Subversion によるバージョン管理

Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato, 、 Translator:

2006 年 2 月 27 日

だめな「よくある質問集 (FAQ)」には実際にユーザが聞きたいことでは なく、著者がユーザに聞いて欲しいことが書いて あります。おそらく経験があるでしょう:

> Q: チームの生産性を最大にするにはどうやって Glorbo ソフト社の XYZ を使えばよいのでしょう?
>
> A: 顧客の多くは私たちの特許であるオフィスグループ ウェアテクノロジを通じた生産性の向上の方法について知りたいと考えて います。答えは簡単: まず「ファイル」 メニューをクリックし、「生産性 向上」 メニューを選択しましょう、それから …

このような FAQ の問題点は、文字通り FAQ でも何でもないというところ です。技術サポートに電話をして「どうやったら生産性が最大になる のでしょうか?」などと聞く人は一人もいないのです。そうではなく、 本当はもっとずっと具体的な質問がしたいのです、たとえば「どうやったら カレンダーシステムを変更して一日前でなく、二日前に通知するように できますか?」のような。しかし本当の問題点を明らかにするより、 仮想的な FAQ を作るほうがずっとやさしいのです。本当の FAQ を作るには 忍耐強い、組織的な努力が必要なのです: ソフトウェアの一生を通じて やってくる問いを追いかけ、それに対する答えを見守り、それらすべてを 集めて経験の浅いユーザの集約的な経験を反映するように検索可能な形に まとめる必要があります。それは忍耐が必要で、自然主義者のように物事 を黙って観察する態度が必要になります。ここには権威に基づいた仮定や希望的な観測が入り込む余地はありません — 開かれた態度と正確に 物事を記録する態度こそが必要なのです。

この本について私が気に入っているところは、そんな過程を通じて絶えず 本が育っていくところであり、それはすべてのページに現れています。 この本はユーザに対する著者の対峙そのものの結果なのです。それは Subversion メーリングリストで繰り返し問われた基本的な質問を Ben Collins-Sussman が観察することから始まりました: Subversion を使う場合の標準的なワークフローとはいったいどのような ものなのだろうか? ブランチやタグは他のバージョン管理システムと 同じように機能するのだろうか? 誰が特定の変更を加えたということを どうやって把握すればよいのだろうか?

毎日毎日同じ質問を目にすることに強い不満を感じ、Ben は 2002 年の 夏に一ヵ月以上かけて *The Subversion* ハンドブック を書き上げました。これは 60 ページのマニュアルで、Subversion を利用 する際のすべての基本を扱っていました。マニュアルは完成したような 顔をしませんでしたが、Subversion と共に配布され、学習曲線の最初の 障害を取り除きました。O'Reilly and Associates が完全な Subversion の本を出版しようと決めたとき一番手っ取り早い方法は明らかでした: 単に Subversion ハンドブックを拡張すればよかったのです。

新しい本の三人の共著者は普通ではない幸運に恵まれていました。公式には 彼らの仕事は本をトップダウンに書き下すために目次を作ることからはじめ、 最初の版を作ることでした。しかし彼らはまた確固とした — 確かにそれは制御不能な形でわきあがるような性質のものでしたが — 生の素材に直接触れることもしました。Subversion はすでに何千と言う 初期ユーザの手にあり、それらのユーザは無数のフィードバックをもたらしそれは Subversion 本体のみならず、すでに存在しているドキュメントに 対してもそうなのでした。

彼らがこの本を書いている間じゅう、Ben, Mike そして Brian は Subversion メーリングリストとチャットチャンネルをうろつき、注意深く実際の状況下でユーザが実際に陥る本当の問題を記録してきました。 そのようなフィードバックを監視することは、とにかく CollabNet での彼ら の作業の一部だったわけで、このフィードバックは Subversion をドキュメン ト化する上で非常に有益なものでした。 彼らが書き上げたこの本は、そんな作業を反映しています。しっかりとした 経験を基礎とし、希望的観測に流されず、この本はユーザマニュアルと FAQ の最良の部分をまとめたものです。この二重性は一度読んだだけでは 気がつかないでしょう。順

序良く、最初から最後まで、この本はソフトウェア の一片の率直な記述になっています。概略について書かれ、不可欠な同伴 ガイドがあり、管理用設定の章があり、いくつかの進んだトピックに触れ、 そしてもちろんコマンドリファレンスと、障害時の対応法があります。 それは具体的な問題の解法を探しに後で戻ってきてはじめて意味が理解 できるでしょう: そこで語られている詳細は不測の事態に陥った時にしか 関係してきませんし、利用例は本当のユースケースを洗練したものですし ほとんどすべての部分がユーザのニーズとユーザの視点への配慮であふれて います。

もちろん、誰もこの本が Subversion についてのすべての疑問に 答えられるとは約束できる人はいません。質問の期待に、ときどき テレパシーのような精密さで答えることがあるかと思えば、Subversion コミュニティーの知識の中の落とし穴にはまりこんでしまい、手ぶらで 出てくるようなことも、しばしばおこるでしょう。そんな時の一番よい 方法は、users@subversion.tigris.org <mailto:users@subversion.tigris.org>にメールを 送って自分の問題を示すことです。著者らはまだそこにいますし、依然と してリストを監視していますし、本の扉に書かれた三人以外にもたくさんの 人が誤りの訂正や最初の資料について貢献してくれています。コミュニティー の観点から言うと、あなたの問題の解決は単にもっとずっと大きなプロジェクト の喜ばしい副作用でしかありません — そのプロジェクトとはつまり、 ゆっくりとこの本の内容を調整し、そして最終的には Subversion そのものが 実際に利用する人々に、より役立つものにすることです。皆は単にあなたを助けることができるということだけではなく逆に皆を助けるということができる という理由であなたの話しによろこんで耳を貸すでしょう。これは Subversion も 他のすべての活発なフリーソフトウェアプロジェクトでも同じです。 あなたは一人ではないのです。

どうかこの本をあなたの最良の共とせんことを。

— Fogel Karl, Chicago, 14 March, 2004

> 「If C gives you enough rope to hang yourself, think of Subversion as a sort of rope storage facility.」 — Brian W. Fitzpatrick

オープンソースの世界では、コンカレントバージョン管理システム (CVS) が長い間よく使われてきました。またそれは正しい選択でした。CVS は フリーソフトですし、その制約のないワークフローモデルと、ネットワーク機能のサポート — それは地理的に さまざまな場所に分散したプログラマに作業内容を共有させるものですが — は、オープンソースの世界での共同作業のやり方に非常によく合っています。 CVS と、CVS のある程度ルーズな開発手法モデルは、オープンソース文化のかなめになりました。

しかし、どんなツールでもそうですが、CVS も年をとりました。 Subversion は比較的新しいバージョン管理システムで、CVS の後継となるように 設計されています。設計者は二つの方法で CVS ユーザのハートをつかもうとして います。CVS とよく似たデザイン (と、「見栄え」) を持ったオープンソースシステムを 作ることによって、もう一つは、CVS でわかっている欠点のほとんどを 解決しようとすることによって、です。その結果は、バージョン管理システム ソフトの世界に次世代革命をもたらすものではないかも知れませんが、Subversion は確かに とても強力で使いやすく柔軟です。

この本は Subversion バージョン管理システムのバージョン 1.2 系のために 書かれたものです。私たちはできるだけ完全な記述を目指しましたが、Subversion は 活発で精力的な開発コミュニティーを持ち、既に今後計画されているさまざまな機能や 改良点があるため、この本にあるコマンドや特殊な注意事項のいくつかは変更される かも知れません。

対象者

この本はデータを管理するのに Subversion を使おうとする コンピュータの知識のある人たちのために書かれています。Subversion はいろいろな オペレーティングシステム上で動きますが、一番力を入れているユーザインターフェースは コマンドラインベースのものです。それでこの本で議論したり利用するのもコマンド ラインツール ((**svn**) が対象になります。一貫性を保つためこの 本での例は読者が Unix 風のオペレーティングシステムを利用し、Unix と Unix の コマンドラインインターフェースに比較的慣れていることを前提にしています。

しかし **svn** は Microsoft Windows のような Unix 以外の プラットフォームでも動かすことができます。バックスラッシュ (}) をスラッシュ (/) のかわりにパス区切り文字として利用しなくては ならないなどの僅かな違いをのぞけば、Windows 上でこのツールを動作させた時の 入力と出力の内容は Unix のものと同一です。しかし、Windows ユーザは Cygwin Unix エミュレータ環境下でこの本の例を実行すれば、よりよい結果を 得られるかも知れません。

ほとんどの読者はおそらくプログラマか管理者で、ソースコードの 変更内容を追う必要のある人になると思います。それが Subversion の 一番普通の使い方なので、この本の例もそういう状況を前提にしています。 ただ、Subversion はどのようなタイプの情報に対しても変更点を管理するのに 使えます。画像、音楽、データベース、ドキュメント、 などなどにも利用できます。Subversion にとっては、どんな種類のデータも、 単なるデータにすぎません。

この本は読者がいままでバージョン管理システムを一度も使ったことは ないものとして書かれていますが、CVS の利用者に対しては、Subversion への 移行を楽にするように工夫しました。しばしば補足として CVS に

触れるかも知れませんし、特に用意した補遺では、CVS と Subversion の大部分の相違点をまとめて あります。

この本の読み方

この本は非常にさまざまな背景を持った人々にとって有用であることを 目的としています — つまりバージョン管理についてまったく経験のない人から、経験を 積んだシステム管理者までのすべての人たちです。どのような知識を既に持っているか に応じて、特定の章が何らかの意味で重要になるでしょう。以下はさまざまな読者層 ごとの「おすすめの読み方」と考えてください。

経験を積んだシステム管理者　あなたはおそらく既に CVS を利用したことがあり、とっとと Subversion を ダウンロードしてサーバを立ち上げたいのでしょう。第 5 章 と 第 6 章を読めばどのよう にして最初のリポジトリを作り、ネットワーク越しに利用できるようになるかが わかるでしょう。それが済んだら第 3 章 と 付録 A をものにするのが CVS 経験者と してのあなたが Subversion クライアントを理解するのに一番早い方法です。

初心者　あなたの管理者は多分もう Subversion を設定しているはずで、あなたに 必要なのはどうやって Subversion クライアントを利用するかを理解する ことだけです。CVS を利用した経験がないのなら (あるいはバージョン管理 システムなどといったものを一度も使ったことがないのなら)、 第 2 章 と 第 3 章 は粋な とっかかりになります。既に CVS を使ったことがあるので あれば第 3 章と補遺 A から始めるのが最良でしょう。

より進んだユーザ　ユーザであれ管理者であれ、最終的にはあなたのプロジェクトは大きくなって いくでしょう。そして Subversion のより進んだ機能を理解したくなるはず です。たとえばブランチ化とマージ (第 4 章)、 メタデータの設定と実行時オプションの設定 (第 7 章)、 などなどです。このふたつの章は最初はピンとこないかも知れません が、基本的なことを理解した後でぜひ読んでみてください。

開発者　おそらくあなたは既に Subversion になじんでいて、どうやってそれを拡張するか、あるいは Subversion のたくさんある API を使ってどうやって新しいソフトウェア を作るかに興味があるでしょう。第 8 章はまさにそんな人に うってつけです。

この本のしめくくりはリファレンス情報です — 第 9 章は、 すべての Subversion コマンドのリファレンスガイドと、いろいろな役に立つトピック に関する補足情報です。この本全体を一度読み終えたあとで戻ってくるのはきっとこの章でしょう。

この本での約束ごと

ここではこの本で利用されるさまざまな規約について触れます。

印刷上の規約

固定幅　コマンド、コマンド出力、スイッチに使います

イタリックな固定幅　プログラムやテキスト中で置き換え可能なアイテムに対して使います

イタリック　ファイルやディレクトリの名前に使います

アイコン

> 注意
>
>  このアイコンは周りにあるテキストに関連した注意を 表します。

> ティップ
>
>  このアイコンは周りにあるテキストに関連したヘルプ 情報を表します。

> 警告
>
>  このアイコンは回りにあるテキストに関連した警告を 表します。

ソースコードのサンプルは、単なる一例です. 普通のやり方でコンパイルできるとは思いますが、問題点を簡単に示すためのものであり、 良いプログラミングスタイルの例として載せたものではありません。

