way things are organized in the repository, just juggle the directories around.

Remember, though, that while moving directories may be easy to do, you need to be considerate of your users as well. Your juggling can be disorienting to users with existing working copies. If a user has a working copy of a particular repository directory, your **svn move** operation might remove the path from the latest revision. When the user next runs **svn update**, she will be told that her working copy represents a path that no longer exists, and the user will be forced to **svn switch** to the new location.

Data Lifetimes

Another nice feature of Subversion's model is that branches and tags can have finite lifetimes, just like any other versioned item. For example, suppose you eventually finish all your work on your personal branch of the `calc` project. After merging all of your changes back into `/calc/trunk`, there's no need for your private branch directory to stick around anymore:

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

Committed revision 375.
```

And now your branch is gone. Of course it's not really gone: the directory is simply missing from the `HEAD` revision, no longer distracting anyone. If you use **svn checkout**, **svn switch**, or **svn list** to examine an earlier revision, you'll still be able to see your old branch.

If browsing your deleted directory isn't enough, you can always bring it back. Resurrecting data is very easy in Subversion. If there's a deleted directory (or file) that you'd like to bring back into `HEAD`, simply use **svn copy -r** to copy it from the old revision:

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

Committed revision 376.
```

In our example, your personal branch had a relatively short lifetime: you may have created it to fix a bug or implement a new feature. When your task is done, so is the branch. In software development, though, it's also common to have two “main” branches running side-by-side for very long periods. For example, suppose it's time to release a stable version of the `calc` project to the public, and you know it's going to take a couple of months to shake bugs out of the software. You don't want people to add new features to the project, but you don't want to tell all developers to stop programming either. So instead, you create a “stable” branch of the software that won't change much:

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

Committed revision 377.
```

And now developers are free to continue adding cutting-edge (or experimental) features to `/calc/trunk`, and you can declare a project policy that only bug fixes are to be committed to `/calc/branches/stable-1.0`. That is, as people continue to work on the trunk, a human selectively ports bug fixes over to

the stable branch. Even after the stable branch has shipped, you'll probably continue to maintain the branch for a long time—that is, as long as you continue to support that release for customers.

Vendor branches

As is especially the case when developing software, the data that you maintain under version control is often closely related to, or perhaps dependent upon, someone else's data. Generally, the needs of your project will dictate that you stay as up-to-date as possible with the data provided by that external entity without sacrificing the stability of your own project. This scenario plays itself out all the time—anywhere that the information generated by one group of people has a direct effect on that which is generated by another group.

For example, software developers might be working on an application which makes use of a third-party library. Subversion has just such a relationship with the Apache Portable Runtime library (see “The Apache Portable Runtime Library”). The Subversion source code depends on the APR library for all its portability needs. In earlier stages of Subversion's development, the project closely tracked APR's changing API, always sticking to the “bleeding edge” of the library's code churn. Now that both APR and Subversion have matured, Subversion attempts to synchronize with APR's library API only at well-tested, stable release points.

Now, if your project depends on someone else's information, there are several ways that you could attempt to synchronize that information with your own. Most painfully, you could issue oral or written instructions to all the contributors of your project, telling them to make sure that they have the specific versions of that third-party information that your project needs. If the third-party information is maintained in a Subversion repository, you could also use Subversion's externals definitions to effectively “pin down” specific versions of that information to some location in your own working copy directory (see “Definições Externas”).

But sometimes you want to maintain custom modifications to third-party data in your own version control system. Returning to the software development example, programmers might need to make modifications to that third-party library for their own purposes. These modifications might include new functionality or bug fixes, maintained internally only until they become part of an official release of the third-party library. Or the changes might never be relayed back to the library maintainers, existing solely as custom tweaks to make the library further suit the needs of the software developers.

Now you face an interesting situation. Your project could house its custom modifications to the third-party data in some disjointed fashion, such as using patch files or full-fledged alternate versions of files and directories. But these quickly become maintenance headaches, requiring some mechanism by which to apply your custom changes to the third-party data, and necessitating regeneration of those changes with each successive version of the third-party data that you track.

The solution to this problem is to use *vendor branches*. A vendor branch is a directory tree in your own version control system that contains information provided by a third-party entity, or vendor. Each version of the vendor's data that you decide to absorb into your project is called a *vendor drop*.

Vendor branches provide two benefits. First, by storing the currently supported vendor drop in your own version control system, the members of your project never need to question whether they have the right version of the vendor's data. They simply receive that correct version as part of their regular working copy updates. Secondly, because the data lives in your own Subversion repository, you can store your custom changes to it in-place—you have no more need of an automated (or worse, manual) method for swapping in your customizations.

General Vendor Branch Management Procedure

Managing vendor branches generally works like this. You create a top-level directory (such as `/vendor`) to hold the vendor branches. Then you import the third party code into a subdirectory of that top-level

directory. You then copy that subdirectory into your main development branch (for example, `/trunk`) at the appropriate location. You always make your local changes in the main development branch. With each new release of the code you are tracking you bring it into the vendor branch and merge the changes into `/trunk`, resolving whatever conflicts occur between your local changes and the upstream changes.

Perhaps an example will help to clarify this algorithm. We'll use a scenario where your development team is creating a calculator program that links against a third-party complex number arithmetic library, libcomplex. We'll begin with the initial creation of the vendor branch, and the import of the first vendor drop. We'll call our vendor branch directory `libcomplex`, and our code drops will go into a subdirectory of our vendor branch called `current`. And since **svn import** creates all the intermediate parent directories it needs, we can actually accomplish both of these steps with a single command.

```
$ svn import /path/to/libcomplex-1.0 \
             http://svn.example.com/repos/vendor/libcomplex/current \
             -m 'importing initial 1.0 vendor drop'
…
```

We now have the current version of the libcomplex source code in `/vendor/libcomplex/current`. Now, we tag that version (see “Rótulos”) and then copy it into the main development branch. Our copy will create a new directory called `libcomplex` in our existing `calc` project directory. It is in this copied version of the vendor data that we will make our customizations.

```
$ svn copy http://svn.example.com/repos/vendor/libcomplex/current  \
           http://svn.example.com/repos/vendor/libcomplex/1.0      \
           -m 'tagging libcomplex-1.0'
…
$ svn copy http://svn.example.com/repos/vendor/libcomplex/1.0  \
           http://svn.example.com/repos/calc/libcomplex        \
           -m 'bringing libcomplex-1.0 into the main branch'
…
```

We check out our project's main branch—which now includes a copy of the first vendor drop—and we get to work customizing the libcomplex code. Before we know it, our modified version of libcomplex is now completely integrated into our calculator program. [5]

A few weeks later, the developers of libcomplex release a new version of their library—version 1.1—which contains some features and functionality that we really want. We'd like to upgrade to this new version, but without losing the customizations we made to the existing version. What we essentially would like to do is to replace our current baseline version of libcomplex 1.0 with a copy of libcomplex 1.1, and then re-apply the custom modifications we previously made to that library to the new version. But we actually approach the problem from the other direction, applying the changes made to libcomplex between versions 1.0 and 1.1 to our modified copy of it.

To perform this upgrade, we check out a copy of our vendor branch, and replace the code in the `current` directory with the new libcomplex 1.1 source code. We quite literally copy new files on top of existing files, perhaps exploding the libcomplex 1.1 release tarball atop our existing files and directories. The goal here is to make our `current` directory contain only the libcomplex 1.1 code, and to ensure that all that code is under version control. Oh, and we want to do this with as little version control history disturbance as possible.

After replacing the 1.0 code with 1.1 code, **svn status** will show files with local modifications as well as, perhaps, some unversioned or missing files. If we did what we were supposed to do, the unversioned files

[5] And entirely bug-free, of course!

are only those new files introduced in the 1.1 release of libcomplex—we run **svn add** on those to get them under version control. The missing files are files that were in 1.0 but not in 1.1, and on those paths we run **svn delete**. Finally, once our `current` working copy contains only the libcomplex 1.1 code, we commit the changes we made to get it looking that way.

Our `current` branch now contains the new vendor drop. We tag the new version (in the same way we previously tagged the version 1.0 vendor drop), and then merge the differences between the tag of the previous version and the new current version into our main development branch.

```
$ cd working-copies/calc
$ svn merge http://svn.example.com/repos/vendor/libcomplex/1.0      \
            http://svn.example.com/repos/vendor/libcomplex/current  \
            libcomplex
… # resolve all the conflicts between their changes and our changes
$ svn commit -m 'merging libcomplex-1.1 into the main branch'
…
```

In the trivial use case, the new version of our third-party tool would look, from a files-and-directories point of view, just like the previous version. None of the libcomplex source files would have been deleted, renamed or moved to different locations—the new version would contain only textual modifications against the previous one. In a perfect world, our modifications would apply cleanly to the new version of the library, with absolutely no complications or conflicts.

But things aren't always that simple, and in fact it is quite common for source files to get moved around between releases of software. This complicates the process of ensuring that our modifications are still valid for the new version of code, and can quickly degrade into a situation where we have to manually recreate our customizations in the new version. Once Subversion knows about the history of a given source file—including all its previous locations—the process of merging in the new version of the library is pretty simple. But we are responsible for telling Subversion how the source file layout changed from vendor drop to vendor drop.

svn_load_dirs.pl

Vendor drops that contain more than a few deletes, additions and moves complicate the process of upgrading to each successive version of the third-party data. So Subversion supplies the **svn_load_dirs.pl** script to assist with this process. This script automates the importing steps we mentioned in the general vendor branch management procedure to make sure that mistakes are minimized. You will still be responsible for using the merge commands to merge the new versions of the third-party data into your main development branch, but **svn_load_dirs.pl** can help you more quickly and easily arrive at that stage.

In short, **svn_load_dirs.pl** is an enhancement to **svn import** that has several important characteristics:

- It can be run at any point in time to bring an existing directory in the repository to exactly match an external directory, performing all the necessary adds and deletes, and optionally performing moves, too.
- It takes care of complicated series of operations between which Subversion requires an intermediate commit—such as before renaming a file or directory twice.
- It will optionally tag the newly imported directory.
- It will optionally add arbitrary properties to files and directories that match a regular expression.

svn_load_dirs.pl takes three mandatory arguments. The first argument is the URL to the base Subversion directory to work in. This argument is followed by the URL—relative to the first argument—into which the

current vendor drop will be imported. Finally, the third argument is the local directory to import. Using our previous example, a typical run of **svn_load_dirs.pl** might look like:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

You can indicate that you'd like **svn_load_dirs.pl** to tag the new vendor drop by passing the `-t` command-line option and specifying a tag name. This tag is another URL relative to the first program argument.

```
$ svn_load_dirs.pl -t libcomplex-1.1                              \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

When you run **svn_load_dirs.pl**, it examines the contents of your existing “current” vendor drop, and compares them with the proposed new vendor drop. In the trivial case, there will be no files that are in one version and not the other, and the script will perform the new import without incident. If, however, there are discrepancies in the file layouts between versions, **svn_load_dirs.pl** will ask you how to resolve those differences. For example, you will have the opportunity to tell the script that you know that the file `math.c` in version 1.0 of libcomplex was renamed to `arithmetic.c` in libcomplex 1.1. Any discrepancies not explained by moves are treated as regular additions and deletions.

The script also accepts a separate configuration file for setting properties on files and directories matching a regular expression that are *added* to the repository. This configuration file is specified to **svn_load_dirs.pl** using the `-p` command-line option. Each line of the configuration file is a whitespace-delimited set of two or four values: a Perl-style regular expression to match the added path against, a control keyword (either `break` or `cont`), and then optionally a property name and value.

```
\.png$              break   svn:mime-type   image/png
\.jpe?g$            break   svn:mime-type   image/jpeg
\.m3u$              cont    svn:mime-type   audio/x-mpegurl
\.m3u$              break   svn:eol-style   LF
.*                  break   svn:eol-style   native
```

For each added path, the configured property changes whose regular expression matches the path are applied in order, unless the control specification is `break` (which means that no more property changes should be applied to that path). If the control specification is `cont`—an abbreviation for `continue`—then matching will continue with the next line of the configuration file.

Any whitespace in the regular expression, property name, or property value must be surrounded by either single or double quote characters. You can escape quote characters that are not used for wrapping whitespace by preceding them with a backslash (\) character. The backslash escapes only quotes when parsing the configuration file, so do not protect any other characters beyond what is necessary for the regular expression.

Sumário

Nós cobrimos muito chão nesse capítulo. Nós discutimos conceitos de rótulo*tags* e ramos *branches*, e demonstramos como Subversion implementa estes conceitos através da cópia de diretórios com o

comando **svn copy**. Nós mostramos como usar **svn merge** para copiar mudanças de um ramo *branch* para outro, ou reverter mudanças indesejadas. Nós passamos pelo uso do **svn switch** para criar locais mistos de copias de trabalho. E nós falamos sobre como eles podem gerenciar a organização e vida dos ramos *branches* em um respositório.

Lembre-se do mantra do Subversion: ramos *branches* e rótulos *tags* são baratos. Então use-os livremente! Ao mesmo tempo, não esqueça de usar bons hábitos de fusão *merge*. Copias baratas são úteis apenas quando você é cuidadoso ao rastrear suas fusões*merges*.

Capítulo 5. Administração do Respositório

O respositório Subversion é a central de todos os dados que estão sobre o controle de versão. Assim, ele se transforma num candidato óbivo para receber todo amor e atenção que o administrador pode oferecer. Enquanto o respositório é geralmente um item de manutenção baixa, é importante entender como configurar e cuidar propriamente para que potenciais problemas sejam evitados e eventuais problemas sejam resolvidos de maneira segura.

Neste capítulo, vamos discutir como criar e configura um respositório Subversion. Vamos falar também sobre manutenção, dando exemplos de como e quando usar as ferramentas **svnlook** e **svnadmin** providas pelo Subversion. Vamos apontar alguns questionamentos e erros, e dar algumas sugestões em como organizar seus dados em um repositório.

Se você planeja acessar um respositório Subversion apenas como um usuário cujos dados estão em um controle de versão (que seria via um cliente Subversion), você pode pular esse capítulo todo. Entretanto, se você é, ou deseja se tornar, um administrador de um respositório Subversion, [1] este capítulo é para você.

O Respositório Subversion, Definição

Antes de entrarmos no vasto tópico da administração do repositório, vamos primeiro definir o que é um respositório. Como ele se parece? Como é a sensação? Ele gosta de chá gelado ou quente, doce, e com limão? Como um administrador, será esperado que você entenda a composição de um repositório tanto da perspectiva do Sistema Operacional—como o repositório se parece e comporta em respeito a ferramentas que não são do Subversion—e da perspectiva lógica— lidando com como os dados são representados *dentro* do respositório.

Vendo pelos olhos de um típico browser de arquivos (como o Windows Explorer) ou linha de comando baseado em ferramentas de navegação, o repositório Subversion é apenas outro diretório cheio de coisas. Há alguns sub-diretórios que possuem arquivos de configurção que podem ser lidos por humanos, alguns que são não tão fáceis de serem lidos assim e assim por diante. Como em outras áreas do desenho do Subversion, modularidade tem grande importância, e a organizção hierarquizada é usada pra controlar o caos. Então uma superficial olhada nas partes essenciais é suficiente para revelar os componentes básicos do respositório:

```
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

Aqui está uma pequena pincelada do que exatamente você está vendendo nessa lista do diretório. (Não fique assustado com a terminologia—uma explicação mais detalhada desses componentes existem em algum lugar nesse e em outros capítulos.)

conf
: Diretório contentdo arquivos de configuração do repostiório.

dav
: Diretório onde ficam os arquivos usados pelo mod_dav_svn.

db
: Local onde é armazenado todos os seus dados versionados.

[1] Isto pode soar bem metido ou arrogante, mas nós estomos apenas falando de alguém que tenha interesse no misterioso local por trás das cópias de trabalho onde os dados de todos ficam.

format
: Arquivo que contém um único inteiro que indica o número da versão do respositório.

hooks
: Diretório cheio de modelos de scripts (e scripts, para quando você instalar alguns).

locks
: Diretórios para arquivos travados do Subversion, usado para rastrear acessos ao respositório.

README.txt
: Arquivo que meramente informa a seus leitores que eles estão olhando para um respositório Subversion.

Claro, quando acesso via bibliotecas do Subversion, essa estranha coleção de arquivos e diretórios de repente se torna uma implementação de completo virtual, sistema de arquivos, com customizáveis triggers. Este sistema de arquivo tem suas próprias noções de diretórios e arquivos, muito similar a noções de coisas usadas em sistemas de arquivos reais (como NTFS, FAT32, ext3, e assim por diante). Mas isto é um sistema de arquivos especial—ele controla esses diretórios e arquivos desde revisões, manter todas as mudanças que você fez neles e guarda com segurança para ser acessando quando quiser. É aqui onde todos os seus dados versionados moram.

Strategies for Repository Deployment

Due largely to the simplicity of the overall design of the Subversion repository and the technologies on which it relies, creating and configuring a repository are fairly straightforward tasks. There are a few preliminary decisions you'll want to make, but the actual work involved in any given setup of a Subversion repository is pretty straightforward, tending towards mindless repetition if you find yourself setting up multiples of these things.

Some things you'll want to consider up front, though, are:

- What data do you expect to live in your repository (or repositories), and how will that data be organized?
- Where will your repository live, and how will it be accessed?
- What types of access control and repository event reporting do you need?
- Which of the available types of data store do you want to use?

In this section, we'll try to help you answer those questions.

Planning Your Repository Organization

While Subversion allows you to move around versioned files and directories without any loss of information, and even provides ways of moving whole sets of versioned history from one repository to another, doing so can greatly disrupt the workflow of those who access the repository often and come to expect things to be at certain locations. So before creating a new repository, try to peer into the future a bit; plan ahead before placing your data under version control. By conscientiously “laying out” your repository or repositories and their versioned contents ahead of time, you can prevent many future headaches.

Let's assume that as repository administrator, you will be responsible for supporting the version control system for several projects. Your first decision is whether to use a single repository for multiple projects, or to give each project its own repository, or some compromise of these two.

There are benefits to using a single repository for multiple projects, most obviously the lack of duplicated maintenance. A single repository means that there is one set of hook programs, one thing to routinely

backup, one thing to dump and load if Subversion releases an incompatible new version, and so on. Also, you can move data between projects easily, and without losing any historical versioning information.

The downside of using a single repository is that different projects may have different requirements in terms of the repository event triggers, such as needing to send commit notification emails to different mailing lists, or having different definitions about what does and does not constitute a legitimate commit. These aren't insurmountable problems, of course—it just means that all of your hook scripts have to be sensitive to the layout of your repository rather than assuming that the whole repository is associated with a single group of people. Also, remember that Subversion uses repository-global revision numbers. While those numbers don't have any particular magical powers, some folks still don't like the fact that even though no changes have been made to their project lately, the youngest revision number for the repository keeps climbing because other projects are actively adding new revisions. [2]

A middle-ground approach can be taken, too. For example, projects can be grouped by how well they relate to each other. You might have a few repositories with a handful of projects in each repository. That way, projects that are likely to want to share data can do so easily, and as new revisions are added to the repository, at least the developers know that those new revisions are at least remotely related to everyone who uses that repository.

After deciding how to organize your projects with respect to repositories, you'll probably want to think about directory hierarchies within the repositories themselves. Because Subversion uses regular directory copies for branching and tagging (see Capítulo 4, *Fundir e Ramificar*), the Subversion community recommends that you choose a repository location for each *project root*—the “top-most” directory which contains data related to that project—and then create three subdirectories beneath that root: `trunk`, meaning the directory under which the main project development occurs; `branches`, which is a directory in which to create various named branches of the main development line; `tags`, which is a collection of tree snapshots that are created, and perhaps destroyed, but never changed. [3]

For example, your repository might look like:

```
/
   calc/
      trunk/
      tags/
      branches/
   calendar/
      trunk/
      tags/
      branches/
   spreadsheet/
      trunk/
      tags/
      branches/
   …
```

Note that it doesn't matter where in your repository each project root is. If you have only one project per repository, the logical place to put each project root is at the root of that project's respective repository. If you have multiple projects, you might want to arrange them in groups inside the repository, perhaps putting projects with similar goals or shared code in the same subdirectory, or maybe just grouping them alphabetically. Such an arrangement might look like:

[2] Whether founded in ignorance or in poorly considered concepts about how to derive legitimate software development metrics, global revision numbers are a silly thing to fear, and *not* the kind of thing you should weigh when deciding how to arrange your projects and repositories.

[3] The `trunk`, `tags`, and `branches` trio are sometimes referred to as “the TTB directories”.

```
/
   utils/
      calc/
         trunk/
         tags/
         branches/
      calendar/
         trunk/
         tags/
         branches/
      …
   office/
      spreadsheet/
         trunk/
         tags/
         branches/
      …
```

Lay out your repository in whatever way you see fit. Subversion does not expect or enforce a particular layout—in its eyes, a directory is a directory is a directory. Ultimately, you should choose the repository arrangement that meets the needs of the people who work on the projects that live there.

In the name of full disclosure, though, we'll mention another very common layout. In this layout, the `trunk`, `tags`, and `branches` directories live in the root directory of your repository, and your projects are in subdirectories beneath those, like:

```
/
   trunk/
      calc/
      calendar/
      spreadsheet/
      …
   tags/
      calc/
      calendar/
      spreadsheet/
      …
   branches/
      calc/
      calendar/
      spreadsheet/
      …
```

There's nothing particularly incorrect about such a layout, but it may or may not seem as intuitive for your users. Especially in large, multi-project situations with many users, those users may tend to be familiar with only one or two of the projects in the repository. But the projects-as-branch-siblings tends to de-emphasize project individuality and focus on the entire set of projects as a single entity. That's a social issue though. We like our originally suggested arrangement for purely practical reasons—it's easier to ask about (or modify, or migrate elsewhere) the entire history of a single project when there's a single repository path that holds the entire history—past, present, tagged, and branched—for that project and that project alone.

Deciding Where and How to Host Your Repository

Before creating your Subversion repository, an obvious question you'll need to answer is where the thing is going to live. This is strongly connected to a myriad of other questions involving how the repository will be accessed (via a Subversion server or directly), by whom (users behind your corporate firewall or the whole world out on the open Internet), what other services you'll be providing around Subversion (repository browsing interfaces, e-mail based commit notification, etc.), your data backup strategy, and so on.

We cover server choice and configuration in Capítulo 6, *Configuração do Servidor*, but the point we'd like to briefly make here is simply that the answers to some of these other questions might have implications that force your hand when deciding where your repository will live. For example, certain deployment scenarios might require accessing the repository via a remote filesystem from multiple computers, in which case (as you'll read in the next section) your choice of a repository back-end data store turns out not to be a choice at all because only one of the available back-ends will work in this scenario.

Addressing each possible way to deploy Subversion is both impossible, and outside the scope of this book. We simply encourage you to evaluate your options using these pages and other sources as your reference material, and plan ahead.

Choosing a Data Store

As of version 1.1, Subversion provides two options for the type of underlying data store—often referred to as “the back-end” or, somewhat confusingly, “the (versioned) filesystem”—that each repository uses. One type of data store keeps everything in a Berkeley DB (or BDB) database environment; repositories that use this type are often referred to as being “BDB-backed”. The other type stores data in ordinary flat files, using a custom format. Subversion developers have adopted the habit of referring to this latter data storage mechanism as *FSFS* [4] —a versioned filesystem implementation that uses the native OS filesystem directly —rather than via a database library or some other abstraction layer—to store data.

Tabela 5.1, “Repository Data Store Comparison” gives a comparative overview of Berkeley DB and FSFS repositories.

[4] Often pronounced “fuzz-fuzz”, if Jack Repenning has anything to say about it. (This book, however, assumes that the reader is thinking “eff-ess-eff-ess”.)

Tabela 5.1. Repository Data Store Comparison

Category	Feature	Berkeley DB	FSFS
Reliability	Data integrity	when properly deployed, extremely reliable; Berkeley DB 4.4 brings auto-recovery	older versions had some rarely demonstrated, but data-destroying bugs
	Sensitivity to interruptions	very; crashes and permission problems can leave the database “wedged”, requiring journaled recovery procedures	quite insensitive
Accessibility	Usable from a read-only mount	no	yes
	Platform-independent storage	no	yes
	Usable over network filesystems	generally, no	yes
	Group permissions handling	sensitive to user umask problems; best if accessed by only one user	works around umask problems
Scalability	Repository disk usage	larger (especially if logfiles aren't purged)	smaller
	Number of revision trees	database; no problems	some older native filesystems don't scale well with thousands of entries in a single directory
	Directories with many files	slower	faster
Performance	Checking out latest revision	no meaningful difference	no meaningful difference
	Large commits	slower overall, but cost is amortized across the lifetime of the commit	faster overall, but finalization delay may cause client timeouts

There are advantages and disadvantages to each of these two back-end types. Neither of them is more “official” than the other, though the newer FSFS is the default data store as of Subversion 1.2. Both are reliable enough to trust with your versioned data. But as you can see in Tabela 5.1, “Repository Data Store Comparison”, the FSFS backend provides quite a bit more flexibility in terms of its supported deployment scenarios. More flexibility means you have to work a little harder to find ways to deploy it incorrectly. Those reasons—plus the fact that not using Berkeley DB means there's one fewer component in the system—largely explain why today almost everyone uses the FSFS backend when creating new repositories.

Fortunately, most programs which access Subversion repositories are blissfully ignorant of which back-end data store is in use. And you aren't even necessarily stuck with your first choice of a data store—in the event that you change your mind later, Subversion provides ways of migrating your repository's data into another repository that uses a different back-end data store. We talk more about that later in this chapter.

The following subsections provide a more detailed look at the available data store types.

Berkeley DB

When the initial design phase of Subversion was in progress, the developers decided to use Berkeley DB for a variety of reasons, including its open-source license, transaction support, reliability, performance, API simplicity, thread-safety, support for cursors, and so on.

Berkeley DB provides real transaction support—perhaps its most powerful feature. Multiple processes accessing your Subversion repositories don't have to worry about accidentally clobbering each other's data. The isolation provided by the transaction system is such that for any given operation, the Subversion repository code sees a static view of the database—not a database that is constantly changing at the hand of some other process—and can make decisions based on that view. If the decision made happens to conflict with what another process is doing, the entire operation is rolled back as if it never happened, and Subversion gracefully retries the operation against a new, updated (and yet still static) view of the database.

Another great feature of Berkeley DB is *hot backups*—the ability to backup the database environment without taking it “offline”. We'll discuss how to backup your repository in “Repository Backup”, but the benefits of being able to make fully functional copies of your repositories without any downtime should be obvious.

Berkeley DB is also a very reliable database system when properly used. Subversion uses Berkeley DB's logging facilities, which means that the database first writes to on-disk log files a description of any modifications it is about to make, and then makes the modification itself. This is to ensure that if anything goes wrong, the database system can back up to a previous *checkpoint*—a location in the log files known not to be corrupt—and replay transactions until the data is restored to a usable state. See “Managing Disk Space” for more about Berkeley DB log files.

But every rose has its thorn, and so we must note some known limitations of Berkeley DB. First, Berkeley DB environments are not portable. You cannot simply copy a Subversion repository that was created on a Unix system onto a Windows system and expect it to work. While much of the Berkeley DB database format is architecture independent, there are other aspects of the environment that are not. Secondly, Subversion uses Berkeley DB in a way that will not operate on Windows 95/98 systems—if you need to house a BDB-backed repository on a Windows machine, stick with Windows 2000 or newer.

While Berkeley DB promises to behave correctly on network shares that meet a particular set of specifications, [5] most networked filesystem types and appliances do *not* actually meet those requirements. And in no case can you allow a BDB-backed repository that resides on a network share to be accessed by multiple clients of that share at once (which quite often is the whole point of having the repository live on a network share in the first place).

If you attempt to use Berkeley DB on a non-compliant remote filesystem, the results are unpredictable—you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted. You should strongly consider using the FSFS data store for repositories that need to live on a network share.

Finally, because Berkeley DB is a library linked directly into Subversion, it's more sensitive to interruptions than a typical relational database system. Most SQL systems, for example, have a dedicated server process that mediates all access to tables. If a program accessing the database crashes for some reason, the database daemon notices the lost connection and cleans up any mess left behind. And because the database daemon is the only process accessing the tables, applications don't need to worry about permission conflicts. These things are not the case with Berkeley DB, however. Subversion (and programs

[5] Berkeley DB requires that the underlying filesystem implement strict POSIX locking semantics, and more importantly, the ability to map files directly into process memory.

using Subversion libraries) access the database tables directly, which means that a program crash can leave the database in a temporarily inconsistent, inaccessible state. When this happens, an administrator needs to ask Berkeley DB to restore to a checkpoint, which is a bit of an annoyance. Other things can cause a repository to “wedge” besides crashed processes, such as programs conflicting over ownership and permissions on the database files.

Berkeley DB 4.4 brings (to Subversion 1.4 and better) the ability for Subversion to automatically and transparently recover Berkeley DB environments in need of such recovery. When a Subversion process attaches to a repository's Berkeley DB environment, it uses some process accounting mechanisms to detect any unclean disconnections by previous processes, performs any necessary recovery, and then continues on as if nothing happened. This doesn't completely eliminate instances of repository wedging, but it does drastically reduce the amount of human interaction required to recover from them.

So while a Berkeley DB repository is quite fast and scalable, it's best used by a single server process running as one user—such as Apache's **httpd** or **svnserve** (see Capítulo 6, *Configuração do Servidor*)—rather than accessing it as many different users via `file://` or `svn+ssh://` URLs. If using a Berkeley DB repository directly as multiple users, be sure to read “Supporting Multiple Repository Access Methods”.

FSFS

In mid-2004, a second type of repository storage system—one which doesn't use a database at all—came into being. An FSFS repository stores the changes associated with a revision in a single file, and so all of a repository's revisions can be found in a single subdirectory full of numbered files. Transactions are created in separate subdirectories as individual files. When complete, the transaction file is renamed and moved into the revisions directory, thus guaranteeing that commits are atomic. And because a revision file is permanent and unchanging, the repository also can be backed up while “hot”, just like a BDB-backed repository.

The FSFS revision files describe a revision's directory structure, file contents, and deltas against files in other revision trees. Unlike a Berkeley DB database, this storage format is portable across different operating systems and isn't sensitive to CPU architecture. Because there's no journaling or shared-memory files being used, the repository can be safely accessed over a network filesystem and examined in a read-only environment. The lack of database overhead also means that the overall repository size is a bit smaller.

FSFS has different performance characteristics too. When committing a directory with a huge number of files, FSFS is able to more quickly append directory entries. On the other hand, FSFS writes the latest version of a file as a delta against an earlier version, which means that checking out the latest tree is a bit slower than fetching the fulltexts stored in a Berkeley DB HEAD revision. FSFS also has a longer delay when finalizing a commit, which could in extreme cases cause clients to time out while waiting for a response.

The most important distinction, however, is FSFS's imperviousness to “wedging” when something goes wrong. If a process using a Berkeley DB database runs into a permissions problem or suddenly crashes, the database can be left in an unusable state until an administrator recovers it. If the same scenarios happen to a process using an FSFS repository, the repository isn't affected at all. At worst, some transaction data is left behind.

The only real argument against FSFS is its relative immaturity compared to Berkeley DB. Unlike Berkeley DB, which has years of history, its own dedicated development team and, now, Oracle's mighty name attached to it, [6] FSFS is a much newer bit of engineering. Prior to Subversion 1.4, it was still shaking out some pretty serious data integrity bugs which, while only triggered in very rare cases, nonetheless did occur. That said, FSFS has quickly become the back-end of choice for some of the largest public and private Subversion repositories, and promises a lower barrier to entry for Subversion across the board.

[6] Oracle bought Sleepycat and its flagship software, Berkeley DB, on Valentine's Day in 2006.

Creating and Configuring Your Repository

In “Strategies for Repository Deployment”, we looked at some of the important decisions that should be made before creating and configuring your Subversion repository. Now, we finally get to get our hands dirty! In this section, we'll see how to actually create a Subversion repository and configure it to perform custom actions when special repository events occur.

Creating the Repository

Subversion repository creation is an incredibly simple task. The **svnadmin** utility that comes with Subversion provides a subcommand (`create`) for doing just that.

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

This creates a new repository in the directory `/path/to/repos`, and with the default filesystem data store. Prior to Subversion 1.2, the default was to use Berkeley DB; the default is now FSFS. You can explicitly choose the filesystem type using the `--fs-type` argument, which accepts as a parameter either `fsfs` or `bdb`.

```
$ # Create an FSFS-backed repository
$ svnadmin create --fs-type fsfs /path/to/repos
$


# Create a Berkeley-DB-backed repository
$ svnadmin create --fs-type bdb /path/to/repos
$
```

After running this simple command, you have a Subversion repository.

The path argument to **svnadmin** is just a regular filesystem path and not a URL like the **svn** client program uses when referring to repositories. Both **svnadmin** and **svnlook** are considered server-side utilities—they are used on the machine where the repository resides to examine or modify aspects of the repository, and are in fact unable to perform tasks across a network. A common mistake made by Subversion newcomers is trying to pass URLs (even “local” `file://` ones) to these two programs.

Present in the `db/` subdirectory of your repository is the implementation of the versioned filesystem. Your new repository's versioned filesystem begins life at revision 0, which is defined to consist of nothing but the top-level root (`/`) directory. Initially, revision 0 also has a single revision property, `svn:date`, set to the time at which the repository was created.

Now that you have a repository, it's time to customize it.

While some parts of a Subversion repository—such as the configuration files and hook scripts—are meant to be examined and modified manually, you shouldn't (and shouldn't need to) tamper with the other parts of the repository “by hand”. The **svnadmin** tool should be sufficient for any changes necessary to your repository, or you can look to third-party tools (such as Berkeley DB's tool suite) for tweaking relevant subsections of the repository. Do *not* attempt manual manipulation of your version control history by poking and prodding around in your repository's data store files!

Implementing Repository Hooks

A *hook* is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Some hooks (the so-called “pre hooks”) run in advance of a repository operation and provide a means by which to both report what is about to happen and to prevent it from happening at all. Other hooks (the “post hooks”) run after the completion of a repository event, and are useful for performing tasks that examine—but don't modify—the repository. Each hook is handed enough information to tell what that event is (or was), the specific repository changes proposed (or completed), and the username of the person who triggered the event.

The `hooks` subdirectory is, by default, filled with templates for various repository hooks.

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

There is one template for each hook that the Subversion repository supports, and by examining the contents of those template scripts, you can see what triggers each script to run and what data is passed to that script. Also present in many of these templates are examples of how one might use that script, in conjunction with other Subversion-supplied programs, to perform common useful tasks. To actually install a working hook, you need only place some executable program or script into the `repos/hooks` directory which can be executed as the name (like **start-commit** or **post-commit**) of the hook.

On Unix platforms, this means supplying a script or program (which could be a shell script, a Python program, a compiled C binary, or any number of other things) named exactly like the name of the hook. Of course, the template files are present for more than just informational purposes—the easiest way to install a hook on Unix platforms is to simply copy the appropriate template file to a new file that lacks the `.tmpl` extension, customize the hook's contents, and ensure that the script is executable. Windows, however, uses file extensions to determine whether or not a program is executable, so you would need to supply a program whose basename is the name of the hook, and whose extension is one of the special extensions recognized by Windows for executable programs, such as `.exe` for programs, and `.bat` for batch files.

For security reasons, the Subversion repository executes hook programs with an empty environment—that is, no environment variables are set at all, not even `$PATH` (or `%PATH%`, under Windows). Because of this, many administrators are baffled when their hook program runs fine by hand, but doesn't work when run by Subversion. Be sure to explicitly set any necessary environment variables in your hook program and/or use absolute paths to programs.

Subversion executes hooks as the same user who owns the process which is accessing the Subversion repository. In most cases, the repository is being accessed via a Subversion server, so this user is the same user as which that server runs on the system. The hooks themselves will need to be configured with OS-level permissions that allow that user to execute them. Also, this means that any file or programs (including the Subversion repository itself) accessed directly or indirectly by the hook will be accessed as the same user. In other words, be alert to potential permission-related problems that could prevent the hook from performing the tasks it is designed to perform.

There are nine hooks implemented by the Subversion repository, and you can get details about each of them in “Repository Hooks”. As a repository administrator, you'll need to decide which of hooks you wish to implement (by way of providing an appropriately named and permissioned hook program), and how. When you make this decision, keep in mind the big picture of how your repository is deployed. For example, if you are using server configuration to determine which users are permitted to commit changes to your repository, then you don't need to do this sort of access control via the hook system.

There is no shortage of Subversion hook programs and scripts freely available either from the Subversion community itself or elsewhere. These scripts cover a wide range of utility—basic access control, policy adherence checking, issue tracker integration, email- or syndication-based commit notification, and beyond. See Apêndice D, *Third Party Tools* for discussion of some of the most commonly used hook programs. Or, if you wish to write your own, see Capítulo 8, *Incorporando o Subversion*.

While hook scripts can do almost anything, there is one dimension in which hook script authors should show restraint: do *not* modify a commit transaction using hook scripts. While it might be tempting to use hook scripts to automatically correct errors or shortcomings or policy violations present in the files being committed, doing so can cause problems. Subversion keeps client-side caches of certain bits of repository data, and if you change a commit transaction in this way, those caches become indetectably stale. This inconsistency can lead to surprising and unexpected behavior. Instead of modifying the transaction, you should simply *validate* the transaction in the `pre-commit` hook and reject the commit if it does not meet the desired requirements. As a bonus, your users will learn the value of careful, compliance-minded work habits.

Berkeley DB Configuration

A Berkeley DB environment is an encapsulation of one or more databases, log files, region files and configuration files. The Berkeley DB environment has its own set of default configuration values for things like the number of database locks allowed to be taken out at any given time, or the maximum size of the journaling log files, etc. Subversion's filesystem logic additionally chooses default values for some of the Berkeley DB configuration options. However, sometimes your particular repository, with its unique collection of data and access patterns, might require a different set of configuration option values.

The producers of Berkeley DB understand that different applications and database environments have different requirements, and so they have provided a mechanism for overriding at runtime many of the configuration values for the Berkeley DB environment: BDB checks for the presence of a file named `DB_CONFIG` in the environment directory (namely, the repository's `db` subdirectory), and parses the options found in that file . Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file, and makes no direct use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is at odds with what Subversion might expect. Also, changes made to `DB_CONFIG` won't take effect until you recover the database environment (using **svnadmin recover**).

Repository Maintenance

Maintaining a Subversion repository can be daunting, mostly due to the complexities inherent in systems which have a database backend. Doing the task well is all about knowing the tools—what they are, when to use them, and how to use them. This section will introduce you to the repository administration tools provided by Subversion, and how to wield them to accomplish tasks such as repository data migration, upgrades, backups and cleanups.

An Administrator's Toolkit

Subversion provides a handful of utilities useful for creating, inspecting, modifying and repairing your repository. Let's look more closely at each of those tools. Afterward, we'll briefly examine some of the utilities included in the Berkeley DB distribution that provide functionality specific to your repository's database backend not otherwise provided by Subversion's own tools.

svnadmin

The **svnadmin** program is the repository administrator's best friend. Besides providing the ability to create Subversion repositories, this program allows you to perform several maintenance operations on those repositories. The syntax of **svnadmin** is similar to that of other Subversion command-line programs:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type 'svnadmin help <subcommand>' for help on a specific subcommand.
Type 'svnadmin --version' to see the program version and FS modules.

Available subcommands:
   crashtest
   create
   deltify
…
```

We've already mentioned **svnadmin**'s `create` subcommand (see “Creating the Repository”). Most of the others we will cover later in this chapter. And you can consult “**svnadmin**” for a full rundown of subcommands and what each of them offers.

svnlook

svnlook is a tool provided by Subversion for examining the various revisions and *transactions* (which are revisions in-the-making) in a repository. No part of this program attempts to change the repository. **svnlook** is typically used by the repository hooks for reporting the changes that are about to be committed (in the case of the **pre-commit** hook) or that were just committed (in the case of the **post-commit** hook) to the repository. A repository administrator may use this tool for diagnostic purposes.

svnlook has a straightforward syntax:

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

Nearly every one of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision (-r)` and `--transaction (-t)` options to specify which revision or transaction, respectively, to examine. In the absence of both the `--revision (-r)` and `--transaction (-t)` options, **svnlook** will examine the youngest (or “HEAD”) revision in the repository. So the following two commands do exactly the same thing when 19 is the youngest revision in the repository located at `/path/to/repos`:

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

The only exception to these rules about subcommands is the **svnlook youngest** subcommand, which takes no options, and simply prints out the repository's youngest revision number.

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

Keep in mind that the only transactions you can browse are uncommitted ones. Most repositories will have no such transactions, because transactions are usually either committed (in which case, you should access them as revision with the `--revision (-r)` option) or aborted and removed.

Output from **svnlook** is designed to be both human- and machine-parsable. Take as an example the output of the `info` subcommand:

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

The output of the `info` subcommand is defined as:

1. The author, followed by a newline.

2. The date, followed by a newline.

3. The number of characters in the log message, followed by a newline.

4. The log message itself, followed by a newline.

This output is human-readable, meaning items like the datestamp are displayed using a textual representation instead of something more obscure (such as the number of nanoseconds since the Tasty Freeze guy drove by). But the output is also machine-parsable—because the log message can contain multiple lines and be unbounded in length, **svnlook** provides the length of that message before the message itself. This allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

svnlook can perform a variety of other queries: displaying subsets of bits of information we've mentioned previously, recursively listing versioned directory trees, reporting which paths were modified in a given revision or transaction, showing textual and property differences made to files and directories, and so on. See “**svnlook**” for a full reference of **svnlook**'s features.

svndumpfilter

While it won't be the most commonly used tool at the administrator's disposal, **svndumpfilter** provides a very particular brand of useful functionality—the ability to quickly and easily modify streams of Subversion repository history data by acting as a path-based filter.

The syntax of **svndumpfilter** is as follows:

```
$ svndumpfilter help
general usage: svndumpfilter SUBCOMMAND [ARGS & OPTIONS ...]
Type "svndumpfilter help <subcommand>" for help on a specific subcommand.
Type 'svndumpfilter --version' to see the program version.
```

```
Available subcommands:
   exclude
   include
   help (?, h)
```

There are only two interesting subcommands. They allow you to make the choice between explicit or implicit inclusion of paths in the stream:

`exclude`
: Filter out a set of paths from the dump data stream.

`include`
: Allow only the requested set of paths to pass through the dump data stream.

You can learn more about these subcommands and **svndumpfilter**'s unique purpose in “Filtering Repository History”.

svnsync

The **svnsync** program, which is new to the 1.4 release of Subversion, provides all the functionality required for maintaining a read-only mirror of a Subversion repository. The program really has one job—to transfer one repository's versioned history into another repository. And while there are few ways to do that, its primary strength is that it can operate remotely—the “source” and “sink” [7] repositories may be on different computers from each other and from **svnsync** itself.

As you might expect, **svnsync** has a syntax that looks very much like every other program we've mentioned in this chapter:

```
$ svnsync help
general usage: svnsync SUBCOMMAND DEST_URL  [ARGS & OPTIONS ...]
Type 'svnsync help <subcommand>' for help on a specific subcommand.
Type 'svnsync --version' to see the program version and RA modules.

Available subcommands:
   initialize (init)
   synchronize (sync)
   copy-revprops
   help (?, h)
$
```

We talk more about replication repositories with **svnsync** in “Repository Replication”.

Berkeley DB Utilities

If you're using a Berkeley DB repository, then all of your versioned filesystem's structure and data live in a set of database tables within the `db/` subdirectory of your repository. This subdirectory is a regular Berkeley DB environment directory, and can therefore be used in conjunction with any of the Berkeley database tools, typically provided as part of the Berkeley DB distribution.

For day-to-day Subversion use, these tools are unnecessary. Most of the functionality typically needed for Subversion repositories has been duplicated in the **svnadmin** tool. For example, **svnadmin list-unused-**

[7] Or is that, the “sync”?

dblogs and **svnadmin list-dblogs** perform a subset of what is provided by the Berkeley **db_archive** command, and **svnadmin recover** reflects the common use cases of the **db_recover** utility.

However, there are still a few Berkeley DB utilities that you might find useful. The **db_dump** and **db_load** programs write and read, respectively, a custom file format which describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system. As we describe later in this chapter, you can also use **svnadmin dump** and **svnadmin load** for similar purposes, but **db_dump** and **db_load** can do certain jobs just as well and much faster. They can also be useful if the experienced Berkeley DB hacker needs to do in-place tweaking of the data in a BDB-backed repository for some reason, which is something Subversion's utilities won't allow. Also, the **db_stat** utility can provide useful information about the status of your Berkeley DB environment, including detailed statistics about the locking and storage subsystems.