この本の構成

以下の章とその内容をここで一覧にしておきます:

第 1 章　Subversion の歴史、その機能、構成、 構成要素、そしてインストール方法についての章です。またクイック スタートガイドもあります。

第 2 章　バージョン管理の基礎と異なるバージョン管理モデルを、Subversion の リポジトリ、作業コピー、リビジョンとの関連で説明します。

第 3 章　Subversion ユーザとしての日常的な利用方法に沿った説明をします。 Subversion を使ってどのようにデータを取得し、修正し、コミットするか についてのデモンストレーションです。

第 4 章　ブランチ、マージ、そしてタグについて議論しますが、これにはブランチと マージの最良の方法、一般的な利用例、変更をどうやって取り消すか、 そしてあるブランチから別ブランチにどうやって簡単に乗り換えるかなども 含まれます。

第 5 章　Subversion リポジトリの基本について議論します。どうやってリポジトリを 作成し、設定し、管理するかについて、また、そのためにどんなツールを 利用できるかについても議論します。

第 6 章　Subversion サーバの設定方法と、リポジトリにアクセスする三種類の方法に ついて説明します: HTTP、svn プロトコル、そしてローカルアクセスです。また認証、認可、匿名アクセス についての詳細にも触れます。

第 7 章　Subversion クライアントの設定ファイル、ファイルとディレクトリの 属性、作業コピー中のファイルを無視する方法、 作業コピー中に外部ツリーを含める方法、そして最後にベンダーブランチ の取り扱いについて説明します。

第 8 章　Subversion の内部構造、Subversion ファイルシステム、そして作業コピー の管理領域についてプログラマーの視点から説明します。 Subversion を利用するプログラムを書くために公開された API を使う 例をあげ、そして最も大切なことですが、どうやって Subversion の開発に 貢献するかを示します。

第 9 章　**svn**、**svnadmin**、そして **svnlook** のそれぞれのサブコマンドについてすべてのケースでの豊富な例をまじえながら詳細に 説明します。

付録 A　Subversion と CVS の間の類似点と相違点に触れ、CVS を長年使ってきたことによる 悪い習慣からどうやって抜け出すかについてのさまざまなアドバイスをします。 具体的には Subversion のリビジョン番号、バージョン化されたディレクトリ、オフラインでの操作、**update** と **status** の違い、ブランチ、タグ、メタデータ、衝突の解消、そして認証です。

付録 B　WebDAV と DeltaV の詳細と、DAV 共有を読み書き可能な形にマウントするために どうやって Subversion リポジトリを設定するかを説明します。

付録 C　Subversion を支援したり利用したりするツールについて議論します。これには 別のクライアントプログラム、リポジトリ参照ツール、などが含まれます。

この本はフリーです

この本は Subversion プロジェクト開発チームによって書かれたちょっとしたドキュメント から始めたものを、一つにまとめて書き直したものです。そんなわけで、この本は 常にフリーライセンス下にあります (付録 D を参 照してください)。実際、この本は公開された状況のもとで、Subversion の一部として書かれました。これは二つのことを意味します:

- この本の最新版は、この本専用の Subversion リポジトリにあります。
- フリーライセンスの下で、誰でもこの本を好きなように変更し、配布することができます。 もちろんこの本のプライベートバージョンを配布するよりも、Subversion 開発チームに パッチの形で送ってもらうほうがずっと助かりますが。 コミュニティへの参加方法については項 8.7 を参照してください。

この本の比較的最近のバージョンは、 <http://svnbook.red-bean.com>にあります。

謝辞

この本は Subversion が存在しなければ不可能でした (し、役に立つものに なることもありませんでした)。そういうわけで、著者はこのようなハイリスク で野心的な新しいオープンソースプロジェクトを支援してくれた Brian Behlendorf と CollabNet にまず感謝します; 次に Subversion の名称とその原型を設計した Jim Blandy に対して感謝します — Jim、みな君を愛しているよ; そして最後に 良き友であると同時に偉大なコミュニティー指導者である Karl Fogel に感謝します。 [*1]

O'Reilly と我々の編集者である Linda Mui と、Tatiana Diaz の忍耐と支援にたいして感謝します。

最後に、この本に対して非公式のレビュー、示唆、修正をしてくれた無数の人々に 感謝します: 確かに完全なリストではありませんが、この本は以下の人々の支援なしに は不完全で不正確なものだったでしょう: 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, Matthew Gregan, Art Haas, 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, Mats Nilsson, Joe Orton, Amy Lyn Pilato, Kevin Pilch-Bisson, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, 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, そして Subversion コミュニティー全体に対して。

Ben Collins-Sussman より

Thanks to my wife Frances, who, for many months, got to hear, 「But honey, I'm still working on the book」, rather than the usual, 「But honey, I'm still doing email.」 I don't know where she gets all that patience! She's my perfect counterbalance.

Thanks to my extended family for their sincere encouragement, despite having no actual interest in the subject. (You know, the ones who say, 「Ooh, you're writing a book?」, and then when you tell them it's a computer book, sort of glaze over.)

Thanks to all my close friends, who make me a rich, rich man. Don't look at me that way — you know who you are.

Brian W. Fitzpatrick より

Huge thanks to my wife Marie for being incredibly understanding, supportive, and most of all, patient. Thank you to my brother Eric who first introduced me to UNIX programming way back when. Thanks to my Mom and Grandmother for all their support, not to mention enduring a Christmas holiday where I came home and promptly buried my head in my laptop to work on the book.

To Mike and Ben: It was a pleasure working with you on the book. Heck, it's a pleasure working with you at work!

To everyone in the Subversion community and the Apache Software Foundation, thanks for having me. Not a

[*1] そして、そうだ Karl、君にはこの本のことで、ずいぶん仕事させてしまったね。

day goes by where I don’t learn something from at least one of you.

Lastly, thanks to my Grandfather who always told me that 「freedom equals responsibility.」 I couldn’t agree more.

C. Michael Pilato より

Special thanks to my wife, Amy, for her love and patient support, for putting up with late nights, and for even reviewing entire sections of this book — you always go the extra mile, and do so with incredible grace. Gavin, when you’re old enough to read, I hope you’re as proud of your Daddy as he is of you. Mom and Dad (and the rest of the family), thanks for your constant support and enthusiasm.

Hats off to Shep Kendall, through whom the world of computers was first opened to me; Ben Collins-Sussman, my tour-guide through the open-source world; Karl Fogel — you *are* my `.emacs`; Greg Stein, for oozing practical programming know-how; Brian Fitzpatrick — for sharing this writing experience with me. To the many folks from whom I am constantly picking up new knowledge — keep dropping it!

Finally, to the One who perfectly demonstrates creative excellence — thank you.

第 1 章

導入

1.1

バージョン管理は情報に対する変更を管理するための技法です。 それは小さな変更をソフトウェアにしたあと、次の日にはその変更を取り消すという ような作業をするプログラマにとっては、長い間非常に重要なことでした。 しかしバージョン管理ソフトウェアの有用性はソフトウェア開発の世界をはる かに越えた汎用性があります。頻繁に変更されるような情報を管理しなくては ならないようなコンピュータを使っている人々がいる場所では常にバージョン 管理システムを導入する余地があります。そして Subversion が力を発揮する のはそのような場所においてです。

この章には Subversion の高レベルの導入があります — つまりそれは何であり、 何をするものであり、そしてそのためにはどうしたらよいか、についての導入 です。

1.2 Subversion って何?

Subversion は、フリーなオープンソースのバージョン管理システムで、 時間とともに変化するファイルやディレクトリを管理します。 ファイルの階層構造全体は、リポジトリと呼ばれる 中心的な場所に置かれます。リポジトリは通常のファイルサーバとよく似ていますが メンバーがファイルやディレクトリにしたすべての変更を記録しています。 このため、メンバーは古いバージョンのデータを戻したり、変更履歴を確認したり することができます。この意味で、バージョン管理システムを、「タイムマシン」の一種と考える人も います。

Subversion はリポジトリにネットワーク越しにアクセスするので、別々の コンピュータで作業する人々によって利用することができます。ある範囲で それぞれの場所からの同じデータの集まりをさまざまな人が修正し管理する 仕組みは共同作業を支援することができます。すべての変更を一つの流れに そって行うわけではないので作業効率をより高めることができます。 さらに作業はバージョン化されているので、作業品質が流れを中断 するかどうかの兼ね合いであるかどうかを心配する必要はありません — データに対して間違った変更をしてしまった場合には単にそれを取り消せばよい のです。

バージョン管理システムのいくつかは、ソフトウェア構成管理システム (SCM) でもあります。そういうシステムは、ソースコードのツリーを管理するために 特別便利に作られています — たとえばプログラム言語をじかに理解する ことができたり、ソフトウェアを構成するのに必要なツールが付属していたり といった具合です。しかし Subversion はそのような種類のシステムでは ありません。 Subversion はどのような タイプのファイルの集合も管理できる一般的なシステムです。あなたにとって それはプログラムのソースコードかも知れません — しかし別の人にとっては 食料品の買い物リストから、デジタルビデオの編集、そしてもっと他のもの ですらあるでしょう。

1.3 Subversion の歴史

2000 年の初め、CollabNet, Inc. (<http://www.collab.net>) は CVS の置き換えを 書く開発者を探し始めていました。CollabNet は CollabNet Enterprise Edition (CEE) [*1] という 共同作業用のソフトウェアを提供しています。それはバージョン管理システムをその 一部として含んでいました。CEE は最初のバージョン管理システムと して CVS を利用していましたが、CVS の持っている制限は最初から明らかで あり、CollabNet は最終的にもっと良いものを見つけなくてはならないと悟り ました。不幸にも CVS はオープンソースの世界において 事実上の標準となって いましたが、それは単に、少なくともフリーライセンスの下ではそれより良いも のが何もなかったというのが理由の大部分でした。 そこで CollabNet は一から新しいバージョン管理システムを開発することを決め ました。ただし、CVS の基本的な考え方は保持したまま、バグやまずい実装を含まないようにする形で、です。

2000 年の 2 月、彼らは *Open Source Development with CVS* (Coriolis, 1999) の著者である Karl Fogel に連絡をとり、この新しい プロジェクトに参加する気はないかどうかたずねました。ちょうど同じころ Karl は既に新しいバージョン管理システムの設計について友人の Jim Blandy と 議論していました。1995 年に二人は CVS のサポート契約を提供する会社、 Cyclic Software を設立し、後にそのビジネスを売却しはしましたが、やはり 自分たちの日常の作業に CVS を利用していました。CVS に関する不満がもと で Jim はバージョン化されたデータの管理について、より良い方法を注意深く 考えることになり、「Subversion」 という名前だけではなく、Subversion リポジトリ の基本的な設計についても既に思いついていました。CollabNet が Karl を 呼ぶと彼はすぐにそのプロジェクトで働くことに同意し、また Jim は 雇用主である Red Hat Software が、不定期の期間にわたって彼を事実上 そのプロジェクトに無償で送り込ませることに成功しました。CollabNet は Karl と Ben Collins-Sussman を雇い、5 月から詳細設計が始まりました。 CollabNet の Brian Behlendorf と Jason Robbins、そして Greg Stein (当時は WebDAV/DeltaV の仕様決めを独立した開発者として行っていました が) からのタイミングの良い刺激に助けられ、Subversion は急速に活発な 開発者コミュニティの注意を引きました。多くの人々は CVS について不満を 持っていたことがわかり、最終的に自分たちがその企画に対して何らか貢献 できることを歓迎しました。