For more information on the Berkeley DB tool chain, visit the documentation section of the Berkeley DB section of Oracle's website, located at http://www.oracle.com/technology/documentation/berkeley-db/db/.

Commit Log Message Correction

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` hook; see “Implementing Repository Hooks”) to accept changes to this log message after the commit is finished, then the user can “fix” her log message remotely using the **svn** program's `propset` command (see svn propset). However, because of the potential to lose information forever, Subversion repositories are not, by default, configured to allow changes to unversioned properties—except by an administrator.

If a log message needs to be changed by an administrator, this can be done using **svnadmin setlog**. This command changes the log message (the `svn:log` property) on a given revision of a repository, reading the new value from a provided file.

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

The **svnadmin setlog** command, by default, is still bound by the same protections against modifying unversioned properties as a remote client is—the `pre-` and `post-revprop-change` hooks are still triggered, and therefore must be set up to accept changes of this nature. But an administrator can get around these protections by passing the `--bypass-hooks` option to the **svnadmin setlog** command.

Remember, though, that by bypassing the hooks, you are likely avoiding such things as email notifications of property changes, backup systems which track unversioned property changes, and so on. In other words, be very careful about what you are changing, and how you change it.

Managing Disk Space

While the cost of storage has dropped incredibly in the past few years, disk usage is still a valid concern for administrators seeking to version large amounts of data. Every bit of version history information stored in the live repository needs to be backed up elsewhere, perhaps multiple times as part of rotating backup schedules. It is useful to know what pieces of Subversion's repository data need to remain on the live site, which need to be backed up, and which can be safely removed.

How Subversion saves disk space

To keep the repository small, Subversion uses *deltification* (or, “deltified storage”) within the repository itself. Deltification involves encoding the representation of a chunk of data as a collection of differences against

some other chunk of data. If the two pieces of data are very similar, this deltification results in storage savings for the deltified chunk—rather than taking up space equal to the size of the original data, it takes up only enough space to say, “I look just like this other piece of data over here, except for the following couple of changes”. The result is that most of the repository data that tends to be bulky—namely, the contents of versioned files—is stored at a much smaller size than the original “fulltext” representation of that data. And for repositories created with Subversion 1.4 or later, the space savings are even better—now those fulltext representations of file contents are themselves compressed.

Because all of the data that is subject to deltification in a BDB-backed repository is stored in a single Berkeley DB database file, reducing the size of the stored values will not immediately reduce the size of the database file itself. Berkeley DB will, however, keep internal records of unused areas of the database file, and consume those areas first before growing the size of the database file. So while deltification doesn't produce immediate space savings, it can drastically slow future growth of the database.

Removing dead transactions

Though they are uncommon, there are circumstances in which a Subversion commit process might fail, leaving behind in the repository the remnants of the revision-to-be that wasn't—an uncommitted transaction and all the file and directory changes associated with it. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure occurred in the middle of an operation. Regardless of the reason, dead transactions can happen. They don't do any real harm, other than consuming disk space. A fastidious administrator may nonetheless wish to remove them.

You can use **svnadmin**'s `lstxns` command to list the names of the currently outstanding transactions.

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

Each item in the resultant output can then be used with **svnlook** (and its `--transaction (-t)` option) to determine who created the transaction, when it was created, what types of changes were made in the transaction—information that is helpful in determining whether or not the transaction is a safe candidate for removal! If you do indeed want to remove a transaction, its name can be passed to **svnadmin rmtxns**, which will perform the cleanup of the transaction. In fact, the `rmtxns` subcommand can take its input directly from the output of `lstxns`!

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

If you use these two subcommands like this, you should consider making your repository temporarily inaccessible to clients. That way, no one can begin a legitimate transaction before you start your cleanup. Exemplo 5.1, “txn-info.sh (Reporting Outstanding Transactions)” contains a bit of shell-scripting that can quickly generate information about each outstanding transaction in your repository.

Exemplo 5.1. txn-info.sh (Reporting Outstanding Transactions)

```
#!/bin/sh

### Generate informational output for all outstanding transactions in
### a Subversion repository.

REPOS="${1}"
if [ "x$REPOS" = x ] ; then
  echo "usage: $0 REPOS_PATH"
  exit
fi

for TXN in `svnadmin lstxns ${REPOS}`; do
  echo "---[ Transaction ${TXN} ]-------------------------------------------"
  svnlook info "${REPOS}" -t "${TXN}"
done
```

The output of the script is basically a concatenation of several chunks of **svnlook info** output (see “svnlook”), and will look something like:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
39
Trying to commit over a faulty network.
---[ Transaction a45 ]-------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

A long-abandoned transaction usually represents some sort of failed or interrupted commit. A transaction's datestamp can provide interesting information—for example, how likely is it that an operation begun nine months ago is still active?

In short, transaction cleanup decisions need not be made unwisely. Various sources of information—including Apache's error and access logs, Subversion's operational logs, Subversion revision history, and so on—can be employed in the decision-making process. And of course, an administrator can often simply communicate with a seemingly dead transaction's owner (via email, for example) to verify that the transaction is, in fact, in a zombie state.

Purging unused Berkeley DB logfiles

Until recently, the largest offender of disk space usage with respect to BDB-backed Subversion repositories was the log files in which Berkeley DB performs its pre-writes before modifying the actual database files. These files capture all the actions taken along the route of changing the database from one state to another

—while the database files, at any given time, reflect a particular state, the log files contain all the many changes along the way *between* states. Thus, they can grow and accumulate quite rapidly.

Fortunately, beginning with the 4.2 release of Berkeley DB, the database environment has the ability to remove its own unused log files automatically. Any repositories created using an **svnadmin** which is compiled against Berkeley DB version 4.2 or greater will be configured for this automatic log file removal. If you don't want this feature enabled, simply pass the `--bdb-log-keep` option to the **svnadmin create** command. If you forget to do this, or change your mind at a later time, simply edit the `DB_CONFIG` file found in your repository's db directory, comment out the line which contains the `set_flags DB_LOG_AUTOREMOVE` directive, and then run **svnadmin recover** on your repository to force the configuration changes to take effect. See “Berkeley DB Configuration” for more information about database configuration.

Without some sort of automatic log file removal in place, log files will accumulate as you use your repository. This is actually somewhat of a feature of the database system—you should be able to recreate your entire database using nothing but the log files, so these files can be useful for catastrophic database recovery. But typically, you'll want to archive the log files that are no longer in use by Berkeley DB, and then remove them from disk to conserve space. Use the **svnadmin list-unused-dblogs** command to list the unused log files:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033
…
$ rm `svnadmin list-unused-dblogs /path/to/repos`
## disk space reclaimed!
```

BDB-backed repositories whose log files are used as part of a backup or disaster recovery plan should *not* make use of the log file autoremoval feature. Reconstruction of a repository's data from log files can only be accomplished when *all* the log files are available. If some of the log files are removed from disk before the backup system has a chance to copy them elsewhere, the incomplete set of backed-up log files is essentially useless.

Berkeley DB Recovery

As mentioned in “Berkeley DB”, a Berkeley DB repository can sometimes be left in frozen state if not closed properly. When this happens, an administrator needs to rewind the database back into a consistent state. This is unique to BDB-backed repositories, though—if you are using FSFS-backed ones instead, this won't apply to you. And for those of you using Subversion 1.4 with Berkeley DB 4.4 or better, you should find that Subversion has become much more resilient in these types of situations. Still, wedged Berkeley DB repositories do occur, and an administrator needs to know how to safely deal with this circumstance.

In order to protect the data in your repository, Berkeley DB uses a locking mechanism. This mechanism ensures that portions of the database are not simultaneously modified by multiple database accessors, and that each process sees the data in the correct state when that data is being read from the database. When a process needs to change something in the database, it first checks for the existence of a lock on the target data. If the data is not locked, the process locks the data, makes the change it wants to make, and then unlocks the data. Other processes are forced to wait until that lock is removed before they are permitted to continue accessing that section of the database. (This has nothing to do with the locks that you, as a user, can apply to versioned files within the repository; we try to clear up the confusion caused by this terminology collision in Os três significados de “trava”.)

In the course of using your Subversion repository, fatal errors or interruptions can prevent a process from having the chance to remove the locks it has placed in the database. The result is that the back-end database

system gets “wedged”. When this happens, any attempts to access the repository hang indefinitely (since each new accessor is waiting for a lock to go away—which isn't going to happen).

If this happens to your repository, don't panic. The Berkeley DB filesystem takes advantage of database transactions and checkpoints and pre-write journaling to ensure that only the most catastrophic of events [8] can permanently destroy a database environment. A sufficiently paranoid repository administrator will have made off-site backups of the repository data in some fashion, but don't head off to the tape backup storage closet just yet.

Instead, use the following recipe to attempt to “unwedge” your repository:

1. Make sure that there are no processes accessing (or attempting to access) the repository. For networked repositories, this means shutting down the Apache HTTP Server or svnserve daemon, too.

2. Become the user who owns and manages the repository. This is important, as recovering a repository while running as the wrong user can tweak the permissions of the repository's files in such a way that your repository will still be inaccessible even after it is “unwedged”.

3. Run the command **svnadmin recover /path/to/repos**. You should see output like this:

   ```
   Repository lock acquired.
   Please wait; recovering the repository may take some time...

   Recovery completed.
   The latest repos revision is 19.
   ```

   This command may take many minutes to complete.

4. Restart the server process.

This procedure fixes almost every case of repository lock-up. Make sure that you run this command as the user that owns and manages the database, not just as `root`. Part of the recovery process might involve recreating from scratch various database files (shared memory regions, for example). Recovering as `root` will create those files such that they are owned by `root`, which means that even after you restore connectivity to your repository, regular users will be unable to access it.

If the previous procedure, for some reason, does not successfully unwedge your repository, you should do two things. First, move your broken repository directory aside (perhaps by renaming it to something like `repos.BROKEN`) and then restore your latest backup of it. Then, send an email to the Subversion user list (at <`users@subversion.tigris.org`>) describing your problem in detail. Data integrity is an extremely high priority to the Subversion developers.

Migrating Repository Data Elsewhere

A Subversion filesystem has its data spread throughout files in the repository, in a fashion generally understood by (and of interest to) only the Subversion developers themselves. However, circumstances may arise that call for all, or some subset, of that data to be copied or moved into another repository.

Subversion provides such functionality by way of repository dump streams. A repository dump stream (often referred to as a “dumpfile” when stored as a file on disk) is a portable, flat file format that describes the various revisions in your repository—what was changed, by whom, when, and so on. This dump stream is

[8] E.g.: hard drive + huge electromagnet = disaster.

the primary mechanism used to marshal versioned history—in whole or in part, with or without modification—between repositories. And Subversion provides the tools necessary for creating and loading these dump streams—the **svnadmin dump** and **svnadmin load** subcommands, respectively.

While the Subversion repository dump format contains human-readable portions and a familiar structure (it resembles an RFC-822 format, the same type of format used for most email), it is *not* a plaintext file format. It is a binary file format, highly sensitive to meddling. For example, many text editors will corrupt the file by automatically converting line endings.

There are many reasons for dumping and loading Subversion repository data. Early in Subversion's life, the most common reason was due to the evolution of Subversion itself. As Subversion matured, there were times when changes made to the back-end database schema caused compatibility issues with previous versions of the repository, so users had to dump their repository data using the previous version of Subversion, and load it into a freshly created repository with the new version of Subversion. Now, these types of schema changes haven't occurred since Subversion's 1.0 release, and the Subversion developers promise not to force users to dump and load their repositories when upgrading between minor versions (such as from 1.3 to 1.4) of Subversion. But there are still other reasons for dumping and loading, including re-deploying a Berkeley DB repository on a new OS or CPU architecture, switching between the Berkeley DB and FSFS back-ends, or (as we'll cover in “Filtering Repository History”) purging versioned data from repository history.

Whatever your reason for migrating repository history, using the **svnadmin dump** and **svnadmin load** subcommands is straightforward. **svnadmin dump** will output a range of repository revisions that are formatted using Subversion's custom filesystem dump format. The dump format is printed to the standard output stream, while informative messages are printed to the standard error stream. This allows you to redirect the output stream to a file while watching the status output in your terminal window. For example:

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

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions. Note that **svnadmin dump** is reading revision trees from the repository just like any other “reader” process would (**svn checkout**, for example), so it's safe to run this command at any time.

The other subcommand in the pair, **svnadmin load**, parses the standard input stream as a Subversion repository dump file, and effectively replays those dumped revisions into the target repository for that operation. It also gives informative feedback, this time using the standard output stream:

```
$ svnadmin load newrepos < dumpfile
<<< Started new txn, based on original revision 1
     * adding path : A ... done.
     * adding path : A/B ... done.
     …
------- Committed new rev 1 (loaded from original rev 1) >>>
```

```
<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.

------- Committed new rev 2 (loaded from original rev 2) >>>

…

<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

The result of a load is new revisions added to a repository—the same thing you get by making commits against that repository from a regular Subversion client. And just as in a commit, you can use hook programs to perform actions before and after each of the commits made during a load process. By passing the `--use-pre-commit-hook` and `--use-post-commit-hook` options to **svnadmin load**, you can instruct Subversion to execute the pre-commit and post-commit hook programs, respectively, for each loaded revision. You might use these, for example, to ensure that loaded revisions pass through the same validation steps that regular commits pass through. Of course, you should use these options with care—if your post-commit hook sends emails to a mailing list for each new commit, you might not want to spew hundreds or thousands of commit emails in rapid succession at that list! You can read more about the use of hook scripts in “Implementing Repository Hooks”.

Note that because **svnadmin** uses standard input and output streams for the repository dump and load process, people who are feeling especially saucy can try things like this (perhaps even using different versions of **svnadmin** on each side of the pipe):

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

By default, the dump file will be quite large—much larger than the repository itself. That's because by default every version of every file is expressed as a full text in the dump file. This is the fastest and simplest behavior, and nice if you're piping the dump data directly into some other process (such as a compression program, filtering program, or into a loading process). But if you're creating a dump file for longer-term storage, you'll likely want to save disk space by using the `--deltas` option. With this option, successive revisions of files will be output as compressed, binary differences—just as file revisions are stored in a repository. This option is slower, but results in a dump file much closer in size to the original repository.

We mentioned previously that **svnadmin dump** outputs a range of revisions. Use the `--revision (-r)` option to specify a single revision to dump, or a range of revisions. If you omit this option, all the existing repository revisions will be dumped.

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

As Subversion dumps each new revision, it outputs only enough information to allow a future loader to re-create that revision based on the previous one. In other words, for any given revision in the dump file, only the items that were changed in that revision will appear in the dump. The only exception to this rule is the first revision that is dumped with the current **svnadmin dump** command.

By default, Subversion will not express the first dumped revision as merely differences to be applied to the previous revision. For one thing, there is no previous revision in the dump file! And secondly, Subversion cannot know the state of the repository into which the dump data will be loaded (if it ever is). To ensure that the output of each execution of **svnadmin dump** is self-sufficient, the first dumped revision is by default a full representation of every directory, file, and property in that revision of the repository.

However, you can change this default behavior. If you add the `--incremental` option when you dump your repository, **svnadmin** will compare the first dumped revision against the previous revision in the repository, the same way it treats every other revision that gets dumped. It will then output the first revision exactly as it does the rest of the revisions in the dump range—mentioning only the changes that occurred in that revision. The benefit of this is that you can create several small dump files that can be loaded in succession, instead of one large one, like so:

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

These dump files could be loaded into a new repository with the following command sequence:

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

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran. Used like this, **svnadmin dump** can be one way to back up changes to your repository over time in case of a system crash or some other catastrophic event.

The dump format can also be used to merge the contents of several different repositories into a single repository. By using the `--parent-dir` option of **svnadmin load**, you can specify a new virtual root directory for the load process. That means if you have dump files for three repositories, say `calc-dumpfile`, `cal-dumpfile`, and `ss-dumpfile`, you can first create a new repository to hold them all:

```
$ svnadmin create /path/to/projects
$
```

Then, make new directories in the repository which will encapsulate the contents of each of the three previous repositories:

```
$ svn mkdir -m "Initial project roots" \
      file:///path/to/projects/calc \
      file:///path/to/projects/calendar \
      file:///path/to/projects/spreadsheet
```

```
Committed revision 1.
$
```

Lastly, load the individual dump files into their respective locations in the new repository:

```
$ svnadmin load /path/to/projects --parent-dir calc < calc-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir calendar < cal-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir spreadsheet < ss-dumpfile
…
$
```

We'll mention one final way to use the Subversion repository dump format—conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human-readable, it should be relatively easy to describe generic sets of changes—each of which should be treated as a new revision—using this file format. In fact, the **cvs2svn** utility (see “Convertendo um Repositório de CVS para Subversion”) uses the dump format to represent the contents of a CVS repository so that those contents can be copied into a Subversion repository.

Filtering Repository History

Since Subversion stores your versioned history using, at the very least, binary differencing algorithms and data compression (optionally in a completely opaque database system), attempting manual tweaks is unwise, if not quite difficult, and at any rate strongly discouraged. And once data has been stored in your repository, Subversion generally doesn't provide an easy way to remove that data. [9] But inevitably, there will be times when you would like to manipulate the history of your repository. You might need to strip out all instances of a file that was accidentally added to the repository (and shouldn't be there for whatever reason). [10] Or, perhaps you have multiple projects sharing a single repository, and you decide to split them up into their own repositories. To accomplish tasks like this, administrators need a more manageable and malleable representation of the data in their repositories—the Subversion repository dump format.

As we described in “Migrating Repository Data Elsewhere”, the Subversion repository dump format is a human-readable representation of the changes that you've made to your versioned data over time. You use the **svnadmin dump** command to generate the dump data, and **svnadmin load** to populate a new repository with it (see “Migrating Repository Data Elsewhere”). The great thing about the human-readability aspect of the dump format is that, if you aren't careless about it, you can manually inspect and modify it. Of course, the downside is that if you have three years' worth of repository activity encapsulated in what is likely to be a very large dump file, it could take you a long, long time to manually inspect and modify it.

That's where **svndumpfilter** becomes useful. This program acts as path-based filter for repository dump streams. Simply give it either a list of paths you wish to keep, or a list of paths you wish to not keep, then pipe your repository dump data through this filter. The result will be a modified stream of dump data that contains only the versioned paths you (explicitly or implicitly) requested.

Let's look a realistic example of how you might use this program. We discuss elsewhere (see “Planning Your Repository Organization”) the process of deciding how to choose a layout for the data in your repositories—using one repository per project or combining them, arranging stuff within your repository, and so on. But sometimes after new revisions start flying in, you rethink your layout and would like to make some

[9] That's rather the reason you use version control at all, right?

[10] Conscious, cautious removal of certain bits of versioned data is actually supported by real use-cases. That's why an “obliterate” feature has been one of the most highly requested Subversion features, and one which the Subversion developers hope to soon provide.

changes. A common change is the decision to move multiple projects which are sharing a single repository into separate repositories for each project.

Our imaginary repository contains three projects: `calc`, `calendar`, and `spreadsheet`. They have been living side-by-side in a layout like this:

```
/
   calc/
      trunk/
      branches/
      tags/
   calendar/
      trunk/
      branches/
      tags/
   spreadsheet/
      trunk/
      branches/
      tags/
```

To get these three projects into their own repositories, we first dump the whole repository:

```
$ svnadmin dump /path/to/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

Next, run that dump file through the filter, each time including only one of our top-level directories, and resulting in three new dump files:

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile > ss-dumpfile
…
$
```

At this point, you have to make a decision. Each of your dump files will create a valid repository, but will preserve the paths exactly as they were in the original repository. This means that even though you would have a repository solely for your `calc` project, that repository would still have a top-level directory named `calc`. If you want your `trunk`, `tags`, and `branches` directories to live in the root of your repository, you might wish to edit your dump files, tweaking the `Node-path` and `Node-copyfrom-path` headers to no longer have that first `calc/` path component. Also, you'll want to remove the section of dump data that creates the `calc` directory. It will look something like:

```
Node-path: calc
Node-action: add
```

```
Node-kind: dir
Content-length: 0
```

If you do plan on manually editing the dump file to remove a top-level directory, make sure that your editor is not set to automatically convert end-of-line characters to the native format (e.g. \r\n to \n), as the content will then not agree with the metadata. This will render the dump file useless.

All that remains now is to create your three new repositories, and load each dump file into the right repository:

```
$ svnadmin create calc; svnadmin load calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar; svnadmin load calendar < cal-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : cal.c ... done.
…
$ svnadmin create spreadsheet; svnadmin load spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : ss.c ... done.
…
$
```

Both of **svndumpfilter**'s subcommands accept options for deciding how to deal with “empty” revisions. If a given revision contained only changes to paths that were filtered out, that now-empty revision could be considered uninteresting or even unwanted. So to give the user control over what to do with those revisions, **svndumpfilter** provides the following command-line options:

`--drop-empty-revs`
: Do not generate empty revisions at all—just omit them.

`--renumber-revs`
: If empty revisions are dropped (using the `--drop-empty-revs` option), change the revision numbers of the remaining revisions so that there are no gaps in the numeric sequence.

`--preserve-revprops`
: If empty revisions are not dropped, preserve the revision properties (log message, author, date, custom properties, etc.) for those empty revisions. Otherwise, empty revisions will only contain the original datestamp, and a generated log message that indicates that this revision was emptied by **svndumpfilter**.

While **svndumpfilter** can be very useful, and a huge timesaver, there are unfortunately a couple of gotchas. First, this utility is overly sensitive to path semantics. Pay attention to whether paths in your dump file are specified with or without leading slashes. You'll want to look at the `Node-path` and `Node-copyfrom-path` headers.

```
…
Node-path: spreadsheet/Makefile
```

...

If the paths have leading slashes, you should include leading slashes in the paths you pass to **svndumpfilter include** and **svndumpfilter exclude** (and if they don't, you shouldn't). Further, if your dump file has an inconsistent usage of leading slashes for some reason, [11] you should probably normalize those paths so they all have, or lack, leading slashes.

Also, copied paths can give you some trouble. Subversion supports copy operations in the repository, where a new path is created by copying some already existing path. It is possible that at some point in the lifetime of your repository, you might have copied a file or directory from some location that **svndumpfilter** is excluding, to a location that it is including. In order to make the dump data self-sufficient, **svndumpfilter** needs to still show the addition of the new path—including the contents of any files created by the copy—and not represent that addition as a copy from a source that won't exist in your filtered dump data stream. But because the Subversion repository dump format only shows what was changed in each revision, the contents of the copy source might not be readily available. If you suspect that you have any copies of this sort in your repository, you might want to rethink your set of included/excluded paths, perhaps including the paths that served as sources of your troublesome copy operations, too.

Finally, **svndumpfilter** takes path filtering quite literally. If you are trying to copy the history of a project rooted at `trunk/my-project` and move it into a repository of its own, you would, of course, use the **svndumpfilter include** command to keep all the changes in and under `trunk/my-project`. But the resulting dump file makes no assumptions about the repository into which you plan to load this data. Specifically, the dump data might begin with the revision which added the `trunk/my-project` directory, but it will *not* contain directives which would create the `trunk` directory itself (because `trunk` doesn't match the include filter). You'll need to make sure that any directories which the new dump stream expect to exist actually do exist in the target repository before trying to load the stream into that repository.

Repository Replication

There are several scenarios in which it is quite handy to have a Subversion repository whose version history is exactly the same as some other repository's. Perhaps the most obvious one is the maintenance of a simple backup repository, used when the primary repository has become inaccessible due to a hardware failure, network outage, or other such annoyance. Other scenarios include deploying mirror repositories to distribute heavy Subversion load across multiple servers, use as a soft-upgrade mechanism, and so on.

As of version 1.4, Subversion provides a program for managing scenarios like these—**svnsync**. **svnsync** works by essentially asking the Subversion server to “replay” revisions, one at a time. It then uses that revision information to mimic a commit of the same to another repository. Neither repository needs to be locally accessible to machine on which **svnsync** is running—its parameters are repository URLs, and it does all its work through Subversion's repository access (RA) interfaces. All it requires is read access to the source repository and read/write access to the destination repository.

When using **svnsync** against a remote source repository, the Subversion server for that repository must be running Subversion version 1.4 or better.

Assuming you already have a source repository that you'd like to mirror, the next thing you need is an empty target repository which will actually serve as that mirror. This target repository can use either of the available filesystem data-store back-ends (see “Choosing a Data Store”), but it must not yet have any version history in it. The protocol via which **svnsync** communicates revision information is highly sensitive to mismatches between the versioned histories contained in the source and target repositories. For this reason, while

[11] While **svnadmin dump** has a consistent leading slash policy—to not include them—other programs which generate dump data might not be so consistent.

svnsync cannot *demand* that the target repository be read-only, [12] allowing the revision history in the target repository to change by any mechanism other than the mirroring process is a recipe for disaster.

Do *not* modify a mirror repository in such a way as to cause its version history to deviate from that of the repository it mirrors. The only commits and revision property modifications that ever occur on that mirror repository should be those performed by the **svnsync** tool.

Another requirement of the target repository is that the **svnsync** process be allowed to modify certain revision properties. **svnsync** stores its bookkeeping information in special revision properties on revision 0 of the destination repository. Because **svnsync** works within the framework of that repository's hook system, the default state of the repository (which is to disallow revision property changes; see pre-revprop-change) is insufficient. You'll need to explicitly implement the pre-revprop-change hook, and your script must allow **svnsync** to set and change its special properties. With those provisions in place, you are ready to start mirroring repository revisions.

It's a good idea to implement authorization measures which allow your repository replication process to perform its tasks while preventing other users from modifying the contents of your mirror repository at all.

Let's walk through the use of **svnsync** in a somewhat typical mirroring scenario. We'll pepper this discourse with practical recommendations which you are free to disregard if they aren't required by or suitable for your environment.

As a service to the fine developers of our favorite version control system, we will be mirroring the public Subversion source code repository and exposing that mirror publicly on the Internet, hosted on a different machine than the one on which the original Subversion source code repository lives. This remote host has a global configuration which permits anonymous users to read the contents of repositories on the host, but requires users to authenticate in order to modify those repositories. (Please forgive us for glossing over the details of Subversion server configuration for the moment—those are covered thoroughly in Capítulo 6, *Configuração do Servidor*.) And for no other reason than that it makes for a more interesting example, we'll be driving the replication process from a third machine, the one which we currently find ourselves using.

First, we'll create the repository which will be our mirror. This and the next couple of steps do require shell access to the machine on which the mirror repository will live. Once the repository is all configured, though, we shouldn't need to touch it directly again.

```
$ ssh admin@svn.example.com \
      "svnadmin create /path/to/repositories/svn-mirror"
admin@svn.example.com's password: ********
$
```

At this point, we have our repository, and due to our server's configuration, that repository is now “live” on the Internet. Now, because we don't want anything modifying the repository except our replication process, we need a way to distinguish that process from other would-be committers. To do so, we use a dedicated username for our process. Only commits and revision property modifications performed by the special username `syncuser` will be allowed.

We'll use the repository's hook system both to allow the replication process to do what it needs to do, and to enforce that only it is doing those things. We accomplish this by implementing two of the repository event hooks—pre-revprop-change and start-commit. Our `pre-revprop-change` hook script is found in Exemplo 5.2, “Mirror repository's pre-revprop-change hook script”, and basically verifies that the user attempting the property changes is our `syncuser` user. If so, the change is allowed; otherwise, it is denied.

[12] In fact, it can't truly be read-only, or **svnsync** itself would have a tough time copying revision history into it.

Exemplo 5.2. Mirror repository's pre-revprop-change hook script

```
#!/bin/sh

USER="$3"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may change revision properties" >&2
exit 1
```

That covers revision property changes. Now we need to ensure that only the `syncuser` user is permitted to commit new revisions to the repository. We do this using a `start-commit` hook scripts like the one in Exemplo 5.3, “Mirror repository's start-commit hook script”.

Exemplo 5.3. Mirror repository's start-commit hook script

```
#!/bin/sh

USER="$2"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may commit new revisions" >&2
exit 1
```

After installing our hook scripts and ensuring that they are executable by the Subversion server, we're finished with the setup of the mirror repository. Now, we get to actually do the mirroring.

The first thing we need to do with **svnsync** is to register in our target repository the fact that it will be a mirror of the source repository. We do this using the **svnsync initialize** subcommand. Note that the various **svnsync** subcommands provide several of the same authentication-related options that **svn** does: `--username`, `--password`, `--non-interactive`, `--config-dir`, and `--no-auth-cache`.

```
$ svnsync help init
initialize (init): usage: svnsync initialize DEST_URL SOURCE_URL

Initialize a destination repository for synchronization from
another repository.

The destination URL must point to the root of a repository with
no committed revisions.  The destination repository must allow
revision property changes.

You should not commit to, or make revision property changes in,
the destination repository by any method other than 'svnsync'.
In other words, the destination repository should be a read-only
mirror of the source repository.

Valid options:
  --non-interactive        : do no interactive prompting
  --no-auth-cache          : do not cache authentication tokens
```

```
  --username arg           : specify a username ARG
  --password arg           : specify a password ARG
  --config-dir arg         : read user configuration files from directory ARG

$ svnsync initialize http://svn.example.com/svn-mirror \
                     http://svn.collab.net/repos/svn \
                     --username syncuser --password syncpass
Copied properties for revision 0.
$
```

Our target repository will now remember that it is a mirror of the public Subversion source code repository. Notice that we provided a username and password as arguments to **svnsync**—that was required by the pre-revprop-change hook on our mirror repository.

The URLs provided to **svnsync** must point to the root directories of the target and source repositories, respectively. The tool does not handle mirroring of repository subtrees.

The initial release of **svnsync** (in Subversion 1.4) has a small shortcoming—the values given to the `--username` and `--password` command-line options get used for authentication against both the source and destination repositories. Obviously, there's no guarantee that the synchronizing user's credentials are the same in both places. In the event that they are not the same, users trying to run **svnsync** in non-interactive mode (with the `--non-interactive` option) might experience problems.

And now comes the fun part. With a single subcommand, we can tell **svnsync** to copy all the as-yet-unmirrored revisions from the source repository to the target. [13] The **svnsync synchronize** subcommand will peek into the special revision properties previously stored on the target repository, and determine what repository it is mirroring and that the most recently mirrored revision was revision 0. Then it will query the source repository and determine what the latest revision in that repository is. Finally, it asks the source repository's server to start replaying all the revisions between 0 and that latest revision. As **svnsync** get the resulting response from the source repository's server, it begins forwarding those revisions to the target repository's server as new commits.

```
$ svnsync help synchronize
synchronize (sync): usage: svnsync synchronize DEST_URL

Transfer all pending revisions from source to destination.
…
$ svnsync synchronize http://svn.example.com/svn-mirror \
                      --username syncuser --password syncpass
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 23406.
Copied properties for revision 23406.
Committed revision 23407.
```

[13] Be forewarned that while it will take only a few seconds for the average reader to parse this paragraph and the sample output which follows it, the actual time required to complete such a mirroring operation is, shall we say, quite a bit longer.

```
Copied properties for revision 23407.
Committed revision 23408.
Copied properties for revision 23408.
```

Of particular interest here is that for each mirrored revision, there is first a commit of that revision to the target repository, and then property changes follow. This is because the initial commit is performed by (and attributed to) the user `syncuser`, and datestamped with the time as of that revision's creation. Also, Subversion's underlying repository access interfaces don't provide a mechanism for setting arbitrary revision properties as part of a commit. So **svnsync** follows up with an immediate series of property modifications which copy all the revision properties found for that revision in the source repository into the target repository. This also has the effect of fixing the author and datestamp of the revision to match that of the source repository.

Also noteworthy is that **svnsync** performs careful bookkeeping that allows it to be safely interrupted and restarted without ruining the integrity of the mirrored data. If a network glitch occurs while mirroring a repository, simply repeat the **svnsync synchronize** command and it will happily pick up right where it left off. In fact, as new revisions appear in the source repository, this is exactly what you to do in order to keep your mirror up-to-date.

There is, however, one bit of inelegance in the process. Because Subversion revision properties can be changed at any time throughout the lifetime of the repository, and don't leave an audit trail that indicates when they were changed, replication processes have to pay special attention to them. If you've already mirrored the first 15 revisions of a repository and someone then changes a revision property on revision 12, **svnsync** won't know to go back and patch up its copy of revision 12. You'll need to tell it to do so manually by using (or with some additionally tooling around) the **svnsync copy-revprops** subcommand, which simply re-replicates all the revision properties for a particular revision.

```
$ svnsync help copy-revprops
copy-revprops: usage: svnsync copy-revprops DEST_URL REV

Copy all revision properties for revision REV from source to
destination.
…
$ svnsync copy-revprops http://svn.example.com/svn-mirror 12 \
                        --username syncuser --password syncpass
Copied properties for revision 12.
$
```

That's repository replication in a nutshell. You'll likely want some automation around such a process. For example, while our example was a pull-and-push setup, you might wish to have your primary repository push changes to one or more blessed mirrors as part of its post-commit and post-revprop-change hook implementations. This would enable the mirror to be up-to-date in as near to realtime as is likely possible.

Also, while it isn't very commonplace to do so, **svnsync** does gracefully mirror repositories in which the user as whom it authenticates only has partial read access. It simply copies only the bits of the repository that it is permitted to see. Obviously such a mirror is not useful as a backup solution.

As far as user interaction with repositories and mirrors goes, it *is* possible to have a single working copy that interacts with both, but you'll have to jump through some hoops to make it happen. First, you need to ensure that both the primary and mirror repositories have the same repository UUID (which is not the case by default). You can set the mirror repository's UUID by loading a dump file stub into it which contains the UUID of the primary repository, like so:

```
$ cat - <<EOF | svnadmin load --force-uuid dest
```

```
SVN-fs-dump-format-version: 2

UUID: 65390229-12b7-0310-b90b-f21a5aa7ec8e
EOF
$
```

Now that the two repositories have the same UUID, you can use **svn switch --relocate** to point your working copy to whichever of the repositories you wish to operate against, a process which is described in svn switch. There is a possible danger here, though, in that if the primary and mirror repositories aren't in close synchronization, a working copy up-to-date with, and pointing to, the primary repository will, if relocated to point to an out-of-date mirror, become confused about the apparent sudden loss of revisions it fully expects to be present, and throws errors to that effect. If this occurs, you can relocate your working copy back to the primary repository and then either wait until the mirror repository is up-to-date, or backdate your working copy to a revision you know is present in the sync repository and then retry the relocation.

Finally, be aware that the revision-based replication provided by **svnsync** is only that—replication of revisions. It does not include such things as the hook implementations, repository or server configuration data, uncommitted transactions, or information about user locks on repository paths. Only information carried by the Subversion repository dump file format is available for replication.

Repository Backup

Despite numerous advances in technology since the birth of the modern computer, one thing unfortunately rings true with crystalline clarity—sometimes, things go very, very awry. Power outages, network connectivity dropouts, corrupt RAM and crashed hard drives are but a taste of the evil that Fate is poised to unleash on even the most conscientious administrator. And so we arrive at a very important topic—how to make backup copies of your repository data.

There are two types of backup methods available for Subversion repository administrators—full and incremental. A full backup of the repository involves squirreling away in one sweeping action all the information required to fully reconstruct that repository in the event of a catastrophe. Usually, it means, quite literally, the duplication of the entire repository directory (which includes either a Berkeley DB or FSFS environment). Incremental backups are lesser things, backups of only the portion of the repository data that has changed since the previous backup.

As far as full backups go, the naive approach might seem like a sane one, but unless you temporarily disable all other access to your repository, simply doing a recursive directory copy runs the risk of generating a faulty backup. In the case of Berkeley DB, the documentation describes a certain order in which database files can be copied that will guarantee a valid backup copy. A similar ordering exists for FSFS data. But you don't have to implement these algorithms yourself, because the Subversion development team has already done so. The **svnadmin hotcopy** command takes care of the minutia involved in making a hot backup of your repository. And its invocation is as trivial as Unix's **cp** or Windows' **copy** operations:

```
$ svnadmin hotcopy /path/to/repos /path/to/repos-backup
```

The resulting backup is a fully functional Subversion repository, able to be dropped in as a replacement for your live repository should something go horribly wrong.

When making copies of a Berkeley DB repository, you can even instruct **svnadmin hotcopy** to purge any unused Berkeley DB logfiles (see “Purging unused Berkeley DB logfiles”) from the original repository upon completion of the copy. Simply provide the `--clean-logs` option on the command-line.

```
$ svnadmin hotcopy --clean-logs /path/to/bdb-repos /path/to/bdb-repos-backup
```

Additional tooling around this command is available, too. The `tools/backup/` directory of the Subversion source distribution holds the **hot-backup.py** script. This script adds a bit of backup management atop **svnadmin hotcopy**, allowing you to keep only the most recent configured number of backups of each repository. It will automatically manage the names of the backed-up repository directories to avoid collisions with previous backups, and will "rotate off" older backups, deleting them so only the most recent ones remain. Even if you also have an incremental backup, you might want to run this program on a regular basis. For example, you might consider using **hot-backup.py** from a program scheduler (such as **cron** on Unix systems) which will cause it to run nightly (or at whatever granularity of Time you deem safe).

Some administrators use a different backup mechanism built around generating and storing repository dump data. We described in "Migrating Repository Data Elsewhere" how to use **svnadmin dump --incremental** to perform an incremental backup of a given revision or range of revisions. And of course, there is a full backup variation of this achieved by omitting the `--incremental` option to that command. There is some value in these methods, in that the format of your backed-up information is flexible—it's not tied to a particular platform, versioned filesystem type, or release of Subversion or Berkeley DB. But that flexibility comes at a cost, namely that restoring that data can take a long time—longer with each new revision committed to your repository. Also, as is the case with so many of the various backup methods, revision property changes made to already-backed-up revisions won't get picked up by a non-overlapping, incremental dump generation. For these reasons, we recommend against relying solely on dump-based backup approaches.

As you can see, each of the various backup types and methods has its advantages and disadvantages. The easiest is by far the full hot backup, which will always result in a perfect working replica of your repository. Should something bad happen to your live repository, you can restore from the backup with a simple recursive directory copy. Unfortunately, if you are maintaining multiple backups of your repository, these full copies will each eat up just as much disk space as your live repository. Incremental backups, by contrast, tend to be quicker to generate and smaller to store. But the restoration process can be a pain, often involving applying multiple incremental backups. And other methods have their own peculiarities. Administrators need to find the balance between the cost of making the backup and the cost of restoring it.

The **svnsync** program (see "Repository Replication") actually provides a rather handy middle-ground approach. If you are regularly synchronizing a read-only mirror with your main repository, then in a pinch, your read-only mirror is probably a good candidate for replacing that main repository if it falls over. The primary disadvantage of this method is that only the versioned repository data gets synchronized—repository configuration files, user-specified repository path locks, and other items which might live in the physical repository directory but not *inside* the repository's virtual versioned filesystem are not handled by svnsync.

In any backup scenario, repository administrators need to be aware of how modifications to unversioned revision properties affect their backups. Since these changes do not themselves generate new revisions, they will not trigger post-commit hooks, and may not even trigger the pre-revprop-change and post-revprop-change hooks. [14] And since you can change revision properties without respect to chronological order—you can change any revision's properties at any time—an incremental backup of the latest few revisions might not catch a property modification to a revision that was included as part of a previous backup.

Generally speaking, only the truly paranoid would need to backup their entire repository, say, every time a commit occurred. However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (like per-commit emails or incremental dumps), a hot backup of the database might be something that a repository administrator would want to include as part of a system-wide nightly backup. It's your data—protect it as much as you'd like.

Often, the best approach to repository backups is a diversified one which leverages combinations of the methods described here. The Subversion developers, for example, back up the Subversion source code repository nightly using **hot-backup.py** and an offsite **rsync** of those full backups; keep multiple archives of

[14] **svnadmin setlog** can be called in a way that bypasses the hook interface altogether.

all the commit and property change notification emails; and have repository mirrors maintained by various volunteers using **svnsync**. Your solution might be similar, but should be catered to your needs and that delicate balance of convenience with paranoia. And whatever you do, validate your backups from time to time—what good is a spare tire that has a hole in it? While all of this might not save your hardware from the iron fist of Fate, [15] it should certainly help you recover from those trying times.

Sumário

Até agora você deve ter tido um entendimento de como criar, configurar e manter repositórios Subversion. Nós o introduzimos a várias ferramentas que o ajudarão nessa tarefa. Ao longo desse capítulo, nós mostramos obstáculos comuns e sugestões para evitá-los.

Tudo que falta é para você é decidir que tipo de informação armazenar no seu repositório, e finalmente, como deixá-lo disponível na rede. O próximo capítulo é todo sobre rede.

[15] You know—the collective term for all of her “fickle fingers”.

Capítulo 6. Configuração do Servidor

Um repositório Subversion pode ser acessado simultaneamente por clientes executando na mesma máquina na qual o repositório se encontra usando o método `file://`. Mas a configuração típica do Subversion envolve ter-se uma única máquina servidora sendo acessada por clientes em computadores por todo um escritório—ou, talvez, por todo o mundo.

Este capítulo descreve como ter seu repositório Subversion acessível a partir da máquina onde estiver instalado para uso por clientes remotos. Vamos cobrir os mecanismos de servidor do Subversion disponíveis atualmente, discutindo a configuração e o uso de cada um. Depois de ler esta seção, você deve ser capaz de decidir qual configuração é a adequada às suas necessidades, e entender como habilitar tal configuração em seu servidor.

Visão Geral

O Subversion foi desenvolvido com uma camada de rede abstrata. Isso significa que um repositório pode ser acessado via programação por qualquer tipo de processo servidor, e a API de “acesso ao repositório” do cliente permite aos programadores escrever plugins que falem com protocolos de rede relevantes. Em teoria, o Subversion pode usar um número infinito de implementações de rede. Na prática, há apenas dois servidores até o momento em que este livro estava sendo escrito.

O Apache é um servidor web extremamente popular; usando o módulo **mod_dav_svn**, o Apache pode acessar um repositório e torná-lo disponível para os clientes através do protocolo WebDAV/DeltaV, que é uma extensão do HTTP. Como o Apache é um servidor web extremamente extensível, ele provê um conjunto de recursos “de graça”, tais como comunicação SSL criptografada, sistema de log, integração com diversos sistemas de autenticação de terceiros, além de navegação simplificada nos repositórios.

Por outro lado está o **svnserve**: um programa servidor pequeno e leve que conversa com os clientes por meio de um protocolo específico. Pelo fato de ter sido explicitamente desenvolvido para o Subversion e de manter informações de estado (diferentemente do HTTP), este seu protocolo permite operações de rede significativamente mais rápidas—ainda que ao custo de alguns recursos. Este protocolo só entende autenticação do tipo CRAM-MD5, não possui recursos de log, nem de navegação web nos repositórios, e não tem opção de criptografar o tráfego de rede. Mas é, no entanto, extremamente fácil de configurar e é quase sempre a melhor opção para pequenas equipes que ainda estão iniciando com o Subversion.

Uma terceira opção é o uso do **svnserve** através de uma conexão SSH. Por mais que este cenário ainda use o **svnserve**, ele difere um pouco no que diz respeito aos recursos de uma implantação **svnserve** tradicional. SSH é usado para criptografar toda a comunicação. O SSH também é usado exclusivamente para autenticação, então são necessárias contas reais no sistema do host servidor (diferentemente do uso do **svnserve** tradicional, que possui suas próprias contas de usuário particulares.) Finalmente, pelo fato desta configuração precisar que cada usuário dispare um processo temporário **svnserve** particular, esta opção é equivalente (do ponto de vista de permissões) a permitir total acesso de um grupo local de usuários no repositório por meio de URLs `file://`. Assim, controle de acesso com base em caminhos não faz sentido, já que cada usuário está acessando os arquivos da base de dados diretamente.

Aqui está um breve sumário destas três configurações típicas de servidor.