最初の設計チームはいくつかのシンプルな目標を決めました。それはバージョン 管理手法の新しい地平を切り開くようなことを目的とはせず、単に CVS の 不具合を修正するものであるとされました。Subversion は CVS の機能に合致し、 同じ開発モデルを踏襲するが、CVS のほとんどの明らかな不具合については 繰り返さないと決められました。そしてそれは CVS を単純な置き換えである必要 はないにせよ、CVS ユーザがわずかな労力によって移行できる程度には十分 似ているべきであるとされました。

14 ヶ月のコーディングの後、Subversion は 2001/8/31 に 「自分で自分自身のソースコード管理」が できるようになりました。Subversion 開発者は、Subversion の自身のソースコード 管理に CVS を使うのをやめて Subversion 自身を使えるようになったということです。

CollabNet がこのプロジェクトを始め、いまだに作業の大部分に出資している わけです (Subversion のフルタイム開発者の給料を払っています) が、Subverion は大部分のオープンソースプロジェクトのように実力主義を促進するような緩やかで オープンないくつかの規則によって成り立っています。CollabNet のコピーライト ライセンスは Debian Free Software Guidelines に完全に合致したものです。 言い換えると、 誰でも自由に Subversion をダウンロードし、修正し、再配布できるということです。 CollabNet や他の誰かの許可を得る必要はありません。

[*1] より小さなグループ作業を狙った CollabNet Team Edition (CTE) という 製品もあります。

1.4 Subversion の機能

Subversion がバージョン管理の問題に提供しようとする機能についての議論 は CVS のデザインをどのように改良したかという観点から話しをすることが しばしば有用です。CVS になじみがないのであればこれらのすべての機能を 理解する必要はありません。そしてバージョン管理についてまったく知らない のであれば、眠くなるだけかも知れません。まず最初に第 2 章を読んでください。バージョン管理システム一般 についての親切な手引きを用意してあります。

Subversion は以下の機能を提供します:

ディレクトリのバージョン化　CVS は個々のファイルの履歴を追うことができるだけですが、 Subversion は時間とともにディレクトリツリー全体の変化も追うことのできる、「仮想的な」バージョン化ファイルシステムを実装しています。 ファイルと、さらに ディレクトリもバージョン付け します。

真のバージョン履歴機能　CVS はファイルのバージョン化に機能が制限されているので、コピー や名称変更 — これはファイルだけではなくディレクトリの内容も 変更する可能性があります — は CVS ではサポートされていません。 さらに CVS では古い履歴を継承しなければ同じ名前の全く新しい ファイル — おそらく全く無関係のファイル — によってすでに バージョン化されているファイルを置き換えることはできません。 Subversion ではファイルとディレクトリの両者に対して追加、削除、 コピー、名称変更をすることができます。そして新規追加されるすべての ファイルは、そこから新しく始まるきれいな履歴を持つことになります。

不分割 (Atomic) なコミット　変更点の集まりは、それ全体がリポジトリに完全に反映されるか、 まったく反映されないかのどちらかです。これにより開発者は 論理的にひとまとまりの変更を作りコミットすることができ、 一部だけがリポジトリに反映されてしまうような問題を回避する ことができます。

バージョン化されたメタデータ　ファイルとディレクトリはそれぞれ関連した属性 — キーと 値の組のことです — を持つことができます。任意の キー/値の組を生成し保存することができます。属性もファイルの内容と 同じようにバージョン化されます。

ネットワーク層の選択　Subversion はリポジトリアクセス用の抽象レイアがあり、新しい ネットワークプログラムを簡単に実装できるようになっています。 Subversion は HTTP サーバの拡張モジュールとして組み込むことも できます。こうすると Subversion は信頼性や相互連携性において 非常に有利になりサーバが提供している既存の機能をすぐに利用 できるようになります — 認証、認可、データ圧縮、などです。 より簡易なスタンドアロンの Subversion プロセスも利用できます。 このサーバは独自のプロトコルによって SSH を利用したトンネル 通信を簡単に実行できます。

データ処理の一貫性　Subversion は、バイナリ差分アルゴリズムを使ってファイルの差分を 表現します。これはテキスト (読むことのできるデータ) にも、バイナリ (簡単に読むことのできないデータ) に対しても同じ方法で働きます。 どちらのタイプのデータもリポジトリ中に同じ形式で圧縮されて格納され、 差分はネットワーク上どちらの方向にも転送されます。

効率的なブランチ、タグの作成　ブランチとタグを作成するコストはプロジェクトのサイズに比例するわけで

は ありません。Subversion はハードリンクとして知られている方法とよく似た方法を使って、 単にプロジェクトをコピーすることでブランチとタグを作ります。そのため ブランチ、タグの作成は非常に短い、一定の時間しかかかりません。

拡張しやすさ　Subversion は歴史的な遺物ではありません。よく設計された API でできた C の共有ライブラリの集まりとして実装されています。 このことは Subversion の保守をとてもやりやすいものにしますし、 他のアプリケーションや 言語から利用しやすいものにします。

1.5 Subversion の構成

図 1.1 は Subversion の「概略」と でも呼べるようなものです。

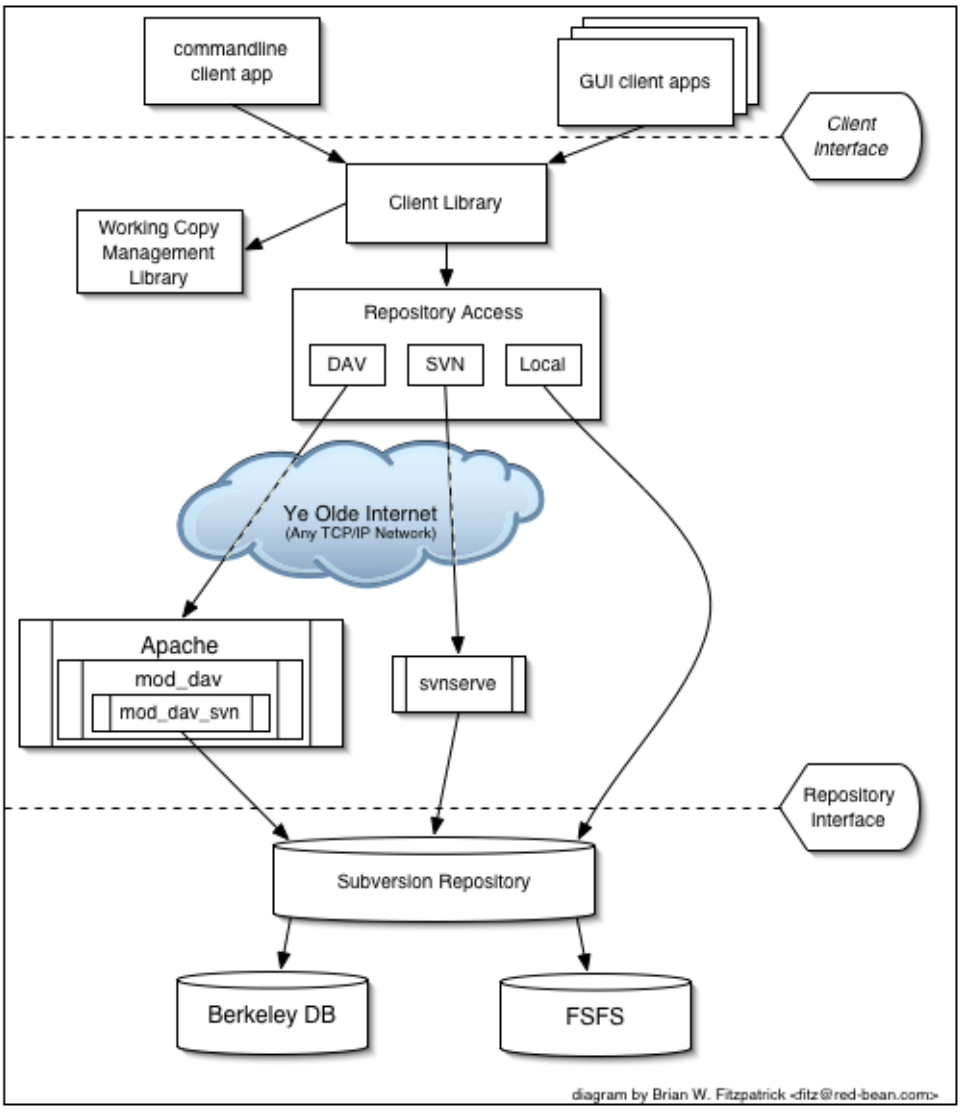

図 1.1　Subversion の構成

一方の端はバージョン化されたすべてのデータがある Subversion リポジトリ です。もう一方はクライアントプログラムで、バージョン化されたデータの ローカルマシン上のコピー (これを「作業コピー」と言います) を 管理します。この二つの間にさまざまなリポジトリアクセス (RA) 層を通じた 通信路があります。そのいく

つかはコンピュータネットワークをまたいで リポジトリにアクセスするためのネットワークサーバ越しに通信します。 他のものはネットワークを利用せず直接リポジトリにアクセスします。

1.6 Subversion のインストール

Subversion は APR (the Apache Portable Runtime library) と呼ばれる 可搬性のあるインターフェースの上に作られています。これで Subversion は Apache の httpd サーバが使えるオペレーティングシステムならどれでも 実行させることができます: Windows, Linux, すべての BSD の変種、Mac OS X, ネットウェア、その他です。

Subversion is built on a portability layer called APR — the Apache Portable Runtime library. The APR library provides all the interfaces that Subversion needs to function on different operating systems: disk access, network access, memory management, and so on. While Subversion is able to use Apache as one of its network server programs, its dependence on APR *does not* mean that Apache is a required component. APR is a standalone library useable by any application. It does mean, however, that like Apache, Subversion clients and servers run on any operating system that the Apache httpd server runs on: Windows, Linux, all flavors of BSD, Mac OS X, Netware, and others.

Subversion を手に入れる一番簡単な方法は自分のオペレーティング システム用のバイナリパッケージをダウンロードすることです。Subversion の ウェブサイト (<http://subversion.tigris.org>) には、ボランティアによって作られたダウンロード可能なパッケージが たくさんあります。このサイトには普通、Microsoft Windows のための グラフィックインストーラパッケージもあります。Unix 系のオペレーティングシステムを使っているなら、(RPMs, DEBs, ports tree などといった、) システム 固有のパッケージ配布システムを使うこともできます。

あるいは直接ソースコードから Subversion を作ることもできます。Subversion ウェブサイトから最新のソースコードリリースを取得してください。解凍した あと `INSTALL` ファイル中の説明に従って作ってください。ソースパッケージには リモートリポジトリにアクセスするためのコマンドラインクライアントを 作るのに必要なものはすべてそろっていますが、(特に apr, apr-util, そして neon ライブラリなど)、Subversion のオプション部分は Berkeley DB や、 潜在的には Apache httpd など、ほかのいろいろなソフトに依存していることに 注意してください。もし完全に ビルドしようとするなら、`INSTALL` ファイルに書かれた すべてのパッケージが手元にあることを確認してください。既にある Subversion リポジトリの上で作業するなら、クライアントプログラムを使っ て、最新の一番新しいソースコードを取得することができます。 このやり方は項 8.7.2 の章に書いてあります。

1.7 Subversion の構成要素

Subversion はさまざま部品からできています。以下はその簡単な概要です。 ここでの簡単な説明で混乱してもあわてないでください; — 混乱を 減らすために非常に多くのページがこの後に用意して ありますので。

svn コマンドラインのクライアントプログラムです。

svnversion 作業コピーの (アイテムが存在するリビジョンに関係した) 状態についての報告をするプログラムです。

svnlook Subversion のリポジトリを調べるためのツールです。

svnadmin Subversion のリポジトリを調整したり修復するためのプログラムで主 システム管理者によって使われます。

svndumpfilter Subversion リポジトリのダンプファイル形式のデータに対 するフィルタプログラムです。

mod_dav_svn Apache HTTP サーバ用のプラグインモジュールです。 リポジトリをネットワーク上の別のユーザが利用できるようにするものです。

svnserve デーモンとして、または SSH から起動されるスタンドアロンのサーバプログラムです。 ネットワーク越しにリポジトリを使えるようにする別の方法です。