Tabela 6.1. Comparação das Opções para o Servidor Subversion

Característica	Apache + mod_dav_svn	svnserve	svnserve sobre SSH
Opções de autenticação	Autenticação básica HTTP(S), certificados X.509, LDAP, NTLM, ou quaisquer outros mecanismos disponíveis ao Apache httpd	CRAM-MD5	SSH
Opções para contas de usuários	arquivo 'users' privativo	arquivo 'users' privativo	contas no sistema
Opções de authorização	acesso leitura/escrita pode ser dado para o repositório como um todo, ou especificado por caminho	acesso leitura/escrita pode ser dado para o repositório como um todo, ou especificado por caminho	acesso leitura/escrita passível de ser dado apenas ao repositório como um todo
Criptografia	através de SSL opcional	nenhuma	túnel SSH
Registro de log	logs completos do Apache para cada requisição HTTP, com opcional log “alto nível” para operações do cliente em geral	sem log	sem log
Interoperabilidade	parcialmente usável por outros clientes WebDAV	se comunica apenas com clientes svn	se comunica apenas com clientes svn
Visualização pela web	suporte existente limitado, ou também por meio de ferramentas de terceiros como o ViewVC	apenas por meio de ferramentas de terceiros como o ViewVC	apenas por meio de ferramentas de terceiros como o ViewVC
Velocidade	um pouco mais lento	um pouco mais rápido	um pouco mais rápido
Configuração inicial	um tanto complexa	extremamente simples	moderadamente simples

Escolhendo uma Configuração de Servidor

Então, que servidor você deve usar? Qual é melhor?

Obviamente, não há uma resposta definitiva para esta pergunta. Cada equipe tem diferentes necessidades e os diferentes servidores todos representam diferentes conjuntos de características. O projeto Subversion em si não endossa um ou outro servidor, nem mesmo considera um servidor mais “oficial” que outro.

Aqui estão algumas razões pelas quais você deveria escolher uma configuração ao invés de outra, bem como as razões pelas quais você *não* deveria escolher uma delas.

O Servidor svnserve

Porque você pode querer usá-lo:

- Rápido e fácil de configurar.
- Protocolo de rede orientado a estado e notavelmente mais rápido que o WebDAV.

- Dispensa necessidade da criação de contas de usuário no sistema servidor.
- Senhas não trafegam através da rede.

Porque você pode querer evitá-lo:

- Protocolo de rede não é criptografado.
- Apenas um único método de autenticação disponível.
- Senhas são armazenadas em texto puro no servidor.
- Sem nenhum tipo de log, mesmo para erros.

svnserve sobre SSH

Porque você pode querer usá-lo:

- Protocolo de rede orientado a estado e notavelmente mais rápido que o WebDAV.
- Você pode aproveitar a existência de contas ssh e infraestrutura de usuários existente.
- Todo o tráfego de rede é criptografado.

Porque você pode querer evitá-lo:

- Apenas um único método de autenticação disponível.
- Sem nenhum tipo de log, mesmo para erros.
- Necessita que os usuários estejam num mesmo grupo no sistema, ou que usem uma chave ssh compartilhada.
- Seu uso inadequado pode resultar em problemas com permissões de arquivos.

O Servidor Apache HTTP

Porque você pode querer usá-lo:

- Permite que o Subversion use quaisquer dos inúmeros sistemas de autenticação já disponíveis e integrados com o Apache.
- Dispensa necessidade de criação de contas de usuário no sistema servidor.
- Logs completos do Apache.
- Tráfego de rede pode ser criptografado com SSL.
- HTTP(S) quase sempre não tem problemas para passar por firewalls.
- Navegação no repositório através de um navegador web.
- Repositório pode ser montado como um drive de rede para controle de versão transparente. (Veja “Autoversioning”.)

Porque você pode querer evitá-lo:

- Notavelmente mais lento que o svnserve, pelo fato do HTTP ser um protocolo sem informação de estado e acabar demandando mais requisições.

- Configuração inicial pode ser complexa.

Recomendações

No geral, os autores deste livro recomendam uma instalação tradicional do **svnserve** para pequenas equipes que ainda estão tentando familiarizar-se com o servidor Subversion; é a forma mais simples de utilização, e a que demanda menos esforço de manutenção. Você sempre pode trocar para uma implantação de servidor mais complexa conforme suas necessidades mudem.

Aqui seguem algumas recomendações e dicas em geral, baseadas na experiência de vários anos de suporte a usuários:

- Se você está tentando configurar o servidor mais simples possível para seu grupo, então uma instalação tradicional do **svnserve** é o caminho mais fácil e rápido. Note, entretanto, que os dados de seu repositório vão ser transmitidos às claras pela rede. Se você estiver fazendo uma implantação inteiramente dentro de sua rede LAN ou VPN da sua empresa, isto não chega a ser nenhum problema. Mas se o repositório tiver de ser acessível pela internet, então você deveria se assegurar que o conteúdo do repositório não contém dados sensíveis (p.ex. se é apenas código-fonte.)

- Se você precisar de integração com alguns sistemas de identificação existentes (LDAP, Active Directory, NTLM, X.509, etc.), então uma configuração baseada no Apache será sua única opção. Similarmente, se você indispensavelmente precisar de log de servidor tanto para registro de erros de servidor quanto para atividades dos clientes, então um servidor com base no Apache será necessário.

- Se você optou por usar ou o Apache ou o **svnserve**, crie uma única conta no sistema para o usuário `svn` e faça com que o processo do servidor seja executado por este usuário. Certifique-se de fazer com que o diretório do repositório pertença totalmente ao usuário `svn` também. Do ponto de vista da segurança, isto deixa os dados do repositório adequadamente seguros e protegidos pelas permissões do sistema de arquivos do sistema operacional, alteráveis apenas pelo próprio processo do servidor Subversion.

- Se você já tiver uma infraestrutura fortemente baseada em contas SSH, e se seus usuários já possuirem contas no servidor, então faz sentido implantar uma solução usando o svnserve sobre SSH. Caso contrário, não recomendamos esta opção largamente ao público. Ter seus usuários acessando o repositório por meio de contas (imaginárias) gerenciadas pelo **svnserve** ou pelo Apache é geralmente considerado mais seguro do que ter acesso por meio de contas reais no sistema. Se o motivo para você fazer isso for apenas obter uma comunicação criptografada, nós recomendamos que você utilize Apache com SSL no lugar.

- *Não* e entusiasme simplesmente com a idéia de ter todos os seus usuários acessando o repositório diretamente através de URLs `file://`. Mesmo se o repositório estiver disponível a todos para leitura através de um compartilhamento de rede, isto é uma má idéia. Esta configuração remove quaisquer camadas de proteção entre os usuários e o repositório: os usuários podem acidentalmente (ou intencionalmente) corromper a base de dados do repositório, dificulta deixar o repositório offline para fins de inspeção ou atualização, e ainda podem surgir problemas relacionados a permissões de arquivos (veja “Supporting Multiple Repository Access Methods”.) Perceba que esta é uma das razões pelas quais nós alertamos acerca do acesso aos repositórios através de URLs `svn+ssh://`—o que sob a perspectiva de segurança, é efetivamente o mesmo que ter usuários locais acessando via `file://`, e pode resultar nos mesmos problemas se o administrador não for cuidadoso.

svnserve, um servidor especializado

O programa **svnserve** é um servidor leve, capaz de falar com clientes via TCP/IP usando um protocolo específico e robusto. Os clientes contactam um servidor **svnserve** usando URLs que começam com o esquema `svn://` ou `svn+ssh://`. Esta seção vai explicar as diversas formas de se executar o **svnserve**,

como os clientes se autenticam para o servidor, e como configurar o controle de acesso apropriado aos seus repositórios.

Invocando o Servidor

Há poucas maneiras distintas de se invocar o programa **svnserve**:

- Executar o **svnserve** como um daemon independente, aguardando por requisições.
- Fazer com que o daemon **inetd** do Unix dispare temporariamente o **svnserve** a cada vez que uma requisição chegar numa dada porta.
- Fazer com que o SSH execute um **svnserve** temporário sobre um túnel criptografado.
- Executar o **svnserve** como um serviço do Windows.

svnserve como Daemon

A opção mais fácil é executar o **svnserve** como um “daemon” independente. Use a opção `-d` para isto:

```
$ svnserve -d
$              # o svnserve está rodando agora, ouvindo na porta 3690
```

Ao executar o **svnserve** no modo daemon, você pode usar as opções `--listen-port=` e `--listen-host=` para especificar a porta e o hostname exatos aos quais o servidor estará “associado”.

Uma vez que tenhamos iniciado o **svnserve** como mostrado acima, isto torna todos os repositórios do sistema disponíveis na rede. Um cliente precisa especificar uma URL com um caminho *absoluto* do repositório. Por exemplo, se um repositório estiver localizado em `/usr/local/repositories/project1`, então um cliente deveria acessá-lo com `svn://host.example.com/usr/local/repositories/project1`. Para aumentar a segurança, você pode passar a opção `-r` para o **svnserve**, o que limita a exportar apenas os repositórios sob o caminho especificado. Por exemplo:

```
$ svnserve -d -r /usr/local/repositories
…
```

O uso da opção `-r` efetivamente modifica o local que o programa considera como a raiz do sistema de arquivos remoto. Os clientes então usam URLs com aquela parte do caminho removida, tornando-as mais curtas (e bem menos informativas):

```
$ svn checkout svn://host.example.com/project1
…
```

svnserve através do inetd

Se você quiser que o **inetd** execute o processo, então você precisa passar a opção `-i` (`--inetd`). No exemplo, mostramos a saída da execução do comando `svnserve -i` na linha de comando, mas note que atualmente não é assim que se inicia o daemon; leia os parágrafos depois do exemplo para saber como configurar o **inetd** para iniciar o **svnserve**.

```
$ svnserve -i
( success ( 1 2 ( ANONYMOUS ) ( edit-pipeline ) ) )
```

Quando invocado com a opção `--inetd`, o **svnserve** tenta se comunicar com um cliente Subversion por meio do *stdin* e *stdout* usando um protocolo específico. Este é o comportamento padrão para um programa sendo executado através do **inetd**. A IANA reservou a porta 3690 para o protocolo Subversion, assim, em um sistema Unix-like você poderia adicionar linhas como estas ao arquivo `/etc/services` (se elas já não existirem):

```
svn             3690/tcp    # Subversion
svn             3690/udp    # Subversion
```

E se seu sistema Unix-like estiver usando um daemon **inetd** clássico, você pode adicionar esta linha ao arquivo `/etc/inetd.conf`:

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

Assegure-se de que “svnowner” seja um usuário com permissões apropriadas para acesso aos seus repositórios. Agora, quando uma conexão do cliente atingir seu servidor na porta 3690, o **inetd** vai disparar um processo **svnserve** para atendê-la. Obviamente, você também pode querer adicionar a opção `-r` para restringir quais repositórios serão exportados.

svnserve sobre um Túnel

Uma terceira forma de se invocar o **svnserve** é no “modo túnel”, com a opção `-t`. Este modo assume que um programa de acesso remoto como o **RSH** ou o **SSH** autenticou um usuário com sucesso e está agora invocando um processo **svnserve** particular *como aquele usuário*. (Note que você, o usuário, vai raramente, ou talvez nunca, precisar invocar o **svnserve** com a opção `-t` na linha de comando; já que o próprio daemon **SSH** faz isso para você.) O programa **svnserve** funciona normalmente (se comunicando por meio do *stdin* e *stdout*), e assume que o tráfego está sendo automaticamente redirecionado por algum tipo de túnel de volta para o cliente. Quando o **svnserve** é invocado por um túnel agente como este, assegure-se de que o usuário autenticado tenha completo acesso de leitura e escrita aos arquivos da base de dados do repositório. É essencialmente o mesmo que um usuário local acessando o repositório por meio de URLs `file://`.

Esta opção está descrita com mais detalhes em “Tunelamento sobre SSH”.

svnserve como um Serviço do Windows

Se seu sistema Windows é descendente dos Windows NT (2000, 2003, XP, Vista), então você pode executar o **svnserve** como um serviço padrão do Windows. Esta é tipicamente uma experiência mais proveitosa do que executá-lo como um daemon independente com a opção `--daemon (-d)`. Usar o modo daemon implica em executar um console, digitar um comando, e então deixar a janela do console executando indefinidamente. Um serviço do Windows, no entanto, executa em segundo plano, pode ser executado automaticamente na inicialização, e pode se iniciado e parado através da mesma interface de administração como os outros serviços do Windows.

Você vai precisar definir o novo serviço usando a ferramenta de linha de comando **SC.EXE.**. Semelhantemente à linha de configuração do **inetd**, você deve especificar a forma exata de invocação do **svnserve** para que o Windows o execute na inicialização:

```
C:\> sc create svn
        binpath= "C:\svn\bin\svnserve.exe --service -r C:\repos"
        displayname= "Servidor Subversion"
        depend= Tcpip
        start= auto
```

Isto define um novo serviço do Windows chamado “svn”, o qual executa um comando **svnserve.exe** particular quando iniciado (neste caso, com raiz em `C:\repos`.) No entanto, há diversos pontos a considerar neste exemplo anterior.

Primeiramente, note que o programa **svnserve.exe** deve sempre ser chamado com a opção `--service`. Quaisquer outras opções para o **svnserve** então devem ser especificadas na mesma linha, mas você não pode adicionar opções conflitantes tais como `--daemon (-d)`, `--tunnel`, ou `--inetd (-i)`. Já opções como `-r` ou `--listen-port` não terão problemas. Em segundo lugar, tenha cuidado com relação a espaços ao invocar o comando **SC.EXE**: padrões `chave= valor` não devem conter espaços entre `chave=` e devem ter exatamente um espaço antes de `valor`. Por último, tenha cuidado também com espaços na sua linha de comando a ser executada. Se um nome de diretório contiver espaços (ou outros caracteres que precisem de escape), coloque todo o valor interno de `binpath` entre aspas duplas, escapando-as:

```
C:\> sc create svn
        binpath= "\"C:\arquivos de programas\svn\bin\svnserve.exe\" --service -r C:\repos"
        displayname= "Servidor Subversion"
        depend= Tcpip
        start= auto
```

Também observe que o termo `binpath` é confuso—seu valor é a *linha de comando*, não o caminho para um executável. Por isso que você precisa delimitá-lo com aspas se o valor contiver espaços.

Uma vez que o serviço esteja definido, ele pode ser parado, iniciado, ou consultado usando-se as ferramentas GUI (o painel de controle administrativo Serviços), bem como através da linha decomando:

```
C:\> net stop svn
C:\> net start svn
```

O serviço também pode ser desinstalado (i.e. indefinido) excluindo-se sua definição: `sc delete svn`. Apenas certifique-se de parar o serviço antes! O programa **SC.EXE** tem diversos outros subcomandos e opções; digite `sc /?` para saber mais sobre ele.

Autenticação e autorização internos

Quando um cliente se conecta a um processo **svnserve**, as seguintes coisas acontecem:

- O cliente seleciona um repositório específico.
- O servidor processa o arquivo `conf/svnserve.conf` do repositório e começa a tomar medidas para ratificar quaisquer políticas de autenticação e autorização nele definidas.
- Dependendo da situação e das políticas de autorização,
  - ao cliente pode ser permitido fazer requisições de forma anônima, sem mesmo precisar receber um desafio de autenticação, OU
  - o cliente pode ser desafiado para se autenticar a qualquer tempo, OU
  - se operando em “modo túnel”, o cliente irá declarar a si próprio como já tendo sido externamente autenticado.

Até o momento em que este livro estava sendo escrito, o servidor sabia apenas como enviar desafios de autenticação do tipo CRAM-MD5 [1] Essencialmente, o servidor envia uma pequena quantidade de dados

[1] Consulte a RFC 2195.

para o cliente. O cliente usa o algoritmo de hash MD5 para criar uma impressão digital dos dados e da senha combinados, então envia esta impressão digital como resposta. O servidor realiza a mesma computação com a senha armazenada para verificar se os resultados coincidem. *Em nenhum momento a senha atual é trafegada pela rede.*

E claro, também é possível para o cliente ser autenticado externamente por meio de um túnel agente, tal como o **SSH**. Neste caso, o servidor simplesmente examina o usuário com o qual está sendo executado, e o utiliza como nome de usuário autenticado. Para mais detalhes sobre isto, veja “Tunelamento sobre SSH”.

Como você já deve ter percebido, o arquivo `svnserve.conf` do repositório é o mecanismo central para controle das políticas de autenticação e autorização. O arquivo possui o mesmo formato que outros arquivos de configuração (veja “Runtime Configuration Area”): nomes de seção são marcados por colchetes (`[` e `]`), comentários iniciam por cerquilha (#), e cada seção contém variáveis específicas que podem ser definidas (`variável = valor`). Vamos conferir estes arquivos e aprender como usá-los.

Criar um arquivo 'users' e um domínio

Por hora, a seção `[general]` do `svnserve.conf` tem todas as variáveis que você precisa. Comece alterando os valores dessas variáveis: escolha um nome para um arquivo que irá conter seus nomes de usuários e senha, e escolha um domínio de autenticação:

```
[general]
password-db = userfile
realm = example domain
```

O domínio (`realm`) é um nome que você define. Ele informa aos clientes a que tipo de “espaço de nomes de autenticação” você está se conectando; o cliente Subversion o exibe no prompt de autenticação, e o utiliza como chave (junto com o hostname do servidor e a porta) para fazer cache de credenciais no disco (veja “Client Credentials Caching”). A variável `password-db` aponta para um arquivo em separado que contém uma lista de nomes de usuários e senhas, usando o mesmo formato familiar. Por exemplo:

```
[users]
harry = foopassword
sally = barpassword
```

O valor de `password-db` pode ser um caminho absoluto ou relativo para o arquivo de usuários. Para muitos administradores, é fácil manter o arquivo logo dentro da área `conf/` do repositório, juntamente com o `svnserve.conf`. Por outro lado, é possível que você queira ter dois ou mais repositórios compartilhando o mesmo arquivo de usuários; neste caso, o arquivo provavelmente deve ficar em um local mais público. Repositórios que compartilhem o arquivo de usuários também devem ser configurados para ter um mesmo domínio, uma vez que a lista de usuários essencialmente define um domínio de autenticação. Onde quer que este arquivo esteja, certifique-se de definir as permissões de leitura e escrita adequadamente. Se você sabe com qual(is) usuário(s) o **svnserve** irá rodar, restrinja o acesso de leitura ao arquivo de usuários conforme necessário.

Definindo controles de acesso

Há ainda mais duas variáveis para definir no arquivo `svnserve.conf`: elas determinam o que os usuários não autenticados (anônimos) e os usuários autenticados têm permissão de fazer. As variáveis `anon-access` e `auth-access` podem ser definidas para os valores `none`, `read`, ou `write`. Atribuindo o valor `none` você proíbe tanto a leitura quanto a escrita; com `read` você permite acesso somente leitura ao repositório, enquanto que `write` permite acesso completo de leitura/escrita ao repositório. Por exemplo:

```
[general]
password-db = userfile
realm = example domain

# usuários anônimos pode apenas ler o repositório
anon-access = read

# usuários autenticados podem tanto ler quanto escrever
auth-access = write
```

De fato, as configurações deste exemplo são os valores default para as variáveis, você poderia esquecer de definí-las. Se você quer ser ainda mais conservador, você pode bloquear o acesso anônimo completamente:

```
[general]
password-db = userfile
realm = example realm

# usuários anônimos não são permitidos
anon-access = none

# usuários autenticados podem tanto ler quanto escrever
auth-access = write
```

O processo servidor não entende apenas esta “restrição” no controle de acesso ao repositório, mas também restrições de acesso mais granularizadas definidas para arquivos ou diretórios específicos dentro do repositório. Para usar este recurso, você precisa criar um arquivo contendo regras mais detalhadas, e então definir o valor da variável `authz-db` para o caminho que o aponte:

```
[general]
password-db = userfile
realm = example realm

# Regras de acesso para locais específicos
authz-db = authzfile
```

A sintaxe do arquivo `authzfile` é discutida em mais detalhes em “Path-Based Authorization”. Atente que a variável `authz-db` não é mutuamente exclusiva com as variáveis `anon-access` e `auth-access`; se todas elas estiverem definidas ao mesmo tempo, então *todas* as regras devem ser satisfeitas antes que o acesso seja permitido.

Tunelamento sobre SSH

A autenticação interna do **svnserve** pode ser bastante útil, pois evita a necessidade de se criar contas reais no sistema. Por outro lado, alguns administradores já possuem frameworks de autenticação com SSH bem estabelecidos em funcionamento. Nestas situações, todos os usuários do projeto devem já ter contas no sistema e a possibilidade “dar um SSH” para acessar a máquina servidora.

É fácil usar o SSH juntamente com o **svnserve**. O cliente simplesmente usa o esquema `svn+ssh://` na URL para conectar:

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harry@host.example.com's password:  *****

foo
bar
baz
…
```

Neste exemplo, o cliente Subversion está invocando um processo **ssh** local, conectando-se a `host.example.com`, autenticando-se como usuário `harry`, então disparando um processo **svnserve** particular na máquina remota rodando como o usuário `harry` O comando **svnserve** está sendo invocado através em modo túnel (`-t`) e seu protocolo de rede está sendo "tunelado" pela conexão criptografada pelo **ssh**, o túnel agente. O **svnserve** sabe que está sendo executando pelo usuário `harry`, e se o cliente executar um commit, o nome do usuário autenticado será usado como autor da nova revisão.

A coisa importante a compreender aqui é que o cliente Subversion *não* está se conectando a um daemon **svnserve** em execução. Este método de acesso não requer um daemon, nem tampouco percebe se há algum daemon presente. Este método se baseia totalmente na capacidade do **ssh** de executar um processo **svnserve** temporário, que então termina quando a conexão de rede é fechada.

Ao usar URLs `svn+ssh://` para acessar um repositório, lembre-se que é o programa **ssh** que está solicitando autenticação, e *não* o programa cliente **svn**. Isto quer dizer que aqui não há cache automático de senhas acontecendo (veja "Client Credentials Caching"). O cliente Subversion quase sempre faz múltiplas conexões ao repositório, apesar de que os usuários normalmente não percebem isto devido a este recurso de cache de senhas. Ao usar URLs `svn+ssh://`, entretanto, os usuários podem ser incomodados repetidamente pelo **ssh** solicitando senhas a cada conexão que inicie. A solição é usar uma ferramenta separada para cache de senhas do SSH como o **ssh-agent** em um sistema Unix-like, ou o **pageant** no Windows.

Quando executada sobre um túnel, a autorização é principalmente controlada pelas permissões do sistema operacional para os arquivos da base dados do repositório; o que é praticamente o mesmo como se Harry estivesse accesando o repositório diretamente através de uma URL `file://`. Se múltiplos usuários no sistema vão acessar o repositório diretamente, você pode querer colocá-los num mesmo grupo, e você precisará ter cuidado com as umasks. (Não deixe de ler "Supporting Multiple Repository Access Methods".) Mas mesmo no caso do tunelamento, o arquivo `svnserve.conf` ainda pode ser usado para bloquear acesso, simplesmente definindo `auth-access = read` ou `auth-access = none`. [2]

Você pode ter pensado que essa história de tunelamento com SSH acabou por aqui, mas não. O Subversion permite que você crie comportamentos específicos para o modo túnel em seu arquivo `config` de execução (veja "Runtime Configuration Area"). Por exemplo, suponha que você queira usar o RSH ao invés do SSH[3]. Na seção `[tunnels]` do seu arquivo `config`, simplesmente defina algo parecido com isto:

```
[tunnels]
rsh = rsh
```

E agora, você pode usar este nova definição de túnel usando um esquema de URL que casa com o nome de sua nova variável: `svn+rsh://host/path`. Ao usar o novo esquema de URL, o cliente Subversion atualmente vai ser executado pelo comando **rsh host svnserve -t** por trás dos panos. Se você incluir

[2] Perceba que usar qualquer tipo de controle de acesso através do **svnserve** acaba não fazendo muito sentido; o usuário sempre possui acesso direto à base de dados do repositório.
[3] Atualmente nós não recomendamos isto, uma vez que o RSH é sabidamente menos seguro que o SSH.

um nome de usuário na URL (por exemplo, `svn+rsh://username@host/path`) o cliente também vai incluí-lo em seu comando (**rsh username@host svnserve -t**). Mas você pode definir novos esquemas de tunelamento que sejam muito mais inteligentes que isto:

```
[tunnels]
joessh = $JOESSH /opt/alternate/ssh -p 29934
```

Este exemplo demonstra uma porção de coisas. Primeiro, ele mostra como fazer o cliente do Subversion executar um binário de tunelamento bem específico (este que está localizado em `/opt/alternate/ssh`) com opções específicas. Neste caso, acessando uma URL `svn+joessh://` deveria invocar o binário SSH em questão com `-p 29934` como argumentos—útil se você quer que o programa do túnel se conecte a uma porta não-padrão.

Segundo, esse exemplo mostra como definir uma variável de ambiente personalizada que pode sobrescrever o nome do programa de tunelamento. Definir a variável de ambiente `SVN_SSH` é uma maneira conveniente de sobrescrever o túnel agente SSH padrão. Mas se você precisar fazer sobrescrita diversas vezes para diferentes servidores, cada um talvez contactando uma porta diferente ou passando diferentes conjuntos de opções para o SSH, você pode usar o mecanismo demonstrado neste exemplo. Agora se formos definir a variável de ambiente `JOESSH`, seu valor irá sobrescrever o valor inteiro da variável túnel —**$JOESSH** deverá ser executado ao invés de **/opt/alternate/ssh -p 29934**.

Dicas de configuração do SSH

Não é possível apenas controlar a forma como o cliente invoca o **ssh**, mas também controlar o comportamento do **ssh** em sua máquina servidora. Nesta seção, vamos mostrar como controlar exatamente o comando **svnserve** executado pelo **sshd**, além de como ter múltiplos usuários compartilhando uma única conta no sistema.

Configuração inicial

Para começar, localize o diretório home da conta que você vai usar para executar o **svnserve**. Certifique-se de que a conta tenha um par de chaves pública/privada instalado, e que o usuário consiga ter acesso ao sistema por meio de autenticação com chave pública. A autenticação por senha não irá funcionar, já que todas as seguintes dicas sobre SSH estão relacionadas com o uso do arquivo `authorized_keys` do SSH.

Se ainda não existir, crie o arquivo `authorized_keys` (no Unix, tipicamente em `~/.ssh/authorized_keys`). Cada linha neste arquivo descreve uma chave pública com permissão para conectar. As linhas são comumente da forma:

```
ssh-dsa AAAABtce9euch… user@example.com
```

O primeiro campo descreve o tipo da chave, o segundo campo é a chave em si codificada em base64, e o terceiro campo é um comentário. Porém, é pouco conhecido o fato de que a linha inteira pode ser precedida por um campo `command`:

```
command="program" ssh-dsa AAAABtce9euch… user@example.com
```

Quando o campo `command` é definido, o daemon SSH irá rodar o programa descrito em vez da típica invocação **svnserve -t** que o cliente Subversion está esperando. Isto abre espaço para um conjunto de truques no lado do servidor. Nos exemplos a seguir, abreviamos a linha do arquivo para:

```
command="program" TIPO CHAVE COMENTÁRIO
```

Controlando o comando invocado

Pelo fato de podermos especificar o comando executado no lado do servidor, é fácil determinar um binário **svnserve** específico para rodar e para o qual passar argumentos extras:

```
command="/caminho/do/svnserve -t -r /virtual/root" TIPO CHAVE
COMENTÁRIO
```

Neste exemplo, `/caminho/do/svnserve` pode ser um script específico que encapsule uma chamada ao **svnserve** e que configure o umask (veja “Supporting Multiple Repository Access Methods”). O exemplo também mostra como prender o **svnserve** em um diretório raiz virtual, tal como sempre ocorre ao se executar o **svnserve** como um processo daemon. Isto pode ser feito tanto para restringir o acesso a partes do sistema, ou simplesmente para facilitar para que o usuário não tenha de digitar um caminho absoluto na URL `svn+ssh://`.

Também é possível ter múltiplos usuários compartilhando uma única conta. Ao invés de criar uma conta em separado no sistema para cada usuário, gere um par de chaves pública/privada para cada pessoa. Então ponha cada chave pública dentro do arquivo `authorized_keys`, uma por linha, e utilize a opção `--tunel-user`:

```
command="svnserve -t --tunnel-user=harry" TIPO1 CHAVE1 harry@example.com
command="svnserve -t --tunnel-user=sally" TIPO2 CHAVE2 sally@example.com
```

Este exemplo permite que Harry e Sally se conectem à mesma conta por meio da autenticação de chave pública. Cada um deles tem um comando específico a ser executado; a opção `--tunnel-user` diz ao **svnserve -t** para assumir que o argumento informado é um nome de usuário autenticado. Não fosse pelo `--tunnel-user` pareceria como se todos os commits viéssem de uma única mesma conta de sistema compartilhada.

Uma última palavra de precaução: dando acesso ao usuário através de uma chave pública numa conta compartilhada deve permitir ainda outras formas de acesso por SSH, ainda que você já tenha definido o valor do campo `command` no `authorized_keys`. Por exemplo, o usuário pode ainda ter acesso a um shell através do SSH, ou ser capaz de executar o X11 ou outro tipo de redirecionamento de portas através do servidor. Para dar ao usuário o mínimo de permissão possível, você pode querer especificar algumas opções restritivas imediatamente após o `command`:

```
command="svnserve -t --tunnel-user=harry",no-port-forwarding,\
         no-agent-forwarding,no-X11-forwarding,no-pty \
         TIPO1 CHAVE1 harry@example.com
```

httpd, o servidor HTTP Apache

O servidor Apache HTTP é um servidor de rede “robusto” do qual o Subversion pode tirar proveito. Por meio de um módulo específico, o **httpd** torna os repositórios Subversion disponíveis aos clientes por meio do protocolo WebDAV/DeltaV, que é uma extensão ao HTTP 1.1 (consulte http://www.webdav.org/ para mais informações). Este protocolo usa o onipresente protocolo HTTP, que é o coração da World Wide Web, e adiciona capacidades de escrita—especificamente, escrita sob controle de versão. O resultado é um sistema robusto, padronizado e que está convenientemente empacotado como uma parte do software Apache 2.0, é suportado por vários sistemas operacionais e produtos de terceiros, e não requer administradores de rede para abrir nenhuma outra porta específica. [4] Apesar de o servidor Apache-

[4] Eles realmente detestam fazer isso.

Subversion ter mais recursos que o **svnserve**, ele também é um pouco mais difícil de configurar. Um pouco mais de complexidade é o preço a se pagar por um pouco mais de flexibilidade.

Muitas das discussões a seguir incluem referências a diretivas de configuração do Apache. Por mais que sejam dados exemplos de uso destas diretivas, descrevê-las por completo está fora do escopo deste capítulo. A equipe do Apache mantém excelente documentação, disponível publicamente no seu website em http://httpd.apache.org. Por exemplo, uma referência geral das diretivas de configuração está localizada em http://httpd.apache.org/docs-2.0/mod/directives.html.

Além disso, quando você faz alteração na configuração do seu Apache, é bem provável que erros sejam cometidos em algum ponto. Se você ainda não está ambientado com o subsistema de logs do Apache, você deveria familiarizar-se com ele. Em seu arquivo `httpd.conf` estão as diretivas que especificam os caminhos em disco dos logs de acesso e de erros gerados pelo Apache (as diretivas `CustomLog` e `ErrorLog`, respectivamente). O módulo mod_dav_svn do Subversion também usa a interface de logs de erro do Apache. A qualquer momento você pode verificar o conteúdo destes arquivos, buscando informação que pode revelar a causa de um problema que não seria claramente percebido de outra forma.

Por que o Apache 2?

Se você é um administrador de sistemas, é bem provável que você já esteja rodando o servidor web Apache e tenha alguma experiência anterior com ele. No momento em que este livro era escrito, o Apache 1.3 é de longe a versão mais popular do Apache. O mundo tem sido um tanto lento para atualizar para o Apache da série 2.X por várias razões: algumas pessoas têm medo da mudança, especialmente mudança em algo tão crítico como um servidor web. Outras pessoas dependem de módulos de plug-in que só funcionam sobre a API do Apache 1.3, e estão aguardando que estes sejam portados para a 2.X. Qualquer que seja a razão, muitas pessoas começam a se preocupar quando descobrem que o módulo Apache do Subversion é escrito especificamente para a API do Apache 2.

A resposta adequada a este problema é: não se preocupe com isto. É fácil executar o Apache 1.3 e o Apache 2 lado a lado; simplesmente instale-os em locais separados, e use o Apache 2 como um servidor Subversion dedicado que execute em outra porta que não a 80. Os clientes podem acessar o repositório inserindo o número da porta na URL:

```
$ svn checkout http://host.example.com:7382/repos/project
…
```

Pré-requisitos

Para disponibilizar seus repositórios em rede via HTTP, você basicamente necessita de quatro componentes, disponíveis em dois pacotes. Você vai precisar do Apache **httpd** 2.0, do módulo DAV **mod_dav** que já vem com ele, do Subversion, e do **mod_dav_svn**, módulo que provê acesso ao sistema de arquivos distribuído junto com o Subversion. Uma vez que você tenha todos estes componentes, o processo de disponibilizar seus repositórios em rede é tão simples quanto:

- executar o httpd 2.0 com o módulo mod_dav,
- instalar o plugin mod_dav_svn no mod_dav, que utiliza as bibliotecas do Subversion para acessar o repositório e
- configurar seu arquivo `httpd.conf` para exportar (ou expor) o repositório.

Você pode realizar os primeiros dois passos tanto compilando o **httpd** e o Subversion a partir do código-fonte, ou instalando os pacotes binários pré-construídos para o seu sistema. Para informações mais atualizadas sobre como compilar o Subversion para uso com o servidor Apache HTTP, além de como compilar e configurar o próprio Apache para este propósito, veja o arquivo `INSTALL` na pasta de mais alto nível na árvore do código-fonte do Subversion.

Configuração Básica do Apache

Tendo instalado todos os componentes necessários em seu sistema, tudo o que resta é a configuração do Apache por meio de seu arquivo `httpd.conf`. Indique ao Apache para carregar o módulo mod_dav_svn usando a diretiva `LoadModule`. Esta diretiva deve preceder a qualquer outro item de configuração relacionado ao Subversion. Se seu Apache foi instalado usando sua estrutura padrão, seu módulo **mod_dav_svn** deve estar instalado no subdiretório `modules` do local de instalação do Apache (comumente em `/usr/local/apache2`). A diretiva `LoadModule` tem uma sintaxe simples, mapeando um nome de módulo ao local em disco da correspondente biblioteca compartilhada:

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

Note que se o **mod_dav** foi compilado como um objeto compartilhado (ao invés de ter sido lincado diretamente ao binário **httpd**), você vai precisar de uma declaração `LoadModule` para ele, também. Assegure-se de que sua declaração venha antes da linha **mod_dav_svn**:

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

Em outra parta de seu arquivo de configuração, você agora vai dizer ao Apache onde você mantém seu repositório (ou repositórios) Subversion. A diretiva `Location` tem uma notação parecida com a de XML, começando com uma tag de abertura, e terminando com uma tag de fechamento, com várias outras diretivas de configuração no meio. O propósito da diretiva `Location` é instruir o Apache a fazer algo especial quando manipular requisições que sejam direcionadas a uma certa URL ou uma de suas filhas. No caso do Subversion, quer que o Apache simplesmente desconsidere URLs que apontem para recursos sob controle de versão na camada DAV. Você pode instruir o Apache a delegar a manipulação de todas as URL em que cuja parte do caminho (a parte da URL após o nome do servidor e do número de porta opcional) comece com `/repos/` para um provedor DAV cujo repositório está localizado em `/caminho/absoluto/do/repositorio` usando a seguinte sintaxe do `httpd.conf`:

```
<Location /repos>
  DAV svn
  SVNPath /caminho/absoluto/do/repositório
</Location>
```

Se você pretende disponibilizar múltiplos repositórios Subversion que se encontrem sob um mesmo diretório-pai em seu disco local, você pode usar uma diretiva alternativa, a diretiva `SVNParentPath`, para indicar este diretório-pai comum. Por exemplo, se você sabe que você vai criar múltiplos repositórios Subversion num diretório `/usr/local/svn` que poderia ser acessado a partir de URLs como `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2`, e assim por diante, você pode usar a sintaxe de configuração do `httpd.conf` do exemplo a seguir:

```
<Location /svn>
  DAV svn
```

```
  # qualquer URL "/svn/foo" vai mapear para um repositório /usr/local/svn/foo
  SVNParentPath /usr/local/svn
</Location>
```

Usando a sintaxe anterior, o Apache vai delegar a manipulação de todas as URLs cuja a parte do caminho comece com `/svn/` para o provedor DAV do Subversion, que então vai assumir que quaisquer itens no diretório especificado pela diretiva `SVNParentPath` atualmente são repositórios Subversion. Esta é uma sintaxe particularmente conveniente para isto, ao contrário do uso da diretiva `SVNPath`, você não tem que reiniciar o Apache para criar e acessar via rede os novos repositórios.

Certifique-se de que ao definir seu novo `Location`, este não se sobreponha a outros Locations exportados. Por exemplo, se seu `DocumentRoot` principal estiver exportado para `/www`, não exporta um repositório Subversion em `<Location /www/repos>`. Se vier uma requisição para a URL `/www/repos/foo.c`, o Apache não vai saber se deve procurar pelo arquivo `repos/foo.c` no `DocumentRoot`, ou se delega ao **mod_dav_svn** para que este retorne `foo.c` a partir do repositório Subversion. O resultado quase sempre é um erro de servidor no formato `301 Moved Permanently`.

Nomes de Servidores e a Requisição COPY

Subversion faz uso da requisição do tipo `COPY` para executar cópias de arquivos e diretórios do lado do servidor. Como parte da verificação de integridade é feita pelos módulos do Apache, espera-se que a origem da cópia esteja localizada na mesma máquina que o destino da cópia. Para satisfazer este requisito, você vai precisar dizer ao mod_dav o nome do hostname usado em seu servidor. Geralmente, você pode usar a diretiva `ServerName` no `httpd.conf` para fazer isto.

```
ServerName svn.example.com
```

Se você está usando o suporte a hosts virtuais do Apache através da diretiva `NameVirtualHost`, você pode precisar usar a diretiva `ServerAlias` para especificar nomes adicionais pelos quais seu servidor também é conhecido. Mais uma vez, verifique a documentação do Apache para mais detalhes.

Neste estágio, você deve considerar maciçamente a questão das permissões. Se você já estiver executando o Apache há algum tempo como seu servidor web regular, você provavelmente já tem uma porção de conteúdos—páginas web, scripts e similares. Estes itens já devem ter sido configurados com um conjunto de permissões que lhes possibilitam trabalhar com o Apache, ou mais adequadamente, que permite ao Apache trabalhar com estes arquivos. O Apache, quando usado como um servidor Subversion, também vai precisar das permissões corretas para ler e escrever em seu repositório Subversion.

Você vai precisar determinar uma configuração do sistema de permissões que satisfaça os requisitos do Subvresion sem causar problemas em nenhuma página ou script previamente instalado. Isto pode significar alterar as permissões de seu repositório Subversion para corresponder àquelas em uso por outras coisas que o Apache lhe serve, ou pode significar usar as diretivas `User` e `Group` no `httpd.conf` para especificar que o Apache deve executar como o usuário e grupo de que seja dono do seu repositório Subversion. Não há apenas uma maneira correta de configurar suas permissões, e cada administrador vai ter diferentes razões para fazer as coisas de uma dada maneira. Apenas tenha cuidado pois problemas de permissão são talvez a falha mais comum ao se configurar o repositório Subversion para uso com o Apache.

Opções de Autenticação

Neste ponto, se você configurou o `httpd.conf` para conter algo como

```
<Location /svn>
```

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

…então seu repositório está “anonimamente” acessível ao mundo. Até que você configure algumas políticas de autenticação e autorização, os repositórios Subversion que você disponibilizar através da diretiva `Location` vão estar globalmente acessíveis a qualquer um. Em outras palavras,

- qualquer pessoa pode usar seu cliente Subversion para dar checkout numa cópia de trabalho de uma URL do repositório (ou qualquer de seus subdiretórios),
- qualquer pessoa pode navegar interativamente pela última revisão do repositório, simplesmente direcionando seu navegador web para a URL do repositório, e
- qualquer pessoa pode dar commit no repositório.

Certamente, você pode já ter definido um hook script para previnir commits (veja “Implementing Repository Hooks”). Mas conforme você for lendo, verá que também é possível usar os métodos inerentes ao Apache para restringir o acesso de maneiras específicas.

Autenticação HTTP Básica

A maneira mais fácil de autenticar um cliente é através do mecanismo de autenticação HTTP Basic, o qual simplesmente usa um nome de usuário e senha para verificar se o usuário é quem ele diz ser. O Apache provê um utilitário **htpasswd** para gerenciar a lista de nomes de usuários e senhas aceitáveis. Vamos permitir o acesso a commit a Sally e Harry. Primeiro, precisamos adicioná-los ao arquivo de senhas.

```
$ ### Primeira vez: use -c para criar o arquivo
$ ### Use -m para usar cripgrafia MD5 na senha, o que é mais seguro
$ htpasswd -cm /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth-file sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

A seguir, você precisa adicionar mais algumas diretivas do `httpd.conf` dentro de seu bloco `Location` para indicar ao Apache o que fazer com seu novo arquivo de senhas. A diretiva `AuthType` especifica o tipo de sistema de autenticação a usar. Neste caso, vamos especificar o sistema de autenticação `Basic`. `AuthName` é um nome arbitrário que você dá para o domínio da autenticação. A maioria dos navegadores web vai mostrar este nome da caixa de diálogo quando o navegador estiver perguntando ao usuário por seu nome e senha. Finalmente, use a diretiva `AuthUserFile` para especificar a localização do arquivo de senhas que você criou usando o comando **htpasswd**.

Depois de adicionar estas três diretivas, seu bloco `<Location>` deve ser algo parecido com isto:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
```

```
  AuthUserFile /etc/svn-auth-file
</Location>
```

Este bloco `<Location>` ainda não está completo, e não fará nada de útil. Está meramente dizendo ao Apache que sempre que uma autorização for requerida, o Apache deve obter um nome de usuário e senha do cliente Subversion. O que está faltando aqui, entretanto, são diretivas que digam ao Apache *quais* tipos de requisições do cliente necessitam de autorização. Toda vez que uma autorização for requerida, o Apache também irá exigir uma autorização. A coisa mais simples a se fazer é proteger todas as requisições. Adicionar `Require valid-user` indica ao Apache que todas as requisições requerem um usuário autenticado:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

Não deixe de ler a próxima seção (“Opções de Autorização”) para mais detalhes sobre a diretiva `Require` e outras formas de definir políticas de autorização.

Uma palavra de alerta: senhas de autenticação HTTP Basic trafegam na rede de forma bem parecida com texto plano, e portanto são extremamente inseguras. Se você está preocupado com a privacidade de suas senhas, pode ser melhor usar algum tipo de criptografia SSL para que o cliente se autentique através de `https://` ao invés de `http://`; no mínimo, você pode configurar o Apache para usar um certificado de servidor auto-assinado. [5] Consulte a documentação do Apache (e a documentação do OpenSSL) sobre como fazê-lo.

Gerência de Certificados SSL

Negócios que precisam expor seus repositórios para acesso por fora do firewall corporativos devem estar conscientes da possibilidade de que pessoas não autorizadas podem “vasculhar” seu tráfego de rede. SSL faz com que esse tipo de análise indesejada tenha menor possibilidade de resultar na exposição de dados sensíveis.

Se um cliente Subversion é compilado para usar OpenSSL, então ele ganha a habilidade de falar com um servidor Apache através de URLs `https://`. A biblioteca Neon usada pelo cliente Subversion não é apenas capaz de verificar certificados de servidor, mas também de prover certificados de cliente quando necessário. Quando cliente e o servidor trocam certificados SSL e se autenticam mutuamente com sucesso, toda a comunicação subseqüente é criptografada por meio de uma chave de sessão.

Está fora do escopo deste livro descrever como gerar certificados de cliente e de servidor, e como configurar o Apache para usá-los. Muitos outros livros, incluindo a própria documentação do Apache, descrevem esta tarefa. Mas o que *pode* ser coberto aqui é como gerenciar os certificados de cliente e servidor a partir de um cliente Subversion ordinário.

Ao se comunicar com o Apache através de `https://`, um cliente Subversion pode receber dois diferentes tipos de informação:

- um certificado de servidor

[5] Por mais que certificados auto-assinados ainda sejam vulneráveis ao “ataque do homem do meio”, tal ataque é muito mais difícil de ser executado por um observador casual, do que o comparado a vasculhar senhas desprotegidas.

- uma demanda por um certificado de cliente

Se o cliente recebe um certificado de servidor, ele precisa verificar se confia no certificado: o servidor é quem ele realmente diz ser? A biblioteca OpenSSL faz isto examinando o assinador do certificado de servidor ou *autoridade certificadora* (AC). Se o OpenSSL for incapaz de confiar automaticamente na AC, ou se algum outro problema ocorrer (como o certificado ter expirado ou o hostname não corresponder), o cliente de linha de comando do Subversion vai lhe perguntar se você quer confiar no certificado do servidor de qualquer maneira:

```
$ svn list https://host.example.com/repos/project

Error validating server certificate for 'https://host.example.com:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: host.example.com
 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56 2006 GMT
 - Issuer: CA, example.com, Sometown, California, US
 - Fingerprint: 7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b

(R)eject, accept (t)emporarily or accept (p)ermanently?
```

Este diálogo deve parecer familiar: é essencialmente a mesma pergunta que você provavelmente já tenha visto vindo de seu navegador web (o qual é apenas outro cliente HTTP como o cliente Subversion). Se você escolher a opção (p)ermanente, o certificado do servidor será armazenado em cache em sua área privativa de tempo de execução, `auth/`, da mesma forma que seu nome de usuário e senha são armazenados (veja “Client Credentials Caching”). Uma vez armazenado, o Subversion automaticamente lembrará de confiar neste certificado em negociações futuras.

Seu arquivo em tempo de execução `servers` também lhe possibilita fazer seu cliente Subversion confiar automaticamente em ACs específicas, tanto de forma global quanto discriminadamente por `ssl-authority-files` para uma lista de certificados das ACs com codificação PEM separados por ponto e vírgula:

```
[global]
ssl-authority-files = /caminho/do/CAcert1.pem;/caminho/do/CAcert2.pem
```

Muitas instalações OpenSSL também possuem um conjunto pré-definido de ACs “padrão” que são quase que universalmente confiáveis. Para fazer o cliente Subversion confiar automaticamente nestas autoridades padrão, atribua o valor da variável `ssl-trust-default-ca` para `true`.

Ao conversar com o Apache, o cliente Subversion pode também receber um desafio para um certificado de cliente. O Apache está solicitando que o cliente identifique a si próprio: o cliente é quem ele realmente diz ser? Se tudo prosseguir corretamente, o cliente Subversion manda de volta um certificado privativo assinado por uma AC na qual o Apache confia. O certificado do cliente é comumente armazenado em disco num formato criptografado, protegido por uma senha local. Quando o Subversion recebe este desafio, ele solicitará a você tanto o caminho do certificado quando a senha que o protege:

```
$ svn list https://host.example.com/repos/project

Authentication realm: https://host.example.com:443
Client certificate filename: /caminho/do/meu/cert.p12
Passphrase for '/caminho/do/meu/cert.p12':  ********
```

...

Note que o certificado do cliente é um arquivo no formato “p12”. Para usar um certificado de cliente com o Subversion, este deve estar no formato PKCS#12, que é um padrão portável. A maioria dos navegadores já são capazes de importar e exportar certificados neste formato. Outra opção é usar as ferramentas de linha de comando do OpenSSL para converter certificados existentes para PKCS#12.

Novamente, o arquivo em tempo de execução `servers` permite a você automatizar este desafio numa configuração baseada em host. Qualquer um ou mesmo os dois tipos de informação podem estar descritos em variáveis em tempo de execução:

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = /caminho/do/meu/cert.p12
ssl-client-cert-password = somepassword
```

Uma vez que você tenha definido as variáveis `ssl-client-cert-file` e `ssl-client-cert-password`, o cliente Subversion pode automaticamente responder a um desafio de certificado de cliente sem solicitar nada a você. [6]

Opções de Autorização

Até este ponto, você configurou a autenticação, mas não a autorização. O Apache é capaz de desafiar cliente e confirmar identidades, mas não está sendo dito como permitir ou restringir acesso a clientes que possuam estas identidades. Esta seção descreve duas estratégias para controle de acesso de seus repositórios.

Controle de Acesso Geral

A maneira mais simples de controle de acesso é autorizar certos usuários ou para acesso somente leitura a um repositório, ou acesso leitura/escrita a um repositório.

The simplest form of access control is to authorize certain users for either read-only access to a repository, or read/write access to a repository.

Você pode restringir acesso em todas as operações de repositório adicionando a diretiva `Require valid-user` ao seu bloco `<Location>`. Usando nosso exemplo anterior, isto significa que apenas aos clientes que afirmaram ser `harry` ou `sally`, e forneceram a senha correta para seus respectivos nomes de usuário será permitido fazer qualquer coisa com o repositório Subversion:

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

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users
```

[6] Pessoas mais preocupadas com segurança podem não querer armazenar a senha do certificado de cliente no arquivo `servers` em tempo de execução.

```
  # apenas usuários autenticados podem acessar o repositório
  Require valid-user
</Location>
```

Algumas vezes você não precisa de regras tão rígidas. Por exemplo, o próprio repositório do código-fonte do Subversion em http://svn.collab.net/repos/svn permite a qualquer um no mundo executar tarefas somente-leitura no repositório (como verificar cópias de trabalho e navegar pelo repositório com um navegador web), mas restringe todas as operações de escrita a usuários autenticados. Para fazer este tipo de restrição seletiva, você pode usar as diretivas de configuração `Limit` e `LimitExcept`. Como a diretiva `Location`, estes blocos possuem tags de abertura e de fechamento, e você deve inserí-las dentro de seu bloco `<Location>`.

Os parâmetros presentes nas diretivas `Limit` e `LimitExcept` são tipos de requisições HTTP que são afetadas por aquele bloco. Por exemplo, se você quiser desabilitar todo o acesso a seu repositório exceto as operações somente-leitura atualmente suportadas, você deve usar a diretiva `LimitExcept`, passando os tipos de requisição `GET`, `PROPFIND`, `OPTIONS`, e `REPORT` como parâmetros. Então a já mencionada diretiva `Require valid-user` deve ser colocada dentro do bloco `<LimitExcept>` ao invés de apenas dentro do bloco `<Location>`.

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

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users

  # Para quaiquer operações além destas, requeira um usuário autenticado
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

Estes são apenas uns poucos exemplos simples. Para informação mais aprofundada sobre controle de acesso e a diretiva `Require`, dê uma olhada na seção `Security` da coleção de tutoriais da documentação do Apache em http://httpd.apache.org/docs-2.0/misc/tutorials.html.

Controle de Acesso por Diretório

É possível configurar permissões mais granularizadas usando um segundo módulo do Apache httpd, **mod_authz_svn**. Este módulo captura várias URLs opacas passando do cliente para o servidor, pede ao **mod_dav_svn** para decodificá-las, e então possivelmente restringe requisições baseadas em políticas de acesso definidas em um arquivo de configuração.

Se você compilou o Subversion a partir do código-fonte, o **mod_authz_svn** é construído automaticamente e instalado juntamente com o **mod_dav_svn**. Muitas distribuições binárias também o instalam automaticamente. Para verificar se está instalado corretamente, assegure-se de que ele venha logo depois da diretiva `LoadModule` do **mod_dav_svn** no `httpd.conf`:

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

```
LoadModule authz_svn_module   modules/mod_authz_svn.so
```

Para ativar este módulo, você precisa configurar seu bloco `Location` para usar a diretiva `AuthzSVNAccessFile`, a qual especifica um arquivo contendo políticas de permissões para caminhos dentro de seus repositórios. (Logo, logo, vamos discutir o formato deste arquivo.)

O Apache é flexível, então você tem a opção de configurar seu bloco em um destes três padrões gerais. Para começar, escolha um destes três padrões de configuração. (Os exemplos abaixo são muito simples; consulte a documentação do próprio Apache para ter muito mais detalhes sobre as opções de autenticação e autorização do Apache.)

O bloco mais simples é permitir acesso abertamente a todo mundo. Neste cenário, o Apache nunca envia desafios de autenticação, então todos os usuários são tratados como “anonymous”.

Exemplo 6.1. Um exemplo de configuração para acesso anônimo.

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

  # nossa política de controle de acesso
  AuthzSVNAccessFile /caminho/do/arquivo/access
</Location>
```

No outro extremo da escala de paranóia, você pode configurar seu bloco para requisitar autenticação de todo mundo. Todos os clientes devem prover suas credenciais para se identificarem. Seu bloco requer autenticação incondicional através da diretiva `Require valid-user`, e define meios para autenticação.

Exemplo 6.2. Um exemplo de configuração para acesso autenticado.

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

  # nossa política de controle de acesso
  AuthzSVNAccessFile /caminho/do/arquivo/access

  # apenas usuários autenticados podem acessar o repositório
  Require valid-user

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users
</Location>
```

Um terceiro padrão bem popular é permitir uma combinação de acesso autenticado e anônimo. Por exemplo, muitos administradores querem permitir usuários anônimos a ler certos diretórios do repositório, mas querem que apenas usuários autenticados leiam (ou escrevam) em áreas mais sensíveis. Nesta configuração, todos os usuários começam acessando o repositório anonimamente. Se sua política de controle de acesso solicitar um nome de usuário real em algum ponto, o Apache vai solicitar autenticação

para o cliente. Para fazer isto, você usa ambas as diretivas `Satisfy Any` e `Require valid-user` em conjunto.

Exemplo 6.3. Um exemplo de configuração para acesso misto autenticado/ anônimo.

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

  # nossa política de controle de acesso
  AuthzSVNAccessFile /caminho/do/arquivo/access

  # tente o acesso anônimo primeiro, ajuste para autenticação
  # real se necessário
  Satisfy Any
  Require valid-user

  # como autenticar um usuário
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /caminho/do/arquivo/users
</Location>
```

Tendo se decidido por um destes três modelos básicos de `httpd.conf`, você então precisa criar seu arquivo contendo regras de acesso para determinados caminhos dentro do repositório. Isto é descrito em “Path-Based Authorization”.

Desabilitando Verificação baseada em Caminhos

O módulo **mod_dav_svn** realiza bastante trabalho para se assegurar de que os dados que você marcou como “unreadable” não sejam corrompidos acidentalmente. Isto significa que ele precisa monitorar de perto todos os caminhos e conteúdos de arquivos retornados por comandos como **svn checkout** ou **svn update**. Se estes comandos encontram um caminho que não seja legível de acordo com alguma política de autorização, então, tipicamente, o caminho como um todo é omitido. No caso de histórico ou acompanhamento de renomeações—p.ex. ao se executar um comando como **svn cat -r OLD foo.c** em um arquivo que foi renomeado há bastante tempo—o acompanhamento da renomeação irá simplesmente parar se um dos nomes anteriores do objeto for determinado como sendo de leitura restrita.

Toda esta verificação de caminhos algumas vezes pode ser um pouco custosa, especialmente no caso do **svn log**. Ao obter uma lista de revisões, o servidor olha para cada caminho alterado em cada revisão e verifica sua legibilidade. Se um caminho ilegível é descoberto, então ele é omitido da lista de caminhos alterados da revisão (normalmente vista com a opção `--verbose`), e a mensagem de log completa é suprimida. Desnecessário dizer que isto pode consumir bastante tempo em revisões que afetam um grande número de arquivos. Este é o custo da segurança: mesmo se você ainda nunca tiver configurado um módulo como **mod_authz_svn**, o módulo **mod_dav_svn** ainda fica solicitando para que o Apache **httpd** execute verificações de autorização em cada caminho. O módulo **mod_dav_svn** não faz idéia de que módulos de autorização estão instalados, então tudo o que ele pode fazer é solicitar que o Apache invoque todos os que podem estar presentes.

Por outro lado, também há um recurso de válvula de escape, o qual permite a você troque características de segurança por velocidade. Se você não estiver impondo nenhum tipo de autorização por diretório (i.e.

não está usando o módulo **mod_authz_svn** ou similar), então você pode desabilitar toda esta checagem de caminhos. Em seu arquivo `httpd.conf`, use a diretiva `SVNPathAuthz`:

Exemplo 6.4. Desabilitando verificações de caminho como um todo

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

  SVNPathAuthz off
</Location>
```

A diretiva `SVNPathAuthz` é definida como “on” por padrão. Quando definida para “off”, toda a checagem de autorização baseada em caminhos é desabilitada; o **mod_dav_svn** pára de solicitar checagem de autorização em cada caminho que ele descobre.

Facilidades Extras

Cobrimos a maior parte das opções de autenticação e autorização para o Apache e o mod_dav_svn. Mas há alguns outros poucos bons recursos que o Apache provê.

Navegação de Repositório

Um dos mais úteis benefícios de uma configuração do Apache/WebDAV para seu repositório Subversion é que as revisões mais recentes de seus arquivos e diretórios sob controle de versão ficam imediatamente disponíveis para visualização por meio de um navegador web comum. Como o Subversion usa URLs para identificar recursos sob controle de versão, estas URLs usadas para acesso ao repositório baseado em HTTP podem ser digitadas diretamente num navegador Web. Seu navegador vai emitir uma requisição HTTP `GET` para aquela URL, e se aquela URL representar um diretório ou arquivo sobre controle de versão, o mod_dav_svn irá responder com uma listagem de diretório ou com o conteúdo do arquivo.

Já que as URLs não contém nenhuma informação sobre quais versões do recurso você quer ver, o mod_dav_svn sempre irá responder com a versão mais recente. Esta funcionalidade tem um maravilhoso efeito colateral que é a possibilidade de informar URLs do Subversion a seus parceiros como referências aos documentos, e estas URLs sempre vão apontar para a versão mais recente do documento. É claro, você ainda pode usar URLs como hiperlinks em outros web sites, também.

[7] No início, ele chamava-se “ViewCVS”.

Posso ver revisões antigas?

Com um navegador web comum? Em uma palavra: não. Ao menos, não com o **mod_dav_svn** como sua única ferramenta.

Seu navegador web entende apenas HTTP padrão. Isso significa que ele apenas sabe como obter (GET) URLs públicas, as quais representam as últimas versõs dos arquivos e diretórios. De acordo com a especificação WebDAV/DeltaV, cada servidor define uma sintaxe de URL particular para versões mais antigas dos recursos, e esta sintaxe é opaca aos clientes. Para encontrar uma versão mais antiga de um arquivo, um cliente deve seguir um processo específico para “descobrir” a URL adequada; o procedimento envolve um conjunto de requisições PROPFIND do WebDAV e a compreensão de conceitos do DeltaV. Isto é algo que seu navegador web simplesmente não consegue fazer.

Então para responder à pergunta, a maneira óbvia de ver revisões antigas de arquivos e diretórios é pela passagem do argumento `--revision (-r)` para os comando **svn list** e **svn cat**. Para navegar em revisões antigas com seu navegador web, entretanto, você precisar usar software de terceiros. Um bom exemplo disto é o ViewVC (http://viewvc.tigris.org/). Originalmente o ViewVC foi escritp para exibir repositórios CVS pela web, [7] mas as versões mais recentes trabalham com repositórios Subversion, também.

Tipo MIME Adequado

Ao navegar em um repositório Subversion, o navegador web obtém um indício sobre como renderizar o conteúdo do arquivo consultando o cabeçalho `Content-Type:` retornado na resposta da requisição HTTP `GET`. O valor deste cabeçalho é o valor de um tipo MIME. Por padrão, o Apache vai indicar aos navegadores web que todos os arquivos do repositório são do tipo MIME “default”, usualmente o tipo `text/plain`. Isto pode ser frustrante, entretanto, se um usuário quiser que os arquivos do repositório sejam renderizados como algo com mais significado—por exemplo, seria ótimo que um arquivo `foo.html` pudesse ser renderizado como HTML na navegação.

Para fazer isto acontecer, você só precisa garantir que seus arquivos tenham o `svn:mime-type` adequadamente configurado. Isto é discutido em mais detalhes em “Tipo de Conteúdo do Arquivo”, a você ainda pode configurar seu cliente para anexar as propriedades `svn:mime-type` automaticamente aos arquivos que estejam entrando no repositório pela primeira vez; veja “Automatic Property Setting”.

Assim, em nosso exemplo, se alguém definiu a propriedade `svn:mime-type` para `text/html` no arquivo `foo.html`, então o Apache deve avisar adequadamente para que seu navegador web renderize o arquivo como HTML. Alguém também poderia anexar propriedades `image/*` de tipos mime para imagens, e fazendo isso, no final das contas obter um site web completo podendo ser visualizado diretamente do repositório! Geralmente não há problema em se fazer isto, desde que o website não contenha nenhum conteúdo gerado dinamicamente.

Personalizando a Aparência

Você normalmente vai fazer mais uso das URLs para arquivos versionados—afinal, é onde o conteúdo interessante tende a estar. Mas pode haver certas situações você pode precisar navegar na listagem de diretórios, no que você rapidamente irá notar que o HTML gerado para exibir estas listagens é muito básico, e certamente não pretende ser esteticamente agradável (ou mesmo interessante). Para possibilitar a personalização destas exibições de diretório, o Subversion provê um recurso de índice em XML. Uma única diretiva `SVNIndexXSLT` no bloco `Location` do seu `httpd.conf` vai orientar o mod_dav_svn a gerar saída XML ao exibir uma listagem de diretório, e referenciar uma folha de estilos XSLT à sua escolha:

```
<Location /svn>
```

```
  DAV svn
  SVNParentPath /usr/local/svn
  SVNIndexXSLT "/svnindex.xsl"
  …
</Location>
```

Usando a diretiva `SVNIndexXSLT` e uma folha de estilos criativa, você pode fazer com que suas listagens de diretórios sigam os esquemas de cores e imagens usados em outras partes de seu website. Ou, se você preferir, você pode usar folhas de estilo de exemplo que já vêm no diretório `tools/xslt` dos fontes do Subversion. Tenha em mente que o caminho informado para o diretório em `SVNIndexXSLT` atualmente é um caminho de URL—os navegadores precisam conseguir ler suas folhas de estilo para que possam fazer uso delas!

Listando Repositórios

Se você está servindo um conjunto de repositórios a partir de uma única URL por meio da diretiva `SVNParentPath`, então também é possível fazer o Apache exibir todos os repositórios disponíveis para o navegador web. Apenas ative a diretiva `SVNListParentPath`:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  SVNListParentPath on
  …
</Location>
```

Se um usuário agora apontar seu navegador web para a URL `http://host.example.com/svn/`, ele irá ver uma lista de todos os repositórios Subversion situados em `/usr/local/svn`. É claro que isto pode representar um problema de segurança, por isso este recurso é desabilitado por padrão.

Apache Logging

Because Apache is an HTTP server at heart, it contains fantastically flexible logging features. It's beyond the scope of this book to discuss all ways logging can be configured, but we should point out that even the most generic `httpd.conf` file will cause Apache to produce two logs: `error_log` and `access_log`. These logs may appear in different places, but are typically created in the logging area of your Apache installation. (On Unix, they often live in `/usr/local/apache2/logs/`.)

The `error_log` describes any internal errors that Apache runs into as it works. The `access_log` file records every incoming HTTP request received by Apache. This makes it easy to see, for example, which IP addresses Subversion clients are coming from, how often particular clients use the server, which users are authenticating properly, and which requests succeed or fail.

Unfortunately, because HTTP is a stateless protocol, even the simplest Subversion client operation generates multiple network requests. It's very difficult to look at the `access_log` and deduce what the client was doing—most operations look like a series of cryptic `PROPPATCH`, `GET`, `PUT`, and `REPORT` requests. To make things worse, many client operations send nearly-identical series of requests, so it's even harder to tell them apart.

`mod_dav_svn`, however, can come to your aid. By activating an “operational logging” feature, you can ask `mod_dav_svn` to create a separate log file describing what sort of high-level operations your clients are performing.

To do this, you need to make use of Apache's `CustomLog` directive (which is explained in more detail in Apache's own documentation). Be sure to invoke this directive *outside* of your Subversion `Location` block:

```
<Location /svn>
  DAV svn
  …
</Location>

CustomLog logs/svn_logfile "%t %u %{SVN-ACTION}e" env=SVN-ACTION
```

In this example, we're asking Apache to create a special logfile `svn_logfile` in the standard Apache `logs` directory. The `%t` and `%u` variables are replaced by the time and username of the request, respectively. The really important part are the two instances of `SVN-ACTION`. When Apache sees that variable, it substitutes the value of the `SVN-ACTION` environment variable, which is automatically set by `mod_dav_svn` whenever it detects a high-level client action.

So instead of having to interpret a traditional `access_log` like this:

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/vcc/default HTTP/1.1" 207 398
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/bln/59 HTTP/1.1" 207 449
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc HTTP/1.1" 207 647
[26/Jan/2007:22:25:29 -0600] "REPORT /svn/calc/!svn/vcc/default HTTP/1.1" 200 607
[26/Jan/2007:22:25:31 -0600] "OPTIONS /svn/calc HTTP/1.1" 200 188
[26/Jan/2007:22:25:31 -0600] "MKACTIVITY /svn/calc/!svn/act/e6035ef7-5df0-4ac0-b811-4be7c8
…
```

… you can instead peruse a much more intelligible `svn_logfile` like this:

```
[26/Jan/2007:22:24:20 -0600] - list-dir '/'
[26/Jan/2007:22:24:27 -0600] - update '/'
[26/Jan/2007:22:25:29 -0600] - remote-status '/'
[26/Jan/2007:22:25:31 -0600] sally commit r60
```

Other Features

Several of the features already provided by Apache in its role as a robust Web server can be leveraged for increased functionality or security in Subversion as well. Subversion communicates with Apache using Neon, which is a generic HTTP/WebDAV library with support for such mechanisms as SSL (the Secure Socket Layer, discussed earlier). If your Subversion client is built to support SSL, then it can access your Apache server using `https://`.

Equally useful are other features of the Apache and Subversion relationship, such as the ability to specify a custom port (instead of the default HTTP port 80) or a virtual domain name by which the Subversion repository should be accessed, or the ability to access the repository through an HTTP proxy. These things are all supported by Neon, so Subversion gets that support for free.

Finally, because **mod_dav_svn** is speaking a subset of the WebDAV/DeltaV protocol, it's possible to access the repository via third-party DAV clients. Most modern operating systems (Win32, OS X, and Linux) have the built-in ability to mount a DAV server as a standard network share. This is a complicated topic; for details, read Apêndice C, *WebDAV and Autoversioning*.

Path-Based Authorization

Both Apache and **svnserve** are capable of granting (or denying) permissions to users. Typically this is done over the entire repository: a user can read the repository (or not), and she can write to the repository (or

not). It's also possible, however, to define finer-grained access rules. One set of users may have permission to write to a certain directory in the repository, but not others; another directory might not even be readable by all but a few special people.

Both servers use a common file format to describe these path-based access rules. In the case of Apache, one needs to load the **mod_authz_svn** module and then add the `AuthzSVNAccessFile` directive (within the `httpd.conf` file) pointing to your own rules-file. (For a full explanation, see “Controle de Acesso por Diretório”.) If you're using **svnserve**, then you need to make the `authz-db` variable (within `svnserve.conf`) point to your rules-file.

Do you really need path-based access control?

A lot of administrators setting up Subversion for the first time tend to jump into path-based access control without giving it a lot of thought. The administrator usually knows which teams of people are working on which projects, so it's easy to jump in and grant certain teams access to certain directories and not others. It seems like a natural thing, and it appeases the administrator's desire to maintain tight control of the repository.

Note, though, that there are often invisible (and visible!) costs associated with this feature. In the visible category, the server needs to do a lot more work to ensure that the user has the right to read or write each specific path; in certain situations, there's very noticeable performance loss. In the invisible category, consider the culture you're creating. Most of the time, while certain users *shouldn't* be committing changes to certain parts of the repository, that social contract doesn't need to be technologically enforced. Teams can sometimes spontaneously collaborate with each other; someone may want to help someone else out by committing to an area she doesn't normally work on. By preventing this sort of thing at the server level, you're setting up barriers to unexpected collaboration. You're also creating a bunch of rules that need to be maintained as projects develop, new users are added, and so on. It's a bunch of extra work to maintain.

Remember that this is a version control system! Even if somebody accidentally commits a change to something they shouldn't, it's easy to undo the change. And if a user commits to the wrong place with deliberate malice, then it's a social problem anyway, and that the problem needs to be dealt with outside of Subversion.

So before you begin restricting users' access rights, ask yourself if there's a real, honest need for this, or if it's just something that “sounds good” to an administrator. Decide whether it's worth sacrificing some server speed for, and remember that there's very little risk involved; it's bad to become dependent on technology as a crutch for social problems.[8].

As an example to ponder, consider that the Subversion project itself has always had a notion of who is allowed to commit where, but it's always been enforced socially. This is a good model of community trust, especially for open-source projects. Of course, sometimes there *are* truly legitimate needs for path-based access control; within corporations, for example, certain types of data really can be sensitive, and access needs to be genuinely restricted to small groups of people.

Once your server knows where to find your rules-file, it's time to define the rules.

The syntax of the file is the same familiar one used by **svnserve.conf** and the runtime configuration files. Lines that start with a hash (#) are ignored. In its simplest form, each section names a repository and path within it, and the authenticated usernames are the option names within each section. The value of each option describes the user's level of access to the repository path: either `r` (read-only) or `rw` (read-write). If the user is not mentioned at all, no access is allowed.

[8] A common theme in this book!

To be more specific: the value of the section-names are either of the form `[repos-name:path]` or the form `[path]`. If you're using the `SVNParentPath` directive, then it's important to specify the repository names in your sections. If you omit them, then a section like `[/some/dir]` will match the path `/some/dir` in *every* repository. If you're using the `SVNPath` directive, however, then it's fine to only define paths in your sections—after all, there's only one repository.

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

In this first example, the user `harry` has full read and write access on the `/branches/calc/bug-142` directory in the `calc` repository, but the user `sally` has read-only access. Any other users are blocked from accessing this directory.

Of course, permissions are inherited from parent to child directory. That means that we can specify a subdirectory with a different access policy for Sally:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

# give sally write access only to the 'testing' subdir
[calc:/branches/calc/bug-142/testing]
sally = rw
```

Now Sally can write to the `testing` subdirectory of the branch, but can still only read other parts. Harry, meanwhile, continues to have complete read-write access to the whole branch.

It's also possible to explicitly deny permission to someone via inheritance rules, by setting the username variable to nothing:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

[calc:/branches/calc/bug-142/secret]
harry =
```

In this example, Harry has read-write access to the entire `bug-142` tree, but has absolutely no access at all to the `secret` subdirectory within it.

The thing to remember is that the most specific path always matches first. The server tries to match the path itself, and then the parent of the path, then the parent of that, and so on. The net effect is that mentioning a specific path in the accessfile will always override any permissions inherited from parent directories.

By default, nobody has any access to the repository at all. That means that if you're starting with an empty file, you'll probably want to give at least read permission to all users at the root of the repository. You can do this by using the asterisk variable (*), which means “all users”:

```
[/]
```

```
* = r
```

This is a common setup; notice that there's no repository name mentioned in the section name. This makes all repositories world readable to all users. Once all users have read-access to the repositories, you can give explicit `rw` permission to certain users on specific subdirectories within specific repositories.

The asterisk variable (`*`) is also worth special mention here: it's the *only* pattern which matches an anonymous user. If you've configured your server block to allow a mixture of anonymous and authenticated access, all users start out accessing anonymously. The server looks for a `*` value defined for the path being accessed; if it can't find one, then it demands real authentication from the client.

The access file also allows you to define whole groups of users, much like the Unix `/etc/group` file:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = harry, sally, joe, frank, sally, jane
```

Groups can be granted access control just like users. Distinguish them with an “at” (@) prefix:

```
[calc:/projects/calc]
@calc-developers = rw

[paint:/projects/paint]
@paint-developers = rw
jane = r
```

Groups can also be defined to contain other groups:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = @calc-developers, @paint-developers
```

Partial Readability and Checkouts

If you're using Apache as your Subversion server and have made certain subdirectories of your repository unreadable to certain users, then you need to be aware of a possible non-optimal behavior with **svn checkout**.

When the client requests a checkout or update over HTTP, it makes a single server request, and receives a single (often large) server response. When the server receives the request, that is the *only* opportunity Apache has to demand user authentication. This has some odd side-effects. For example, if a certain subdirectory of the repository is only readable by user Sally, and user Harry checks out a parent directory, his client will respond to the initial authentication challenge as Harry. As the server generates the large response, there's no way it can re-send an authentication challenge when it reaches the special subdirectory; thus the subdirectory is skipped altogether, rather than asking the user to re-authenticate as Sally at the right moment. In a similar way, if the root of the repository is anonymously world-readable, then the entire checkout will be done without authentication—again, skipping the unreadable directory, rather than asking for authentication partway through.

Supporting Multiple Repository Access Methods

You've seen how a repository can be accessed in many different ways. But is it possible—or safe—for your repository to be accessed by multiple methods simultaneously? The answer is yes, provided you use a bit of foresight.

At any given time, these processes may require read and write access to your repository:

- regular system users using a Subversion client (as themselves) to access the repository directly via `file://` URLs;
- regular system users connecting to SSH-spawned private **svnserve** processes (running as themselves) which access the repository;
- an **svnserve** process—either a daemon or one launched by **inetd**—running as a particular fixed user;
- an Apache **httpd** process, running as a particular fixed user.

The most common problem administrators run into is repository ownership and permissions. Does every process (or user) in the previous list have the rights to read and write the Berkeley DB files? Assuming you have a Unix-like operating system, a straightforward approach might be to place every potential repository user into a new `svn` group, and make the repository wholly owned by that group. But even that's not enough, because a process may write to the database files using an unfriendly umask—one that prevents access by other users.

So the next step beyond setting up a common group for repository users is to force every repository-accessing process to use a sane umask. For users accessing the repository directly, you can make the **svn** program into a wrapper script that first sets **umask 002** and then runs the real **svn** client program. You can write a similar wrapper script for the **svnserve** program, and add a **umask 002** command to Apache's own startup script, `apachectl`. For example:

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

Another common problem is often encountered on Unix-like systems. As a repository is used, Berkeley DB occasionally creates new log files to journal its actions. Even if the repository is wholly owned by the **svn** group, these newly created files won't necessarily be owned by that same group, which then creates more permissions problems for your users. A good workaround is to set the group SUID bit on the repository's `db` directory. This causes all newly-created log files to have the same group owner as the parent directory.

Once you've jumped through these hoops, your repository should be accessible by all the necessary processes. It may seem a bit messy and complicated, but the problems of having multiple users sharing write-access to common files are classic ones that are not often elegantly solved.

Fortunately, most repository administrators will never *need* to have such a complex configuration. Users who wish to access repositories that live on the same machine are not limited to using `file://` access URLs—they can typically contact the Apache HTTP server or **svnserve** using `localhost` for the server

name in their `http://` or `svn://` URLs. And to maintain multiple server processes for your Subversion repositories is likely to be more of a headache than necessary. We recommend you choose the server that best meets your needs and stick with it!

The svn+ssh:// server checklist

It can be quite tricky to get a bunch of users with existing SSH accounts to share a repository without permissions problems. If you're confused about all the things that you (as an administrator) need to do on a Unix-like system, here's a quick checklist that resummarizes some of things discussed in this section:

- All of your SSH users need to be able to read and write to the repository, so: put all the SSH users into a single group.
- Make the repository wholly owned by that group.
- Set the group permissions to read/write.
- Your users need to use a sane umask when accessing the repository, so: make sure that **svnserve** (`/usr/bin/svnserve`, or wherever it lives in `$PATH`) is actually a wrapper script which sets **umask 002** and executes the real **svnserve** binary.
- Take similar measures when using **svnlook** and **svnadmin**. Either run them with a sane umask, or wrap them as described above.

Capítulo 7. Customizando sua Experiência com Subversion

Controle de versão pode ser um tema complexo, assim como arte ou ciência, e oferecer variadas maneiras fazer as coisas. Através desse livro você leu sobre vários sub-comandos do cliente de linha de comando e as opções para modificar seu comportamento. Nesse capítulo, vamos dar uma olhada e mais maneiras de customizar o jeito que o Subversion trabalha para você—setando as configurações em tempo de execução, usando ajuda de aplicações externas, a interação do Subversion com as configurações locais do sistema operacional, e assim por diante.

Runtime Configuration Area

Subversion provides many optional behaviors that can be controlled by the user. Many of these options are of the kind that a user would wish to apply to all Subversion operations. So, rather than forcing users to remember command-line arguments for specifying these options, and to use them for every operation they perform, Subversion uses configuration files, segregated into a Subversion configuration area.

The Subversion *configuration area* is a two-tiered hierarchy of option names and their values. Usually, this boils down to a special directory that contains *configuration files* (the first tier), which are just text files in standard INI format (with “sections” providing the second tier). These files can be easily edited using your favorite text editor (such as Emacs or vi), and contain directives read by the client to determine which of several optional behaviors the user prefers.

Configuration Area Layout

The first time that the **svn** command-line client is executed, it creates a per-user configuration area. On Unix-like systems, this area appears as a directory named `.subversion` in the user's home directory. On Win32 systems, Subversion creates a folder named `Subversion`, typically inside the `Application Data` area of the user's profile directory (which, by the way, is usually a hidden directory). However, on this platform the exact location differs from system to system, and is dictated by the Windows registry. [1] We will refer to the per-user configuration area using its Unix name, `.subversion`.

In addition to the per-user configuration area, Subversion also recognizes the existence of a system-wide configuration area. This gives system administrators the ability to establish defaults for all users on a given machine. Note that the system-wide configuration area does not alone dictate mandatory policy—the settings in the per-user configuration area override those in the system-wide one, and command-line arguments supplied to the **svn** program have the final word on behavior. On Unix-like platforms, the system-wide configuration area is expected to be the `/etc/subversion` directory; on Windows machines, it looks for a `Subversion` directory inside the common `Application Data` location (again, as specified by the Windows Registry). Unlike the per-user case, the **svn** program does not attempt to create the system-wide configuration area.

The per-user configuration area currently contains three files—two configuration files (`config` and `servers`), and a `README.txt` file which describes the INI format. At the time of their creation, the files contain default values for each of the supported Subversion options, mostly commented out and grouped with textual descriptions about how the values for the key affect Subversion's behavior. To change a certain behavior, you need only to load the appropriate configuration file into a text editor, and modify the desired

[1] The `APPDATA` environment variable points to the `Application Data` area, so you can always refer to this folder as `%APPDATA%\Subversion`.

option's value. If at any time you wish to have the default configuration settings restored, you can simply remove (or rename) your configuration directory and then run some innocuous **svn** command, such as **svn --version**. A new configuration directory with the default contents will be created.

The per-user configuration area also contains a cache of authentication data. The `auth` directory holds a set of subdirectories that contain pieces of cached information used by Subversion's various supported authentication methods. This directory is created in such a way that only the user herself has permission to read its contents.

Configuration and the Windows Registry

In addition to the usual INI-based configuration area, Subversion clients running on Windows platforms may also use the Windows registry to hold the configuration data. The option names and their values are the same as in the INI files. The “file/section” hierarchy is preserved as well, though addressed in a slightly different fashion—in this schema, files and sections are just levels in the registry key tree.

Subversion looks for system-wide configuration values under the `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` key. For example, the `global-ignores` option, which is in the `miscellany` section of the `config` file, would be found at `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`. Per-user configuration values should be stored under `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

Registry-based configuration options are parsed *before* their file-based counterparts, so are overridden by values found in the configuration files. In other words, Subversion looks for configuration information in the following locations on a Windows system; lower-numbered locations take precedence over higher-numbered locations:

1. Command-line options

2. The per-user INI files

3. The per-user Registry values

4. The system-wide INI files

5. The system-wide Registry values

Also, the Windows Registry doesn't really support the notion of something being “commented out”. However, Subversion will ignore any option key whose name begins with a hash (#) character. This allows you to effectively comment out a Subversion option without deleting the entire key from the Registry, obviously simplifying the process of restoring that option.

The **svn** command-line client never attempts to write to the Windows Registry, and will not attempt to create a default configuration area there. You can create the keys you need using the **REGEDIT** program. Alternatively, you can create a `.reg` file, and then double-click on that file from the Explorer shell, which will cause the data to be merged into your registry.

Exemplo 7.1. Sample Registration Entries (.reg) File.

```
REGEDIT4

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

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-proxy-host"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-proxy-password"=""
"#http-proxy-exceptions"=""
"#http-timeout"="0"
"#http-compression"="yes"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-trust-default-ca"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#store-passwords"="yes"
"#store-auth-creds"="yes"

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

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

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#global-ignores"="*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store"
"#log-encoding"=""
"#use-commit-times"=""
"#no-unlock"=""
"#enable-auto-props"=""

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

The previous example shows the contents of a `.reg` file which contains some of the most commonly used configuration options and their default values. Note the presence of both system-wide (for network proxy-related options) and per-user settings (editor programs and password storage, among others). Also note that all the options are effectively commented out. You need only to remove the hash (#) character from the beginning of the option names, and set the values as you desire.

Configuration Options

In this section, we will discuss the specific run-time configuration options that are currently supported by Subversion.

Servers

The `servers` file contains Subversion configuration options related to the network layers. There are two special section names in this file—`groups` and `global`. The `groups` section is essentially a cross-reference table. The keys in this section are the names of other sections in the file; their values are *globs*—textual tokens which possibly contain wildcard characters—that are compared against the hostnames of the machine to which Subversion requests are sent.

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

[beanie-babies]
…

[collabnet]
…
```

When Subversion is used over a network, it attempts to match the name of the server it is trying to reach with a group name under the `groups` section. If a match is made, Subversion then looks for a section in the `servers` file whose name is the matched group's name. From that section it reads the actual network configuration settings.

The `global` section contains the settings that are meant for all of the servers not matched by one of the globs under the `groups` section. The options available in this section are exactly the same as those valid for the other server sections in the file (except, of course, the special `groups` section), and are as follows:

`http-proxy-exceptions`
: This specifies a comma-separated list of patterns for repository hostnames that should accessed directly, without using the proxy machine. The pattern syntax is the same as is used in the Unix shell for filenames. A repository hostname matching any of these patterns will not be proxied.

`http-proxy-host`
: This specifies the hostname of the proxy computer through which your HTTP-based Subversion requests must pass. It defaults to an empty value, which means that Subversion will not attempt to route HTTP requests through a proxy computer, and will instead attempt to contact the destination machine directly.

`http-proxy-port`
: This specifies the port number on the proxy host to use. It defaults to an empty value.

`http-proxy-username`
: This specifies the username to supply to the proxy machine. It defaults to an empty value.

`http-proxy-password`
: This specifies the password to supply to the proxy machine. It defaults to an empty value.

`http-timeout`
: This specifies the amount of time, in seconds, to wait for a server response. If you experience problems with a slow network connection causing Subversion operations to time out, you should increase the value of this option. The default value is `0`, which instructs the underlying HTTP library, Neon, to use its default timeout setting.

`http-compression`
: This specifies whether or not Subversion should attempt to compress network requests made to DAV-ready servers. The default value is `yes` (though compression will only occur if that capability is compiled

into the network layer). Set this to `no` to disable compression, such as when debugging network transmissions.

`neon-debug-mask`
: This is an integer mask that the underlying HTTP library, Neon, uses for choosing what type of debugging output to yield. The default value is `0`, which will silence all debugging output. For more information about how Subversion makes use of Neon, see Capítulo 8, *Incorporando o Subversion*.

`ssl-authority-files`
: This is a semicolon-delimited list of paths to files containing certificates of the certificate authorities (or CAs) that are accepted by the Subversion client when accessing the repository over HTTPS.

`ssl-trust-default-ca`
: Set this variable to `yes` if you want Subversion to automatically trust the set of default CAs that ship with OpenSSL.

`ssl-client-cert-file`
: If a host (or set of hosts) requires an SSL client certificate, you'll normally be prompted for a path to your certificate. By setting this variable to that same path, Subversion will be able to find your client certificate automatically without prompting you. There's no standard place to store your certificate on disk; Subversion will grab it from any path you specify.

`ssl-client-cert-password`
: If your SSL client certificate file is encrypted by a passphrase, Subversion will prompt you for the passphrase whenever the certificate is used. If you find this annoying (and don't mind storing the password in the `servers` file), then you can set this variable to the certificate's passphrase. You won't be prompted anymore.

Config

The `config` file contains the rest of the currently available Subversion run-time options, those not related to networking. There are only a few options in use as of this writing, but they are again grouped into sections in expectation of future additions.

The `auth` section contains settings related to Subversion's authentication and authorization against the repository. It contains:

`store-passwords`
: This instructs Subversion to cache, or not to cache, passwords that are supplied by the user in response to server authentication challenges. The default value is `yes`. Set this to `no` to disable this on-disk password caching. You can override this option for a single instance of the **svn** command using the `--no-auth-cache` command-line parameter (for those subcommands that support it). For more information, see “Client Credentials Caching”.

`store-auth-creds`
: This setting is the same as `store-passwords`, except that it enables or disables disk-caching of *all* authentication information: usernames, passwords, server certificates, and any other types of cacheable credentials.

The `helpers` section controls which external applications Subversion uses to accomplish its tasks. Valid options in this section are:

`editor-cmd`
: This specifies the program Subversion will use to query the user for a log message during a commit operation, such as when using **svn commit** without either the `--message` (`-m`) or `--file` (`-F`) options. This program is also used with the **svn propedit** command—a temporary file is populated with the

current value of the property the user wishes to edit, and the edits take place right in the editor program (see “Properties”). This option's default value is empty. The order of priority for determining the editor command (where lower-numbered locations take precedence over higher-numbered locations) is:

1. Command-line option `--editor-cmd`
2. Environment variable `SVN_EDITOR`
3. Configuration option `editor-cmd`
4. Environment variable `VISUAL`
5. Environment variable `EDITOR`
6. Possibly, a default value built in to Subversion (not present in the official builds)

The value of any of these options or variables is (unlike `diff-cmd`) the beginning of a command line to be executed by the shell. Subversion appends a space and the pathname of the temporary file to be edited. The editor should modify the temporary file and return a zero exit code to indicate success.

`diff-cmd`
: This specifies the absolute path of a differencing program, used when Subversion generates “diff” output (such as when using the **svn diff** command). By default Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See “Using External Differencing Tools” for more details on using such programs.

`diff3-cmd`
: This specifies the absolute path of a three-way differencing program. Subversion uses this program to merge changes made by the user with those received from the repository. By default Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See “Using External Differencing Tools” for more details on using such programs.

`diff3-has-program-arg`
: This flag should be set to `true` if the program specified by the `diff3-cmd` option accepts a `--diff-program` command-line parameter.

The `tunnels` section allows you to define new tunnel schemes for use with **svnserve** and `svn://` client connections. For more details, see “Tunelamento sobre SSH”.

The `miscellany` section is where everything that doesn't belong elsewhere winds up. [2] In this section, you can find:

`global-ignores`
: When running the **svn status** command, Subversion lists unversioned files and directories along with the versioned ones, annotating them with a ? character (see “See an overview of your changes”). Sometimes, it can be annoying to see uninteresting, unversioned items—for example, object files that result from a program's compilation—in this display. The `global-ignores` option is a list of whitespace-delimited globs which describe the names of files and directories that Subversion should not display unless they are versioned. The default value is `*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store`.

  As well as **svn status**, the **svn add** and **svn import** commands also ignore files that match the list when they are scanning a directory. You can override this behaviour for a single instance of any of these commands by explicitly specifying the file name, or by using the `--no-ignore` command-line flag.

  For information on more fine-grained control of ignored items, see “Ignorando Itens Não-Versionados”.

[2] Anyone for potluck dinner?

`enable-auto-props`
: This instructs Subversion to automatically set properties on newly added or imported files. The default value is `no`, so set this to `yes` to enable Auto-props. The `auto-props` section of this file specifies which properties are to be set on which files.

`log-encoding`
: This variable sets the default character set encoding for commit log messages. It's a permanent form of the `--encoding` option (see “**svn** Options”). The Subversion repository stores log messages in UTF-8, and assumes that your log message is written using your operating system's native locale. You should specify a different encoding if your commit messages are written in any other encoding.

`use-commit-times`
: Normally your working copy files have timestamps that reflect the last time they were touched by any process, whether that be your own editor or by some **svn** subcommand. This is generally convenient for people developing software, because build systems often look at timestamps as a way of deciding which files need to be recompiled.

  In other situations, however, it's sometimes nice for the working copy files to have timestamps that reflect the last time they were changed in the repository. The **svn export** command always places these “last-commit timestamps” on trees that it produces. By setting this config variable to `yes`, the **svn checkout**, **svn update**, **svn switch**, and **svn revert** commands will also set last-commit timestamps on files that they touch.

The `auto-props` section controls the Subversion client's ability to automatically set properties on files when they are added or imported. It contains any number of key-value pairs in the format `PATTERN = PROPNAME=PROPVALUE` where `PATTERN` is a file pattern that matches a set of filenames and the rest of the line is the property and its value. Multiple matches on a file will result in multiple propsets for that file; however, there is no guarantee that auto-props will be applied in the order in which they are listed in the config file, so you can't have one rule “override” another. You can find several examples of auto-props usage in the `config` file. Lastly, don't forget to set `enable-auto-props` to `yes` in the `miscellany` section if you want to enable auto-props.

Localization

Localization is the act of making programs behave in a region-specific way. When a program formats numbers or dates in a way specific to your part of the world, or prints messages (or accepts input) in your native language, the program is said to be *localized*. This section describes steps Subversion has made towards localization.

Understanding locales

Most modern operating systems have a notion of the “current locale”—that is, the region or country whose localization conventions are honored. These conventions—typically chosen by some runtime configuration mechanism on the computer—affect the way in which programs present data to the user, as well as the way in which they accept user input.

On most Unix-like systems, you can check the values of the locale-related runtime configuration options by running the **locale** command:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
```

```
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
```

The output is a list of locale-related environment variables and their current values. In this example, the variables are all set to the default `C` locale, but users can set these variables to specific country/language code combinations. For example, if one were to set the `LC_TIME` variable to `fr_CA`, then programs would know to present time and date information formatted according a French-speaking Canadian's expectations. And if one were to set the `LC_MESSAGES` variable to `zh_TW`, then programs would know to present human-readable messages in Traditional Chinese. Setting the `LC_ALL` variable has the effect of changing every locale variable to the same value. The value of `LANG` is used as a default value for any locale variable that is unset. To see the list of available locales on a Unix system, run the command **locale -a**.