Subversion が正しくインストールされていれば、これで利用できるようになって いるはずです。次の二つの章では、コマンドラインクライアントプログラム、 **svn** の使い方を説明します。

1.8 クイックスタート

人によってはこの本での「トップダウン」的なアプローチによって 新しい技術を習得するのが困難かも知れません。この節では Subversion の 非常に短い導入方法を用意し、「ボトムアップ」的な読者にも挑戦の機会を与える ことにします。あなたが経験によって学ぶやり方を好むようなタイプの人であ れば、以下のやり方がうまくいくでしょう。途中、この本の関連した章へのリ ンクをつけてあります。

バージョン管理モデルや CVS と Subversion の両者で利用される 「コピー・修正・マージ」モデルについて全く聞いたことがない のであればまず第 2 章を読んでから先に進んだほうが よいでしょう。

> 注意
>
> 
>
> 以下の例では Subversion のコマンドラインクライアントで ある **svn**、管理用ツールである **svnadmin** が利用可能な形で手元にあることを前提とします。 さらに Subversion 1.2 かそれ以降を利用していることも仮定します (これを確認するには **svn –version** を実行してください)。

Subversion はすべてのバージョン化されたデータを中心的な リポジトリに格納します。最初に新しいリポジトリを作りましょう:

```
$ svnadmin create /path/to/repos
$ ls /path/to/repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

このコマンドは Subversion リポジトリを含む新しいディレクトリ `/path/to/repos` を作ります。 この新しいディレクトリには (他のファイルに混じって) データベースファイルの 集まりを含んでいます。内部を詳細に知る必要がないのであれば このバージョン化されたファイルを見る必要はないでしょう。 リポジトリ生成と保守についてのより詳しい情報は 第 5 章を見てください。

Subversion には「プロジェクト」という概念はありません。 リポジトリは単なる仮想的にバージョン化され

たファイルシステム であり、どんなデータも含むことのできる大きなツリー構造です。 管理者によってはひとつのリポジトリにひとつのプロジェクトだけを 入れることを好みますが、他の管理者はディレクトリを分割した形で 複数のプロジェクトを格納することを好みます。両者のメリット、 デメリットについては項 5.5.1 で議論します。どちらの方法でもリポジトリは単にファイルとディレクトリ を管理するだけなので、特定のディレクトリを「プロジェクト」 であると解釈するかどうかは人間にまかされています。それでこの本をつうじてプロジェクトを参照するときには、リポジトリに存在する、いま言ったような形 のいくつかのディレクトリ (あるいはディレクトリの集まり) についてだけ話を することに注意してください。

この例では、新しく作った Subversion リポジトリにインポートを済ませた 何かのプロジェクト (ファイルとディレクトリの集まり) があるものと 仮定しています。このデータ内容は `myproject` という単一のディレクトリに編成されているものとしましょう (もちろん実際には好きな名前にすることができます)。 後で説明する理由により (第 4 章 参照)、ツリーの構造は `branches`, `tags`, そして `trunk` という 名前の三つの最上位ディレクトリを含む必要があります。 `trunk` ディレクトリはすべてのデータを含んでいる はずですが、`branches` と `tags` ディレクトリは空です:

```
/tmp/myproject/branches/
/tmp/myproject/tags/
/tmp/myproject/trunk/
                    foo.c
                    bar.c
                    Makefile
                    …
```

`branches`, `tags`, `trunk` サブディレクトリは実際には Subversion に 必要なものではありません。後で利用する時におそらくもっとも便利になる ように考えられた、よく利用される命名規約にすぎません。

ツリー中にデータを作ったら **svn import** コマンドでリポジトリにインポートします (項 3.8.2 を見てください):

```
$ 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.
$
```

これでリポジトリにツリーのデータが入りました。この時点で `trunk` ディレクトリの「作業コピー」 を作ります。ここが実際の作業を行う場所になります:

これでリポジトリにツリーのデータが入りました。 すでに注意したように、リポジトリ中のファイルやディ

レクトリを詳しく 調べる必要はありません; すべてはデータベース中に格納されているもの だからです。しかしリポジトリの仮想的なファイルシステムを考えると、 いまの場合、最上位にディレクトリ `myproject` があり、その下にあなたのデータが含まれている形になります。

もとの `/tmp/myproject` にはなにも変更がないことに注意してく ださい。(実際、必要ならこのディレクトリを消してしまうこともできま す)。リポジトリのデータを操作するためには、このデータのために、一種の 個人用の作業領域となる新しい「作業コピー」を作らなくてはな りません。Subversion に、リポジトリの `myproject/trunk` ディレ クトリ用の作業コピーを「チェックアウト」するように指示して みましょう:

```
$ svn checkout file:///path/to/repos/myproject/trunk myproject
A  myproject/foo.c
A  myproject/bar.c
A  myproject/Makefile
…
Checked out revision 1.
```

これで `myproject` という名前の新しい ディレクトリ中にリポジトリのプライベートなコピーを手にしたことに なります。作業コピー中のファイルを編集し、その変更点をリポジトリに 書き戻すためにコミットすることができます。

- 作業コピーに行ってファイル内容を修正します。
- **svn diff** を実行して 変更点に対する unified diff 出力を確認します。
- **svn commit** を実行して リポジトリに自分のファイルの新しいバージョンをコミットします。
- **svn update** を実行して リポジトリの「最新の」状態を自分の作業コピーに反映します。

作業コピーに対してできるすべてのことについての完全な手引き については第 3 章を読んでください。

この時点で、ネットワーク越しに別の人々にリポジトリを 利用可能にすることもできます。 第 6 章を読んで利用可能ないくつかのサーバプロセス の違いについて把握し、どのように設定すれば良いかを理解してください。

第 2 章

基本概念

2.1

この章では Subversion の概要を説明します。 バージョン管理システムの利用が初めての人は、必ずこの章を読んでください。 一般的なバージョン管理の概念から始めて、Subversion の背後にあるアイディア を説明し、Subversion の使い方の簡単な例をお見せします。

この章の例では複数のプログラムソースコードの共有を扱いますが、Subversion はどのようなファイルの集まりも管理できることに注意してください — コンピュータプログラマだけを助けるものではないのです。

2.2 リポジトリ

Subversion は共有情報の一元管理システムです。最も重要なのは リポジトリと呼ばれる、データの格納庫です。リポジトリは情報をファイルシステムツリー — 一般的なファイルとディレクトリの階層構造 — の形で格納します。 任意の数のクライアントがリポジトリにアクセスし このようなファイルの読み書きをします。データを書き込むことでクライアントは 他の人たちがその情報を使えるようにします。データを読み出すことでクライアントは 他の人たちの情報を受け取ります。図 2.1 はこれ を表したものです。

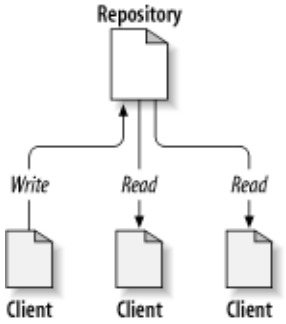

図 2.1　典型的なクライアント/サーバシステム

どうしてこんなことが興味深いのか? ここまでのところでは、典型的なファイルサーバの 定義にすぎないように思います。そして実際、リポジトリはファイルサーバの 一種です。が、普通言うようなものとは少し 違います。Subversion のリポジトリの特徴はそれまで書き込まれた すべての修正をすべて憶えているところ です。すべてのファイル変更についても、また、ディレクトリツリーの自身の変更に ついてもそうです。このような変更は、ファイルやディレクトリの追加、削除、 再配置、などによって起こります。

クライアントがリポジトリからデータを読み出すときには、普通はファイルシステム ツリーの最後のバージョンだけが見えます。が、ファイルシステムの 以前の状態も閲覧することができます。 たとえばクライア

ントは、「先週の水曜日にこのディレクトリにはどのファイルがあったの?」、とか「最後にこのファイルを変更したのは誰で、その人は何を変更したの?」 といった履歴に関する質問をすることができます。 この手の質問はすべてのバージョン管理システム のキモになるような質問です。つまりバージョン管理システムとは時間と共に 修正されるデータを記録したり、修正内容を追跡したりするようにデザイン されています。

2.3 バージョン管理モデル

バージョン管理システムの中核となる役割は共同作業での編集とデー タの共有を可能にすることです。しかしこれにはシステムごとに違った戦略が必要 になります。

2.3.1 ファイル共有の問題

あらゆるバージョン管理システムはどれも基本的な一つの問題を解かなくてはなりません: どうやってユーザに情報を共有させつつ、お互いの変更点が重ならないようにするか、です。 リポジトリ上の別の人の変更を間違って上書きしてしまうことは簡単に 起こりえます。

図 2.2 に示したこんな状況を考えてみてください: 二人の同僚、Harry と Sally がいます。 二人は同時に同じリポジトリ内のファイルを編集することにしました。 もし Harry が先に彼の変更をリポジトリに書き込めば、多分、(その少し あとで) Sally は間違って彼女の新しいバージョンでそれを上書きしてしまう でしょう。Harry のバージョンは永久に失われることはありません (と、いうのは バージョン管理システムはすべての変更を記録しているため) が、 Harry がやった修正は、どれも Sally の新しいバージョンには 現れることがありません。編集時には 彼女は Harry の変更を見ることはできないからです。Harry の作業は、 実質的には失われてしまい、 — あるいは少なくとも最新のバージョンからは 失われてしまい、 — しかもおそらくそれは二人が意図したことではないで しょう。これこそわれわれが避けなくてはならない状況です。

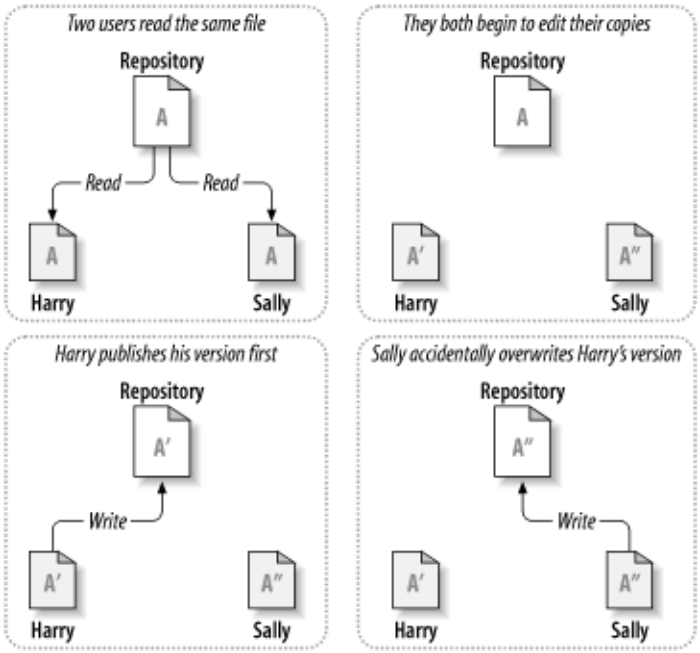

図 2.2 避けなくてはならない問題

2.3.2 ロック・修正・ロック解除の解法

多くのバージョン管理システムでは、ロック・修正・ロック解除のモデルを使ってこの問題を扱います。そのようなシステムではリポジトリ中のファイルを変更できるのは一度に一人だけです。最初 Harry はファイルに変更を加える前に、「ロック」しなくてはなりません。ファイルのロックは、図書館から本を借りるのにいろんな意味でよく似ています。もし Harry がファイルをロックすると、Sally は同じファイルに変更することができなくなります。ロックしようとすれば、リポジトリはその要求を拒否します。彼女ができるのはそのファイルを読むことと、Harry が仕事を終えてロック解除してくれるのを待つことだけです。Harry がロックを解除したあと、彼の番は終わり、今度は Sally がロックして編集することができる番になります。図 2.3 はこの単純な解法の例です。

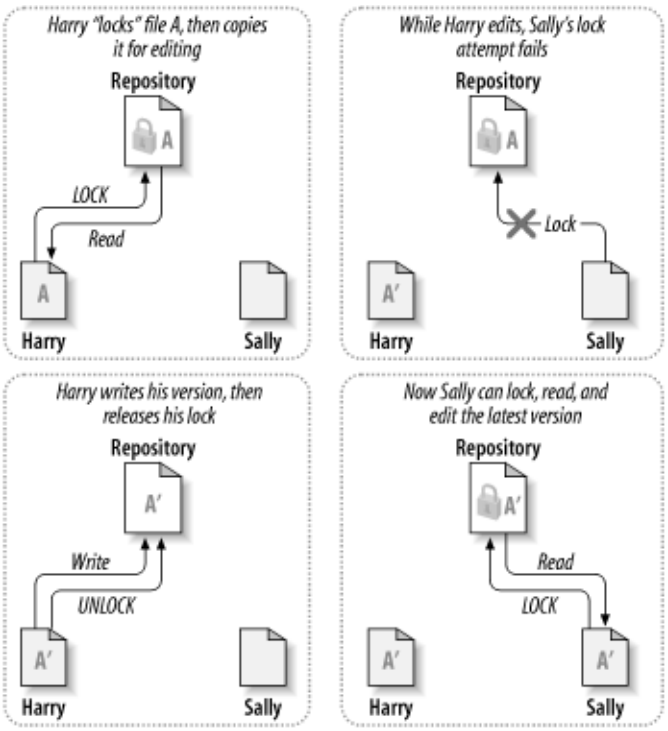

図 2.3 ロック・修正・ロック解除の解法

ロック・修正・ロック解除のモデルの問題は、ファイル管理が少し厳しすぎることで、しばしば、ユーザとって作業の障害になります:

- ロックすることは管理上の問題を起こすかも知れません。ときどき Harry はファイルをロックしたあとでそのことを忘れてしまいます。いっぽう Sally はずっと自分の番を待っているので、その間何もすることができません。そして Harry はそのままバカンスに行ってしまい、Sally としては管理者に対して Harry のロックを解除してもらうように頼まなくてはならなくなります。この状況は不要な遅れと、時間の無駄を起こします。
- ロックは不要な直列化を起こすかも知れません。Harry はそのテキストファイルの先頭の部分を修正して、Sally は同じファイルの最後の部分を修正したいだけだとしたら? 二人の修正はまったく重なっていません。適当な形でマージされることさえ保証できれば、二人は同じファイルを同時に編集することができ、それが大きな問題にはならないでしょう。
- ロックは間違った意味の安心感を与えてしまう場合があります。Harry がファイル A をロックしてか

ら編集し、一方 Sally は同時にファイル B を ロックしてから編集しているとします。しかしここで A と B とは意味的に 依存しあっていて、それぞれに対する独立した変更は両立しないとしましょう。 突然 A と B はもう一緒に動作しなくなります。ロックを使ったシステムは このような状況には無力です。 — これはある意味で、間違った意味の 安心感を与えてしまっています。Harry や Sally が、ファイルを ロックする ことでそれぞれ安全な状態に入り、自分の作業は他人から分離されていると 錯覚することは簡単に起こりえます。このことが、最初に述べたような 実は両立しない変更についての議論を妨げてしまうかも知れません。

2.3.3 コピー・修正・マージの解法

Subversion, CVS, その他のバージョン管理システムはロックに変わる アイディアとしてコピー・修正・マージモデルを 使います。このモデルではユーザごとのクライアントプログラムはプロジェクト リポジトリにアクセスして自分だけの作業コピーを 作ります — それはリポジトリにあるファイルやディレクトリをローカルに コピーしてきたものです。それからユーザは ひとりひとりが平行して作業をし、自分の作業コピーを修正します。 最後に自分のコピーは最終的な新しいバージョンにマージされます。 このバージョン管理システムは大部分のマージを手伝いますが 最終的には正しいマージかどうかについては人が責任を持ちます。 ユーザは平行して作業し、変更を同じファイル、ただしそれぞれの作業コピー である”A”に対して行います。

例をあげます。 Harry と Sally が同じプロジェクトに対するそれぞれの作業コピーをリポジトリ の内容をコピーして作ったとします。 彼らは平行して作業し、変更をまずは自分の作業コピーの同じファイル A に対して 行います。Sally は自分の変更を先にリポジトリに保存します。 Harry が変更をあとで保存したいと思ったとき、リポジトリは、彼に対して A は既に最新ではないことを伝えます。 言い換えると、リポジトリにあるファイル A は彼がそれをコピーした後で 別の人によって修正されていることを伝えます。そこで Harry は、Subversion のクライアントプログラムに、自分の作業コピー A に対して、リポジトリにある 新しい変更点をマージするように要求します。 Sally の変更が彼のもので上書きされることはありえません。ひとたび彼が 両方の変更を統合してしまえば、自分の作業コピーをリポジトリに書き戻す ことができます。図 2.4 と 図 2.5 はこの処理を示しています。

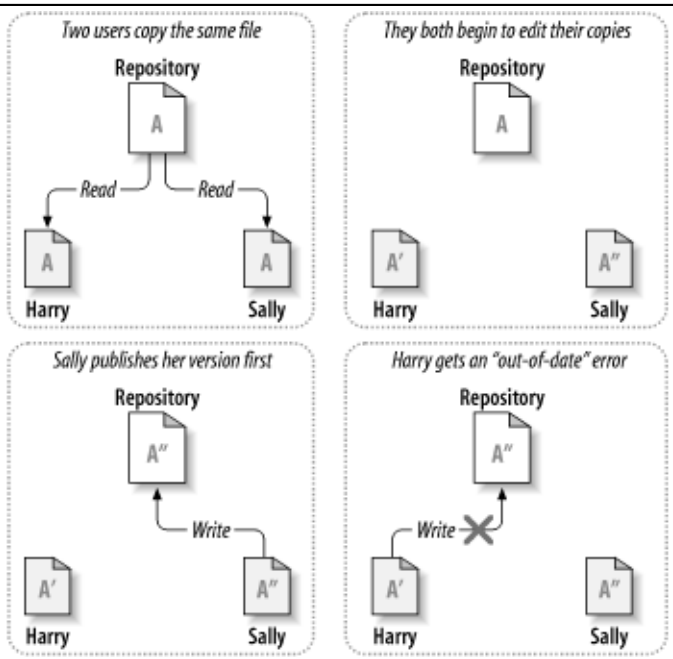

図 2.4 コピー・修正・マージの解法

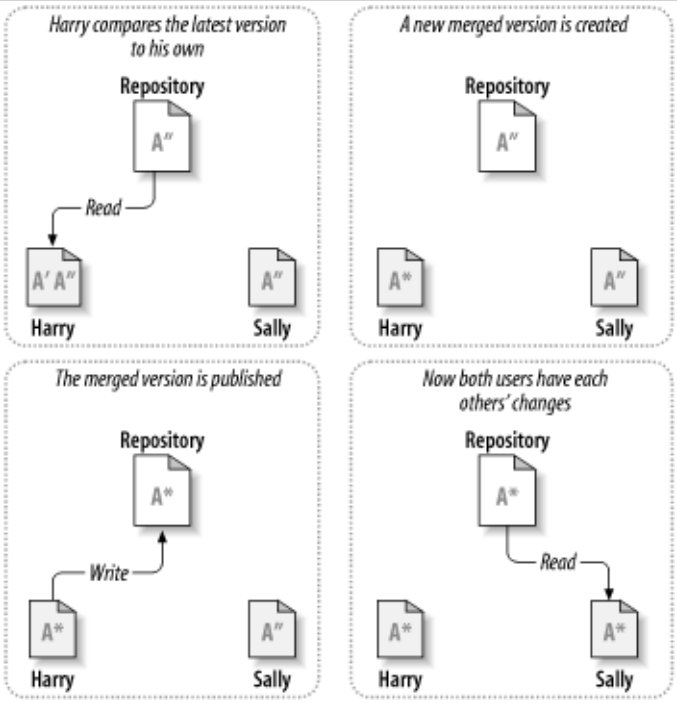

図 2.5 コピー・修正・マージの解法 (続き)

しかし、Sally の変更点が Harry のと重なって いたら? そのときはどうなるのでしょう? この状況は衝突と呼ばれ、普通はあまり大きな問題にはなりません。 Harry が Subversion クライアントプログラムにリポジトリの最新の変更を 自分の作業コピーにマージするように要求したとき、彼の A ファイルの作業 コピーは、衝突の状態としてマークされます。彼は両方の変更の衝突した 部分を見ることができ、どちらを選ぶかを選択します。ソフトウェア自体が 自動的に衝突を解決することはできないのに注意してください; 人間だけが 理解し、正しく選択する力を持っています。Harry がいったん重なっている 部分の修正を手で解消したら — たぶん Sally と衝突について話し合った あと — マージされたファイルをリポジトリに安全に書き戻すことが できます。

コピー・修正・マージのモデルは少々混沌としているように 思うかも知れませんが、実際にはとてもスムーズに行きます。 ユーザは平行して作業することができ、相手の修正を待つことはありません。 同じファイルに対して変更するときでも、ほとんどの変更は、まったく重ならない ことがわかります。そして、衝突を解消するのにかかる時間は、ロックする システムで失われる時間よりもずっと短いのです。

最終的に、これは一つの重要な要因に行き着きます: ユーザ間の コミュニケーションです。ユーザがお互いにあまり意見のやり取りをしなければ、 両方の構文上の、また意味の上の衝突は増えます。どんなシステムもユーザに完全な意思の疎通を強制することはできないので、意味上の衝突を 検出することはできません。そういうわけで、ロックするシステムが衝突を 回避することができるという間違った保証に安心する理由はありません。 実際には、ロックは生産性を落とす以外のなにものでもないように見えます。ロックが必要な場合

ロック・修正・ロック解除のモデルは一般的には複数で協力して作業する場合には 有害だと考えられていますが、場合によってはロックが有効なこともあります。

コピー・修正・マージモデルはファイルが文脈を考慮した上でマージ可能である という前提に基づいています: つまり、リポジトリ中の大部分のファイルが 行単位のテキストファイルである場合を想定しています (プログラムのソース コードなどがその典型です)。しかしバイナリ形式のファイル、たとえば画像や 音声などの場合、衝突した部分の変更点をマージすることは不可能なのが 普通です。そのような場合に必要なのは、ユーザはファイルを変更する上で、 厳密に自分の順番を確保したいということです。順序性を保った形でのアクセ

ス がなければ、最終的には捨ててしまうことになるある人の修正によって時間が 無駄になってしまいます。

CVS と Subversion は原則としてはコピー・修正・マージのシステムですが 両者とも場合によってはファイルにロックする必要があることを認め その仕組みを用意しています。 項 7.4 を見てください。

2.4 実行中の Subversion

そろそろ抽象論から具体的な議論に移るときがきました。この章で は Subversion が利用される実際の例をお見せします。

2.4.1 作業コピー

既に作業コピーについて読んできたことと思いますので、Subversion のクライアント プログラムが作業コピーを作ったり使ったりする様子を見てみます。

Subversion 作業コピーは、自分のローカルシステム上の普通の ディレクトリツリーで、その中には複数のファイルがあります。あなたは 望むファイルを編集することができ、ソースコードファイルなら、それを 普通にコンパイルすることができます。作業コピーは自分だけの作業領域 です: Subversion はほかの人の変更を持ち込んだりしませんし、明示的に そうしてくれと言うまで、自分の変更を他の人に見せたりすることも ありません。同じプロジェクト用に一人で複数の作業コピーを持つことさえできます。

作業コピーのファイルに変更を加え、それがうまく動作することを 確認したあとで、Subversion はその変更を同じプロジェクトであなたと一緒に 作業しているほかの人に「公開」するためのコマンドを (リポジトリに書き込むことで) 用意します。もし他の人が自分自身の変更を公開したとき には Subversion はその変更を自分の作業コピーにマージするコマンドを用意します。 (リポジトリの内容を読み出すことで。)

作業コピーには、Subversion によって管理される、いくつかの特殊な ファイルもあり、その助けによって (読み出し、書き込みなどの) コマンドを 実行します。特に作業コピー中のディレクトリには `.svn` という名前の、管理ディレクトリとして知られる サブディレクトリがあります。管理ディレクトリのそれぞれのファイルは Subversion がどのファイルにまだ公開していない変更があるか、どのファイルが 他の人の作業によって最新でなくなっているか理解するのを助けるものです。

典型的な Subversion リポジトリは複数のプロジェクトのファイル (またはソースコード) をつかんでいます。普通、それぞれのプロジェクトは リポジトリのファイルシステムツリー中のサブディレクトリになっています。 この構成によって、ユーザの作業コピーは普通、リポジトリの特定の 部分木に対応しています。

たとえば二つのソフトウェアプロジェクト、`paint` と `calc` を含むリポジトリが あるとします。それぞれのプロジェクトはそれぞれの最上位サブディレクトリ にあります。図 2.6 のような状況です。

作業コピーを持ってくるため、リポジトリ中のどれかのサブツリーを チェックアウトしなくてはなりません。 (「check out」 という言葉は何かをロックしたり保護したり するような響きがありますがそうではありません; それは単に自分のための プロジェクトのコピーを作るだけです。) たとえば `/calc` をチェックアウトするとこんな 感じで作業コピーを手に入れることができます:

```
$ 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
```

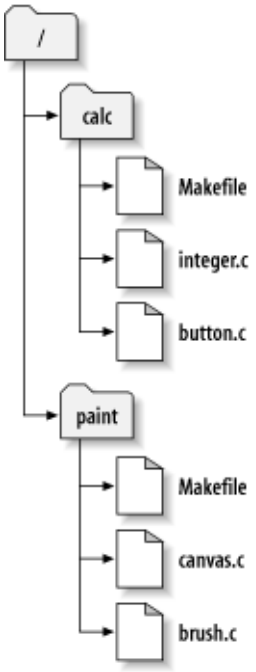

図 2.6 リポジトリのファイルシステム

```
Makefile  integer.c  button.c  .svn/
```

A の文字で始まる一覧は Subversion があなたの作業コピーにいくつかの ファイルを追加したことを示しています。これでリポジトリにある `/calc` ディレクトリの作業コピーを 持ってくることができました。最初に言ったように、この取得時に は、もう一つ、`.svn` が 作成されますが、これが Subversion に必要な追加情報を格納する ための場所になります。リポジトリの URL

Subversion のリポジトリにはいろいろな方法でアクセスすることができます — それはローカルディスクにあるかも知れませんし、さまざまなネット ワークプロトコルを通じてかも知れません。しかしどのような場合でも、リポ ジトリの場所は、常に URL によって表現されます。 表 2.1 はどのようにして異なる URL が適切なアクセ ス方法に結び付けられるかを示しています。

表 2.1 リポジトリにアクセスするための URL

Schema	アクセス方法
`file:///`	リポジトリへの直接アクセス (ローカルディスク上)
`http://`	Subversion を考慮した Apache サーバへの WebDAV プロトコル経由でのアクセス
`https://`	`http://`と同じだが、SSL による暗号化
`svn://`	`svnserve` サーバに対する独自 TCP/IP プロトコル経由での アクセス
`svn+ssh://`	`svn://`と同じですが、SSH トンネルを利用します。

Subversion が URL をどのように構文解析するかについてのより詳しい情報は 項 7.10 を見てください。

`button.c` に変更を加えることを考えてみます。 `.svn` ディレクトリがファイルの修正時刻と もともとの内容を記憶しているので、Subversion はあなたがファイルを 変更したかどうかを見分けることができます。し

かし Subversion は明示的に そうしてくれと言われるまでその変更を公にはしません。 自分の変更を公開する操作のことを変更点のコミット (あるいは チェックイン) と言います。

変更点を他の人に公開するには Subversion の **commit** コマンドを使います:

```
$ svn commit button.c
Sending        button.c
Transmitting file data .
Committed revision 57.
```

これで `button.c` への変更はリポジトリに コミットされました。もし別のユーザが`/calc` の作業コピーを作るのにチェックアウトすれば、最新バージョン中に あなたの変更点を見ることになるでしょう。

一緒に作業している Sally が、あなたがチェックアウトしたのと同じ時刻に `/calc` の作業コピーを自分用にチェックアウト したとしましょう。あなたが `button.c` への自分の 変更をコミットしても、Sally の作業コピーは変更されない状態のままです。 Subversion はユーザの要求によって初めて作業コピーの内容を変更します。

作業内容をプロジェクトの最新の状態にするには、Sally は Subversion に 自分の作業コピーを更新 するように依頼しなくては なりません。これには **update** コマンドを使います。 これはあなたの変更を彼女の作業コピーにマージしますし、彼女がチェック アウトしたあとで他の人がコミットしたすべての部分についてもマージします。

```
$ pwd
/home/sally/calc

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

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

svn update コマンドからの出力は Subversion が `button.c` の内容を 更新したことを示しています。Sally はどのファイルを更新 するかを指定する必要がないのに注意してください。Subversion は `.svn` ディレクトリの情報と リポジトリの情報を使って、どのファイルを更新しなくてはならないか を決定します。

2.4.2 リビジョン

svn commit 操作は一つのトランザクション として任意の数のファイル、ディレクトリに対する変更点を公開する ことができます。作業コピー中で、ファイルの内容を変えたり、新しい ファイルを作ったり、削除したり、名前を変えたり、ファイルや ディレクトリをコピーしたあと、それらの変更点の全体を完全なひと かたまりのものとしてコミットすることができます。

リポジトリでは、それぞれのコミットは、一つの分割できないひと かたまりのトランザクションとして扱います。すべてのコミットに よる変更は、完全に実行されるか、まったく実行されないかの どちらかです。

Subversion は、この不分割の性質を、プログラム 障害、システム障害、ネットワーク障害、その他の操作があった 場合でも保とうとします。

リポジトリがコミットを受け付けるときは常に リビジョンと呼ばれるファイルシステム ツリーの新しい状態を作ります。それぞれのリビジョンには 一意な自然数が割り当てられます。前のバージョンよりも 後のバージョンのほうが数が大きくなります。 リポジトリ新規作成時の最初のバージョンはゼロで、ルートディレクトリ 以外には何も含まれていません。

図 2.7 はリポジトリを視覚化するうまい方法 を示しています。0 から始まるリビジョン番号が、左から右に追加されていく 状況を想像してください。それぞれのリビジョン番号には対応した ファイルシステム木があり、それぞれの木はコミット 後のリポジトリの状態を示す「スナップショット」 です。

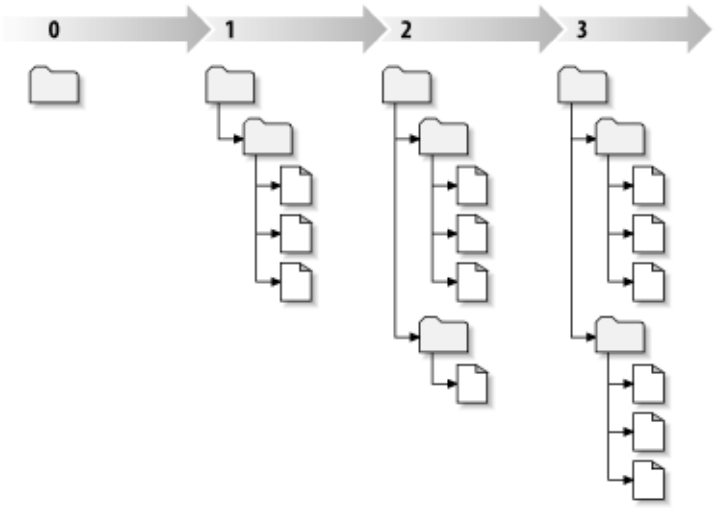

図 2.7 リポジトリ

グローバルリビジョン番号

他の多くのバージョン管理システムとは違って、Subversion のリビジョン番号は 木全体に対して付けられるもので、 個々のファイルに付けられるものではありません。それぞれのリビジョン番号は 木全体を指定し、あるコミットによる変更後のリポジトリの特定の状態を示す ものです。これを想像するための別の方法はリビジョン N は、N 番目のコミット 後のリポジトリファイルシステムの状態をあらわしていると考えることです。Subversion ユーザが、「`foo.c` のリビジョン」 と言うときそれが実際に意味するものは、「リビジョン 5 に現れる `foo.c`」です。 一般的に、あるファイルのリビジョン N と M は、異なっている必要は ありません! CVS はファイルごとのリビジョン番号を使うので、この違いを詳しく知りたい人は 付録 A を見てください。

作業コピーは常にリポジトリのどれか一つのリビジョン対応しているとは 限らないことに注意してください。複数の異なるリビジョンのファイル を含んでいるかも知れません。たとえば、最新リビジョン番号が 4 である リポジトリから作業コピーをチェックアウトしたとします:

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

この時点では、作業コピーはリポジトリのリビジョン 4 と完全に一致しています。 しかし、ここで `button.c` に変更を加え その変更をコミットしたとします。他にコミットした人がいない場合、 今回のコミットはリ

ポジトリのバージョンを 5 にあげ、作業コピーの内容は 以下のようになります:

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

この時点で Sally が `integer.c` に対する修正を コミットし、リビジョンを 6 にあげたとします。ここでもし、**svn update** コマンドであなたの作業コピーを更新すると、次のようになるでしょう:

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

Sally の `integer.c` への変更は あなたの作業コピーに現れますが、`button.c` に対するあなたの変更はそのままです。この例では、 `Makefile` のテキストは、リビジョン 4,5,6 で まったく同一のものですが、Subversion はあなたの作業コピー中の `Makefile` のリビジョンを 6 として、 それが最新であることを表現します。それで自分の作業コピーに きれいなアップデートをかけたときには、一般に作業コピーは リポジトリのある特定のバージョンと完全に一致します。

2.4.3 作業コピーはどのようにリポジトリを追いかけるか

作業コピー中のそれぞれのファイルについて、Subversion は 二つの本質的な情報を`.svn/`管理領域に 記録します:

- あなたの作業ファイルは、どのリビジョンに基づいているか (これはファイルの作業リビジョンと呼ばれます)、 そして
- リポジトリとの対話によって作業コピーが最後に更新された時刻

これらの情報とリポジトリとの対話によって、Subversion は作業ファイルの それぞれが、以下の四つの状態のどれにあるかを見分けることができます:

変更なし、かつ最新　作業コピーのファイルは変更されていないし、その作業リビジョン 以降に起きたリポジトリに対するコミットでもそのファイルに対する変更が ない状態。 そのファイルに対する **svn commit** は何も実行しませんし、 **svn update** も何もしません。

ローカルで変更あり、かつ最新　作業コピー中のファイルは変更されましたが、そのベースリビジョン以降のリポジトリへのコミットで、そのファイルに対する変更が何もなかった 場合。作業コピーにはまだリポジトリにコミットしていない変更がある ので、そのファイルに対する **svn commit** は、 あなたの変更点をそのまま公開することで成功します。**svn update** は何も実行しません。

変更なし、かつ、最新ではない　ファイルは作業コピー中では変更されていませんが、リポジトリには 変更がありました。このファイルは、公開リビジョンによって最新とするために どこかで更新する必要があり

ます。そのファイルに対する **svn commit** コマンドは何もしません。そのファイルに対する **svn update** は あなたの作業コピーに最新の修正点をマージします。