On Windows, locale configuration is done via the “Regional and Language Options” control panel item. There you can view and select the values of individual settings from the available locales, and even customize (at a sickening level of detail) several of the display formatting conventions.

Subversion's use of locales

The Subversion client, **svn**, honors the current locale configuration in two ways. First, it notices the value of the `LC_MESSAGES` variable and attempts to print all messages in the specified language. For example:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

This behavior works identically on both Unix and Windows systems. Note, though, that while your operating system might have support for a certain locale, the Subversion client still may not be able to speak the particular language. In order to produce localized messages, human volunteers must provide translations for each language. The translations are written using the GNU gettext package, which results in translation modules that end with the `.mo` filename extension. For example, the German translation file is named `de.mo`. These translation files are installed somewhere on your system. On Unix, they typically live in `/usr/share/locale/`, while on Windows they're often found in the `\share\locale\` folder in Subversion's installation area. Once installed, a module is named after the program it provides translations for. For example, the `de.mo` file may ultimately end up installed as `/usr/share/locale/de/LC_MESSAGES/subversion.mo`. By browsing the installed `.mo` files, you can see which languages the Subversion client is able to speak.

The second way in which the locale is honored involves how **svn** interprets your input. The repository stores all paths, filenames, and log messages in Unicode, encoded as UTF-8. In that sense, the repository is *internationalized*—that is, the repository is ready to accept input in any human language. This means, however, that the Subversion client is responsible for sending only UTF-8 filenames and log messages into the repository. In order to do this, it must convert the data from the native locale into UTF-8.

For example, suppose you create a file named `caffÃ¨.txt`, and then when committing the file, you write the log message as “Adesso il caffÃ¨ Ã¨ piÃ¹ forte”. Both the filename and log message contain non-ASCII characters, but because your locale is set to `it_IT`, the Subversion client knows to interpret them as Italian. It uses an Italian character set to convert the data to UTF-8 before sending them off to the repository.

Note that while the repository demands UTF-8 filenames and log messages, it *does not* pay attention to file contents. Subversion treats file contents as opaque strings of bytes, and neither client nor server makes an attempt to understand the character set or encoding of the contents.

Character set conversion errors

While using Subversion, you might get hit with an error related to character set conversions:

```
svn: Can't convert string from native encoding to 'UTF-8':
…
svn: Can't convert string from 'UTF-8' to native encoding:
…
```

Errors like this typically occur when the Subversion client has received a UTF-8 string from the repository, but not all of the characters in that string can be represented using the encoding of the current locale. For example, if your locale is `en_US` but a collaborator has committed a Japanese filename, you're likely to see this error when you receive the file during an **svn update**.

The solution is either to set your locale to something which *can* represent the incoming UTF-8 data, or to change the filename or log message in the repository. (And don't forget to slap your collaborator's hand—projects should decide on common languages ahead of time, so that all participants are using the same locale.)

Using External Differencing Tools

The presence of `--diff-cmd` and `--diff3-cmd` options, and similarly named runtime configuration parameters (see “Config”), can lead to a false notion of how easy it is to use external differencing (or “diff”) and merge tools with Subversion. While Subversion can use most of popular such tools available, the effort invested in setting this up often turns out to be non-trivial.

The interface between Subversion and external diff and merge tools harkens back to a time when Subversion's only contextual differencing capabilities were built around invocations of the GNU diffutils toolchain, specifically the **diff** and **diff3** utilities. To get the kind of behavior Subversion needed, it called these utilities with more than a handful of options and parameters, most of which were quite specific to the utilities. Some time later, Subversion grew its own internal differencing library, and as a failover mechanism,[3] the `--diff-cmd` and `--diff3-cmd` options were added to the Subversion command-line client so users could more easily indicate that they preferred to use the GNU diff and diff3 utilities instead of the newfangled internal diff library. If those options were used, Subversion would simply ignore the internal diff library, and fall back to running those external programs, lengthy argument lists and all. And that's where things remain today.

It didn't take long for folks to realize that having such easy configuration mechanisms for specifying that Subversion should use the external GNU diff and diff3 utilities located at a particular place on the system could be applied toward the use of other diff and merge tools, too. After all, Subversion didn't actually verify that the things it was being told to run were members of the GNU diffutils toolchain. But the only configurable aspect of using those external tools is their location on the system—not the option set, parameter order, etc. Subversion continues throwing all those GNU utility options at your external diff tool regardless of whether or not that program can understand those options. And that's where things get unintuitive for most users.

The key to using external diff and merge tools (other than GNU diff and diff3, of course) with Subversion is to use wrapper scripts which convert the input from Subversion into something that your differencing tool can understand, and then to convert the output of your tool back into a format which Subversion expects—the format that the GNU tools would have used. The following sections cover the specifics of those expectations.

[3] Subversion developers are good, but even the best make mistakes.

The decision on when to fire off a contextual diff or merge as part of a larger Subversion operation is made entirely by Subversion, and is affected by, among other things, whether or not the files being operated on are human-readable as determined by their `svn:mime-type` property. This means, for example, that even if you had the niftiest Microsoft Word-aware differencing or merging tool in the Universe, it would never be invoked by Subversion so long as your versioned Word documents had a configured MIME type that denoted that they were not human-readable (such as `application/msword`). For more about MIME type settings, see “Tipo de Conteúdo do Arquivo”

External diff

Subversion calls external diff programs with parameters suitable for the GNU diff utility, and expects only that the external program return with a successful error code. For most alternative diff programs, only the sixth and seventh arguments—the paths of the files which represent the left and right sides of the diff, respectively—are of interest. Note that Subversion runs the diff program once per modified file covered by the Subversion operation, so if your program runs in an asynchronous fashion (or “backgrounded”), you might have several instances of it all running simultaneously. Finally, Subversion expects that your program return an error code of 1 if your program detected differences, or 0 if it did not—any other error code is considered a fatal error. [4]

Exemplo 7.2, “diffwrap.sh” and Exemplo 7.3, “diffwrap.bat” are templates for external diff tool wrappers in the Bourne shell and Windows batch scripting languages, respectively.

Exemplo 7.2. diffwrap.sh

```
#!/bin/sh

# Configure your favorite diff program here.
DIFF="/usr/local/bin/my-diff-tool"

# Subversion provides the paths we need as the sixth and seventh
# parameters.
LEFT=${6}
RIGHT=${7}

# Call the diff command (change the following line to make sense for
# your merge program).
$DIFF --left $LEFT --right $RIGHT

# Return an errorcode of 0 if no differences were detected, 1 if some were.
# Any other errorcode will be treated as fatal.
```

[4] The GNU diff manual page puts it this way: “An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.”

Exemplo 7.3. diffwrap.bat

```
@ECHO OFF

REM Configure your favorite diff program here.
SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"

REM Subversion provides the paths we need as the sixth and seventh
REM parameters.
SET LEFT=%6
SET RIGHT=%7

REM Call the diff command (change the following line to make sense for
REM your merge program).
%DIFF% --left %LEFT% --right %RIGHT%

REM Return an errorcode of 0 if no differences were detected, 1 if some were.
REM Any other errorcode will be treated as fatal.
```

External diff3

Subversion calls external merge programs with parameters suitable for the GNU diff3 utility, expecting that the external program return with a successful error code and that the full file contents which result from the completed merge operation are printed on the standard output stream (so that Subversion can redirect them into the appropriate version controlled file). For most alternative merge programs, only the ninth, tenth, and eleventh arguments, the paths of the files which represent the “mine”, “older”, and “yours” inputs, respectively, are of interest. Note that because Subversion depends on the output of your merge program, you wrapper script must not exit before that output has been delivered to Subversion. When it finally does exit, it should return an error code of 0 if the merge was successful, or 1 if unresolved conflicts remain in the output—any other error code is considered a fatal error.

Exemplo 7.4, “diff3wrap.sh” and Exemplo 7.5, “diff3wrap.bat” are templates for external merge tool wrappers in the Bourne shell and Windows batch scripting languages, respectively.

Exemplo 7.4. diff3wrap.sh

```
#!/bin/sh

# Configure your favorite diff3/merge program here.
DIFF3="/usr/local/bin/my-merge-tool"

# Subversion provides the paths we need as the ninth, tenth, and eleventh
# parameters.
MINE=${9}
OLDER=${10}
YOURS=${11}

# Call the merge command (change the following line to make sense for
# your merge program).
$DIFF3 --older $OLDER --mine $MINE --yours $YOURS

# After performing the merge, this script needs to print the contents
# of the merged file to stdout.  Do that in whatever way you see fit.
# Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
# remain in the result.  Any other errorcode will be treated as fatal.
```

Exemplo 7.5. diff3wrap.bat

```
@ECHO OFF

REM Configure your favorite diff3/merge program here.
SET DIFF3="C:\Program Files\Funky Stuff\My Merge Tool.exe"

REM Subversion provides the paths we need as the ninth, tenth, and eleventh
REM parameters.  But we only have access to nine parameters at a time, so we
REM shift our nine-parameter window twice to let us get to what we need.
SHIFT
SHIFT
SET MINE=%7
SET OLDER=%8
SET YOURS=%9

REM Call the merge command (change the following line to make sense for
REM your merge program).
%DIFF3% --older %OLDER% --mine %MINE% --yours %YOURS%

REM After performing the merge, this script needs to print the contents
REM of the merged file to stdout.  Do that in whatever way you see fit.
REM Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
REM remain in the result.  Any other errorcode will be treated as fatal.
```

Capítulo 8. Incorporando o Subversion

O Subversion tem uma estrutura modular: é implementado como uma coleção de bibliotecas em C. Cada biblioteca tem um propósito bem definido e uma Aplicação de Interface (API - Application Program Interface), e que a interface está disponível não só para o próprio Subversion usar, mas para qualquer software que queira incorporar ou através de programação controlar o Subversion. Adicionalmente, a API do Subversion está disponível não só para outros programas em C, mas também programas em linguagens de alto nível como Python, Perl, Java ou Ruby.

Este capítulo é para aqueles que desejam interagir com o Subversion através da sua API pública ou seus vários bindings de linguagem. Se você deseja escrever scripts robustos ao redor das funcionalidades do Subversion para simplificar sua vida, se está tentando desenvolver integrações mais complexas entre o Subversion e outros partes de um software, ou apenas tem interesse nas várias bibliotecas modulares e o que elas tem a oferecer, este capítulo é para você. Se, entretanto, você não se vê participando com o Subversion nesse nível, sinta-se livre para pular este capítulo certo que suas experiências como usuários do Subversion não serão afetadas.

Desing da Camada de Biblioteca

Para núcleo de bibliotecas do Subversion pode ser dito existir em três principais camadas—a Camada do Respositório, a Camada de Acesso ao Respositório (RA) ou Camada Cliente (veja Figura 1, “Subversion's Architecture”). Nós iremos examinar essas camadas daqui a pouco, mas primeiro, vamos dar uma olhada nas várias bibliotecas do Subversion. Pelo bem da consistência, nós vamos nos referir às bibliotecas pelos seus nomes de extensão de biblioteca Unix (libsvn_fs, libsvn_wc, mod_dav_svn, etc.).

libsvn_client
: Interface primária de programas cliente

libsvn_delta
: Rotinas de diferenciação de Árvore e byte-stream

libsvn_diff
: Rotinas contextuais de fusão e diferenciação

libsvn_fs
: Sistema de arquivos comuns e carregamento de módulos

libsvn_fs_base
: Sistema de arquivos Berkeley DB back-end --FIXME--

libsvn_fs_fs
: Sistema de arquivos native (FSFS) back-end--FIXME--

libsvn_ra
: --FIXME--Repository Access commons and module loader

libsvn_ra_dav
: Módulo WebDAV de acesso ao Repositório

libsvn_ra_local
: Módulo de Accesso Local ao respositório

libsvn_ra_serf
: Outro (em experimentação) módulo WebDAV de accesso ao Repositório

libsvn_ra_svn
: Módulo de Acesso ao Repositório customizado

libsvn_repos
: Interface do Repositório

libsvn_subr
: Várias sub-rotinas de ajuda

libsvn_wc
: Biblioteca de gerênciamente de cópia de trabalho

mod_authz_svn
: Módulo de autorização para acesso a Repositórios Subversion WebDAV

mod_dav_svn
: Módulo Apache para mapear operações WebDAV para operações do Subversion

O fato da palavara “várias” só aparecer uma vez na lista anterior é um bom sinal. O time de desenvolvimento do Subversion leva se esforça para fazer com que funcionalidades fiquem na camada certo e nas bibliotecas certas. Talvez a maior vantagem de um sistema modular é a pouca complexidade do ponto de vista do desenvolvedor. Como desenvolvedor, você pode enxergar rapidamente um tipo de “grande quadro” que permite você achar o local de certos pedaços de funcionalidade com certa facilidade.

Outro benefício da modularidade é a habilidade de substituir um módulo dado por uma biblioteca nova que implementa a mesma API sem afetar o resto do código base. De certo modo, isso já acontece dentro do Subversion. Cada uma das bibliotecas libsvn_ra_dav, libsvn_ra_local, libsvn_ra_serf, e libsvn_ra_svn implementam a mesma interface, todas funcionam como plugins para libsvn_ra. E todas as quatro se comunicam com a camada de Repositório—libsvn_ra_local se conecta com o repositório diretamente; as outras três fazem atrás da rede. As bibliotecas libsvn_fs_base e libsvn_fs_fs são outro par de bibliotecas qeu implementam a mesma funcionalidade de maneiras diferentes—ambos plugins comuns a biblioteca libsvn_fs.

O cliente também mostra os benefícios da modularidade no design do Subversion. A biblioteca lbsvn_client do Subversion é uma boa para a maioria das funcionalidades necessárias para o design do cliente Subversion (veja “Camada Cliente”). Enquanto a distribuição do Subverion provê apenas o **svn** como linha de comando, existe várias ferramentas de terceiros que provêm várias formas de gráficas do cliente. Essas interfaces gráficas usam a mesma API que o clinte de linha de comando usa. Este tipo de modularidade tem tido um papel importante na proliferação de clientes Subversion e integrações em IDEs e, por consequência, pela grande adoção do Subversion.

Camado do respositório

Quando referindo a camada de Repositório do Subversion, nós geralmente estamos falando sobre os conceitos—a implementação do sistema de arquivos (acessado via libsvn_fs, e suportados pelos plugins libsvn_fs_base e libsvn_fs_fs), e o respositório lógico que o contém (como implementado na libsvn_respos). Esta biblioteca provê o armazenamento e mecanimos de relatórios para as várias revisões dos dados seus dados versionados. Esta camada é conectada a camada Cliente através da Camada de Acesso ao Repositório, e é, da perspectiva do usuário Subversion, a coisa do “outro lado da linha”

O Sistema de Arquivos não é um sistema de arquivos no nível do kernel que se pode instalar em um sistema operacional (como o ext2 do Linux ou NTFS), mas um sistema de arquivos virtual. Além de guardar arquivos e diretórios (como os que você navega usando seu programa shell favorito), ele usa um dos dois --FIXEME-- available abstract storage backends—mesmo o ambiente de banco de dados Berkeley DB, uma representação plana de um arquivo. (Para aprender mais sobre os dois --FIXEME-- respository

back-end, veja “Choosing a Data Store”.) Sempre houve um considerável interesse pela comunidade de desenvolvimento em dar a futuros releases do Subversion a habilidade de usar --FIXEME-- back-end database systems, talvez através de um mecanismo como um ODBC (Open Database Connectivity). De fato, Google fez algo similar a isso antes de lançar o serviço Google para Host e Projetos: eles anunciaram em meados de 2006 que os membros do time Open Source teria escrito um novo plugin de sistema de arquivos proprietário para o Subversion que seria usaria um banco de dados de Grande tabela ultra escalonável para armazenamento.

A API do sistema de arquivos exportado pelo libsvn_fs contém funcionalidades que você esperaria de uma API de sistema de arquivos—você pode criar e remover arquivos e diretórios, copiar e move-los, modificar seus conteudos, e assim por diante. Ela também tem funcionalidades que não são muito comuns, como a habilidade de adicionar, modificar, e remover metadata (“propriedades”) em cada arquivo ou diretório. Além do mais, o sistema de arquivos Subversion é um sistema versionado, o que significa que assim que você faz mudanças na sua árvore de diretórios, Subversion lembra como sua árvore estava quando as mudanças foram feitas. E as mudanças anteriores. E as anteriores a essas. E assim por diante, todas as mudanças até durante o tempo de versionamento até (e no máximo) o primeiro momento que as coisas foram adicionadas ao sistema de arquivos.

Todas as modificações que você fez a sua árvore são feitas dentro do contexto de uma transação de commit do Subversion. A seguir veremos uma rotina simplificada e genérica de modificação do seu sistema de arquivos:

1. Iniciar uma transação de commit do Subversion.

2. Fazendo modificações (adicionar, apagar, modificações de propriedades, etc.).

3. Commit as sua transação.

Uma vez que você commitou sua transação, as modificações do seu sistema de arquivo é permanentemente armazenado como um histórico. Cada um desses ciclos gerá um única nova revisão da sua árvore, e cada revisão é para sempre acessível como uma imutável foto de “como as coisas estavam”

--FIXME--The Transaction Distraction

A noção de transação no Subversion pode facilmente ser confundida com as transações suportadas providas pelo próprio banco de dados, especialmente pelo código do libsvn_fs_base se aproximar ao código do baco de dados Berkeley DB. Os 2 tipos de transação existem para prover atomicidade e isolamento. Em outra palavras, transações dão a você a habilidade de fazer um conjunto de ações no modo tudo ou nada—ou todas as ações são completadas com sucesso, ou todas são tratadas como se *nenhuma* tivesse ocorrido—e de um certo modo que não há interefência nos outros processos que estão agindo nos dados.

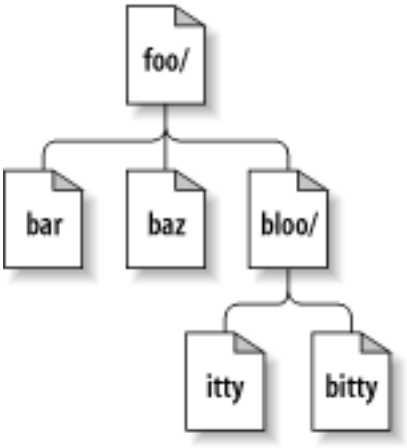
Transações de banco de dados geralmente contém pequenas operações relacionadas com a modificações de dados no banco de dados (como modificações em dados de uma linha de uma tabela). Transações do Subversion são grandes em escopo, com operações de alto nível como fazer modificações de um conjunto de arquivos e diretórios os quais serão guardados como a próxima revisão da árvore de sistema de arquivos. Como se já não fosse confuso o suficiente, considerar o fato que o Subversion criam usa uma transação durante a criação de uma transação Subversion (então se a criação da transação do Subversion falhar, o banco de dados irá travar como se nenhum tentativa de criação tivesse ocorrido em primeiro lugar)!

Felizmente para os usuários da API do sistema de arquivos, o suporte à transação provido pelo sistema de banco de dados é encondido quase totalmente de vista (como é esperado de um esquema de bibliotecas modularizadas). Apenas quando você começa a procurar dentro da implementação do sistema de arquivos que essas coisas começam a ficar visível (ou interessante).

A maioria das funcionalidades providas pela interface dos sitema de arquivos lida com ações que ocorrem em caminhos de sistema de arquivos individuais. Isto é, de fora do sistema de arquivos, o mecanismo primário para descrever e acessar as revisões individuais de arquivos e diretórios que veem do uso de string de caminhos como `/foo/bar`, como se você estivesse endereçando arquivos e diretórios através do seu programa shell favorito. Você adiciona novos arquivos e diretórios passando o futuro caminho para as funções certas da API. Você requisita uma informação sobre eles pelo menos mecanismo.

Ao contrário de muitos sistemas de arquivos, entrentanto, um cmainho sozinho não é informação suficiente para identificar um arquivo ou diretório no Subversion. Pense na árvore de diretório como uma sistema de duas dimensões, onde um nodo irmão representa um conjunto de movimentos direita-esquerda, e descendendo nos sub-diretórios em um movimento de descida.Figura 8.1, “Arquivos e diretórios em duas dimensões” mostrando uma típica representação de uma árvore exatemente assim.

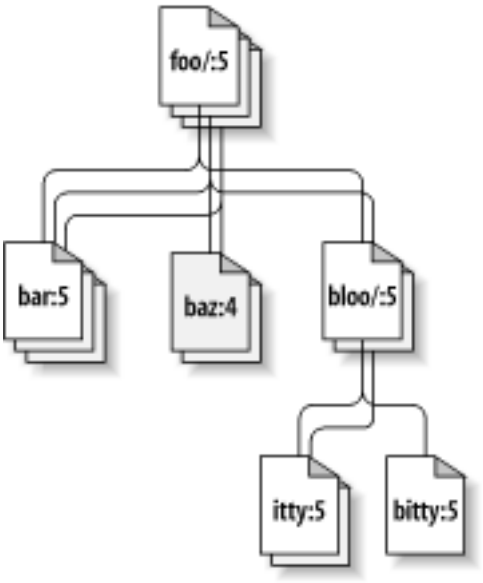
Figura 8.1. Arquivos e diretórios em duas dimensões



A diferença aqui é que o sistema de arquivos Subversion tem uma brilhante terceira dimensão que muitos sistemas de aquivo não tem—Tempo! [1] Numa interface de sistema de arquivos, quase toda função que tem um *`caminho`* como argumento também espera uma *`raíz`* como arguemnto. Este argumento svn_fs_root_t descreve tanto a revisão ou a transação Subversion (que é simplesmente uma --FIXME-- revision-in-the-making), e provê essa terceira dimensão necessário para entender a diferença entre `/foot/bar` na revisão 32, e o mesmo caminho como ele existe na revisão 98.Figura 8.2, “Versioning time—the third dimension!” mostra o histórico de revisão como uma dimensão adicional ao universo do sistema de arquivos Subversion.

[1] Nós entendemos que isso é um shock para fans de ficção científica que tem a impressão que tempo é na verdade a *quarta* dimensão, e nos desculpamos pelo trauma emocional causado pela nossa declaração de uma teoria diferente.

Figura 8.2. Versioning time—the third dimension!



Como mencionado anteriormente, a API libsvn_fs parece qualquer outro sistema de arquivos, exceto que ele tem essa maravilhosa capacidade de versionamento. Ele foi desenhado para ser usável por qualquer programa interessado em versionar arquivos. Não coincidentemente, o próprio Subversion tem interesse nessa funcionalidade. Mas enquanto a API do sistema de arquivos deveria ser suficiente para suporte básico de arquivos e diretórios, Subversion quer mais—e é aí que entra a libsvn_repos.

A biblioteca de respositório Subversion (libsvn_repos) está sentada (logicamente falando) no topo da API libsvn_fs, provendo funcionalidades adicionais além das frisadas pelo versionamenteo logico do sistema de arquivos. Não contém completamente cada uma de todas as funções de um sistema de arquivos—apenas alguns grades passos no ciclo geral da atividade de um sistema de arquivos é contida pela interface do respositório. Algumas dessas incluem a criação e commit de transações Subversion, e as modificações de propriedades de revisão. Esses eventos particulares estão contidos na camada de repositório porque eles tem --FIXME-- hooks associados a eles. Um --FIXME-- respository hook system não está restritamente relacionado a implemnentação de um sistema de arquivos relacionados, então ele fica contido na biblioteca de repositório

--FIXME--The hooks mechanism é uma das razões para a abstração da separação da biblioteca de respositório do resto do código do sistema de arquivo. A API libsvn_repos provê muitas utilidades importantes para o Subversion. Isso inclui habilidades para:

- criar, abrir, destruir, e recuperar passos no respositório Subversion e o sistema de arquivos incluido nesse respositório.
- descrever a diferença entre duas árovres de sistema de arquivos.
- requisitar ao log de messagens do commit associados a todas (ou algumas) das revisões nas quais um conjunto de arquivos foi modificado no sistema de arquivos.
- gerar um “dump” compreensível ao ser humano do sistema de arquivos, uma completa representação da revisão do sitema de arquivos.
- --FIXME--parse that dump, carregando a revisão dumped no repositório Subversion diferente.

Como o Subversion coninua a evoluir, a biblioteca de repositório irá crescer com a biblioteca de sistema de arquivos para oferecer um número crescente de funcionalidades e opções de configuração.

Camada de Acesso ao Repositório

Se a Camada de Repositório do Subversion está “ do outro lado da linha”, a Camada de Accesso (RA) está na linha. Carregado com dados--FIXME--marshaling entre as bibliotecas de clientes e o respositório, esta camada inclue o módulo carregamento de bibliotecas libsvn_ra, os módulos RA mesmos (que normalmente incluem libsvn_ra_dav, libsvn_ra_local, libsvn_ra_serf, e libsvn_ra_svn), e qualquer biblioteca adicional necessária por um ou mais dos módulos RA (assim como o módulo Apache mod_dav_svn ou servidor do libsvn_ra_svn, **svnserv**).

Já que Subversion usa URLs para identificar seus respositórios, o porção de protocolo do esquema URL (normalmente `file://`, `http://`, `https://`, `svn://`, ou `svn+ssh://`) é usado para determinar que módulo RA irá dar conta das comunicações. Cada módulo possui uma lista de protocolos que sabem como “falar” para que o carregador RA possa, em tempo de execução, determinar que módulo usar para faze a tarefa. Você pode saber que módulo RA está disponível para o cliente de linha de comando do Subversion e que protocolos ele suporta, apenas com o comando **svn --version**:

```
$ svn --version
svn, version 1.4.3 (r23084)
   compiled Jan 18 2007, 07:47:40

Copyright (C) 2000-2006 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/).

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' scheme
  - handles 'https' scheme
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' scheme
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' scheme

$
```

A API pública exportada pela Camada RA contém funcionalidades necessárias por enviar e receber dados versionados para e do resposítorio. Cada um dos plugins RA estão disponíveis para fazer a tarefa de usar protocolos específicos—libsvn_ra_dav fala HTTP/WebDAV (opcionalmente usando SSL encryption) com um servidor Apache HTTP que estiver rodando o módulo servidor mod_dav_svn; libsvn_ra_svn fala um protocolo customizável de rede com o programa **svnserve**; e assim por diante.

E para aqueles que desejam acessar um respositório Subversion usando outro protocolo, isso é exatamente por isso que a Camada de Acesso ao Respositório é modularazada! Desenvolvedores podem simplesmente escrever uma nova biblioteca que implementa a interface RA em um dos lados e comunicar com o respositório do outro lado. Sua biblioteca pode usar protocolos já existentes, ou você pode inventar o seu próprio. Você pode usar processos de comunicação (IPC), ou—vamos ficar loucos, podemos?—você pode até implementar um protocolo baseado em email. Subversion provê a API; você vem com a criatividade.

Camada Cliente

No lado do cliente, a cópia de trabalho do Subversion é onde todas as ações tomam lugar. O conjunto de funcionalidades implementadas pelas bibliotecas cliente existem pelo único propósito de gerenciar as cópias de trabalho— diretórios cheios de arquivos e outros sub-diretórios que servem como um tipo de local, editáveis “reflexos” de um ou mais locais de repositórios—e propagando mudanças para e da Camada de Acesso ao Respositório.

Cópia de trabalho Subversion, libsvn_wc, é diretamente responsável por gerenciar os dados nas cópias de trablho. Para conseguir isso, a biblioteca guarda informações administrativas sobre cada diretório da cópia de trabalho com em um sub-diretório especial. Este sub-diretório, chama-se `.svn`, está presente em cada cópia de trabalho e contém vários outros arquivos e diretórios com o estado das informações e provê um espaço privado para ações de administração. Para os familiares com o CVS, este sub-diretório `.svn` é similar em objetivo ao diretório administrativo CVS encontrado nas cópias de trabalho CVS. Para mais informação sobre a área administrativa `.svn`, veja“Por dentro da área de Administração da cópia de trabalho”in this chapter.

A biblioteca do cliente Subversion, libsvn_client, tem uma responsabilidade mais abrangente. o seu trabalho é unir as funcionalidades da biblioteca da cópia de trabalho com as da Camada de Acesso ao Respositório, e então prover uma API de alto nível para qualquer aplicação que desejar fazer controle geral de ações de revisão. Por exemplo, a função `svn_client_checkout()` pega um URL como argumento. E passa a URL para a camada de RA e abre uma sessão com um repositório. Ele então pede ao respositório pora uma certa árvore, e envia está árvore para a biblioteca da cópia de trabalho, que então esrever toda a cópia de trabalho no disco (o diretório `.svn` e tudo mais).

A biblioteca cliente foi desenhada para ser usada por qualquer aplicacação. Enquanto o código fonte inclui um cliente de linha de comando padrão, deveria ser muito fácil escrever qualquer número de clientes gráficos no topo dessa biblioteca cliente. Novas interfaces gráficas (ou qualquer novo cliente) para não precisa ser algo em volta do cliente de linha de comando—eles tem total acesso via API libsvn_client às mesmas funcionalidades, dados e mecanimos de resposta que o cliente de linha de comando usa. De fato, a ávore do código fonte do Subversion contém um pequeno programa em C (que pode ser encontrado em `tools/examples/minimal_client.c` que exemplifica como usar a API do Subversion para criar um programa cliente simples

Fazendo binding diretamente—Uma palavra sobre o que é certo

Porque seu programa de interface deveria fazer o bind diretamente com o libsvn_client ao invés ser um programa em volta do cliente em linha de comando? Além do fato de ser mais eficiente, é mais correto também. Um programa de linha de comando (como o que o Subversion fornece) que faz o bind para a biblioteca do cliente precisa traduzir eficientemente as respostas e requisões de bits de dados de tipos C para tipo em forma entendidas pelo ser humano. Esse tipo de tradução pode ser despendioso. Sendo assim, o programa pode não mostrar todas as informações colhidas pela API, ou pode combinar pedaços de informações para uma apresentação compacta.

Se você puser o programa de linha de comando com outro programa, o segundo programa terá acesso apenas às informações já interpretadas (e como mencionado, possivelemente incompletas), o que é *novamente* uma tradução do *seu próprio* formato de apresentação. Com cada camada de encapsulamento, a integridade do dado original é um pouco mudado mais e mais, parecido com o resultado de fazer uma copia da copia (da cópia …) do seu audio ou vídeo cassete favorito.

Mas o argumento mais contudente para fazer o bind diretamente para as APIs ao invés de usar outrs programas é que o projeto Subversion fez promessas de compatibilidades entre suas APIs. Através de versões menores dessas APIs (como entre 1.3 e 1.4), nenhum protótipo de função irá mudar. Em outras palavras, você não será forçado a atualizar seu código fonte semplesmente porque você atualizou para uma nova versão do Subversion. Algumas funções podem ficar defazadas, msa ainda assim irão funcionar, e isso te dá um intervaldo de tempo para começar a usar as novas APIs. Esse tipo de compatibilidade não é prometido para as mensagens de saída do cliente de linha de comando do Subversion, o que é objeto de mudança de versão para versão.

Por dentro da área de Administração da cópia de trabalho

Como mencionamos anteriormente, cada diretório do Subversion tem um diretório especial chamado `.svn` que hospeda informações administrativas sobre o diretório da cópia de trabalho. Subversion usa essa informação do `.svn` para rastrear coisas como:

- Qual(is) os locais dos repositórios que são representados pelos arquivos e subdiretórios no diretório da cópia de trabalho
- Que revisão de cada arquivo e diretório está representado no momento na sua cópia de trabalho
- Propriedades definidas pelo usuário que podem estar ligadas a esses arquivos e diretórios.
- --FIXME--Pristine (un-edited) cópias de arquivos da cópia de trabalho.

A área de administração da cópia de trabalho do Subversion e o seu conteúdo são consideradas implementações detalhadas e não tem intenção de serem para consumo humano. Desenvolvedores são encorajados a suar APÌs públicas do Subversion, ou as ferramentas que o Subversion provê, para acessar e manipular as informações da cópia local de trabalho, ao invés de ler ou modificar diretamente esses arquivos. Os formatos de arquivos empregados pela biblioteca de cópia local para administração realmente muda de tempos em tempos—um fato é que as APIs públicas fazem um bom trabalho ao se esconder do usuário comum. Nessa seção, vamos expor algumas desses detalhes de implementação para satisfazer sua grande curiosidade.

Os arquivos de Entrada

Talvez o arquivo único de maior importância no diretório `.svn` é o arquivo `entries`. Nele está contido uma quantidade de informações administrativas sobre os ítens versionados no diretório da cópia de trabalho.

É esse arquivo que rastreia as URLs do repositório, arquivos intocados, arquivos de checksum, textos intocados, timestamps de propriedades, agendamento e informações de conflitos, última informações conhecida de commit (autor, revisão, timestamp), cópia local do histórico—praticamente tudo que um cliente Subversion tem interesse em saber sobre dados versionados ou que serão versionados!

Pessoas familiares com diretórios de administração do CVS vão reconhecer nesse ponto que o arquivo `.svn/entries` serve ao propósito de, além de outras coisas, uma combinação dos arquivos `CVS/Entries`, `CVS/Root`, and `CVS/Repository`.

O formato do aquivo `.snv/entries` tem mudado ao longo do tempo. Originalmente como um arquivo XML, usa um formato customizado—mas ainda legível a humanos—. Enquanto o XML foi uma boa escolha para o início do desenvolvimento do Subversion que era frequentemente debugado o conteúdo do arquivos (e o comportamento do Subversion na visão deles), a necessidade de um debug fácil no desenvolvimento foi se reduzindo com a maturidade do Subversion, e vem sendo substituido pela impaciencia do usuário em ter performace. Fique atento que a biblioteca de cópia de trabalho do Subversion atualiza automaticamente cópias de trabalho de um formato para outro—ele lé formatos velhos, e escreve no novo—o que o salva do problema de checar uma nova cópia de trabalho, mas pode complicar em situações onde diferentes versões do Subversion estão tentando usar a mesma cópia de trabablho.

Cópias inalteradas e Propriedade de arquivos

Como mencionado anteriormente, o diretório `.svn` possui o arquivo inalterado de versão de arquivos “text-base”. Eles podem encotrados em `.svn/text-base`. Os benefícios das cópias inalteradas são muitas—checagem livre de rede para modificações locais e diferenças, checagem livre de rede de modificação de revisão e arquivos perdidos, maior eficiência na transmissão das mudanças para o servidor—mas tem o custo de ter cada arquivo armazenado 2 vezes no disco. Nos dias de hoje, isso parece ser uma problema inofensivo para a maioria dos arquivos. Entretanto, a sutiação fica feia quando o tamanho dos seus arquivos versionados cresce. Uma atenção vem sendo dada para que a presença do “text-base” seja opcional. Ironicamente, é quando o tamanho dos seus arquivos fica grande que a existência do “text-base” se torna crucial—quem gosta de transmitir arquivos enormes pela rede apenas para comitar uma pequena mudança?

Com propósito similar aos arquivos “text-base” estão os arquivos de propriedade e suas cópias inalteráveis “prop-base”, localizados em `.svn/props` e `.svn/prop-base` respectivamente. Já que diretórios podem ter propriedades, também, existe também os arquivos `.svn/dir-props` e `.svn/dir-prop-base`.

Using the APIs

Developing applications against the Subversion library APIs is fairly straightforward. Subversion is primarily a set of C libraries, with header (.h) files that live in the `subversion/include` directory of the source tree. These headers are copied into your system locations (for example, `/usr/local/include`) when you build and install Subversion itself from source. These headers represent the entirety of the functions and types meant to be accessible by users of the Subversion libraries. The Subversion developer community is meticulous about ensuring that the public API is well-documented—refer directly to the header files for that documentation.

When examining the public header files, the first thing you might notice is that Subversion's datatypes and functions are namespace protected. That is, every public Subversion symbol name begins with `svn_`, followed by a short code for the library in which the symbol is defined (such as `wc`, `client`, `fs`, etc.), followed by a single underscore (_) and then the rest of the symbol name. 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 help to clarify the scope of a given function or datatype.

Another good source of information about programming against the Subversion APIs is the project's own hacking guidelines, which can be found at http://subversion.tigris.org/hacking.html. This document contains useful information which, while aimed at developers and would-be developers of Subversion itself, is equally applicable to folks developing against Subversion as a set of third-party libraries. [2]

The Apache Portable Runtime Library

Along with Subversion's own datatypes, you will see many references to datatypes that begin with `apr_`—symbols from the Apache Portable Runtime (APR) library. APR is Apache's portability library, originally carved out of its server code as an attempt to separate the OS-specific bits from the OS-independent portions of the code. The result was a library that provides a generic API for performing operations that differ mildly—or wildly—from OS to OS. While the Apache HTTP Server was obviously the first user of the APR library, the Subversion developers immediately recognized the value of using APR as well. This means that there is practically no OS-specific code in Subversion itself. Also, it means that the Subversion client compiles and runs anywhere that Apache HTTP Server itself does. Currently this list includes all flavors of Unix, Win32, BeOS, OS/2, and Mac OS X.

In addition to providing consistent implementations of system calls that differ across operating systems, [3] APR gives Subversion immediate access to many custom datatypes, such as dynamic arrays and hash tables. Subversion uses these types extensively. But perhaps the most pervasive APR datatype, found in nearly every Subversion API prototype, is the apr_pool_t—the APR memory pool. Subversion uses pools internally for all its memory allocation needs (unless an external library requires a different memory management mechanism for data passed through its API), [4] and while a person coding against the Subversion APIs is not required to do the same, 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 create and manage pools for use with Subversion API calls, typically by using `svn_pool_create()`, `svn_pool_clear()`, and `svn_pool_destroy()`.

[2] After all, Subversion uses Subversion's APIs, too.
[3] Subversion uses ANSI system calls and datatypes as much as possible.
[4] Neon and Berkeley DB are examples of such libraries.

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.

Higher-level languages, on the other hand, take the job of memory management away from the developer completely. [5] Languages like Java and Python use *garbage collection*, allocating memory for objects when needed, and automatically freeing that memory when the object is no longer in use.

APR provides a middle-ground approach called pool-based memory management. It allows the developer to control memory usage at a lower resolution—per chunk (or “pool”) of memory, instead of per allocated object. Rather than using `malloc()` and friends to allocate enough memory for a given object, you ask APR to allocate the memory from a memory pool. When you're finished using the objects you've created in the pool, you destroy the entire pool, effectively de-allocating the memory consumed by *all* the objects you allocated from it. Thus, rather than keeping track of individual objects which need to be de-allocated, your program simply considers the general lifetimes of those objects, and allocates the objects in a pool whose lifetime (the time between the pool's creation and its deletion) matches the object's needs.

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 or Perl script—Subversion has some support for this via the Simplified Wrapper and Interface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig`. They are still maturing, but they are usable. These bindings allow you to call Subversion API functions indirectly, using wrappers that translate the datatypes native to your scripting language into the datatypes needed by Subversion's C libraries.

Significant efforts have been made towards creating functional SWIG-generated bindings for Python, Perl, and Ruby. To some extent, the work done preparing the SWIG interface files for these languages is reusable

[5] Or at least make it something you only toy with when doing extremely tight program optimization.

in efforts to generate bindings for other languages supported by SWIG (which include versions of C#, Guile, Java, MzScheme, OCaml, PHP, and Tcl, among others). However, some extra programming is required to compensate for complex APIs that SWIG needs some help translating between languages. For more information on SWIG itself, see the project's website at http://www.swig.org/.

Subversion also has language bindings for Java. The JavaJL bindings (located in `subversion/bindings/java` in the Subversion source tree) aren't SWIG-based, but are instead a mixture of javah and hand-coded JNI. JavaHL most covers Subversion client-side APIs, and is specifically targeted at implementors of Java-based Subversion clients and IDE integrations.

Subversion's language bindings tend to lack the level of developer attention given to the core Subversion modules, but can generally be trusted as production-ready. A number of scripts and applications, alternative Subversion GUI clients and other third-party tools are successfully using Subversion's language bindings today to accomplish their Subversion integrations.

It's worth noting here that there are other options for interfacing with Subversion using other languages: alternative bindings for Subversion which aren't provided by the Subversion development community at all. You can find links to these alternative bindings on the Subversion project's links page (at http://subversion.tigris.org/links.html), but there are a couple of popular ones we feel are especially noteworthy. First, Barry Scott's PySVN bindings (http://pysvn.tigris.org/) are a popular option for binding with Python. PySVN boasts of a more Pythonic interface than the more C-like APIs provided by Subversion's own Python bindings. For folks looking for a pure Java implementation of Subversion, check out SVNKit (http://svnkit.com/), which is Subversion re-written from the ground up in Java. You should exercise caution here, though—because SVNKit doesn't use the core Subversion libraries, it's behavior is not guaranteed to match that of Subversion itself.

Code Samples

Exemplo 8.1, “Using the Repository Layer” contains a code segment (written in C) that illustrates some of the concepts we've been discussing. It uses both the repository and filesystem interfaces (as can be determined by the prefixes `svn_repos_` and `svn_fs_` of the function names, respectively) to create a new revision in which a directory is added. You can see the use of an APR pool, which is passed around for memory allocation purposes. Also, the code reveals a somewhat obscure fact about Subversion error handling—all Subversion errors must be explicitly handled to avoid memory leakage (and in some cases, application failure).

Exemplo 8.1. Using the Repository Layer

```
  /* Get a pointer to the filesystem object that is stored in REPOS.
   */
  fs = svn_repos_fs(repos);

  /* Ask the filesystem to tell us the youngest revision that
   * currently exists.
   */
  INT_ERR(svn_fs_youngest_rev(&youngest_rev, fs, pool));

  /* Begin a new transaction that is based on YOUNGEST_REV.  We are
   * less likely to have our later commit rejected as conflicting if we
   * always try to make our changes against a copy of the latest snapshot
   * of the filesystem tree.
   */
  INT_ERR(svn_fs_begin_txn(&txn, fs, youngest_rev, pool));

  /* Now that we have started a new Subversion transaction, get a root
   * object that represents that transaction.
   */
  INT_ERR(svn_fs_txn_root(&txn_root, txn, pool));

  /* Create our new directory under the transaction root, at the path
   * NEW_DIRECTORY.
   */
  INT_ERR(svn_fs_make_dir(txn_root, new_directory, pool));

  /* Commit the transaction, creating a new revision of the filesystem
   * which includes our added directory path.
   */
  err = svn_repos_fs_commit_txn(&conflict_str, repos,
                                &youngest_rev, txn, pool);
  if (! err)
    {
      /* No error?  Excellent!  Print a brief report of our success.
       */
      printf("Directory '%s' was successfully added as new revision "
             "'%ld'.\n", new_directory, youngest_rev);
    }
  else if (err->apr_err == SVN_ERR_FS_CONFLICT)
    {
      /* Uh-oh.  Our commit failed as the result of a conflict
       * (someone else seems to have made changes to the same area
       * of the filesystem that we tried to modify).  Print an error
       * message.
       */
      printf("A conflict occurred at path '%s' while attempting "
             "to add directory '%s' to the repository at '%s'.\n",
             conflict_str, new_directory, repos_path);
    }
  else
    {
      /* Some other error has occurred.  Print an error message.
       */
      printf("An error occurred while attempting to add directory '%s' "
             "to the repository at '%s'.\n",
             new_directory, repos_path);
    }

  INT_ERR(err);
}
```

Note that in Exemplo 8.1, “Using the Repository Layer”, the code could just as easily have committed the transaction using `svn_fs_commit_txn()`. But the filesystem API knows nothing about the repository library's hook mechanism. If you want your Subversion repository to automatically perform some set of non-Subversion tasks every time you commit a transaction (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, which adds the hook triggering functionality—in this case, `svn_repos_fs_commit_txn()`. (For more information regarding Subversion's repository hooks, see “Implementing Repository Hooks”.)