ローカルで変更あり、かつ最新ではない ファイルは作業コピーでも、リポジトリでも変更されています。ファイルに対する **svn commit** は「out-of-date」 エラーになります。そのファイルはまず更新しなくてはなりません。 ファイルに対する **svn update** は公開されている変更点 を作業コピーの変更にマージしようとします。これが自動的にできないような 状況の場合、Subversion はユーザに衝突の解消をさせるためそのままにして おきます。

これにはいろいろな情報の変化を追う必要があるように思いますが、 **svn status** コマンドを使えば、あなたの作業コピーの どのファイルの状態も表示できます。このコマンドについてのより詳しい情報は 項 3.6.3.1 を見てください。

2.4.4 混合リビジョン状態の作業コピー

原則として、Subversion はできる限り柔軟であろうとします。 この特別な例として、作業コピーに、いろいろな異なる作業リビジョン番号を もったファイルとディレクトリを共存させることが できます。前の例での混合リビジョンに戸惑っている人のために、 なぜこのような機能が必要で、どのように利用したらよいかを以下に示します。

2.4.4.1 更新とコミットは別の処理です

Subversion での基本的な原則の一つは、「作業コピーへの取得 (push)」の動作 が「リポジトリへの反映 (pull)」動作を自動的に引き起こすことは ないし、逆もないということです。これは、あなたがリポジトリへの新しい変更点を 送信する用意ができているということが、他の人たちの変更点を受け取る用意が できていることを意味していることはないという、当たり前の理由によります。 そして自分がまだ引き続き新しい修正を加えている場合、 **svn update** は自分自身の作業コピー中にリポジトリの内容をうまく反映してくれるはずで、 このさい、あなたの側の変更点を強制的に他の人々に公開する必要はありません。

この規則はまた副次的に、作業コピーには混合リビジョンの状態を記録する ための特殊な仕組みが必要になり、またその状態に対して寛容でなければならない ことを意味しています。これはディレクトリ自身もバージョン管理できる ためさらに複雑な話になります。

たとえば、作業コピーが完全にリビジョン 10 にあるとします。`foo.html` を編集して **svn commit** を実行した結果、リポジトリにリビジョン 15 ができたとします。このコミットが成功した直後では、多くの不慣れなユーザは 作業コピーは完全にリビジョン 15 にあるだろうと期待するかも知れませんがそうでは ないのです!。リビジョン 10 とりビジョン 15 までの間にリポジトリに対していろいろな変更が 起こったかも知れないのです。クライアント側ではリポジトリに起きたこの変更については 何も知りません。まだ **svn update** を実行していませんし、 **svn commit** は新しい変更点をリポジトリから取得したりはしないからです。 一方、もしかりに **svn commit** が自動的に最新の変更点をダウンロード するとすれば、作業コピー全体を完全にリビジョン 15 に設定することも可能 でしょう — しかしこれでは「push」と「pull」 が独立した処理であるという基本的な原則を侵すことになります。 このため Subversion クライアントができる唯一の安全な方法は、ある特定の ファイル — `foo.html` — がリビジョン 15 にある という印をつけることだけです。作業コピーの残りのファイルはリビジョン 10 の ままなのです。**svn update** を実行することだけが、最新の変更点 をダウンロードする方法であり、これで作業コピー全体にリビジョン 15 の印がつきます。

2.4.4.2 混合リビジョンは正常な状態です

事実として、**svn commit** を実行するときは 常に、あなたの作業コピーはあるいくつかのリビジョンの 混合状態となります。コミット対象となったファイルだけは、それ以外のファイル よりも新しい作業リビジョンになります。何度かのコミットの後で (その間に update を含めなければ)、作業コピーはいくつかのリビジョンの 混合状態になります。あなたがリポジトリを利用している唯一のユーザであったと してもやはりこの現象に出会うでしょう。作業リビジョンの混合状況を見る には **svn status –verbose** コマンドを利用して ください (さらに詳しい情報については 項 3.6.3.1 を見てください)。

不慣れなユーザは自分の作業コピーが混合リビジョンになっていることには まったく気づかないことがよくあります。多くのクライアントコマンドは 処理対象となるアイテムの作業コピー上でのリビジョンが問題になるので 混乱することになります。例えば **svn log** コマンドは ファイルあるいはディレクトリの変更履歴を表示するために利用されます (項 3.7.1 参照)。ユーザがこのコマンド を作業コピー上野オブジェクトに対して実行するとき、そのオブジェクト の完全な履歴を見れるものだと考えるでしょう。しかしそのオブジェクトの 作業コピー上のりビジョンが非常に古いものであった場合 (これは **svn update** が長い期間にわたって実行されなかったような 場合におこります)、そのオブジェクトの より古い バージョン履歴が表示されるでしょう。

2.4.4.3 混合リビジョンは役にたつものです

プロジェクトが非常に複雑になった場合、作業コピー中の一部のファイル を強制的に「古い日付」をもった以前のリビジョンに戻すことが 有用であることに気づくでしょう; どうやるかについては 3 章で説明します。たぶん、あるサブディレクトリにあるサブモジュールの以前のバージョンを テストしたいか、特定のファイルに存在するバグが最初に紛れ込んだ リビジョンを知りたいとかいった場合でしょう。これはバージョン管理システムの 「タイムマシン」としての性質の一つです — つまり、作業コピーの任意の部分を履歴の中のより新しい状態や古い状態に 移動することができるのです。

2.4.4.4 混合リビジョンには制約があります

作業コピー中を混合リビジョン状態に置くことはできますが、 この柔軟性には制約があります。

まず、完全に最新状態ではないファイルやディレクトリの削除を コミットすることができません。より新しいバージョンのアイテムが リポジトリに存在する場合、この試みは拒否されます。まだあなたが 見ていない変更点を間違って消してしまうことを防ぐためです。

次に、完全に最新状態ではないディレクトリに対するメタデータの変更 はコミットできません。アイテムに対する 「属性」の 付与は 6 章で扱います。ディレクトリの作業リビジョンは特定の エントリと属性の組を定義し、最新のディレクトリへの属性の 変更点のコミットは、やはりまだ見ていない変更点を間違って消して しまうかも知れないからです。

2.5 まとめ

この章では、さまざまな Subversion の基本的な概念を扱いました:

- 中心となるリポジトリ、クライアント作業コピー、リポジトリリビジョンツリーの 並び、といった概念を導入しました。
- どのように二人の共同作業者が Subversion を利用してお互いの修正点を公開したり 受け取ったりするかの簡単な例を見てきました。これには、「コピー・修正・マージ」 のモデルを利用するのでした。
- Subversion が作業コピー内の情報をたどったり管理したりする方法について 少し触れました。

ここでは、最も一般的な意味で、Subversion がどのように動作するかについての 良い考え方が身に着いたはずです。この知識をもとに、次の章に進むことが できます。ここは、Subversion のコマンドと機能についての詳しいツアーに なっています。

第 3 章

同伴ツアー

3.1

さて、Subversion を使った詳細を見ていくことにしましょう。この章を 終えるころには、Subversion を使った日常的にしなくてはならない操作のほとんど すべてをやることができるようになっているでしょう。 ソースコードの最初のチェックアウトから始まって、修正し、その修正内容を 調べます。他の人の修正をどうやって自分の作業コピーにマージし、それが どのようなものかを調べ、起きるかも知れない衝突をどのように扱えば 良いかもわかるでしょう。

この章は、Subversion コマンドの全体を列挙するのではないのに注意してください。 — そうではなく、普段一番よく利用する Subversion の操作についての対話的な手引き にしてあります。この章は、 第 2 章 を読み、理解していることと、Subversion の一般的モデルをよく知っていることを前提 としています。コマンドの完全なリファレンスは、 第 9 章 を見てください。

3.2 おたすけを!

読み進める前に、Subversion を使うときに必要な一番重要なコマンドを載せて おきます: **svn help** Subversion コマンドラインクライアントは、自分自身の中にドキュメントを 持っています。 — いつでも **svn help** <サブコマンド>とやれば 構文、オプションスイッチ、その サブコマンドの振る舞いを見ることができます。

3.3 インポート

svn import で、Subversion のリポジトリに新しいプロジェクトを インポートできます。Subversion サーバを設定するときには、一番最初に 実行するコマンドかも知れませんが、それほどしばしば利用するものでは ありません。import の詳細についてはこの章の後のほうにある、 項 3.8.2 を見て ください。

3.4 リビジョン: 番号、キーワード、そして、時刻、おやおや・・・

話を進める前に、リポジトリ中の特定のリビジョンを特定する方法について少し 知っておく必要があります。 項 2.4.2 で 見たように、リビジョンは、リポジトリのある特定の時点における「スナップショット」です。コミットを繰り返してリポジトリが大きくなるにつれ、スナップショットを特定する 仕組みが必要になってきます。

リビジョンの特定には、 `--revision` (`-r`) スイッチの あとにほしいリビジョン番号を続けます。 (**svn –revision REV**) とするか、 二つのリビジョンをコロンで区切って範囲指定することもできます (**svn –revision REV1:REV2**). また、リビジョンは、番号、キーワード、日付で参照することもできます。

3.4.1 リビジョン番号

新しい Subversion のリポジトリを作ると、最初はリビジョンゼロとなり、その後の コミットのたびにリビジョン番号は一つずつ大きくなります。 コミットが完了した後では、Subversion クライアントはあなたに一番新しい リビジョン番号を知らせます:

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

その後、いつでもこのリビジョンを参照したければ、 (この章の後のほうで、どうしてそんなことが必要かを説明します。)「3」として参照することができます。

3.4.2 リビジョンキーワード

Subversion クライアントはさまざまな リビジョンキーワードを理解できます。 このようなキーワードは `--revision` スイッチの整数引数のかわりに 使うことができ、Subversion によって、特定のリビジョン番号に変換されます。:

> 注意
>
> 
>
> 作業コピーのすべてのディレクトリには `.svn` と呼ばれる管理用のサブディレクトリがあります。Subversion はディレクトリ 中のそれぞれのファイルごとの コピーをこの管理領域中に保存しています。このコピーは修正されることは なく (キーワード展開は起こりませんし、行末変換もしませんし、その他 いっさい何もしません)、最後のリビジョン (これを「BASE」リビジョン といいます) に存在しているファイルで、作業コピーを更新したときのファイルその もののコピーです。このファイルのことを、 プリスティン・コピーあるいはファイルの テキスト-ベースバージョンと言い、リポジトリに存在している元のファイルの厳密なコピーです。

HEAD リポジトリにある最新のリビジョンです。

BASE 作業コピーにあるファイル、ディレクトリの、「修正元」リビジョン です。

COMMITTED ファイル、ディレクトリが変更された BASE 以前の (または BASE リビジョンを含む) 最後のリビジョンです。

PREV 変更があった最後のリビジョンの 直前 のリビジョンです。 (`COMITTED` - 1 番ということになります)。

注意

PREV, BASE, そして COMMITTED はローカルパス名として参照する のに利用できますが、URL では利用できません。

コマンド実行時のリビジョンキーワードの例です。 コマンドの意味はわからなくても大丈夫です。章を進めるごとに 説明します:

```
$ svn diff --revision PREV:COMITTED foo.c
# foo.c にコミットした最後の変更を表示

$ svn log --revision HEAD
# 最後のリポジトリへのコミットで付けたログメッセージを表示

$ svn diff --revision HEAD
# 作業コピー内ファイルを、リポジトリの最新バージョンと比較

$ svn diff --revision BASE:HEAD foo.c
# 作業コピーの修正元 foo.c を
# リポジトリの最新バージョンと比較

$ svn log --revision BASE:HEAD
# 最後に更新した後のすべてのコミットログを表示

$ svn update --revision PREV foo.c
# foo.c の最後の変更をもとに戻す
# (foo.c の作業リビジョン番号は減少する)
```

これらのキーワードを使って特定のリビジョン番号や作業コピーの正確な リビジョンを憶えておくことなしに、いろいろな (役に立つ) 処理を することができます。

3.4.3 リビジョン日付

リビジョン番号やリビジョンキーワードを指定できるところではどこでも、 中かっこ、「{}」 の中に日付を入れて指定することもできます。 日付とリビジョン番号の両方を使ってリポジトリ中の変更範囲にアクセスする ことさえできます。

ここでは Subversion が受け入れることのできる日付形式の例をあげておきます。 空白を含むような日付は常に引用符でくくるのを忘れないでください。

```
$ svn checkout --revision {2002-02-17}
```

```
$ svn checkout --revision {15:30}
$ svn checkout --revision {15:30:00.200000}
$ svn checkout --revision {"2002-02-17 15:30"}
$ svn checkout --revision {"2002-02-17 15:30 +0230"}
$ svn checkout --revision {2002-02-17T15:30}
$ svn checkout --revision {2002-02-17T15:30Z}
$ svn checkout --revision {2002-02-17T15:30-04:00}
$ svn checkout --revision {20020217T1530}
$ svn checkout --revision {20020217T1530Z}
$ svn checkout --revision {20020217T1530-0500}
…
```

リビジョンとして日付を指定するときは Subversion はその日付に 一番近いリビジョンを見つけようとします:

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

Subversion の日付は早くないですか?