Now let's switch languages. Exemplo 8.2, “Using the Repository Layer with Python” is a sample program that uses Subversion's SWIG Python bindings to recursively crawl the youngest repository revision, and print the various paths reached during the crawl.

Exemplo 8.2. Using the Repository Layer with Python

```
import sys
import os.path
import svn.fs, svn.core, svn.repos

def crawl_filesystem_dir(root, directory):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
    a list of all the paths at or below DIRECTORY."""

    # Print the name of this path.
    print directory + "/"

    # Get the directory entries for DIRECTORY.
    entries = svn.fs.svn_fs_dir_entries(root, directory)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Calculate the entry's full path.
        full_path = directory + '/' + name

        # If the entry is a directory, recurse.  The recursion will return
        # a list with the entry and all its children, which we will add to
        # our running list of paths.
        if svn.fs.svn_fs_is_dir(root, full_path):
            crawl_filesystem_dir(root, full_path)
        else:
            # Else it's a file, so print its path here.
            print full_path

def crawl_youngest(repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj)

    # Open a root object representing the youngest (HEAD) revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj, youngest_rev)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "")

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
                         % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize the repository path.
    repos_path = svn.core.svn_path_canonicalize(sys.argv[1])

    # Do the real work.
    crawl_youngest(repos_path)
```

This same program in C would need to deal with APR's memory pool system. But Python handles memory usage automatically, and Subversion's Python bindings adhere to that convention. In C, you'd be working with custom datatypes (such as those provided by the APR library) for representing the hash of entries and the list of paths, but Python has hashes (called “dictionaries”) and lists as built-in datatypes, and provides a rich collection of functions for operating on those types. So SWIG (with the help of some customizations in Subversion's language bindings layer) takes care of mapping those custom datatypes into the native datatypes of the target language. This provides a more intuitive interface for users of that language.

The Subversion Python bindings can be used for working copy operations, too. In the previous section of this chapter, we mentioned the `libsvn_client` interface, and how it exists for the sole purpose of simplifying the process of writing a Subversion client. Exemplo 8.3, “A Python Status Crawler” is a brief example of how that library can be accessed via the SWIG Python bindings to recreate a scaled-down version of the **svn status** command.

```
                svn.wc.svn_wc_status_ignored     : 'I',
                svn.wc.svn_wc_status_external    : 'X',
                svn.wc.svn_wc_status_unversioned : '?',
              }
    return code_map.get(status, '?')

def do_status(wc_path, verbose):
```

Exemplo 8.3. A Python Status Crawler

```
    # Calculate the length of the input working copy path.
    wc_path_len = len(wc_path)

    # Build a client context baton.
    ctx = svn.client.svn_client_ctx_t()

    def _status_callback(path, status, root_path_len=wc_path_len):
        """A callback function for svn_client_status."""

        # Print the path, minus the bit that overlaps with the root of
        # the status crawl
        text_status = generate_status_code(status.text_status)
        prop_status = generate_status_code(status.prop_status)
        print '%s%s  %s' % (text_status, prop_status, path[wc_path_len + 1:])

    # Do the status crawl, using _status_callback() as our callback function.
    svn.client.svn_client_status(wc_path, None, _status_callback,
                                 1, verbose, 0, 0, ctx)

def usage_and_exit(errorcode):
    """Print usage message, and exit with ERRORCODE."""
    stream = errorcode and sys.stderr or sys.stdout
    stream.write("""Usage: %s OPTIONS WC-PATH
Options:
  --help, -h    : Show this usage message
  --verbose, -v : Show all statuses, even uninteresting ones
""" % (os.path.basename(sys.argv[0])))
    sys.exit(errorcode)

if __name__ == '__main__':
    # Parse command-line options.
    try:
        opts, args = getopt.getopt(sys.argv[1:], "hv", ["help", "verbose"])
    except getopt.GetoptError:
        usage_and_exit(1)
    verbose = 0
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage_and_exit(0)
        if opt in ("-v", "--verbose"):
            verbose = 1
    if len(args) != 1:
        usage_and_exit(2)

    # Canonicalize the repository path.
    wc_path = svn.core.svn_path_canonicalize(args[0])

    # Do the real work.
    try:
        do_status(wc_path, verbose)
    except svn.core.SubversionException, e:
        sys.stderr.write("Error (%d): %s\n" % (e[1], e[0]))
        sys.exit(1)
```

As was the case in Exemplo 8.2, “Using the Repository Layer with Python”, this program is pool-free and uses, for the most part, normal Python data types. The call to `svn_client_ctx_t()` is deceiving because the public Subversion API has no such function—this just happens to be a case where SWIG's automatic language generation bleeds through a little bit (the function is a sort of factory function for Python's version of the corresponding complex C structure). Also note that the path passed to this program (like the last one) gets run through `svn_path_canonicalize()`, because to *not* do so runs the risk of triggering the underlying Subversion C library's assertions about such things, which translate into rather immediate and unceremonious program abortion.

Capítulo 9. Referência Completa do Subverion

Este capítulo tem a intenção de ser uma referência completa para o uso do Subversion. Ele inclui o comando (**svn**) e todos os seus sub-comandos, assim como programas de administração de repositório (**svnadmin** e **svnlook**) e seus respectivos sub-comandos.

The Subversion Command Line Client: svn

To use the command line client, you type **svn**, the subcommand you wish to use [1], and any options or targets that you wish to operate on—there is no specific order that the subcommand and the options must appear in. For example, all of the following are valid ways to use **svn status**:

```
$ svn -v status
$ svn status -v
$ svn status -v myfile
```

You can find many more examples of how to use most client commands in Capítulo 2, *Uso Básico* and commands for managing properties in “Properties”.

svn Options

While Subversion has different options for its subcommands, all options are global—that is, each option is guaranteed to mean the same thing regardless of the subcommand you use it with. For example, `--verbose` (`-v`) always means “verbose output”, regardless of the subcommand you use it with.

`--auto-props`
: Enables auto-props, overriding the `enable-auto-props` directive in the `config` file.

`--change` (`-c`) *`ARG`*
: Used as a means to refer to a specific “change” (aka a revision), this option is syntactic sugar for “-r ARG-1:ARG”.

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--diff-cmd` *`CMD`*
: Specifies an external program to use to show differences between files. When **svn diff** is invoked without this option, it uses Subversion's internal diff engine, which provides unified diffs by default. If you want to use an external diff program, use `--diff-cmd`. You can pass options to the diff program with the `--extensions` option (more on that later in this section).

`--diff3-cmd` *`CMD`*
: Specifies an external program to use to merge files.

`--dry-run`
: Goes through all the motions of running a command, but makes no actual changes—either on disk or in the repository.

[1] Yes, yes, you don't need a subcommand to use the `--version` option, but we'll get to that in just a minute.

`--editor-cmd` *`CMD`*
Specifies an external program to use to edit a log message or a property value. See the `editor-cmd` section in “Config” for ways to specify a default editor.

`--encoding` *`ENC`*
Tells Subversion that your commit message is encoded in the charset provided. The default is your operating system's native locale, and you should specify the encoding if your commit message is in any other encoding.

`--extensions` (`-x`) *`ARGS`*
Specifies an argument or arguments that Subversion should pass to an external diff command. This option is valid only when used with the **svn diff** or **svn merge** commands, with the `--diff-cmd` option. If you wish to pass multiple arguments, you must enclose all of them in quotes (for example, **svn diff --diff-cmd /usr/bin/diff -x "-b -E"**).

`--file` (`-F`) *`FILENAME`*
Uses the contents of the named file for the specified subcommand, though different subcommands do different things with this content. For example, **svn commit** uses the content as a commit log, whereas **svn propset** uses it as a property value.

`--force`
Forces a particular command or operation to run. There are some operations that Subversion will prevent you from doing in normal usage, but you can pass the force option to tell Subversion “I know what I'm doing as well as the possible repercussions of doing it, so let me at 'em”. This option is the programmatic equivalent of doing your own electrical work with the power on—if you don't know what you're doing, you're likely to get a nasty shock.

`--force-log`
Forces a suspicious parameter passed to the `--message` (`-m`) or `--file` (`-F`) options to be accepted as valid. By default, Subversion will produce an error if parameters to these options look like they might instead be targets of the subcommand. For example, if you pass a versioned file's path to the `--file` (`-F`) option, Subversion will assume you've made a mistake, that the path was instead intended as the target of the operation, and that you simply failed to provide some other—unversioned—file as the source of your log message. To assert your intent and override these types of errors, pass the `--force-log` option to subcommands that accept log messages.

`--help` (`-h` or `-?`)
If used with one or more subcommands, shows the built-in help text for each subcommand. If used alone, it displays the general client help text.

`--ignore-ancestry`
Tells Subversion to ignore ancestry when calculating differences (rely on path contents alone).

`--ignore-externals`
Tells Subversion to ignore external definitions and the external working copies managed by them.

`--incremental`
Prints output in a format suitable for concatenation.

`--limit` *`NUM`*
Show only the first *`NUM`* log messages.

`--message` (`-m`) *`MESSAGE`*
Indicates that you will specify a either a log message or a lock comment on the command line, following this option. For example:

```
$ svn commit -m "They don't make Sunday."
```

`--new` *ARG*
Uses *ARG* as the newer target (for use with **svn diff**).

`--no-auth-cache`
Prevents caching of authentication information (e.g. username and password) in the Subversion administrative directories.

`--no-auto-props`
Disables auto-props, overriding the `enable-auto-props` directive in the `config` file.

`--no-diff-added`
Prevents Subversion from printing differences for added files. The default behavior when you add a file is for **svn diff** to print the same differences that you would see if you had added the entire contents of an existing (empty) file.

`--no-diff-deleted`
Prevents Subversion from printing differences for deleted files. The default behavior when you remove a file is for **svn diff** to print the same differences that you would see if you had left the file but removed all the content.

`--no-ignore`
Shows files in the status listing that would normally be omitted since they match a pattern in the `global-ignores` configuration option or the `svn:ignore` property. See “Config” and “Ignorando Itens Não-Versionados” for more information.

`--no-unlock`
Don't automatically unlock files (the default commit behavior is to unlock all files listed as part of the commit). See “Travamento” for more information.

`--non-interactive`
In the case of an authentication failure, or insufficient credentials, prevents prompting for credentials (e.g. username or password). This is useful if you're running Subversion inside of an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--non-recursive` (`-N`)
Stops a subcommand from recursing into subdirectories. Most subcommands recurse by default, but some subcommands—usually those that have the potential to remove or undo your local modifications—do not.

`--notice-ancestry`
Pay attention to ancestry when calculating differences.

`--old` *ARG*
Uses *ARG* as the older target (for use with **svn diff**).

`--password` *PASS*
Indicates that you are providing your password for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--quiet` (`-q`)
Requests that the client print only essential information while performing an operation.

`--recursive` (`-R`)
Makes a subcommand recurse into subdirectories. Most subcommands recurse by default.

`--relocate` *`FROM TO [PATH...]`*
Used with the **svn switch** subcommand, changes the location of the repository that your working copy references. This is useful if the location of your repository changes and you have an existing working copy that you'd like to continue to use. See **svn switch** for an example.

`--revision` (`-r`) *`REV`*
Indicates that you're going to supply a revision (or range of revisions) for a particular operation. You can provide revision numbers, revision keywords or dates (in curly braces), as arguments to the revision option. If you wish to provide a range of revisions, you can provide two revisions separated by a colon. For example:

```
$ svn log -r 1729
$ svn log -r 1729:HEAD
$ svn log -r 1729:1744
$ svn log -r {2001-12-04}:{2002-02-17}
$ svn log -r 1729:{2002-02-17}
```

See “Revision Keywords” for more information.

`--revprop`
Operates on a revision property instead of a property specific to a file or directory. This option requires that you also pass a revision with the `--revision` (`-r`) option.

`--show-updates` (`-u`)
Causes the client to display information about which files in your working copy are out-of-date. This doesn't actually update any of your files—it just shows you which files will be updated if you run **svn update**.

`--stop-on-copy`
Causes a Subversion subcommand which is traversing the history of a versioned resource to stop harvesting that historical information when a copy—that is, a location in history where that resource was copied from another location in the repository—is encountered.

`--strict`
Causes Subversion to use strict semantics, a notion which is rather vague unless talking about specific subcommands (namely, **svn propget**).

`--targets` *`FILENAME`*
Tells Subversion to get the list of files that you wish to operate on from the filename you provide instead of listing all the files on the command line.

`--username` *`NAME`*
Indicates that you are providing your username for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--verbose` (`-v`)
Requests that the client print out as much information as it can while running any subcommand. This may result in Subversion printing out additional fields, detailed information about every file, or additional information regarding its actions.

`--version`
Prints the client version info. This information not only includes the version number of the client, but also a listing of all repository access modules that the client can use to access a Subversion repository. With `--quiet` (`-q`) it prints only the version number in a compact form.

`--xml`
Prints output in XML format.

svn Subcommands

Here are the various subcommands:

Nome

svn add — Add files, directories, or symbolic links.

Synopsis

```
svn add PATH...
```

Description

Schedule files, directories, or symbolic links in your working copy for addition to the repository. They will be uploaded and added to the repository on your next commit. If you add something and change your mind before committing, you can unschedule the addition using **svn revert**.

Alternate Names

None

Changes

Working Copy

Accesses Repository

No

Options

```
--targets FILENAME
--non-recursive (-N)
--quiet (-q)
--config-dir DIR
--no-ignore
--auto-props
--no-auto-props
--force
```

Examples

To add a file to your working copy:

```
$ svn add foo.c
A         foo.c
```

When adding a directory, the default behavior of **svn add** is to recurse:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
```

```
A         testdir/d
```

You can add a directory without adding its contents:

```
$ svn add --non-recursive otherdir
A         otherdir
```

Normally, the command **svn add *** will skip over any directories that are already under version control. Sometimes, however, you may want to add every unversioned object in your working copy, including those hiding deeper down. Passing the `--force` option makes **svn add** recurse into versioned directories:

```
$ svn add * --force
A         foo.c
A         somedir/bar.c
A         otherdir/docs/baz.doc
…
```

Nome

svn blame — Show author and revision information in-line for the specified files or URLs.

Synopsis

```
svn blame TARGET[@REV]...
```

Description

Show author and revision information in-line for the specified files or URLs. Each line of text is annotated at the beginning with the author (username) and the revision number for the last change to that line.

Alternate Names

praise, annotate, ann

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) ARG
--verbose (-v)
--incremental
--xml
--extensions (-x) ARG
--force
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
```

Examples

If you want to see blame annotated source for `readme.txt` in your test repository:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry You should read this.
```

Even if **svn blame** says that Harry last modified readme.txt in revision 5, you'll have to examine exactly what the revision changed to be sure that Harry changed the *context* of the line—he may have just adjusted the whitespace.

Nome

svn cat — Output the contents of the specified files or URLs.

Synopsis

```
svn cat TARGET[@REV]...
```

Description

Output the contents of the specified files or URLs. For listing the contents of directories, see **svn list**.

Alternate Names

None

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

If you want to view readme.txt in your repository without checking it out:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```

If your working copy is out of date (or you have local modifications) and you want to see the HEAD revision of a file in your working copy, **svn cat** will automatically fetch the HEAD revision when you give it a path:

```
$ 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!
```

Nome

svn checkout — Check out a working copy from a repository.

Synopsis

```
svn checkout URL[@REV]... [PATH]
```

Description

Check out a working copy from a repository. If *PATH* is omitted, the basename of the URL will be used as the destination. If multiple URLs are given each will be checked out into a subdirectory of *PATH*, with the name of the subdirectory being the basename of the URL.

Alternate Names

co

Changes

Creates a working copy.

Accesses Repository

Yes

Options

```
--revision (-r) REV
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--ignore-externals
--config-dir DIR
```

Examples

Check out a working copy into a directory called `mine`:

```
$ svn checkout file:///tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2.
$ ls
mine
```

Check out two different directories into two separate working copies:

```
$ 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
```

Check out two different directories into two separate working copies, but place both into a directory called `working-copies`:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz working-copies
A  working-copies/test/a
A  working-copies/test/b
Checked out revision 2.
A  working-copies/quiz/l
A  working-copies/quiz/m
Checked out revision 2.
$ ls
working-copies
```

If you interrupt a checkout (or something else interrupts your checkout, like loss of connectivity, etc.), you can restart it either by issuing the identical checkout command again, or by updating the incomplete working copy:

```
$ 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.
```

Nome

svn cleanup — Recursively clean up the working copy.

Synopsis

```
svn cleanup [PATH...]
```

Description

Recursively clean up the working copy, removing working copy locks and resuming unfinished operations. If you ever get a “working copy locked” error, run this command to remove stale locks and get your working copy into a usable state again.

If, for some reason, an **svn update** fails due to a problem running an external diff program (e.g. user input or network failure), pass the `--diff3-cmd` to allow cleanup to complete any merging with your external diff program. You can also specify any configuration directory with the `--config-dir` option, but you should need these options extremely infrequently.

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Options

```
--diff3-cmd CMD
--config-dir DIR
```

Examples

Well, there's not much to the examples here as **svn cleanup** generates no output. If you pass no *PATH*, “`.`” is used.

```
$ svn cleanup

$ svn cleanup /path/to/working-copy
```

Nome

svn commit — Send changes from your working copy to the repository.

Synopsis

```
svn commit [PATH...]
```

Description

Send changes from your working copy to the repository. If you do not supply a log message with your commit by using either the `--file` or `--message` option, **svn** will launch your editor for you to compose a commit message. See the `editor-cmd` section in “Config”.

svn commit will send any lock tokens that it finds and will release locks on all *PATHS* committed (recursively), unless `--no-unlock` is passed.

If you begin a commit and Subversion launches your editor to compose the commit message, you can still abort without committing your changes. If you want to cancel your commit, just quit your editor without saving your commit message and Subversion will prompt you to either abort the commit, continue with no message, or edit the message again.

Alternate Names

ci (short for “check in”; not “co”, which is short for “checkout”)

Changes

Working copy, repository

Accesses Repository

Yes

Options

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--no-unlock
--non-recursive (-N)
--targets FILENAME
--force-log
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--config-dir DIR
```

Examples

Commit a simple modification to a file with the commit message on the command line and an implicit target of your current directory (“`.`”):

```
$ svn commit -m "added howto section."
Sending        a
Transmitting file data .
Committed revision 3.
```

Commit a modification to the file `foo.c` (explicitly specified on the command line) with the commit message in a file named `msg`:

```
$ svn commit -F msg foo.c
Sending        foo.c
Transmitting file data .
Committed revision 5.
```

If you want to use a file that's under version control for your commit message with `--file`, you need to pass the `--force-log` option:

```
$ 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.
```

To commit a file scheduled for deletion:

```
$ svn commit -m "removed file 'c'."
Deleting       c

Committed revision 7.
```

Nome

svn copy — Copy a file or directory in a working copy or in the repository.

Synopsis

```
svn copy SRC DST
```

Description

Copy a file in a working copy or in the repository. *SRC* and *DST* can each be either a working copy (WC) path or URL:

WC -> WC
: Copy and schedule an item for addition (with history).

WC -> URL
: Immediately commit a copy of WC to URL.

URL -> WC
: Check out URL into WC, and schedule it for addition.

URL -> URL
: Complete server-side copy. This is usually used to branch and tag.

You can only copy files within a single repository. Subversion does not support cross-repository copying.

Alternate Names

cp

Changes

Repository if destination is a URL.

Working copy if destination is a WC path.

Accesses Repository

If source or destination is in the repository, or if needed to look up the source revision number.

Options

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

```
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Examples

Copy an item within your working copy (just schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +   bar.txt
```

Copy an item in your working copy to a URL in the repository (an immediate commit, so you must supply a commit message):

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

Copy an item from the repository to your working copy (just schedules the copy—nothing goes into the repository until you commit):

This is the recommended way to resurrect a dead file in your repository!

```
$ svn copy file:///tmp/repos/test/far-away near-here
A         near-here
```

And finally, copying between two URLs:

```
$ svn copy file:///tmp/repos/test/far-away file:///tmp/repos/test/over-there -m "remote co

Committed revision 9.
```

This is the easiest way to “tag” a revision in your repository—just **svn copy** that revision (usually HEAD) into your tags directory.

```
$ svn copy file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prerelease -m "

Committed revision 12.
```

And don't worry if you forgot to tag—you can always specify an older revision and tag anytime:

```
$ svn copy -r 11 file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prereleas
```

```
Committed revision 13.
```

Nome

svn delete — Delete an item from a working copy or the repository.

Synopsis

```
svn delete PATH...

svn delete URL...
```

Description

Items specified by *PATH* are scheduled for deletion upon the next commit. Files (and directories that have not been committed) are immediately removed from the working copy. The command will not remove any unversioned or modified items; use the `--force` option to override this behavior.

Items specified by URL are deleted from the repository via an immediate commit. Multiple URLs are committed atomically.

Alternate Names

del, remove, rm

Changes

Working copy if operating on files, repository if operating on URLs

Accesses Repository

Only if operating on URLs

Options

```
--force
--force-log
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Examples

Using **svn** to delete a file from your working copy deletes your local copy of the file, but merely schedules it to be deleted from the repository. When you commit, the file is deleted in the repository.

```
$ svn delete myfile
D         myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

Deleting a URL, however, is immediate, so you have to supply a log message:

```
$ svn delete -m "Deleting file 'yourfile'" file:///tmp/repos/test/yourfile

Committed revision 15.
```

Here's an example of how to force deletion of a file that has local mods:

```
$ 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
```

Nome

svn diff — Display the differences between two revisions or paths.

Synopsis

```
diff [-c M | -r N[:M]] [TARGET[@REV]...]

diff [-r N[:M]] --old=OLD-TGT[@OLDREV] [--new=NEW-TGT[@NEWREV]] [PATH...]

diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

Description

Display the differences between two paths. The ways you can use **svn diff** are:

Use just **svn diff'**to display local modifications in a working copy.

Display the changes made to *TARGET*s as they are seen in *REV* between two revisions. *TARGET*s may be all working copy paths or all *URL*s. If *TARGET*s are working copy paths, *N* defaults to BASE and *M* to the working copy; if *URL*s, *N* must be specified and *M* defaults to HEAD. The “-c M” option is equivalent to “-r N:M” where `N = M-1`. Using “-c -M” does the reverse: “-r M:N” where `N = M-1`.

Display the differences between *OLD-TGT* as it was seen in *OLDREV* and *NEW-TGT* as it was seen ain *NEWREV*. *PATH*s, if given, are relative to *OLD-TGT* and *NEW-TGT* and restrict the output to differences for those paths. *OLD-TGT* and *NEW-TGT* may be working copy paths or *URL[@REV]*. *NEW-TGT* defaults to *OLD-TGT* if not specified. “-r N” makes OLDREV default to N, -r N:M makes *OLDREV* default to *N* and *NEWREV* default to *M*.

Shorthand for **svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV]**

svn diff -r N:M URL is shorthand for **svn diff -r N:M --old=URL --new=URL**.

svn diff [-r N[:M]] URL1[@N] URL2[@M] is shorthand for **svn diff [-r N[:M]] --old=URL1 --new=URL2**.

If *TARGET* is a URL, then revs N and M can be given either via the `--revision` or by using “@” notation as described earlier.

If *TARGET* is a working copy path, then the `--revision` option means:

`--revision N:M`
: The server compares *TARGET@N* and *TARGET@M*.

`--revision N`
: The client compares *TARGET@N* against working copy.

(no `--revision`)
: The client compares base and working copies of *TARGET*.

If the alternate syntax is used, the server compares *URL1* and *URL2* at revisions *N* and *M* respectively. If either *N* or *M* are omitted, a value of HEAD is assumed.

By default, **svn diff** ignores the ancestry of files and merely compares the contents of the two files being compared. If you use `--notice-ancestry`, the ancestry of the paths in question will be taken into consideration when comparing revisions (that is, if you run **svn diff** on two files with identical contents but different ancestry you will see the entire contents of the file as having been removed and added again).

Alternate Names

di

Changes

Nothing

Accesses Repository

For obtaining differences against anything but BASE revision in your working copy

Options

```
--revision (-r) ARG
--change (-c) ARG
--old ARG
--new ARG
--non-recursive (-N)
--diff-cmd CMD
--extensions (-x) "ARGS"
--no-diff-deleted
--notice-ancestry
--summarize
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Compare BASE and your working copy (one of the most popular uses of **svn diff**):

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
```

See what changed in the file COMMITTERS revision 9115:

```
$ svn diff -c 9115 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

See how your working copy's modifications compare against an older revision:

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

Compare revision 3000 to revision 3500 using “@” syntax:

```
$ svn diff http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000 http://svn.collab.net/rep
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Compare revision 3000 to revision 3500 using range notation (you only pass the one URL in this case):

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Compare revision 3000 to revision 3500 of all files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

Compare revision 3000 to revision 3500 of only three files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk COMMITTERS README HACK
```

If you have a working copy, you can obtain the differences without typing in the long URLs:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Use `--diff-cmd` *`CMD`* `-x` to pass arguments directly to the external diff program

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
```

Nome

svn export — Export a clean directory tree.

Synopsis

```
svn export [-r REV] URL[@PEGREV] [PATH]

svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

Description

The first form exports a clean directory tree from the repository specified by URL, at revision *REV* if it is given, otherwise at HEAD, into *PATH*. If *PATH* is omitted, the last component of the *URL* is used for the local directory name.

The second form exports a clean directory tree from the working copy specified by *PATH1* into *PATH2*. All local changes will be preserved, but files not under version control will not be copied.

Alternate Names

None

Changes

Local disk

Accesses Repository

Only if exporting from a URL

Options

```
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--non-recursive (-N)
--config-dir DIR
--native-eol EOL
--ignore-externals
```

Examples

Export from your working copy (doesn't print every file and directory):

```
$ svn export a-wc my-export
Export complete.
```

Export directly from the repository (prints every file and directory):

```
$ svn export file:///tmp/repos my-export
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

When rolling operating-system-specific release packages, it can be useful to export a tree which uses a specific EOL character for line endings. The `--native-eol` option will do this, but it only affects files that have `svn:eol-style = native` properties attached to them. For example, to export a tree with all CRLF line endings (possibly for a Windows .zip file distribution):

```
$ svn export file:///tmp/repos my-export --native-eol CRLF
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

You can specify `LR`, `CR`, or `CRLF` as a line ending type with the `--native-eol` option.

Nome

svn help — Help!

Synopsis

```
svn help [SUBCOMMAND...]
```

Description

This is your best friend when you're using Subversion and this book isn't within reach!

Alternate Names

?, h

The options `-?`, `-h` and `--help` have the same effect as using the **help** subcommand.

Changes

Nothing

Accesses Repository

No

Options

```
--config-dir DIR
```

Nome

svn import — Commit an unversioned file or tree into the repository.

Synopsis

```
svn import [PATH] URL
```

Description

Recursively commit a copy of *PATH* to *URL*. If *PATH* is omitted "`.`" is assumed. Parent directories are created in the repository as necessary.

Alternate Names

None

Changes

Repository

Accesses Repository

Yes

Options

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
--auto-props
--no-auto-props
--ignore-externals
```

Examples

This imports the local directory `myproj` into `trunk/misc` in your repository. The directory `trunk/misc` need not exist before you import into it—**svn import** will recursively create directories for you.

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/trunk/misc
Adding         myproj/sample.txt
…
Transmitting file data .........
```

```
Committed revision 16.
```

Be aware that this will *not* create a directory named `myproj` in the repository. If that's what you want, simply add `myproj` to the end of the URL:

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/trunk/misc/myproj
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

After importing data, note that the original tree is *not* under version control. To start working, you still need to **svn checkout** a fresh working copy of the tree.

Nome

svn info — Display information about a local or remote item.

Synopsis

```
svn info [TARGET[@REV]...]
```

Description

Print information about the working copy paths or URLs specified. The information shown for both may include:

- Path
- Name
- URL
- Repository Root
- Repository UUID
- Revision
- Node Kind
- Last Changed Author
- Last Changed Revision
- Last Changed Date
- Lock Token
- Lock Owner
- Lock Created (date)
- Lock Expires (date)

Additional kinds of information available only for working copy paths are:

- Schedule
- Copied From URL
- Copied From Rev
- Text Last Updated
- Properties Last Updated
- Checksum
- Conflict Previous Base File

- Conflict Previous Working File
- Conflict Current Base File
- Conflict Properties File

Alternate Names

None

Changes

Nothing

Accesses Repository

Only if operating on URLs

Options

```
--revision (-r) REV
--recursive (-R)
--targets FILENAME
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

svn info will show you all the useful information that it has for items in your working copy. It will show information for files:

```
$ svn info foo.c
Path: foo.c
Name: foo.c
URL: http://svn.red-bean.com/repos/test/foo.c
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 4417
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 20
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)
Checksum: d6aeb60b0662ccceb6bce4bac344cb66
```

It will also show information for directories:

```
$ svn info vendors
Path: vendors
URL: http://svn.red-bean.com/repos/test/vendors
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 19
Node Kind: directory
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 19
Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-16 23:39:02 -0600 (Thu, 16 Jan 2003)
```

svn info also acts on URLs (also note that the file readme.doc in this example is locked, so lock information is also provided):

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc
Path: readme.doc
Name: readme.doc
URL: http://svn.red-bean.com/repos/test/readme.doc
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 1
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 42
Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
Lock Comment (1 line):
My test lock comment
```

Nome

svn list — List directory entries in the repository.

Synopsis

```
svn list [TARGET[@REV]...]
```

Description

List each *TARGET* file and the contents of each *TARGET* directory as they exist in the repository. If *TARGET* is a working copy path, the corresponding repository URL will be used.

The default *TARGET* is "`.`", meaning the repository URL of the current working copy directory.

With `--verbose`, **svn list** shows the following fields for each item:

- Revision number of the last commit
- Author of the last commit
- If locked, the letter "O" (See svn info for details).
- Size (in bytes)
- Date and time of the last commit

With `--xml`, output is in XML format (with a header and an enclosing document element unless `--incremental` is also specified). All of the information is present; the `--verbose` option is not accepted.

Alternate Names

ls

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) REV
--verbose (-v)
--recursive (-R)
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

svn list is most useful if you want to see what files a repository has without downloading a working copy:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
…
```

You can pass the `--verbose` option for additional information, rather like the UNIX command **ls -l**:

```
$ 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/
```

For further details, see “**svn list**”.

Nome

svn lock — Lock working copy paths or URLs in the repository, so that no other user can commit changes to them.

Synopsis

```
svn lock TARGET...
```

Description

Lock each *TARGET*. If any *TARGET* is already locked by another user, print a warning and continue locking the rest of the *TARGET*s. Use `--force` to steal a lock from another user or working copy.

Alternate Names

None

Changes

Working Copy, Repository

Accesses Repository

Yes

Options

```
--targets FILENAME
--message (-m) TEXT
--file (-F) FILE
--force-log
--encoding ENC
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--force
```

Examples

Lock two files in your working copy:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

Lock a file in your working copy that is currently locked by another user:

```
$ svn lock tree.jpg
svn: warning: Path '/tree.jpg is already locked by user 'sally in \
     filesystem '/svn/repos/db'

$ svn lock --force tree.jpg
'tree.jpg' locked by user 'harry'.
```

Lock a file without a working copy:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'harry'.
```

For further details, see “Travamento”.

Nome

svn log — Display commit log messages.

Synopsis

```
svn log [PATH]

svn log URL [PATH...]

svn log URL[@REV] [PATH...]
```

Description

Shows log messages from the repository. If no arguments are supplied, **svn log** shows the log messages for all files and directories inside of (and including) the current working directory of your working copy. You can refine the results by specifying a path, one or more revisions, or any combination of the two. The default revision range for a local path is `BASE:1`.

If you specify a URL alone, then it prints log messages for everything that the URL contains. If you add paths past the URL, only messages for those paths under that URL will be printed. The default revision range for a URL is `HEAD:1`.

With `--verbose`, **svn log** will also print all affected paths with each log message. With `--quiet`, **svn log** will not print the log message body itself (this is compatible with `--verbose`).

Each log message is printed just once, even if more than one of the affected paths for that revision were explicitly requested. Logs follow copy history by default. Use `--stop-on-copy` to disable this behavior, which can be useful for determining branch points.

Alternate Names

None

Changes

Nothing

Accesses Repository

Yes

Options

```
--revision (-r) REV
--quiet (-q)
--verbose (-v)
--targets FILENAME
--stop-on-copy
--incremental
--limit NUM
--xml
--username USER
```

```
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

You can see the log messages for all the paths that changed in your working copy by running **svn log** from the top:

```
$ svn log
------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Tweak.
------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
…
```

Examine all log messages for a particular file in your working copy:

```
$ svn log foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you don't have a working copy handy, you can log a URL:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you want several distinct paths underneath the same URL, you can use the `URL [PATH...]` syntax.

```
$ svn log http://svn.red-bean.com/repos/test/ foo.c bar.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r31 | harry | 2003-01-10 12:25:08 -0600 (Fri, 10 Jan 2003) | 1 line
```

```
Added new file bar.c
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

When you're concatenating the results of multiple calls to the log command, you may want to use the `--incremental` option. **svn log** normally prints out a dashed line at the beginning of a log message, after each subsequent log message, and following the final log message. If you ran **svn log** on a range of two revisions, you would get this:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r15 | …

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

However, if you wanted to gather 2 non-sequential log messages into a file, you might do something like this:

```
$ svn log -r 14 > mylog
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | …

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

You can avoid the clutter of the double dashed lines in your output by using the incremental option:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
r27 | …
```

The `--incremental` option provides similar output control when using the `--xml` option.

If you run **svn log** on a specific path and provide a specific revision and get no output at all

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

That just means that the path was not modified in that revision. If you log from the top of the repository, or know the file that changed in that revision, you can specify it explicitly:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

Nome

svn merge — Apply the differences between two sources to a working copy path.

Synopsis

```
svn merge [-c M | -r N:M] SOURCE[@REV] [WCPATH]

svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]

svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]
```

Description

In the first and second forms, the source paths (URLs in the first form, working copy paths in the second) are specified at revisions *N* and *M*. These are the two sources to be compared. The revisions default to HEAD if omitted.

The `-c M` option is equivalent to `-r N:M` where *N = M-1*. Using `-c -M` does the reverse: `-r M:N` where *N = M-1*.

In the third form, *SOURCE* can be a URL or working copy item, in which case the corresponding URL is used. This URL, at revisions *N* and *M*, defines the two sources to be compared.

WCPATH is the working copy path that will receive the changes. If *WCPATH* is omitted, a default value of "`.`" is assumed, unless the sources have identical basenames that match a file within "`.`": in which case, the differences will be applied to that file.

Unlike **svn diff**, the merge command takes the ancestry of a file into consideration when performing a merge operation. This is very important when you're merging changes from one branch into another and you've renamed a file on one branch but not the other.

Alternate Names

None

Changes

Working copy

Accesses Repository

Only if working with URLs

Options

```
--revision (-r) REV
--change (-c) REV
--non-recursive (-N)
--quiet (-q)
--force
--dry-run
--diff3-cmd CMD
```

```
--extensions (-x) ARG
--ignore-ancestry
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Merge a branch back into the trunk (assuming that you have a working copy of the trunk, and that the branch was created in revision 250):

```
$ svn merge -r 250:HEAD http://svn.red-bean.com/repos/branches/my-branch
U  myproj/tiny.txt
U  myproj/thhgttg.txt
U  myproj/win.txt
U  myproj/flo.txt
```

If you branched at revision 23, and you want to merge changes on trunk into your branch, you could do this from inside the working copy of your branch:

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors
U  myproj/thhgttg.txt
…
```

To merge changes to a single file:

```
$ cd myproj
$ svn merge -r 30:31 thhgttg.txt
U  thhgttg.txt
```

Nome

svn mkdir — Create a new directory under version control.

Synopsis

```
svn mkdir PATH...

svn mkdir URL...
```

Description

Create a directory with a name given by the final component of the *PATH* or URL. A directory specified by a working copy *PATH* is scheduled for addition in the working copy. A directory specified by a URL is created in the repository via an immediate commit. Multiple directory URLs are committed atomically. In both cases all the intermediate directories must already exist.

Alternate Names

None

Changes

Working copy, repository if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Examples

Create a directory in your working copy:

```
$ svn mkdir newdir
A         newdir
```

Create one in the repository (instant commit, so a log message is required):

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir

Committed revision 26.
```

Nome

svn move — Move a file or directory.

Synopsis

```
svn move SRC DST
```

Description

This command moves a file or directory in your working copy or in the repository.

This command is equivalent to an **svn copy** followed by **svn delete**.

Subversion does not support moving between working copies and URLs. In addition, you can only move files within a single repository—Subversion does not support cross-repository moving.

WC -> WC
: Move and schedule a file or directory for addition (with history).

URL -> URL
: Complete server-side rename.

Alternate Names

mv, rename, ren

Changes

Working copy, repository if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV (Deprecated)
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
```

```
--config-dir DIR
```

Examples

Move a file in your working copy:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

Move a file in the repository (an immediate commit, so it requires a commit message):

```
$ 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.
```

Nome

svn propdel — Remove a property from an item.

Synopsis

```
svn propdel PROPNAME [PATH...]

svn propdel PROPNAME --revprop -r REV [TARGET]
```

Description

This removes properties from files, directories, or revisions. The first form removes versioned properties in your working copy, while the second removes unversioned remote properties on a repository revision (*TARGET* only determines which repository to access).

Alternate Names

pdel, pd

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Delete a property from a file in your working copy

```
$ svn propdel svn:mime-type  some-script
property 'svn:mime-type' deleted from 'some-script'.
```

Delete a revision property:

```
$ svn propdel --revprop -r 26 release-date
property 'release-date' deleted from repository revision '26'
```

Nome

svn propedit — Edit the property of one or more items under version control.

Synopsis

```
svn propedit PROPNAME PATH...

svn propedit PROPNAME --revprop -r REV [TARGET]
```

Description

Edit one or more properties using your favorite editor. The first form edits versioned properties in your working copy, while the second edits unversioned remote properties on a repository revision (*TARGET* only determines which repository to access).

Alternate Names

pedit, pe

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--editor-cmd EDITOR
--config-dir DIR
```

Examples

svn propedit makes it easy to modify properties that have multiple values:

```
$ svn propedit svn:keywords  foo.c
    <svn will launch your favorite editor here, with a buffer open
    containing the current contents of the svn:keywords property.  You
    can add multiple values to a property easily here by entering one
    value per line.>
Set new value for property 'svn:keywords' on 'foo.c'
```

Nome

svn propget — Print the value of a property.

Synopsis

```
svn propget PROPNAME [TARGET[@REV]...]

svn propget PROPNAME --revprop -r REV [URL]
```

Description

Print the value of a property on files, directories, or revisions. The first form prints the versioned property of an item or items in your working copy, while the second prints unversioned remote property on a repository revision. See “Properties” for more information on properties.

Alternate Names

pget, pg

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Examine a property of a file in your working copy:

```
$ svn propget svn:keywords foo.c
Author
Date
Rev
```

The same goes for a revision property:

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

Nome

svn proplist — List all properties.

Synopsis

```
svn proplist [TARGET[@REV]...]

svn proplist --revprop -r REV [TARGET]
```

Description

List all properties on files, directories, or revisions. The first form lists versioned properties in your working copy, while the second lists unversioned remote properties on a repository revision (*TARGET* only determines which repository to access).

Alternate Names

plist, pl

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--verbose (-v)
--recursive (-R)
--revision (-r) REV
--quiet (-q)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

You can use proplist to see the properties on an item in your working copy:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

But with the `--verbose` flag, svn proplist is extremely handy as it also shows you the values for the properties:

```
$ svn proplist --verbose foo.c
Properties on 'foo.c':
  svn:mime-type : text/plain
  svn:keywords : Author Date Rev
  owner : sally
```

Nome

svn propset — Set PROPNAME to PROPVAL on files, directories, or revisions.

Synopsis

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...

svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [TARGET]
```

Description

Set *PROPNAME* to *PROPVAL* on files, directories, or revisions. The first example creates a versioned, local property change in the working copy, and the second creates an unversioned, remote property change on a repository revision (*TARGET* only determines which repository to access).

Subversion has a number of “special” properties that affect its behavior. See “Subversion properties” for more on these properties.

Alternate Names

pset, ps

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Options

```
--file (-F) FILE
--quiet (-q)
--revision (-r) REV
--targets FILENAME
--recursive (-R)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force
--config-dir DIR
```

Examples

Set the mime type on a file:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

On a UNIX system, if you want a file to have the executable permission set:

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

Perhaps you have an internal policy to set certain properties for the benefit of your coworkers:

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

If you made a mistake in a log message for a particular revision and want to change it, use `--revprop` and set `svn:log` to the new log message:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

Or, if you don't have a working copy, you can provide a URL.

```
$ svn propset --revprop -r 26 svn:log "Document nap." http://svn.red-bean.com/repos
property 'svn:log' set on repository revision '25'
```

Lastly, you can tell propset to take its input from a file. You could even use this to set the contents of a property to something binary:

```
$ svn propset owner-pic -F sally.jpg moo.c
property 'owner-pic' set on 'moo.c'
```

By default, you cannot modify revision properties in a Subversion repository. Your repository administrator must explicitly enable revision property modifications by creating a hook named `pre-revprop-change`. See “Implementing Repository Hooks” for more information on hook scripts.

Nome

svn resolved — Remove “conflicted” state on working copy files or directories.

Synopsis

```
svn resolved PATH...
```

Description

Remove “conflicted” state on working copy files or directories. This routine does not semantically resolve conflict markers; it merely removes conflict-related artifact files and allows *PATH* to be committed again; that is, it tells Subversion that the conflicts have been “resolved”. See “Resolve Conflicts (Merging Others' Changes)” for an in-depth look at resolving conflicts.

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Options

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Examples

If you get a conflict on an update, your working copy will sprout three new files:

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

Once you've resolved the conflict and `foo.c` is ready to be committed, run **svn resolved** to let your working copy know you've taken care of everything.

You *can* just remove the conflict files and commit, but **svn resolved** fixes up some bookkeeping data in the working copy administrative area in addition to removing the conflict files, so we recommend that you use this command.

Nome

svn revert — Undo all local edits.

Synopsis

```
svn revert PATH...
```

Description

Reverts any local changes to a file or directory and resolves any conflicted states. **svn revert** will not only revert the contents of an item in your working copy, but also any property changes. Finally, you can use it to undo any scheduling operations that you may have done (e.g. files scheduled for addition or deletion can be “unscheduled”).

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Options

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Examples

Discard changes to a file:

```
$ svn revert foo.c
Reverted foo.c
```

If you want to revert a whole directory of files, use the `--recursive` flag:

```
$ svn revert --recursive .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

Lastly, you can undo any scheduling operations:

```
$ svn add mistake.txt whoops
A         mistake.txt
A         whoops
A         whoops/oopsie.c

$ svn revert mistake.txt whoops
Reverted mistake.txt
Reverted whoops

$ svn status
?      mistake.txt
?      whoops
```

svn revert is inherently dangerous, since its entire purpose is to throw away data—namely, your uncommitted changes. Once you've reverted, Subversion provides *no way* to get back those uncommitted changes.

If you provide no targets to **svn revert**, it will do nothing—to protect you from accidentally losing changes in your working copy, **svn revert** requires you to provide at least one target.

Nome

svn status — Print the status of working copy files and directories.

Synopsis

```
svn status [PATH...]
```

Description

Print the status of working copy files and directories. With no arguments, it prints only locally modified items (no repository access). With `--show-updates`, it adds working revision and server out-of-date information. With `--verbose`, it prints full revision information on every item.

The first six columns in the output are each one character wide, and each column gives you information about different aspects of each working copy item.

The first column indicates that an item was added, deleted, or otherwise changed.

' '
: No modifications.

'A'
: Item is scheduled for Addition.

'D'
: Item is scheduled for Deletion.

'M'
: Item has been modified.

'R'
: Item has been replaced in your working copy. This means the file was scheduled for deletion, and then a new file with the same name was scheduled for addition in its place.

'C'
: The contents (as opposed to the properties) of the item conflict with updates received from the repository.

'X'
: Item is present because of an externals definition.

'I'
: Item is being ignored (e.g. with the `svn:ignore` property).

'?'
: Item is not under version control.

'!'
: Item is missing (e.g. you moved or deleted it without using **svn**). This also indicates that a directory is incomplete (a checkout or update was interrupted).

'~'
: Item is versioned as one kind of object (file, directory, link), but has been replaced by different kind of object.

The second column tells the status of a file's or directory's properties.

' '
No modifications.

'M'
Properties for this item have been modified.

'C'
Properties for this item are in conflict with property updates received from the repository.

The third column is populated only if the working copy directory is locked. (See “Sometimes You Just Need to Clean Up”.)

' '
Item is not locked.

'L'
Item is locked.

The fourth column is populated only if the item is scheduled for addition-with-history.

' '
No history scheduled with commit.

'+'
History scheduled with commit.

The fifth column is populated only if the item is switched relative to its parent (see “Traversing Branches”).

' '
Item is a child of its parent directory.

'S'
Item is switched.

The sixth column is populated with lock information.

' '
When `--show-updates` is used, the file is not locked. If `--show-updates` is *not* used, this merely means that the file is not locked in this working copy.

K
File is locked in this working copy.

O
File is locked either by another user or in another working copy. This only appears when `--show-updates` is used.

T
File was locked in this working copy, but the lock has been “stolen” and is invalid. The file is currently locked in the repository. This only appears when `--show-updates` is used.

B
File was locked in this working copy, but the lock has been “broken” and is invalid. The file is no longer locked This only appears when `--show-updates` is used.

The out-of-date information appears in the seventh column (only if you pass the `--show-updates` option).

' '
: The item in your working copy is up-to-date.

'*'
: A newer revision of the item exists on the server.

The remaining fields are variable width and delimited by spaces. The working revision is the next field if the `--show-updates` or `--verbose` options are passed.

If the `--verbose` option is passed, the last committed revision and last committed author are displayed next.

The working copy path is always the final field, so it can include spaces.

Alternate Names

stat, st

Changes

Nothing

Accesses Repository

Only if using `--show-updates`

Options

```
--show-updates (-u)
--verbose (-v)
--non-recursive (-N)
--quiet (-q)
--no-ignore
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Examples

This is the easiest way to find out what changes you have made to your working copy:

```
$ svn status wc
 M     wc/bar.c
A  +   wc/qax.c
```

If you want to find out what files in your working copy are out-of-date, pass the `--show-updates` option (this will *not* make any changes to your working copy). Here you can see that `wc/foo.c` has changed in the repository since we last updated our working copy:

```
$ svn status --show-updates wc
```

```
 M           965    wc/bar.c
       *     965    wc/foo.c
A  +         965    wc/qax.c
Status against revision:    981
```

`--show-updates` *only* places an asterisk next to items that are out of date (that is, items that will be updated from the repository if you run **svn update**). `--show-updates` does *not* cause the status listing to reflect the repository's version of the item (although you can see the revision number in the repository by passing the `--verbose` option).

And finally, the most information you can get out of the status subcommand:

```
$ svn status --show-updates --verbose wc
 M           965       938 sally        wc/bar.c
       *     965       922 harry        wc/foo.c
A  +         965       687 harry        wc/qax.c
             965       687 harry        wc/zig.c
Head revision:   981
```

For many more examples of **svn status**, see “See an overview of your changes”.

Nome

svn switch — Update working copy to a different URL.

Synopsis

```
svn switch URL [PATH]

switch --relocate FROM TO [PATH...]
```

Description

The first variant of this subcommand (without the `--relocate` option) updates your working copy to point to a new URL—usually a URL which shares a common ancestor with your working copy, although not necessarily. This is the Subversion way to move a working copy to a new branch. See “Traversing Branches” for an in-depth look at switching.

The `--relocate` option causes **svn switch** to do something different: it updates your working copy to point to *the same* repository directory, only at a different URL (typically because an administrator has moved the repository to another server, or to another URL on the same server).

Alternate Names

sw

Changes

Working copy

Accesses Repository

Yes

Options

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--diff3-cmd CMD
--relocate FROM TO
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

If you're currently inside the directory `vendors`, which was branched to `vendors-with-fix`, and you'd like to switch your working copy to that branch:

```
$ 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.
```

And to switch back, just provide the URL to the location in the repository from which you originally checked out your working copy:

```
$ svn switch http://svn.red-bean.com/repos/trunk/vendors .
U  myproj/foo.txt
U  myproj/bar.txt
U  myproj/baz.c
U  myproj/qux.c
Updated to revision 31.
```

You can just switch part of your working copy to a branch if you don't want to switch your entire working copy.

Sometimes an administrator might change the “base location” of your repository—in other words, the contents of the repository doesn't change, but the main URL used to reach the root of the repository does. For example, the hostname may change, the URL scheme may change, or any part of the URL which leads to the repository itself may change. Rather than check out a new working copy, you can have the **svn switch** command “rewrite” the beginnings of all the URLs in your working copy. Use the `--relocate` option to do the substitution. No file contents are changed, nor is the repository contacted. It's similar to running a Perl script over your working copy `.svn/` directories which runs **s/OldRoot/NewRoot/**.

```
$ svn checkout file:///tmp/repos test
A  test/a
A  test/b
…

$ mv repos newlocation
$ cd test/

$ svn update
svn: Unable to open an ra_local session to URL
svn: Unable to open repository 'file:///tmp/repos'

$ svn switch --relocate file:///tmp/repos file:///tmp/newlocation .
$ svn update
At revision 3.
```

Be careful when using the `--relocate` option. If you mistype the argument, you might end up creating nonsensical URLs within your working copy that render the whole workspace unusable and tricky to fix. It's also important to understand exactly when one should or shouldn't use `--relocate`. Here's the rule of thumb:

- If the working copy needs to reflect a new directory *within* the repository, then use just **svn switch**.
- If the working copy still reflects the same repository directory, but the location of the repository itself has changed, then use **svn switch --relocate**.

Nome

svn unlock — Unlock working copy paths or URLs.

Synopsis

```
svn unlock TARGET...
```

Description

Unlock each *TARGET*. If any *TARGET* is either locked by another user or no valid lock token exists in the working copy, print a warning and continue unlocking the rest of the *TARGET*s. Use `--force` to break a lock belonging to another user or working copy.

Alternate Names

None

Changes

Working Copy, Repository

Accesses Repository

Yes

Options

```
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--force
```

Examples

Unlock two files in your working copy:

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

Unlock a file in your working copy that is currently locked by another user:

```
$ svn unlock tree.jpg
svn: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

Unlock a file without a working copy:

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

For further details, see “Travamento”.

Nome

svn update — Update your working copy.

Synopsis

```
svn update [PATH...]
```

Description

svn update brings changes from the repository into your working copy. If no revision is given, it brings your working copy up-to-date with the HEAD revision. Otherwise, it synchronizes the working copy to the revision given by the `--revision` option. As part of the synchronization, **svn update** also removes any stale locks (see “Sometimes You Just Need to Clean Up”) found in the working copy.

For each updated item, it prints a line that starts with a character reporting the action taken. These characters have the following meaning:

A
: Added

D
: Deleted

U
: Updated

C
: Conflicted

G
: Merged

A character in the first column signifies an update to the actual file, while updates to the file's properties are shown in the second column.

Alternate Names

up

Changes

Working copy

Accesses Repository

Yes

Options

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--no-ignore
--incremental
```

```
--diff3-cmd CMD
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Examples

Pick up repository changes that have happened since your last update:

```
$ svn update
A  newdir/toggle.c
A  newdir/disclose.c
A  newdir/launch.c
D  newdir/README
Updated to revision 32.
```

You can also “update” your working copy to an older revision (Subversion doesn't have the concept of “sticky” files like CVS does; see Apêndice B, *Subversion para Usuários de CVS*):

```
$ 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.
```

If you want to examine an older revision of a single file, you may want to use **svn cat** instead —it won't change your working copy.

svnadmin

svnadmin is the administrative tool for monitoring and repairing your Subversion repository. For detailed information, see “svnadmin”.

Since **svnadmin** works via direct repository access (and thus can only be used on the machine that holds the repository), it refers to the repository with a path, not a URL.

svnadmin Options

`--bdb-log-keep`
: (Berkeley DB specific) Disable automatic log removal of database log files. Having these log files around can be convenient if you need to restore from a catastrophic repository failure.

`--bdb-txn-nosync`
: (Berkeley DB specific) Disables fsync when committing database transactions. Used with the **svnadmin create** command to create a Berkeley DB backed repository with `DB_TXN_NOSYNC` enabled (which improves speed but has some risks associated with it).

`--bypass-hooks`
Bypass the repository hook system.

`--clean-logs`
Removes unused Berkeley DB logs.

`--force-uuid`
By default, when loading data into repository that already contains revisions, **svnadmin** will ignore the `UUID` from the dump stream. This option will cause the repository's `UUID` to be set to the `UUID` from the stream.

`--ignore-uuid`
By default, when loading an empty repository, **svnadmin** will ignore the `UUID` from the dump stream. This option will force that UUID to be ignored (useful for overriding your configuration file if it has `--force-uuid` set).

`--incremental`
Dump a revision only as a diff against the previous revision, instead of the usual fulltext.

`--parent-dir` *`DIR`*
When loading a dump file, root paths at *`DIR`* instead of `/`.

`--revision` (`-r`) *`ARG`*
Specify a particular revision to operate on.

`--quiet`
Do not show normal progress—show only errors.

`--use-post-commit-hook`
When loading a dump file, run the repository's post-commit hook after finalizing each newly loaded revision.

`--use-pre-commit-hook`
When loading a dump file, run the repository's pre-commit hook before finalizing each newly loaded revision. If the hook fails, abort the commit and terminate the load process.

svnadmin Subcommands

Nome

svnadmin create — Create a new, empty repository.

Synopsis

```
svnadmin create REPOS_PATH
```

Description

Create a new, empty repository at the path provided. If the provided directory does not exist, it will be created for you.[1] As of Subversion 1.2, **svnadmin** creates new repositories with the `fsfs` filesystem backend by default.

Options

```
--bdb-txn-nosync
--bdb-log-keep
--config-dir DIR
--fs-type TYPE
```

Examples

Creating a new repository is just this easy:

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

In Subversion 1.0, a Berkeley DB repository is always created. In Subversion 1.1, a Berkeley DB repository is the default repository type, but an FSFS repository can be created using the `--fs-type` option:

```
$ svnadmin create /usr/local/svn/repos --fs-type fsfs
```

[1] Remember, **svnadmin** works only with local *paths*, not *URLs*.

Nome

svnadmin deltify — Deltify changed paths in a revision range.

Synopsis

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

Description

svnadmin deltify exists in current versions of Subversion only for historical reasons. This command is deprecated and no longer needed.

It dates from a time when Subversion offered administrators greater control over compression strategies in the repository. This turned out to be a lot of complexity for *very* little gain, and this “feature” was deprecated.

Options

```
--revision (-r) REV
--quiet (-q)
```

Nome

svnadmin dump — Dump the contents of filesystem to stdout.

Synopsis

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

Description

Dump the contents of filesystem to stdout in a “dumpfile” portable format, sending feedback to stderr. Dump revisions *LOWER* rev through *UPPER* rev. If no revisions are given, dump all revision trees. If only *LOWER* is given, dump that one revision tree. See “Migrating Repository Data Elsewhere” for a practical use.

By default, the Subversion dumpfile stream contains a single revision (the first revision in the requested revision range) in which every file and directory in the repository in that revision is presented as if that whole tree was added at once, followed by other revisions (the remainder of the revisions in the requested range) which contain only the files and directories which were modified in those revisions. For a modified file, the complete fulltext representation of its contents, as well as all of its properties, are presented in the dumpfile; for a directory, all of its properties are presented.

There are two useful options which modify the dumpfile generator's behavior. The first is the `--incremental` option, which simply causes that first revision in the dumpfile stream to contain only the files and directories modified in that revision, instead of being presented as the addition of a new tree, and in exactly the same way that every other revision in the dumpfile is presented. This is useful for generating a relatively small dumpfile to be loaded into another repository which already has the files and directories that exist in the original repository.

The second useful option is `--deltas`. This option causes **svnadmin dump** to, instead of emitting fulltext representations of file contents and property lists, emit only deltas of those items against their previous versions. This reduces (in some cases, drastically) the size of the dumpfile that **svnadmin dump** creates. There are, however, disadvantages to using this option—deltified dumpfiles are more CPU intensive to create, cannot be operated on by **svndumpfilter**, and tend not to compress as well as their non-deltified counterparts when using third-party tools like **gzip** and **bzip2**.

Options

```
--revision (-r) REV
--incremental
--quiet (-q)
--deltas
```

Examples

Dump your whole repository:

```
$ 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
```

…

Incrementally dump a single transaction from your repository:

```
$ 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
…
```

Nome

svnadmin help — Help!

Synopsis

```
svnadmin help [SUBCOMMAND...]
```

Description

This subcommand is useful when you're trapped on a desert island with neither a net connection nor a copy of this book.

Alternate Names

?, h

Nome

svnadmin hotcopy — Make a hot copy of a repository.

Synopsis

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

Description

This subcommand makes a full “hot” backup of your repository, including all hooks, configuration files, and, of course, database files. If you pass the `--clean-logs` option, **svnadmin** will perform a hotcopy of your repository, and then remove unused Berkeley DB logs from the original repository. You can run this command at any time and make a safe copy of the repository, regardless of whether other processes are using the repository.

Options

`--clean-logs`

As described in “Berkeley DB”, hot-copied Berkeley DB repositories are *not* portable across operating systems, nor will they work on machines with a different “endianness” than the machine where they were created.

Nome

svnadmin list-dblogs — Ask Berkeley DB which log files exist for a given Subversion repository (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See “Managing Disk Space” for more information.

Nome

svnadmin list-unused-dblogs — Ask Berkeley DB which log files can be safely deleted (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-unused-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See “Managing Disk Space” for more information.

Examples

Remove all unused log files from a repository:

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

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## disk space reclaimed!
```

Nome

svnadmin load — Read a “dumpfile”-formatted stream from stdin.

Synopsis

```
svnadmin load REPOS_PATH
```

Description

Read a “dumpfile”-formatted stream from stdin, committing new revisions into the repository's filesystem. Send progress feedback to stdout.

Options

```
--quiet (-q)
--ignore-uuid
--force-uuid
--use-pre-commit-hook
--use-post-commit-hook
--parent-dir
```

Example

This shows the beginning of loading a repository from a backup file (made, of course, with **svnadmin dump**):

```
$ svnadmin load /usr/local/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Or if you want to load into a subdirectory:

```
$ svnadmin load --parent-dir new/subdir/for/project /usr/local/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Nome

svnadmin lslocks — Print descriptions of all locks.

Synopsis

```
svnadmin lslocks REPOS_PATH
```

Description

Print descriptions of all locks in a repository.

Options

None

Example

This lists the one locked file in the repository at `/svn/repos`:

```
$ svnadmin lslocks /svn/repos
Path: /tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Nome

svnadmin lstxns — Print the names of all uncommitted transactions.

Synopsis

```
svnadmin lstxns REPOS_PATH
```

Description

Print the names of all uncommitted transactions. See “Removing dead transactions” for information on how uncommitted transactions are created and what you should do with them.

Examples

List all outstanding transactions in a repository.

```
$ svnadmin lstxns /usr/local/svn/repos/
1w
1x
```

Nome

svnadmin recover — Bring a repository database back into a consistent state (applies only to repositories using the bdb backend). In addition, if `repos/conf/passwd` does not exist, it will create a default password file .

Synopsis

```
svnadmin recover REPOS_PATH
```

Description

Run this command if you get an error indicating that your repository needs to be recovered.

Options

```
--wait
```

Examples

Recover a hung repository:

```
$ svnadmin recover /usr/local/svn/repos/
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Recovering the database requires an exclusive lock on the repository. (This is a “database lock”; see Os três significados de “trava”.) If another process is accessing the repository, then **svnadmin recover** will error:

```
$ svnadmin recover /usr/local/svn/repos
svn: Failed to get exclusive repository access; perhaps another process
such as httpd, svnserve or svn has it open?

$
```

The `--wait` option, however, will cause **svnadmin recover** to wait indefinitely for other processes to disconnect:

```
$ svnadmin recover /usr/local/svn/repos --wait
Waiting on repository lock; perhaps another process has it open?

### time goes by…

Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
```

```
The latest repos revision is 34.
```

Nome

svnadmin rmlocks — Unconditionally remove one or more locks from a repository.

Synopsis

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

Description

Remove lock from each *LOCKED_PATH*.

Options

None

Example

This deletes the locks on `tree.jpg` and `house.jpg` in the repository at `/svn/repos`

```
$ svnadmin rmlocks /svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

Nome

svnadmin rmtxns — Delete transactions from a repository.

Synopsis

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

Description

Delete outstanding transactions from a repository. This is covered in detail in “Removing dead transactions”.

Options

```
--quiet (-q)
```

Examples

Remove named transactions:

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

Fortunately, the output of **lstxns** works great as the input for **rmtxns**:

```
$ svnadmin rmtxns /usr/local/svn/repos/  `svnadmin lstxns /usr/local/svn/repos/`
```

Which will remove all uncommitted transactions from your repository.

Nome

svnadmin setlog — Set the log-message on a revision.

Synopsis

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

Description

Set the log-message on revision REVISION to the contents of FILE.

This is similar to using **svn propset --revprop** to set the `svn:log` property on a revision, except that you can also use the option `--bypass-hooks` to avoid running any pre- or post-commit hooks, which is useful if the modification of revision properties has not been enabled in the pre-revprop-change hook.

Revision properties are not under version control, so this command will permanently overwrite the previous log message.

Options

```
--revision (-r) REV
--bypass-hooks
```

Examples

Set the log message for revision 19 to the contents of the file `msg`:

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

Nome

svnadmin verify — Verify the data stored in the repository.

Synopsis

```
svnadmin verify REPOS_PATH
```

Description

Run this command if you wish to verify the integrity of your repository. This basically iterates through all revisions in the repository by internally dumping all revisions and discarding the output—it's a good idea to run this on a regular basis to guard against latent hard disk failures and “bitrot”. If this command fails—which it will do at the first sign of a problem—that means that your repository has at least one corrupted revision and you should restore the corrupted revision from a backup (you did make a backup, didn't you?).

Examples

Verify a hung repository:

```
$ svnadmin verify /usr/local/svn/repos/
* Verified revision 1729.
```

svnlook

svnlook é um comando de console útil para examinar diferentes aspectos do repositório Subversion. Ele não faz nenhuma mudança no repositório—é usado mesmo para dar uma dar uma “espiada”. **svnlook** é usando tipicamente usado pelos hooks do repositório, mas o administrador do repositório pode achá-lo útil como ferramente de diagnóstico.

Já que **svnlook** funciona via acesso direto ao repositório, (e por isso só pode ser usado em máquinas que tenham o repositórios), ele se refere ao repositório por um caminho, não uma URL.

Se nenhuma revisão ou transação for especificada, o padrão do **svnlook** é da revisão mais jovem(mais recente) do repositório.

Opções do svnlook

Opções no **svnlook** são globais, assim como no **svn** e **svnadmin**; entretanto, a maioria das opções apenas se aplicam a um commando já que as funcionalidades do **svnlook** é (intencionalmente) limitado ao escopo.

`--no-diff-deleted`
: Previne **svnlook** de mostrar diferenças entre arquivos deletados. O comportamento padrão quando um arquivo é deletado numa transação/revisão é mostrar as mesmas diferenças que você veria se tivesse deixado o arquivo mas apagado seu conteudo

`--revision` (`-r`)
: Especifica uma revisão em particular que você deseja examinar.

`--revprop`
: Opera uma propriedade da revisão ao invés da propriedade especificada para o arquivo ou diretório. Essa opção exige que você passe a revisão com a opção `--revision` (`-r`) .

`--transaction` (`-t`)
Especifica um ID de transação particular que você deseja examinar.

`--show-ids`
Mostrar o ID da revisão do nodo do sistema de arquivos para cada caminho da árvore do sistema de arquivos.

Sub-comandos do svnlook

Nome

autor svnlook — Mostrar o autor.

Sinópses

```
svnlook author REPOS_PATH
```

Descrição

Mostrar o autor da revisão ou transação no repositório.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

svnlook author é útil, mas não muito excitante:

```
$ svnlook author -r 40 /usr/local/svn/repos
sally
```

Nome

svnlook cat — Mostra o conteudo de um arquivo.

Synopsis

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

Descrição

Mostra o conteudo de um arquivo.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto motra o conteudo de um arquivo em uma transação `ax8`, localizado no `/trunk/README`:

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README

               Subversion, a version control system.
               =====================================

$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

     I. A FEW POINTERS
    II. DOCUMENTATION
   III. PARTICIPATING IN THE SUBVERSION COMMUNITY
…
```

Nome

svnlook changed — Mostra os caminhos que foram mudados.

Synopsis

```
svnlook changed REPOS_PATH
```

Descrição

Mostra os caminhos que foram mudados em uma revisão ou transação particular, assim como “svn update-style” as letras de estatus nas duas primeiras colunas:

'A '
: Item adicionado ao repositório.

'D '
: Item apagado do respostório.

'U '
: Conteúdo do arquivo foi mudado.

' U'
: Propriedades do tem mudados.--FIXME Note the leading space.--

'UU'
: Conteudo e propriedades mudados.

Arquivos e diretórios podem ser distinguidos, como os caminhos dos diretórios são mostrados com caracter'/' .

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra a lista de todos os diretórios e arquivos mudados na revisão 39 em um diretório de teste. Note que a primeira mudança é um diretório, como evidenciado pela `/`:

```
$ svnlook changed -r 39 /usr/local/svn/repos
A   trunk/vendors/deli/
A   trunk/vendors/deli/chips.txt
A   trunk/vendors/deli/sandwich.txt
A   trunk/vendors/deli/pickle.txt
U   trunk/vendors/baker/bagel.txt
 U  trunk/vendors/baker/croissant.txt
UU  trunk/vendors/baker/pretzel.txt
D   trunk/vendors/baker/baguette.txt
```

Nome

svnlook date — Mostrar data-hora.

Synopsis

```
svnlook date REPOS_PATH
```

Descrição

Mostrar data-hora de uma revisão ou transação em um repositório.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Mostra a data da revisão 40 de um repositório de teste:

```
$ svnlook date -r 40 /tmp/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

Nome

svnlook diff — Mostra diferenças de arquivos e propriedades que foram mudados.

Sinopse

```
svnlook diff REPOS_PATH
```

Descrição

Mostra no estilo GNU diferençasde arquivos e propriedades que foram mudados.

Opções

```
--revision (-r) REV
--transaction (-t)
--no-diff-added
--no-diff-deleted
```

Exemplos

Isto mostra um novo arquivo adicionado, deletado e copiado:

```
$ svnlook diff -r 40 /usr/local/svn/repos/
Copied: egg.txt (from rev 39, trunk/vendors/deli/pickle.txt)

Added: trunk/vendors/deli/soda.txt
===================================================================

Modified: trunk/vendors/deli/sandwich.txt
===================================================================
--- trunk/vendors/deli/sandwich.txt (original)
+++ trunk/vendors/deli/sandwich.txt 2003-02-22 17:45:04.000000000 -0600
@@ -0,0 +1 @@
+Don't forget the mayo!

Modified: trunk/vendors/deli/logo.jpg
===================================================================
(Binary files differ)

Deleted: trunk/vendors/deli/chips.txt
===================================================================

Deleted: trunk/vendors/deli/pickle.txt
===================================================================
```

Se um arquivo tem um conteudo que não é texto, propriedade `svn:mime-type`, então as diferenças não são explicitamente mostradas.

Nome

svnlook dirs-changed — Mostra os diretórios que foram mudados.

Synopsis

```
svnlook dirs-changed REPOS_PATH
```

Descrição

Mostra os diretórios que foram mudados (edição de propriedade) ou tiveram seus filhos mudados.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra os difertórios que foram mudados na revisão 40 no nosso respostório de exemplo:

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos
trunk/vendors/deli/
```

Nome

svnlook help — Help!

Sinopses

```
Também svnlook -h e svnlook -?.
```

Descrição

Mostra a mensagem de ajuda para o svnlook. Este comando, como seu irmão **svn help**, é também seu amigo, mesmo que você não ligue mais pra ele e tenha esquecido de convida-lo para sua última festa.

Nomes alternativos

?, h

Nome

svnlook history — Mostra informções sobre o histórico de um caminhos em um repositório (ou da raiz do diretório se nenhum caminho for informado).

Synopsis

```
svnlook history REPOS_PATH [PATH_IN_REPOS]
```

Descrição

Mostra informações sobre histórico de um caminho em um repositório (ou da raiz do diretório se nenhum caminho for informado).

Opções

```
--revision (-r) REV
--show-ids
```

Exemplos

Isto mostra o histórico de um caminho `/tags/1.0` da revisão 20 no nosso repositório de exemplo.

```
$ 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>
```

Nome

svnlook info — Mostra o autor, data-hora, tamanho da mensagem de log, e a mensagem de log.

Synopsis

```
svnlook info REPOS_PATH
```

Descrição

Mostra o autor, data-hora, tamanho da mensagem de log, e a mensagem de log.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra a saída para a revisão 40 no nosso repositório de exemplo.

```
$ svnlook info -r 40 /usr/local/svn/repos
sally
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
15
Rearrange lunch.
```

Nome

svnlook lock — Se o lock existir no caminho do repositório, o descreve.

Synopsis

```
svnlook lock REPOS_PATH PATH_IN_REPOS
```

Descrição

Mostra todas as informações disponíveis para o lock no *`PATH_IN_REPOS`*. Se *`PATH_IN_REPOS`* não estiver lockado, não mostra nada.

Opções

Nada

Exemplos

Descreve o lock do arquivo `tree.jpg`.

```
$ svnlook lock /svn/repos tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Nome

svnlook log — Mostra a mensagem de log.

Synopsis

```
svnlook log REPOS_PATH
```

Descrição

Mostra a mensagem de log.

Opções

```
--revision (-r) REV
--transaction (-t)
```

Exemplos

Isto mostra o log de saída para a revisão 40 no nosso repositório de exemplo:

```
$ svnlook log /tmp/repos/
Rearrange lunch.
```

Nome

svnlook propget — Print the raw value of a property on a path in the repository.

Synopsis

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

Description

List the value of a property on a path in the repository.

Alternate Names

pg, pget

Options

```
--revision (-r) REV
--transaction (-t)
--revprop
```

Examples

This shows the value of the “seasonings” property on the file `/trunk/sandwich` in the `HEAD` revision:

```
$ svnlook pg /usr/local/svn/repos seasonings /trunk/sandwich
mustard
```

Nome

svnlook proplist — Print the names and values of versioned file and directory properties.

Synopsis

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

Description

List the properties of a path in the repository. With `--verbose`, show the property values too.

Alternate Names

pl, plist

Options

```
--revision (-r) REV
--transaction (-t)
--verbose (-v)
--revprop
```

Examples

This shows the names of properties set on the file `/trunk/README` in the `HEAD` revision:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README
  original-author
  svn:mime-type
```

This is the same command as in the previous example, but this time showing the property values as well:

```
$ svnlook --verbose proplist /usr/local/svn/repos /trunk/README
  original-author : fitz
  svn:mime-type : text/plain
```

Nome

svnlook tree — Print the tree.

Synopsis

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

Description

Print the tree, starting at *PATH_IN_REPOS* (if supplied, at the root of the tree otherwise), optionally showing node revision IDs.

Options

```
--revision (-r) REV
--transaction (-t)
--show-ids
```

Examples

This shows the tree output (with node-IDs) for revision 40 in our sample repository:

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids
/ <0.0.2j>
 trunk/ <p.0.2j>
  vendors/ <q.0.2j>
   deli/ <1g.0.2j>
    egg.txt <1i.e.2j>
    soda.txt <1k.0.2j>
    sandwich.txt <1j.0.2j>
```

Nome

svnlook uuid — Print the repository's `UUID`.

Synopsis

```
svnlook uuid REPOS_PATH
```

Description

Print the `UUID` for the repository. the UUID is the repository's *u*niversal *u*nique *id*entifier. The Subversion client uses this identifier to differentiate between one repository and another.

Examples

```
$ svnlook uuid /usr/local/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

Nome

svnlook youngest — Print the youngest revision number.

Synopsis

```
svnlook youngest REPOS_PATH
```

Description

Print the youngest revision number of a repository.

Examples

This shows the youngest revision of our sample repository:

```
$ svnlook youngest /tmp/repos/
42
```

svnsync

svnsync is the Subversion remote repository mirroring tool. Put simply, it allows you to replay the revisions of one repository into another one.

In any mirroring scenario, there are two repositories: the source repository, and the mirror (or “sink”) repository. The source repository is the repository from which **svnsync** pulls revisions. The mirror repository is the destination for the revisions pulled from the source repository. Each of the repositories may be local or remote—they are only ever addressed by their URLs.

The **svnsync** process requires only read access to the source repository; it never attempts to modify it. But obviously, **svnsync** requires both read and write access to the mirror repository.

svnsync is very sensitive to changes made in the mirror repository that weren't made as part of a mirroring operation. To prevent this from happening, it's best if the **svnsync** process is the only process permitted to modify the mirror repository.

svnsync Options

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--no-auth-cache`
: Prevents caching of authentication information (e.g. username and password) in the Subversion administrative directories.

`--non-interactive`
: In the case of an authentication failure, or insufficient credentials, prevents prompting for credentials (e.g. username or password). This is useful if you're running Subversion inside of an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--password` *PASS*
: Indicates that you are providing your password for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--username` *NAME*
: Indicates that you are providing your username for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

svnsync Subcommands

Here are the various subcommands:

Nome

svnsync copy-revprops — Copy all revision properties for a given revision from the source repository to the mirror repository.

Synopsis

```
svnsync copy-revprops DEST_URL REV
```

Description

Because Subversion revision properties can be changed at any time, it's possible that the properties for some revision might be changed after that revision has already been synchronized to another repository. Because the **svnsync synchronize** command operates only on the range of revisions that have not yet been synchronized, it won't notice a revision property change outside that range. Left as is, this causes a deviation in the values of that revision's properties between the source and mirror repositories. **svnsync copy-revprops** is the answer to this problem. Use it to re-synchronize the revision properties for a particular revision.

Alternate Names

None

Options

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Examples

Re-synchronize revision properties for a single revision:

```
$ svnsync copy-revprops file:///opt/svn/repos-mirror 6
Copied properties for revision 6.
$
```

Nome

svnsync initialize — Initialize a destination repository for synchronization from another repository.

Synopsis

```
svnsync initialize DEST_URL SOURCE_URL
```

Description

svnsync initialize verifies that a repository meets the requirements of a new mirror repository—that it has no previous existing version history, and that it allows revision property modifications—and records the initial administrative information which associates the mirror repository with the source repository. This is the first **svnsync** operation you run on a would-be mirror repository.

Alternate Names

init

Options

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Examples

Fail to initialize a mirror repository due to inability to modify revision properties:

```
$ svnsync initialize file:///opt/svn/repos-mirror http://svn.example.com/repos
svnsync: Repository has not been enabled to accept revision propchanges;
ask the administrator to create a pre-revprop-change hook
$
```

Initialize a repository as a mirror, having already created a pre-revprop-change hook which permits all revision property changes:

```
$ svnsync initialize file:///opt/svn/repos-mirror http://svn.example.com/repos
Copied properties for revision 0.
$
```

Nome

svnsync synchronize — Transfer all pending revisions from the source repository to the mirror repository.

Synopsis

```
svnsync synchronize DEST_URL
```

Description

The **svnsync synchronize** command does all the heavy lifting of a repository mirroring operation. After consulting with the mirror repository to see which revisions have already been copied into it, it then begins copying any not-yet-mirrored revisions from the source repository.

svnsync synchronize can be gracefully cancelled and restarted.

Alternate Names

sync

Options

```
--non-interactive
--no-auth-cache
--username NAME
--password PASS
--config-dir DIR
```

Examples

Copy unsynchronized revisions from the source repository to the mirror repository:

```
$ svnsync synchronize file:///opt/svn/repos-mirror
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
…
Committed revision 45.
Copied properties for revision 45.
Committed revision 46.
Copied properties for revision 46.
Committed revision 47.
Copied properties for revision 47.
$
```

svnserve

svnserve allows access to Subversion repositories using Subversion's custom network protocol.

You can run **svnserve** as a standalone server process (for clients that are using the `svn://` access method); you can have a daemon such as **inetd** or **xinetd** launch it for you on demand (also for `svn://`), or you can have **sshd** launch it on demand for the `svn+ssh://` access method.

Regardless of the access method, once the client has selected a repository by transmitting its URL, **svnserve** reads a file named `conf/svnserve.conf` in the repository directory to determine repository-specific settings such as what authentication database to use and what authorization policies to apply. See “svnserve, um servidor especializado” for details of the `svnserve.conf` file.

svnserve Options

Unlike the previous commands we've described, **svnserve** has no subcommands—**svnserve** is controlled exclusively by options.

`--daemon` (`-d`)
: Causes **svnserve** to run in daemon mode. **svnserve** backgrounds itself and accepts and serves TCP/IP connections on the svn port (3690, by default).

`--listen-port=`*`PORT`*
: Causes svnserve to listen on *`PORT`* when run in daemon mode. (FreeBSD daemons only listen on tcp6 by default—this option tells them to also listen on tcp4.)

`--listen-host=`*`HOST`*
: Causes **svnserve** to listen on the interface specified by *`HOST`*, which may be either a hostname or an IP address.

`--foreground`
: When used together with `-d`, this option causes **svnserve** to stay in the foreground. This option is mainly useful for debugging.

`--inetd` (`-i`)
: Causes **svnserve** to use the stdin/stdout file descriptors, as is appropriate for a daemon running out of **inetd**.

`--help` (`-h`)
: Displays a usage summary and exits.

`--version`
: Displays version information, a list of repository back-end modules available, and exits.

`--root=`*`ROOT`* (`-r=`*`ROOT`*)
: Sets the virtual root for repositories served by **svnserve**. The pathname in URLs provided by the client will be interpreted relative to this root, and will not be allowed to escape this root.

`--tunnel` (`-t`)
: Causes **svnserve** to run in tunnel mode, which is just like the **inetd** mode of operation (both modes serve one connection over stdin/stdout, then exit), except that the connection is considered to be pre-authenticated with the username of the current uid. This flag is automatically passed for you by the client when running over a tunnel agent such as **ssh**. That means there's rarely any need for *you* to pass this option to **svnserve**. So if you find yourself typing `svnserve --tunnel` on the command line, and wondering what to do next, see “Tunelamento sobre SSH”.

`--tunnel-user NAME`
: Used in conjunction with the `--tunnel` option; tells svnserve to assume that *`NAME`* is the authenticated user, rather than the UID of the svnserve process. Useful for users wishing to share a single system account over SSH, but maintaining separate commit identities.

`--threads` (`-T`)
When running in daemon mode, causes **svnserve** to spawn a thread instead of a process for each connection (e.g. for when running on Windows). The **svnserve** process still backgrounds itself at startup time.

`--listen-once` (`-X`)
Causes **svnserve** to accept one connection on the svn port, serve it, and exit. This option is mainly useful for debugging.

svnversion

Nome

svnversion — Summarize the local revision(s) of a working copy.

Synopsis

```
svnversion [OPTIONS] [WC_PATH [TRAIL_URL]]
```

Description

svnversion is a program for summarizing the revision mixture of a working copy. The resultant revision number, or revision range, is written to standard output.

It's common to use this output in your build process when defining the version number of your program.

`TRAIL_URL`, if present, is the trailing portion of the URL used to determine if *`WC_PATH`* itself is switched (detection of switches within *`WC_PATH`* does not rely on *`TRAIL_URL`*).

When *`WC_PATH`* is not defined, the current directory will be used as the working copy path. *`TRAIL_URL`* cannot be defined if *`WC_PATH`* is not explicitly given.

Options

Like **svnserve**, **svnversion** has no subcommands, it only has options.

`--no-newline` (`-n`)
: Omit the usual trailing newline from the output.

`--committed` (`-c`)
: Use the last-changed revisions rather than the current (i.e., highest locally available) revisions.

`--help` (`-h`)
: Print a help summary.

`--version`
: Print the version of **svnversion** and exit with no error.

Examples

If the working copy is all at the same revision (for example, immediately after an update), then that revision is printed out:

```
$ svnversion
4168
```

You can add *`TRAIL_URL`* to make sure that the working copy is not switched from what you expect. Note that the *`WC_PATH`* is required in this command:

```
$ svnversion . /repos/svn/trunk
4168
```

For a mixed-revision working copy, the range of revisions present is printed:

```
$ svnversion
4123:4168
```

If the working copy contains modifications, a trailing "M" is added:

```
$ svnversion
4168M
```

If the working copy is switched, a trailing "S" is added:

```
$ svnversion
4168S
```

Thus, here is a mixed-revision, switched working copy containing some local modifications:

```
$ svnversion
4212:4168MS
```

If invoked on a directory that is not a working copy, **svnversion** assumes it is an exported working copy and prints "exported":

```
$ svnversion
exported
```

mod_dav_svn

Nome

`mod_dav_svn` Configuration Directives — Apache configuration directives for serving Subversion repositories through Apache HTTP Server.

Description

This section briefly describes each of the Subversion Apache configuration directives. For an in-depth description of configuring Apache with Subversion, see “httpd, o servidor HTTP Apache”.)

Directives

`DAV svn`
: This directive must be included in any `Directory` or `Location` block for a Subversion repository. It tells httpd to use the Subversion backend for mod_dav to handle all requests.

`SVNAutoversioning On`
: This directive allows write requests from WebDAV clients to result in automatic commits. A generic log message is auto-generated and attached to each revision. If you enable Autoversioning, you'll likely want to set `ModMimeUsePathInfo On` so that `mod_mime` can set `svn:mime-type` to the correct mime-type automatically (as best as `mod_mime` is able to, of course). For more information, see Apêndice C, *WebDAV and Autoversioning*

`SVNPath`
: This directive specifies the location in the filesystem for a Subversion repository's files. In a configuration block for a Subversion repository, either this directive or `SVNParentPath` must be present, but not both.

`SVNSpecialURI`
: Specifies the URI component (namespace) for special Subversion resources. The default is “`!svn`”, and most administrators will never use this directive. Only set this if there is a pressing need to have a file named `!svn` in your repository. If you change this on a server already in use, it will break all of the outstanding working copies and your users will hunt you down with pitchforks and flaming torches.

`SVNReposName`
: Specifies the name of a Subversion repository for use in `HTTP GET` responses. This value will be prepended to the title of all directory listings (which are served when you navigate to a Subversion repository with a web browser). This directive is optional.

`SVNIndexXSLT`
: Specifies the URI of an XSL transformation for directory indexes. This directive is optional.

`SVNParentPath`
: Specifies the location in the filesystem of a parent directory whose child directories are Subversion repositories. In a configuration block for a Subversion repository, either this directive or `SVNPath` must be present, but not both.

`SVNPathAuthz`
: Control path-based authorization by enabling or disabling subrequests. See “Desabilitando Verificação baseada em Caminhos” for details.

Subversion properties

Subversion allows users to invent arbitrarily-named versioned properties on files and directories, as well as unversioned properties on revisions. The only restriction is on properties whose names begin with `svn:`

(those are reserved for Subversion's own use). While these properties may be set by users to control Subversion's behavior, users may not invent new `svn:` properties.

Versioned Properties

`svn:executable`
: If present on a file, the client will make the file executable in Unix-hosted working copies. See “Executabilidade de Arquivo”.

`svn:mime-type`
: If present on a file, the value indicates the file's mime-type. This allows the client to decide whether line-based contextual merging is safe to perform during updates, and can also affect how the file behaves when fetched via web browser. See “Tipo de Conteúdo do Arquivo”.

`svn:ignore`
: If present on a directory, the value is a list of *unversioned* file patterns to be ignored by **svn status** and other subcommands. See “Ignorando Itens Não-Versionados”

`svn:keywords`
: If present on a file, the value tells the client how to expand particular keywords within the file. See “Substituição de Palavra-Chave”.

`svn:eol-style`
: If present on a file, the value tells the client how to manipulate the file's line-endings in the working copy, and in exported trees. See “Seqüência de Caracteres de Fim-de-Linha” and svn export.

`svn:externals`
: If present on a directory, the value is a multi-line list of other paths and URLs the client should check out. See “Definições Externas”.

`svn:special`
: If present on a file, indicates that the file is not an ordinary file, but a symbolic link or other special object[1].

`svn:needs-lock`
: If present on a file, tells the client to make the file read-only in the working copy, as a reminder that the file should be locked before editing begins. See “Lock Communication”.

Unversioned Properties

`svn:author`
: If present, contains the authenticated username of the person who created the revision. (If not present, then the revision was committed anonymously.)

`svn:date`
: Contains the UTC time the revision was created, in ISO 8601 format. The value comes from the *server* machine's clock, not the client's.

`svn:log`
: Contains the log message describing the revision.

`svn:autoversioned`
: If present, the revision was created via the autoversioning feature. See “Autoversioning”.

[1] As of this writing, symbolic links are indeed the only “special” objects. But there might be more in future releases of Subversion.

Repository Hooks

Nome

start-commit — Notification of the beginning of a commit.

Description

The start-commit hook is run before the commit transaction is even created. It is typically used to decide if the user has commit privileges at all.

If the start-commit hook program returns a non-zero exit value, the commit is stopped before the commit transaction is even created, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. authenticated username attempting the commit

Common Uses

access control

Nome

pre-commit — Notification just prior to commit completion.

Description

The pre-commit hook is run just before a commit transaction is promoted to a new revision. Typically, this hook is used to protect against commits that are disallowed due to content or location (for example, your site might require that all commits to a certain branch include a ticket number from the bug tracker, or that the incoming log message is non-empty).

If the pre-commit hook program returns a non-zero exit value, the commit is aborted, the commit transaction is removed, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. commit transaction name

Common Uses

change validation and control

Nome

post-commit — Notification of a successful commit.

Description

The post-commit hook is run after the transaction is committed, and a new revision created. Most people use this hook to send out descriptive emails about the commit or to notify some other tool (such as an issue tracker) that a commit has happened. Some configurations also use this hook to trigger backup processes.

The output from, and exit value returned by the post-commit hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. revision number created by the commit

Common Uses

commit notification, tool integration

Nome

pre-revprop-change — Notification of a revision property change attempt.

Description

The pre-revprop-change hook is run immediately prior to the modification of a revision property when performed outside the scope of a normal commit. Unlike the other hooks, the default state of this one is to deny the proposed action. The hook must actually exist and return a zero exit value before a revision property modification can happen.

If the pre-revprop-change hook doesn't exist, isn't executable, or returns a non-zero exit value, no change to the property will be made, and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision whose property is about to be modified
3. authenticated username attempting the propchange
4. name of the property changed
5. change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program via standard input the proposed value of the property.

Common Uses

access control, change validation and control

Nome

post-revprop-change — Notification of a successful revision property change.

Description

The post-revprop-change hook is run immediately after to the modification of a revision property when performed outside the scope of a normal commit. As can be derived from the description of its counterpart, the pre-revprop-change hook, this hook will not run at all unless the pre-revprop-change hook is implemented. It is typically used to send email notification of the property change.

The output from, and exit value returned by, the post-revprop-change hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path
2. revision whose property was modified
3. authenticated username of the person making the change
4. name of the property changed
5. change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program, via standard input, the previous value of the property.

Common Uses

propchange notification

Nome

pre-lock — Notification of a path lock attempt.

Description

The pre-lock hook runs whenever someone attempts to lock a path. It can be used to prevent locks altogether, or to create a more complex policy specifying exactly which users are allowed to lock particular paths. If the hook notices a pre-existing lock, then it can also decide whether a user is allowed to “steal” the existing lock.

If the pre-lock hook program returns a non-zero exit value, the lock action is aborted and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. versioned path which is to be locked

3. authenticated username of the person attempting the lock

Common Uses

access control

Nome

post-lock — Notification of a successful path lock.

Description

The post-lock hook runs after one or more paths has been locked. It is typically used to send email notification of the lock event.

The output from and exit value returned by the post-lock hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. authenticated username of the person who locked the paths

Additionally, the list of paths locked is passed to the hook program via standard input, one path per line.

Common Uses

lock notification

Nome

pre-unlock — Notification of a path unlock attempt.

Description

The pre-unlock hook runs whenever someone attempts to remove a lock on a file. It can be used to create policies that specify which users are allowed to unlock particular paths. It's particularly important for determining policies about lock breakage. If user A locks a file, is user B allowed to break the lock? What if the lock is more than a week old? These sorts of things can be decided and enforced by the hook.

If the pre-unlock hook program returns a non-zero exit value, the unlock action is aborted and anything printed to stderr is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. versioned path which is to be locked

3. authenticated username of the person attempting the lock

Common Uses

access control

Nome

post-unlock — Notification of a successful path unlock.

Description

The post-unlock hook runs after one or more paths has been unlocked. It is typically used to send email notification of the unlock event.

The output from and exit value returned by, the post-unlock hook program are ignored.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. repository path

2. authenticated username of the person who unlocked the paths

Additionally, the list of paths unlocked is passed to the hook program via standard input, one path per line.

Common Uses

unlock notification

Apêndice A. Guia Rápido de Subversion

Se você está FIXME eager para ter o Subversion FIXME funcionando (e você gosta de aprender experimentando), este capítulo mostrará a você como criar um repositório, importar código, e então FIXME check it back out novamente como uma cópia de trabalho. FIXME Ao longo do caminho, fornecemos links para os FIXME capítulos relevantes deste livro.

Se você FIXME you're new to the entire conceito de controle de versão ou FIXME to the modelo "copiar-modificar-FIXMEmerge" usado tanto pelo CVS quanto pelo Subversion, então você deve ler Capítulo 1, *Conceitos Fundamentais* antes de ir adiante.

Instalando o Subversion

O Subversion é FIXME construído sobre uma camada de portabilidade chamada APR—a biblioteca Apache Portable Runtime. A biblioteca APR fornece todas as interfaces de que o Subversion necessita para funcionar em diferentes sistemas operacionais: acesso a disco, acesso a rede, gerenciamento de memória, e assim por diante. Embora o Subversion seja capaz de usar o Apache como um de seus programas servidores de rede, sua dependência da APR *não* significa que o Apache é um componente obrigatório. A APR é uma biblioteca FIXME standalone usável por qualquer aplicação. Isso significa, entretanto, que, como o Apache, os clientes e servidores Subversion podem ser executados em qualquer sistema operacional no qual o servidor Apache httpd também possa: Windows, Linux, todas as FIXME variedades de BSD, Mac OS X, Netware, e outros.

A forma mais fácil de obter o Subversion é baixar um pacote binário construído para o seu sistema operacional. O website do Subversion (http://subversion.tigris.org) freqüentemente tem esses pacotes disponíveis para serem baixados, FIXME postados por voluntários. O site normalmente contém pacotes com instaladores gráficos para usuários de sistemas operacionais da Microsoft. Se você executa um sistema operacional FIXME semelhante a Unix, pode usar o sistema de distribuição de pacotes nativo do seu sistema (RPMs, DEBs, a árvore de FIXME ports, etc.) para obter o Subversion.

Alternativamente, você pode construir o Subversion diretamente do código-fonte, embora essa nem sempre seja uma tarefa fácil. (Se você não tem experiência em construir pacotes de software de código aberto, provavelmente FIXME se sairá melhor baixando uma distribuição binária!) Do website do Subversion, baixe o último código-fonte lançado. Após FIXME descompactá-lo, siga as instruções do arquivo `INSTALL` para construí-lo. Observe que um pacote de fontes lançado pode não conter tudo o que você precisa para construir um cliente de linha de comando capaz de conversar com um repositório remoto. A partir do Subversion 1.4, as bibliotecas de que o Subversion depende (apr, apr-util, e neon) são distribuídas em um pacote de fontes separado FIXME sufixado com `-deps`. Estas bibliotecas agora são tão comuns que podem já estar instaladas no seu sistema. Se não, você precisará descompactar o pacote de dependências no mesmo diretório onde descompactou os fontes principais do Subversion. FIXME Regardless, é possível que você queira FIXME to fetch outras dependências opcionais como o Berkeley DB e possivelmente o Apache httpd. Se você quiser fazer uma construção completa, assegure-se de ter todos os pacotes documentados no arquivo `INSTALL`.

Se você é uma das pessoas que gostam de usar software FIXME bleeding-edge, também pode obter o código-fonte do Subversion do repositório Subversion onde ele FIXME reside. Obviamente, você precisará já ter um cliente Subversion à mão para fazer isto. Mas uma vez que o tenha, você pode efetuar check out de uma cópia de trabalho do repositório de fontes do Subversion em http://svn.collab.net/repos/svn/trunk/: [1]

[1] Observe que o checkout realizado a partir da URL acima não resulta em `svn`, mas em um subdiretório FIXME thereof chamado `trunk`. Veja nossa discussão do modelo de ramificação e rotulação do Subversion para entender o FIXME motivo por trás disso.

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subversion
A    subversion/HACKING
A    subversion/INSTALL
A    subversion/README
A    subversion/autogen.sh
A    subversion/build.conf
…
```

O comando acima criará uma cópia de trabalho do último código-fonte (não-lançado) do Subversion em um subdiretório chamado `subversion` em seu diretório de trabalho atual. Você pode ajustar o último argumento como FIXME preferir. FIXME Independentemente de como você chame o diretório da nova cópia de trabalho, entretanto, depois que esta operação termina, você agora terá o código-fonte do Subversion. FIXME Obviamente, você ainda precisará FIXME to fetch algumas bibliotecas FIXME auxiliares (apr, apr-util, etc.)—veja o arquivo `INSTALL` no nível superior da cópia de trabalho para detalhes.

Tutorial de alta velocidade

> “Por favor assegure-se de que os FIXME encostos das suas poltronas estão em FIXME posição vertical, e que suas bandejas estão FIXME recolhidas. FIXME Comissárias de bordo, preparem-se para a decolagem….”

O que segue é um rápido tutorial que o conduz através da configuração e operação básicas do Subversion. Quando você terminá-lo, deverá ter um entendimento básico da utilização típica do Subversion.

Os exemplos usados neste apêndice supõem que você tem **svn**, o cliente de linha de comando do Subversion, e **svnadmin**, a ferramenta administrativa, FIXME prontos para uso em um sistema operacional FIXME semelhante a Unix. (Este tutorial também funciona no prompt de linha de comando do Windows, supondo que você faça alguns FIXME ajustes óbvios.) Nós também supomos que você está usando Subversion 1.2 ou posterior (execute **svn --version** para verificar.)

O Subversion armazena todos os dados versionados em um repositório central. Para começar, crie um novo repositório:

```
$ svnadmin create /path/to/repos
$ ls /path/to/repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

Este comando cria um novo diretório `/path/to/repos` que contém um repositório Subversion. Este novo diretório contém (entre outras coisas) uma coleção de arquivos de banco de dados. Você não verá seus arquivos versionados se FIXME you peek inside. Para mais informação sobre criação e manutenção de repositórios, veja Capítulo 5, *Administração do Respositório*.

O Subversion não tem um conceito de “projeto”. O repositório é apenas um sistema de arquivos virtual versionado, uma grande árvore que pode conter qualquer coisa que você desejar. Alguns administradores preferem armazenar apenas um projeto em um repositório, e outros preferem armazenar múltiplos projetos em um repositório colocando-os em diretórios separados. Os méritos de cada abordagem são discutidos em “Planning Your Repository Organization”. FIXME De uma forma ou de outra, o repositório só gerencia arquivos e diretórios, então fica a cargo dos humanos interpretar diretórios particulares como “projetos”. Assim, embora você possa ver referências a projetos ao longo deste livro, tenha em mente que estamos falando apenas FIXME ever sobre algum diretório (ou coleção de diretórios) no repositório.

Neste exemplo, nós supomos que você já tem algum tipo de projeto (uma coleção de arquivos e diretórios) que você deseja importar para o seu recém-criado repositório Subversion. Comece organizando seus

dados em um único diretório chamado `myproject` (ou o que você desejar). Por razões que se tornarão claras depois (veja Capítulo 4, *Fundir e Ramificar*), a árvore da estrutura do seu projeto deve conter três diretórios no nível superior chamados `branches`, `tags`, e `trunk`. O diretório `trunk` deve conter todos os seus dados, enquanto os diretórios `branches` e `tags` estão vazios:

```
/tmp/myproject/branches/
/tmp/myproject/tags/
/tmp/myproject/trunk/
                    foo.c
                    bar.c
                    Makefile
                    …
```

Os subdiretórios `branches`, `tags`, e `trunk` na verdade não são exigidos pelo Subversion. Eles são meramente uma popular convenção que você provavelmente vai querer usar FIXME later on.

Uma vez que você tenha sua árvore de dados FIXME ready to go, importe-a para o repositório com o comando **svn import** (veja “Colocando dados em seu Repositório”):

```
$ svn import /tmp/myproject file:///path/to/repos/myproject -m "initial import"
Adding         /tmp/myproject/branches
Adding         /tmp/myproject/tags
Adding         /tmp/myproject/trunk
Adding         /tmp/myproject/trunk/foo.c
Adding         /tmp/myproject/trunk/bar.c
Adding         /tmp/myproject/trunk/Makefile
…
Committed revision 1.
$
```

Agora o repositório contém esta árvore de dados. Como mencionado anteriormente, você não verá seus arquivos FIXME by directly peeking into the repositório; eles são todos armazenados em um banco de dados. Mas o sistema de arquivos imaginário do repositório agora contém um diretório no nível superior chamado `myproject`, FIXME que por sua vez contém seus dados.

Observe que o diretório `/tmp/myproject` original está inalterado; O Subversion FIXME não sabe disso. (De fato, você pode até mesmo excluir esse diretório se quiser.) Para começar a manipular os dados do repositório, você precisa criar uma nova “cópia de trabalho” dos dados, um tipo de FIXME workspace privado. Peça ao Subversion para “efetuar check out” de uma cópia de trabalho do diretório `myproject/trunk` do repositório:

```
$ svn checkout file:///path/to/repos/myproject/trunk myproject
A  myproject/foo.c
A  myproject/bar.c
A  myproject/Makefile
…
Checked out revision 1.
```

Agora você tem uma cópia pessoal de parte do repositório em um novo diretório chamado `myproject`. Você pode editar os arquivos da sua cópia de trabalho e então submeter essas mudanças de volta para o repositório.

- FIXME Acesse sua cópia de trabalho e edite o conteúdo de um arquivo.

- Execute **svn diff** para ver a FIXME uma saída contendo a diferença unificada das suas mudanças.
- Execute **svn commit** para submeter a nova versão do seu arquivo ao repositório.
- Execute **svn update** para tornar sua cópia de trabalho “atualizada” com o repositório.

Para um FIXME passeio completo por todas as coisas que você pode fazer com sua cópia de trabalho, leia Capítulo 2, *Uso Básico*.

FIXME Neste ponto, você tem a opção de tornar seu repositório disponível FIXME para outros através de uma rede. Veja Capítulo 6, *Configuração do Servidor* para aprender sobre os diferentes tipos de processos de servidor disponíveis e como configurá-los.

Apêndice B. Subversion para Usuários de CVS

Este apêndice é um guia para usuários de CVS novos no Subversion. É essencialmente uma lista das diferenças entre os dois sistemas como são “vistos a 10.000 pés de altura”. Em cada seção, nós fornecemos referências a capítulos relevantes, quando possível.

Embora o objetivo do Subversion seja assumir a atual e futura base de usuários do CVS, algumas novas características e mudanças de projeto foram necessárias para corrigir certos comportamentos “quebrados” que o CVS apresentava. Isto significa que, como um usuário de CVS, você pode precisar mudar hábitos —a começar pelos que você esqueceu que eram estranhos.

Os Números de Revisão Agora São Diferentes

No CVS, os números de revisão são por arquivo. Isso porque o CVS armazena seus dados em arquivos RCS; cada arquivo tem um arquivo RCS correspondente no repositório, e o repositório é organizado aproximadamente de acordo com a estrutura da árvore do seu projeto.

No Subversion, o repositório parece um sistema de arquivos único. Cada submissão resulta em uma árvore de sistema de arquivos inteiramente nova; em essência, o repositório é um conjunto ordenado de árvores. Cada uma dessas árvores é rotulada com um número de revisão único. Quando alguém fala sobre a “revisão 54”, está falando sobre uma árvore em particular (e, indiretamente, sobre a forma que o sistema de arquivos apresentava após a 54ª submissão).

Tecnicamente, não é válido falar sobre a “revisão 5 de `foo.c`”. Em vez disso, diria-se “`foo.c` como aparece na revisão 5”. Também seja cuidadoso ao fazer suposições sobre a evolução de um arquivo. No CVS, as revisões 5 e 6 de `foo.c` são sempre diferentes. No Subversion, é mais provável que `foo.c` *não* tenha mudado entre as revisões 5 e 6.

Similarmente, no CVS um rótulo ou ramo é uma anotação no arquivo, ou na informação de versão para aquele arquivo individual, enquanto no Subversion um rótulo ou ramo é uma cópia de uma árvore inteira (por convenção, nos diretórios `/branches` ou `/tags` que aparecem no nível superior do repositório, ao lado de `/trunk`). No repositório como um todo, muitas versões de cada arquivo podem estar visíveis: a última versão em cada ramo, cada versão rotulada, e, claro, a última versão no próprio tronco. Assim, para refinar ainda mais os termos, poderia-se freqüentemente dizer “`foo.c` como aparece em `/branches/REL1` na revisão 5.”

Para mais detalhes sobre este tópico, veja “Revisões”.

Versões de Diretório

O Subversion rastreia estruturas de árvores, e não apenas o conteúdo dos arquivos. Esta é uma das maiores razões pelas quais o Subversion foi escrito para substituir o CVS.

Aqui está o que isto significa para você, como antigo usuário de CVS:

- Os comandos **svn add** e **svn delete** agora funcionam em diretórios, da mesma forma como funcionam em arquivos. O mesmo vale para **svn copy** e **svn move**. Entretanto, estes comandos *não* causam nenhum tipo de mudança imediata no repositório. Em vez disso, os itens de trabalho são simplesmente “marcados” para adição ou exclusão. Nenhuma mudança no repositório acontece até que você execute **svn commit**.

- Os diretórios não são mais simples contêineres; eles têm números de revisão como os arquivos. (Ou, mais propriamente, é correto dizer “diretório `foo/` na revisão 5”.)

Vamos falar mais sobre esse último ponto. O versionamento de diretórios é um problema difícil; como nós queremos permitir cópias de trabalho de revisões mistas, há algumas limitações no quanto podemos abusar deste modelo.

De um ponto de vista teórico, nós definimos que a “revisão `5` do diretório `foo`” significa uma coleção específica de entradas de diretório e propriedades. Agora suponha que começamos a adicionar e remover arquivos de `foo`, e então submetemos. Seria mentira dizer que nós ainda temos a revisão `5` de `foo`. Entretanto, se nós mudássemos o número de revisão de `foo` depois da submissão, isso também seria falso; pode haver outras mudanças em `foo` que nós ainda não recebemos, porque ainda não atualizamos.

O Subversion lida com este problema rastreando secretamente na área `.svn` as adições e exclusões submetidas. Quando você eventualmente executa **svn update**, todas as contas são acertadas com o repositório, e o novo número de revisão do diretório é determinado corretamente. *Portanto, apenas depois de uma atualização é realmente seguro dizer que você tem uma “perfeita” revisão de um diretório.* Na maior parte do tempo, sua cópia de trabalho conterá revisões de diretório “imperfeitas”.

Similarmente, surge um problema se você tenta submeter mudanças de propriedades em um diretório. Normalmente, a submissão mudaria o número de revisão local do diretório de trabalho. Mas, novamente, isso seria falso, porque pode haver adições ou exclusões que o diretório ainda não tem, porque nenhuma atualização aconteceu. *Portanto, não é permitido que você submeta mudanças de propriedade em um diretório, a não ser que ele esteja atualizado.*

Para mais discussão sobre as limitações do versionamento de diretórios, veja “Revisões Locais Mistas”.

Mais Operações Desconectadas

Nos últimos anos, espaço em disco tornou-se ultrajantemente barato e abundante, mas largura de banda não. Portanto, a cópia de trabalho do Subversion foi otimizada em função do recurso mais escasso.

O diretório administrativo `.svn` serve ao mesmo propósito que o diretório `CVS`, exceto porque também armazena cópias somente-leitura e “intocadas” dos seus arquivos. Isto permite que você faça muitas coisas desconectado:

svn status
: Mostra quaisquer mudanças locais que você fez (see “See an overview of your changes”)

svn diff
: Mostra os detalhes das suas mudanças (see “Examine the details of your local modifications”)

svn revert
: Remove suas mudanças locais (see “Undoing Working Changes”)

Os arquivos originais guardados também permitem que o cliente Subversion envie diferenças ao submeter, o que o CVS não é capaz de fazer.

O último subcomando da lista é novo; ele não apenas removerá mudanças locais, mas irá desmarcar operações agendadas, como adições e exclusões. É a maneira preferida de reverter um arquivo; executar **rm file; svn update** também irá funcionar, mas isso mancha o propósito da atualização. E, por falar nisso…

Distinção Entre Status e Update

No Subversion, nós tentamos dirimir boa parte da confusão entre os comandos **cvs status** e **cvs update**.

O comando **cvs status** tem dois propósitos: primeiro, mostrar ao usuário qualquer modificação local na cópia de trabalho, e, segundo, mostrar ao usuário quais arquivos estão desatualizados. Infelizmente, devido à dificuldade para ler a saída produzida pelo status no CVS, muitos usuários de CVS não tiram nenhuma vantagem deste comando. Em vez disso, eles desenvolveram um hábito de executar **cvs update** ou **cvs -n update** para ver rapidamente suas mudanças. Se os usuários se esquecem de usar a opção `-n`, isto tem o efeito colateral de fundir alterações no repositório com as quais eles podem não estar preparados para lidar.

Com o Subversion, nós tentamos eliminar esta confusão tornando a saída do **svn status** fácil de ler tanto para humanos quanto para programas analisadores. Além disso, **svn update** só imprime informação sobre arquivos que estão atualizados, e *não* modificações locais.

Status

svn status imprime todos os arquivos que têm modificações locais. Por padrão, o repositório não é contatado. Embora este subcomando aceite um bom número de opções, as seguintes são as mais comumente usadas:

`-u`
: Contatar o repositório para determinar, e então mostrar, informações sobre desatualização.

`-v`
: Mostrar *todas* as entradas sob controle de versão.

`-N`
: Executar não-recursivamente (não descer para os subdiretórios).

O comando **status** tem dois formatos de saída. No formato “curto” padrão, modificações locais parecem com isto:

```
$ svn status
M      foo.c
M      bar/baz.c
```

Se você especificar a opção `--show-updates` (`-u`), um formato mais longo de saída é usado:

```
$ svn status -u
M            1047   foo.c
       *     1045   faces.html
       *            bloo.png
M            1050   bar/baz.c
Status against revision:   1066
```

Neste caso, duas novas colunas aparecem. A segunda coluna contém um asterisco se o arquivo ou diretório está desatualizado. A terceira coluna mostra o número de revisão da cópia de trabalho do item. No exemplo acima, o asterisco indica que `faces.html` seria alterado se nós atualizássemos, e que `bloo.png` é um arquivo recém-adicionado ao repositório. (A falta de qualquer número de revisão próximo a `bloo.png` significa que ele ainda não existe na cópia de trabalho.)

A esta altura, você deve dar uma rápida olhada na lista de todos os códigos de status possíveis em svn status. Aqui estão alguns dos códigos de status mais comuns que você verá:

```
A    O recurso está programado para Adição
```

```
D    O recurso está programado para exclusão
M    O recurso tem Modificações locais
C    O recurso tem Conflitos (as mudanças não foram completamente fundidas
       entre o repositório e a versão da cópia de trabalho)
X    O recurso é eXterno a esta cópia de trabalho (pode vir de outro
       repositório).  Veja “Definições Externas”
?    O recurso não está sob controle de versão
!    O recurso está faltando ou incompleto (removido por outra ferramenta que
       não Subversion)
```

Para uma discussão mais detalhada de **svn status**, veja “See an overview of your changes”.

Update

svn update atualiza sua cópia de trabalho, e só imprime informação sobre arquivos que ele atualiza.

O Subversion combinou os códigos `P` e `U` do CVS em apenas `U`. Quando ocorre uma fusão ou conflito, o Subversion simplesmente imprime `G` ou `C`, em vez de uma sentença inteira.

Para uma discussão mais detalhada de **svn update**, veja “Update Your Working Copy”.

Ramos e Rótulos

O Subversion não distingue entre espaço do sistema de arquivos e espaço do “ramo”; ramos e rótulos são diretórios normais dentro do sistema de arquivos. Esta é provavelmente o único maior obstáculo mental que um usuário de CVS precisará escalar. Leia tudo sobre isso em Capítulo 4, *Fundir e Ramificar*.

Visto que o Subversion trata ramos e rótulos como diretórios normais, sempre se lembre de efetuar checkout do `tronco` (`http://svn.example.com/repos/calc/trunk/`) do seu projeto, e não do projeto em si (`http://svn.example.com/repos/calc/`). Se você cometer o erro de efetuar checkout do projeto em si, vai terminar com uma cópia de trabalho que contém uma cópia do seu projeto para cada ramo e rótulo que você tem.[1]

Propriedades de Metadados

Uma nova característica do Subversion é que você pode atribuir um metadado arbitrário (ou “propriedades”) as arquivos e diretórios. Propriedades são pares nome/valor arbitrários associados com arquivos e diretórios na sua cópia de trabalho.

Para atribuir ou obter o nome de uma propriedade, use os subcomandos **svn propset** e **svn propget** . Para listar todas as propriedades de um objeto, use **svn proplist**.

Para mais informações, veja “Properties”.

Resolução de Conflitos

O CVS marca conflitos com “marcadores de conflito” em-linha, e imprime um `C` durante uma atualização. Historicamente, isso tem causado problemas, porque o CVS não está fazendo o suficiente. Muitos usuários esquecem (ou não vêem) o `C` depois que ele passa correndo pelo terminal. Eles freqüentemente esquecem até mesmo que os marcadores de conflitos estão presentes, e então acidentalmente submetem arquivos contendo marcadores de conflitos.

O Subversion resolve este problema tornando os conflitos mais tangíveis. Ele se lembra de que um arquivo encontra-se em um estado de conflito, e não permitirá que você submeta suas mudanças até que execute **svn resolved**. Veja “Resolve Conflicts (Merging Others' Changes)” para mais detalhes.

Arquivos Binários e Tradução

No sentido mais geral, o Subversion lida com arquivos binários de forma mais elegante que o CVS. Por usar RCS, o CVS só pode armazenar sucessivas cópias inteiras de um arquivo binário que está sendo alterado. O Subversion, entretanto, expressa as diferenças entre arquivos usando um algoritmo de diferenciação binária, não importando se eles contêm dados textuais ou binários. Isso significa que todos os arquivos são armazenados diferencialmente (comprimidos) no repositório.

Os usuários de CVS têm que marcar arquivos binários com flags `-kb`, para prevenir que os dados sejam corrompidos (devido a expansão de palavras-chave e tradução de quebras de linha). Eles algumas vezes se esquecem de fazer isso.

O Subversion segue a rota mais paranóica—primeiro, nunca realiza nenhum tipo de tradução de palavra-chave ou de quebra de linha, a menos que você explicitamente o instrua a fazê-lo (veja “Substituição de Palavra-Chave” e “Seqüência de Caracteres de Fim-de-Linha” para mais detalhes). Por padrão, o Subversion trata todos os dados do arquivo como cadeias de bytes literais, e os arquivos sempre são armazenados no repositório em estado não-traduzido.

Segundo, o Subversion mantém uma noção interna de se um arquivo contém dados “de texto” ou “binários”, mas esta noção existe *apenas* na cópia de trabalho. Durante um **svn update**, o Subversion realizará fusões contextuais em arquivos de texto modificados localmente, mas não tentará fazer o mesmo com arquivos binários.

Para determinar se uma fusão contextual é possível, o Subversion examina a propriedade `svn:mime-type`. Se o arquivo não tem a propriedade `svn:mime-type`, ou tem um mime-type que é textual (por exemplo, `text/*`), o Subversion supõe que ele é texto. Caso contrário, o Subversion supõe que o arquivo é binário. O Subversion também ajuda os usuários executando um algoritmo para detectar arquivos binários nos comandos **svn import** e **svn add**. Estes comandos farão uma boa suposição e então (possivelmente) colocarão uma propriedade `svn:mime-type` binária no arquivo que está sendo adicionado. (Se o Subversion fizer uma suposição errada, o usuário sempre pode remover ou editar manualmente a propriedade.)

Módulos sob Controle de Versão

Diferentemente do que ocorre no CVS, uma cópia de trabalho do Subversion sabe que efetuou checkout de um módulo. Isso significa que se alguém muda a definição de um módulo (por exemplo, adiciona ou remove componentes), então uma chamada a **svn update** irá atualizar a cópia de trabalho apropriadamente, adicionando e removendo componentes.

O Subversion define módulos como uma lista de diretórios dentro de uma propriedade de diretório: see “Definições Externas”.

Autenticação

Com o pserver do CVS, exige-se que você “inicie sessão” no servidor antes de qualquer operação de leitura ou escrita—às vezes você tem de iniciar uma sessão até para operações anônimas. Com um repositório Subversion usando Apache **httpd** ou **svnserve** como servidor, você não fornece quaisquer credenciais de autenticação a princípio —se uma operação que você realiza requer autenticação, o servidor irá pedir suas credenciais (sejam essas credenciais nome de usuário e senha, um certificado de cliente, ou mesmo

ambos). Assim, se o seu repositório pode ser lido por todo o mundo, não será exigido que você se autentique para operações de leitura.

Assim como o CVS, o Subversion ainda guarda suas credenciais em disco (em seu diretório `~/.subversion/auth/`), a menos que você o instrua a não fazê-lo, usando a opção `--no-auth-cache`.

A exceção a este comportamento, entretanto, é no caso de se acessar um servidor **svnserve** através de um túnel SSH, usando o esquema de URL `svn+ssh://`. Nesse caso, o programa **ssh** incondicionalmente requer autenticação para iniciar o túnel.

Convertendo um Repositório de CVS para Subversion

Talvez a forma mais importante de familiarizar usuários de CVS com o Subversion é deixá-los continuar trabalhando em seus projetos usando o novo sistema. E mesmo que isso possa de certa forma ser conseguido usando uma importação simples em um repositório Subversion de um repositório CVS exportado, a solução mais completa envolve transferir não apenas o estado mais recente dos seus dados, mas toda a história atrás dele também, de um sistema para o outro. Isto é um problema extremamente difícil de resolver, que envolve deduzir conjuntos de mudanças na falta de atomicidade, e traduzir entre as políticas de ramificação completamente ortogonais dos dois sistemas, entre outras complicações. Todavia, há um punhado de ferramentas prometendo suportar ao menos parcialmente a habilidade de converter repositórios CVS em repositórios Subversion.

A mais popular (e provavelmente a mais madura) ferramenta de conversão é cvs2svn (http://cvs2svn.tigris.org/), um script Python originalmente criado por membros da própria comunidade de desenvovimento do Subversion. Esta ferramenta é destinada a ser executada exatamente uma vez: ela examina seu repositório CVS diversas vezes e tenta deduzir submissões, ramos e rótulos da melhor forma que consegue. Quando termina, o resultado é ou um repositório Subversion ou um arquivo de despejo portável representando a história do seu código. Veja o website para instruções detalhadas e advertências.

Apêndice C. WebDAV and Autoversioning

WebDAV is an extension to HTTP, and is growing more and more popular as a standard for file-sharing. Today's operating systems are becoming extremely Web-aware, and many now have built-in support for mounting “shares” exported by WebDAV servers.

If you use Apache as your Subversion network server, then to some extent you are also running a WebDAV server. This appendix gives some background on the nature of this protocol, how Subversion uses it, and how well Subversion interoperates with other software that is WebDAV-aware.

What is WebDAV?

DAV stands for “Distributed Authoring and Versioning”. RFC 2518 defines a set of concepts and accompanying extension methods to HTTP 1.1 that make the web into a more universal read/write medium. The basic idea is that a WebDAV-compliant web server can act like a generic file server; clients can “mount” shared folders over HTTP that behave much like other network filesystems (such as NFS or SMB.)

The tragedy, though, is that despite the acronym, the RFC specification doesn't actually describe any sort of version control. Basic WebDAV clients and servers assume only one version of each file or directory exists, and can be repeatedly overwritten.

Because RFC 2518 left out versioning concepts, another committee was left with the responsibility of writing RFC 3253 a few years later. The new RFC adds versioning concepts to WebDAV, placing the “V” back in “DAV” — hence the term “DeltaV”. WebDAV/DeltaV clients and servers are often called just “DeltaV” programs, since DeltaV implies the existence of basic WebDAV.

The original WebDAV standard has been widely successful. Every modern computer operating system has a general WebDAV client built-in (details to follow), and a number of popular standalone applications are also able to speak WebDAV— Microsoft Office, Dreamweaver, and Photoshop to name a few. On the server end, the Apache webserver has been able to provide WebDAV services since 1998 and is considered the de-facto open-source standard. There are several other commercial WebDAV servers available, including Microsoft's own IIS.

DeltaV, unfortunately, has not been so successful. It's very difficult to find any DeltaV clients or servers. The few that do exist are relatively unknown commercial products, and thus it's very difficult to test interoperability. It's not entirely clear as to why DeltaV has remained stagnant. Some argue that the specification is just too complex, others argue that while WebDAV's features have mass appeal (even the least technical users appreciate network file-sharing), version control features aren't interesting or necessary for most users. Finally, some have argued that DeltaV remains unpopular because there's still no open-source server product which implements it well.

When Subversion was still in its design phase, it seemed like a great idea to use Apache as a network server. It already had a module to provide WebDAV services. DeltaV was a relatively new specification. The hope was that the Subversion server module (**mod_dav_svn**) would eventually evolve into an open-source DeltaV reference implementation. Unfortunately, DeltaV has a very specific versioning model that doesn't quite line up with Subversion's model. Some concepts were mappable, others were not.

What does this mean, then?

First, the Subversion client is not a fully-implemented DeltaV client. It needs certain types of things from the server that DeltaV itself cannot provide, and thus is largely dependent on a number of Subversion-specific HTTP `REPORT` requests that only **mod_dav_svn** understands.

Second, **mod_dav_svn** is not a fully-realized DeltaV server. Many portions of the DeltaV specification were irrelevant to Subversion, and thus left unimplemented.

There is still some debate in the developer community as to whether or not it's worthwhile to remedy either of these situations. It's fairly unrealistic to change Subversion's design to match DeltaV, so there's probably no way the client can ever learn to get everything it needs from a general DeltaV server. On the other hand, **mod_dav_svn** *could* be further developed to implement all of DeltaV, but it's hard to find motivation to do so—there are almost no DeltaV clients to interoperate with.

Autoversioning

While the Subversion client is not a full DeltaV client, nor the Subversion server a full DeltaV server, there's still a glimmer of WebDAV interoperability to be happy about: it's called autoversioning.

Autoversioning is an optional feature defined in the DeltaV standard. A typical DeltaV server will reject an ignorant WebDAV client attempting to do a `PUT` to a file that's under version control. To change a version-controlled file, the server expects a series of proper versioning requests: something like `MKACTIVITY`, `CHECKOUT`, `PUT`, `CHECKIN`. But if the DeltaV server supports autoversioning, then write-requests from basic WebDAV clients are accepted. The server behaves as if the client *had* issued the proper series of versioning requests, performing a commit under the hood. In other words, it allows a DeltaV server to interoperate with ordinary WebDAV clients that don't understand versioning.

Because so many operating systems already have integrated WebDAV clients, the use case for this feature can be incredibly appealing to administrators working with non-technical users: imagine an office of ordinary users running Microsoft Windows or Mac OS. Each user “mounts” the Subversion repository, which appears to be an ordinary network folder. They use the shared folder as they always do: open files, edit them, save them. Meanwhile, the server is automatically versioning everything. Any administrator (or knowledgeable user) can still use a Subversion client to search history and retrieve older versions of data.

This scenario isn't fiction: it's real and it works, as of Subversion 1.2 and later. To activate autoversioning in **mod_dav_svn**, use the `SVNAutoversioning` directive within the `httpd.conf` Location block, like so:

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

When SVNAutoversioning is active, write requests from WebDAV clients result in automatic commits. A generic log message is auto-generated and attached to each revision.

Before activating this feature, however, understand what you're getting into. WebDAV clients tend to do *many* write requests, resulting in a huge number of automatically committed revisions. For example, when saving data, many clients will do a `PUT` of a 0-byte file (as a way of reserving a name) followed by another `PUT` with the real file data. The single file-write results in two separate commits. Also consider that many applications auto-save every few minutes, resulting in even more commits.

If you have a post-commit hook program that sends email, you may want to disable email generation either altogether, or on certain sections of the repository; it depends on whether you think the influx of emails will still prove to be valuable notifications or not. Also, a smart post-commit hook program can distinguish between a transaction created via autoversioning and one created through a normal **svn commit**. The trick is to look for a revision property named `svn:autoversioned`. If present, the commit was made by a generic WebDAV client.

Another feature that may be a useful complement for `SVNAutoversioning` comes from Apache's `mod_mime` module. If a WebDAV client adds a new file to the repository, there's no opportunity for the user to set the the `svn:mime-type` property. This might cause the file to appear as generic icon when viewed within a WebDAV shared folder, not having an association with any application. One remedy is to have a sysadmin (or other Subversion-knowledgeable person) check out a working copy and manually set the `svn:mime-type` property on necessary files. But there's potentially no end to such cleanup tasks. Instead, you can use the `ModMimeUsePathInfo` directive in your Subversion `<Location>` block:

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

  ModMimeUsePathInfo on

</Location>
```

This directive allows `mod_mime` to attempt automatic deduction of the mime-type on new files that enter the repository via autoversioning. The module looks at the file's named extension and possibly the contents as well; if the file matches some common patterns, then the the file's `svn:mime-type` property will be set automatically.

Client Interoperability

All WebDAV clients fall into one of three categories—standalone applications, file-explorer extensions, or filesystem implementations. These categories broadly define the types of WebDAV functionality available to users. Tabela C.1, “Common WebDAV Clients” gives our categorization and a quick description of some common pieces of WebDAV-enabled software. More details about these software offerings, as well as their general category, can be found in the sections that follow.

	application				WebDAV shares
Macromedia Dreamweaver	Standalone WebDAV application	X			Web production software able to directly read from and write to WebDAV URLs

WebDAV and Autoversioning

Tabela C.1. Common WebDAV Clients

Microsoft Office	Standalone WebDAV application	X			Office productivity suite with several components able to directly read from and write to WebDAV URLs
Microsoft Web Folders	File-explorer WebDAV extension	X			GUI file explorer program able to perform tree operations on a WebDAV share
GNOME Nautilus	File-explorer WebDAV extension			X	GUI file explorer able to perform tree operations on a WebDAV share
KDE Konqueror	File-explorer WebDAV extension			X	GUI file explorer able to perform tree operations on a WebDAV share
Mac OS X	WebDAV filesystem implementation		X		Operating system has built-in support for mounting WebDAV shares.
Novell NetDrive	WebDAV filesystem implementation	X			Drive-mapping program for assigning Windows drive letters to a mounted remote WebDAV share
SRT WebDrive	WebDAV filesystem implementation	X			File transfer software which, among other things, allows the assignment of Windows drive letters to a mounted remote WebDAV share
davfs2	WebDAV filesystem implementation			X	Linux file system driver that allows you to mount a WebDAV share

Standalone WebDAV applications

A WebDAV application is a program which speakes WebDAV protocols with a WebDAV server. We'll cover some of the most popular programs with this kind of WebDAV support.

Microsoft Office, Dreamweaver, Photoshop

On Windows, there are several well-known applications that contain integrated WebDAV client functionality, such as Microsoft's Office, [1] Adobe's Photoshop, and Macromedia's Dreamweaver programs. They're able to directly open and save to URLs, and tend to make heavy use of WebDAV locks when editing a file.

Note that while many of these programs also exist for the Mac OS X, they do not appear to support WebDAV directly on that platform. In fact, on Mac OS X, the File->Open dialog box doesn't allow one to type a path or URL at all. It's likely that the WebDAV features were deliberately left out of Macintosh versions of these programs, since OS X already provides such excellent low-level filesystem support for WebDAV.

Cadaver, DAV Explorer

Cadaver is a bare-bones Unix commandline program for browsing and changing WebDAV shares. Like the Subversion client, it uses the neon HTTP library—not surprisingly, since both neon and cadaver are written by the same author. Cadaver is free software (GPL license) and is available at http://www.webdav.org/cadaver/.

Using cadaver is similar to using a commandline FTP program, and thus it's extremely useful for basic WebDAV debugging. It can be used to upload or download files in a pinch, and also to examine properties, and to copy, move, lock or unlock files:

```
$ cadaver http://host/repos
dav:/repos/> ls
Listing collection `/repos/': succeeded.
Coll: > foobar                                0  May 10 16:19
      > playwright.el                      2864  May  4 16:18
      > proofbypoem.txt                    1461  May  5 15:09
      > westcoast.jpg                     66737  May  5 15:09

dav:/repos/> put README
Uploading README to `/repos/README':
Progress: [============================>] 100.0% of 357 bytes succeeded.

dav:/repos/> get proofbypoem.txt
Downloading `/repos/proofbypoem.txt' to proofbypoem.txt:
Progress: [============================>] 100.0% of 1461 bytes succeeded.
```

DAV Explorer is another standalone WebDAV client, written in Java. It's under a free Apache-like license and is available at http://www.ics.uci.edu/~webdav/. DAV Explorer does everything cadaver does, but has the advantages of being portable and being a more user-friendly GUI application. It's also one of the first clients to support the new WebDAV Access Control Protocol (RFC 3744).

Of course, DAV Explorer's ACL support is useless in this case, since **mod_dav_svn** doesn't support it. The fact that both Cadaver and DAV Explorer support some limited DeltaV commands isn't particularly useful either, since they don't allow `MKACTIVITY` requests. But it's not relevant anyway; we're assuming all of these clients are operating against an autoversioning repository.

[1] WebDAV support was removed from Microsoft Access for some reason, but exists in the rest of the Office suite.

File-explorer WebDAV extensions

Some popular file explorer GUI programs support WebDAV extensions which allow a user to browse a DAV share as if it was just another directory on the local computer, and to perform basic tree editing operations on the items in that share. For example, Windows Explorer is able to browse a WebDAV server as a “network place”. Users can drag files to and from the desktop, or can rename, copy, or delete files in the usual way. But because it's only a feature of the file-explorer, the DAV share isn't visible to ordinary applications. All DAV interaction must happen through the explorer interface.

Microsoft Web Folders

Microsoft was one of the original backers of the WebDAV specification, and first started shipping a client in Windows 98, known as “Web Folders”. This client was also shipped in Windows NT4 and 2000.

The original Web Folders client was an extension to Explorer, the main GUI program used to browse filesystems. It works well enough. In Windows 98, the feature might need to be explicitly installed if Web Folders aren't already visible inside “My Computer”. In Windows 2000, simply add a new “network place”, enter the URL, and the WebDAV share will pop up for browsing.

With the release of Windows XP, Microsoft started shipping a new implementation of Web Folders, known as the “WebDAV mini-redirector”. The new implementation is a filesystem-level client, allowing WebDAV shares to be mounted as drive letters. Unfortunately, this implementation is incredibly buggy. The client usually tries to convert http URLs (`http://host/repos`) into UNC share notation (`\\host\repos`); it also often tries to use Windows Domain authentication to respond to basic-auth HTTP challenges, sending usernames as `HOST\username`. These interoperability problems are severe and documented in numerous places around the web, to the frustration of many users. Even Greg Stein, the original author of Apache's WebDAV module, recommends against trying to use XP Web Folders against an Apache server.

It turns out that the original “Explorer-only” Web Folders implementation isn't dead in XP, it's just buried. It's still possible to find it by using this technique:

1. Go to 'Network Places'.
2. Add a new network place.
3. When prompted, enter the URL of the repository, but *include a port number* in the URL. For example, `http://host/repos` would be entered as `http://host:80/repos` instead.
4. Respond to any authentication prompts.

There are a number of other rumored workarounds to the problems, but none of them seem to work on all versions and patchlevels of Windows XP. In our tests, only the previous algorithm seems to work consistently on every system. The general consensus of the WebDAV community is that you should avoid the new Web Folders implementation and use the old one instead, and that if you need a real filesystem-level client for Windows XP, then use a third-party program like WebDrive or NetDrive.

A final tip: if you're attempting to use XP Web Folders, make sure you have the absolute latest version from Microsoft. For example, Microsoft released a bug-fixed version in January 2005, available at http://support.microsoft.com/?kbid=892211. In particular, this release is known to fix a bug whereby browsing a DAV share shows an unexpected infinite recursion.

Nautilus, Konqueror

Nautilus is the official file manager/browser for the GNOME desktop (http://www.gnome.org), and Konqueror is the manager/browser for the KDE desktop (http://www.kde.org). Both of these applications have an explorer-level WebDAV client built-in, and operate just fine against an autoversioning repository.

In GNOME's Nautilus, from the File menu, select Open location and enter the URL. The repository should then be displayed like any other filesystem.

In KDE's Konqueror, you need to use the `webdav://` scheme when entering the URL in the location bar. If you enter an `http://` URL, Konqueror will behave like an ordinary web browser. You'll likely see the generic HTML directory listing produced by **mod_dav_svn**. By entering `webdav://host/repos` instead of `http://host/repos`, Konqueror becomes a WebDAV client and displays the repository as a filesystem.

WebDAV filesystem implementation

The WebDAV filesystem implementation is arguably the best sort of WebDAV client. It's implemented as a low-level filesystem module, typically within the operating system's kernel. This means that the DAV share is mounted like any other network filesystem, similar to mounting an NFS share on Unix, or attaching an SMB share as drive letter in Windows. As a result, this sort of client provides completely transparent read/write WebDAV access to all programs. Applications aren't even aware that WebDAV requests are happening.

WebDrive, NetDrive

Both WebDrive and NetDrive are excellent commercial products which allow a WebDAV share to be attached as drive letters in Windows. We've had nothing but success with these products. At the time of writing, WebDrive can be purchased from South River Technologies (http://www.southrivertech.com). NetDrive ships with Netware, is free of charge, and can be found by searching the web for “netdrive.exe”. Though it is freely available online, users are required to have a Netware license. (If any of that sounds odd to you, you're not alone. See this page on Novell's website: http://www.novell.com/coolsolutions/qna/999.html)

Mac OS X

Apple's OS X operating system has an integrated filesystem-level WebDAV client. From the Finder, select the Connect to Server item from the Go menu. Enter a WebDAV URL, and it appears as a disk on the desktop, just like any other mounted volume. You can also mount a WebDAV share from the Darwin terminal by using the `webdav` filesystem type with the **mount** command:

```
$ mount -t webdav http://svn.example.com/repos/project /some/mountpoint
$
```

Note that if your **mod_dav_svn** is older than version 1.2, OS X will refuse to mount the share as read-write; it will appear as read-only. This is because OS X insists on locking support for read-write shares, and the ability to lock files first appeared in Subversion 1.2.

One more word of warning: OS X's WebDAV client can sometimes be overly sensitive to HTTP redirects. If OS X is unable to mount the repository at all, you may need to enable the BrowserMatch directive in the Apache server's `httpd.conf`:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Linux davfs2

Linux davfs2 is a filesystem module for the Linux kernel, whose development is located at http://dav.sourceforge.net/. Once installed, a WebDAV network share can be mounted with the usual Linux mount command:

```
$ mount.davfs http://host/repos /mnt/dav
```

Apêndice D. Third Party Tools

Subversion's modular design (covered in “Desing da Camada de Biblioteca”) and the availability of language bindings (as described in “Using Languages Other than C and C++”) make it a likely candidate for use as an extension or backend to other pieces of software. For a listing of many third-party tools that are using Subversion functionality under-the-hood, check out the Links page on the Subversion website (http://subversion.tigris.org/project_links.html).

Apêndice E. Copyright

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

   c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

   d. "Original Author" means the individual or entity who created the Work.

   e. "Work" means the copyrightable work of authorship offered under the terms of this License.

   f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License,

Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

   b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF

TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and

no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

==========

Índice Remissivo