リビジョンとして一つの日付を選択し、時刻までは指定しなかった 場合 (たとえば `2002-11-27`), Subversion は 11 月 27 日に追加された最後のリビジョンをあつかうべきだと 思うかも知れません。しかし、そうではなく、26 日か、それより前の リビジョンを得ることになるでしょう。Subversion は 指定された日付以前でリポジトリ中の最新のリビジョンを見つけようとすることに注意してください。 `2002-11-27` のような タイムスタンプなしの日付を指定すると、 Subversion は、時刻として 00:00:00 が指定されたと仮定し、 このため 27 日のコミットについては何も返さないことに なってしまいます。

検索に 27 日を含めたければ 27 日の時刻を指定するか、(`{"2002-11-27 23:59"}`) 単に次の日を指定する (`2002-11-28`) 必要があります。

日付範囲を使うこともできます。 Subversion は両方の日付の間にあるすべてのリビジョンを検索対象と します。両端の日付は検索に含みます:

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

既に指摘したように、日付とリビジョン番号を混在させることもできます:

```
$ svn log --revision {2002-11-20}:4040
```

Subversion で日付を扱う場合、面倒なことが起こるかも知れないことに注意 してください。リビジョンのタ

イムスタンプはリビジョン属性として保存 されます — つまりバージョン化されていない、修正可能な属性として、ということです — リビジョンのタイムスタンプは本当の時間 を完全に偽造する形で変更可能ですし、削除してしまうことすら可能です。 このようなことは Subversion が処理する内部的な日付-リビジョン変換 に大混乱を与えるかも知れません。

3.5 最初のチェックアウト

たいていの場合、Subversion はリポジトリからプロジェクトを チェックアウト することで始まります。 リポジトリをチェックアウトすると あなたのマシンにはリポジトリの作業コピーができます。 このコピーはコマンドラインで指定した Subversion リポジトリの HEAD (最新のリビジョン) になります:

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

利用可能な名前は?

Subversion はバージョン管理するデータ型に できる限り制約を置かないように動作します。ファイルの内容と 属性値はバイナリデータとして保存されたり転送され、 項 7.3.3.2 は どのファイルが 「テキスト的な」処理を受け付けないかについての情報を Subversion に与えます。 しかし Subversion が保存情報に対して制約を置かなくてはならない 場合もいくつかあります。

Subversion は内部的には UTF-8 エンコードされた Unicode としてテキストを処理するので、属性値、パス名、ログメッセージ、などの ような本来「テキスト的な」 性質を持ったアイテムは、 正しい UTF-8 文字だけを含むことができます。また `svn:mime-type` 属性の利用についても最低限度の決まりがあります — ファイルの内容が UTF-8 互換でなければ、バイナリファイルの設定をすべきです。そうしないと Subversion は UTF-8 を使った差分のマージを実行しようとするので、おそらく ファイルにはゴミが残ってしまうでしょう。

さらにパス名は Subversion のいくつかの管理ファイル中だけではなく WebDAV データ交換での XML 属性値としても使われます。これはパス名は 正しい XML (1.0) 文字だけを含まなければならないことを意味します。 Subversion はまた、ファイル名に TAB, CR, LF 文字を使うことも禁止していますが、 これによって diff コマンドや、

名前

svn log や

名前

svn status などのコマンド出力中で問題が起こらなく なります。

こう聞くと何かたくさんのことを覚えておく必要があるかの ようですが、実際にはこのような制約が問題となることはほとんどありません。 ロケールの設定が UTF-8 と互換になっていて、パス名で制御文字を使わなければ Subversion を利用する上で何の問題も起こらないはずです。 コマンドライン・クライアントの場合にはもう少し便利です — 内部的な 利用での「構文的に正しい」バージョンを作るために入力した パス名を、URL 中で必要となる形式に自動的にエスケープしてくれます。

Subversion の利用に経験を積んだユーザは リポジトリ中のディレクトリ配置について、もっとも便利な決まりを見つけて います。そのような規約は、上で説明したような構文的に厳密な要求ではありませんが 通常のよくおこなう処理を楽にしてくれるものです。 この本全体を通じてあらわれる URL の`/trunk` の部分は そのような規約のひとつです: これについては関連情報とともに 第 4 章でより詳しく説明します。

上の例は trunk ディレクトリのチェックアウトでしたが、チェックアウトの URL 中にサブディレクトリを指定することでどのような深い階層にある サブディレクトリも簡単にチェックアウトできます:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk/doc/book/tools
A  tools/readme-dblite.html
A  tools/fo-stylesheet.xsl
A  tools/svnbook.el
A  tools/dtd
A  tools/dtd/dblite.dtd
…
Checked out revision 2499.
```

Subversion は「ロック・修正・ロック解除」 のかわりに「コピー・修正・マージ」 モデルを使うので (第 2 章)、すでに作業コピーのファイルやディレクトリ に対して変更する準備ができています。作業コピーはあなたのシステムに ある他のファイルやディレクトリのようなものです。編集したり変更を 加えたり移動することもできますし、作業コピー全体を削除してから、その ことを忘れてしまうこともできます。

> 注意
>
> 
>
> 作業コピーは「システム中のほかのファイルやディレクトリ の集まり」となんら変わることはありませんが、作業コピー中のファイル やディレクトリを編成しなおした場合には常に Subversion にそのことを知らせ なくてはなりません。もし作業コピー中のファイル、ディレクトリをコピーまたは 移動したい場合には、オペレーティングシステムで用意されているコピーや 移動コマンドを使うかわりに **svn copy** や **svn move** を使ってください。この章の後のほうでこれについてもっと詳しく 説明します。

新しいファイルやディレクトリを作ったり、既に存在するものを 変更したりした結果をコミットする用意ができるまで、何をやろうと Subversion サーバに追加報告する必要はまったく ありません。`.svn` ディレクトリって何?

作業コピー中のどのディレクトリにも `.svn` という名前の管理領域があります。 普通のディレクトリ一覧表示コマンドはこのディレクトリを表示しませんが、 にもかかわらずこれは非常に重要なディレクトリです。どんなことをするときでも、管理領域を消したり変更したりしないでください! Subversion は作業コピーを管

理するのにこのディレクトリを使います。

リポジトリの URL を唯一の引数として作業コピーをチェックアウトする こともできますが、リポジトリ URL の後に、ディレクトリを指定することも できます。この場合、指定した新規のディレクトリ中に作業コピーを作ろうとします。 たとえば:

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

これは、既にやったような `trunk` という名前のディレクトリのかわりに `subv` という名前のディレクトリに、作業コピーを作ります。

3.6 基本的な作業サイクル

Subversion はたくさんの機能、オプション、おまけが付いていますが、 日々の作業では、おそらくその中のいくつかを使うだけでしょう。 この章では一番よく起こることを説明します。

典型的な作業サイクルは次のようなものです:

- 作業コピーの更新
  - **svn update**
- 変更
  - **svn add**
  - **svn delete**
  - **svn copy**
  - **svn move**
- 自分の変更点の確認
  - **svn status**
  - **svn diff**
  - **svn revert**
- 他の人の変更の、作業コピーへのマージ
  - **svn update**
  - **svn resolved**
- 自分の変更のコミット
  - **svn commit**

3.6.1 作業コピーの更新

チームを作って作業してるプロジェクトでは、自分の作業コピーを 更新してプロジェクトの他のメンバーが自分の 更新処理後に加えた変更点をすべて受け取りたくなるでしょう。 **svn update** を使って自分の作業コピーを リポジトリの最新バージョンにあわせてください。

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

この場合、あなたが最後に更新してから、誰か別の人が `foo.c` と `bar.c` の両方に加えた変更をコミットし、Subversion はこの変更をあなたの 作業コピーに加えるために更新しました。

svn update の出力をもう少し詳しく見てみましょう。 サーバが変更点を作業コピーに送るとき、文字コードがそれぞれのファイル の横に表示されて、あなたの作業コピーを最新にするために、どのような 動作を起こしたかを知らせます:

`U foo` ファイル `foo` は 更新 (Updated) されました (サーバから 変更を受け取りました)。

`A foo` ファイルかディレクトリである `foo` は あなたの作業コピーに追加 (Added) されました。

`D foo` ファイルかディレクトリである `foo` は あなたの作業コピーから削除 (Deleted) されました。

`R foo` ファイルかディレクトリである `foo` は あなたの作業コピー中で置き換え (Replaced) られました。つまり `foo` は削除されて、同じ名前の 新しいファイルまたはディレクトリが追加されました。両方は同じ名前ですが、 リポジトリはそれらを別の履歴を持った別のものであるとみなします。

`G foo` ファイル `foo` は新しい変更点を リポジトリから受け取りましたが、そのファイルのローカルコピーにも 修正が加えられていました。しかし両方の修正は重なっていないか、あるいは 変更の内容が自分自身のものとまったく同じであったため、Subversion はリポジトリの変更を、問題を起こすことなしにマージ (merGed) しました。

`C foo` ファイル `foo` は、サーバから 衝突 (Conflicting) のある変更を 受け取りました。サーバからの変更は、あなた自身の変更と直接重なっています。 でも心配はいりません。この衝突は人間 (つまりあなた) が解消しなくては なりません。この章の後でこの状況について議論します。

3.6.2 作業コピーに変更を加えること

さて、これで自分の作業コピーに変更を加えることができます。 以下のような、比較的特殊な変更をすることもできます。新しい機能 を書いたり、バグをフィックスしたり、などです。このような場合に 使う Subversion コマンドは、 **svn add**, **svn delete**, **svn copy**, **svn move** などです。しかし、既に Subversion 管理下

にあるファイルを単に編集するだけなら、コミットするまでに そのようなコマンドを使う必要はありません:

ファイルの変更　これは一番単純なタイプの変更です。ファイルを変更することについて Subversion に報告する必要はありません。どのファイルが変更された については Subversion 自身が自動的に検出することができます。

ツリーの変更　Subversion に対して、削除、追加、コピー、移動の予告として ファイルやディレクトリを「マーク」するように 依頼することができます。このような変更は作業コピー上では直ちに 起こりますが、次にあなたがコミットするまでリポジトリ上では 追加削除は一切起きません。

ファイルを変更するには、テキストエディタ、ワードプロセッサ、 グラフィックプログラム、その他の通常利用しているツールなら なんでも使うことができます。Subversion はバイナリファイルを テキストファイルを扱うのと同じくらい簡単に扱うことができます — し、十分効率的にあつかえます。

ここでは、Subversion でツリーの変更として一番よく利用される 四つのサブコマンドを概観しておきます (あとで、**svn import** と **svn mkdir** も見ていきます)。

> 警告
>
> どんなツールを使ってファイルを編集する場合でも、その内容 を Subversion に伝えずに作業コピーの構成を変えるべきではあり ません。作業コピーの構成を変えるときには **svn copy**, **svn delete**, **svn move** コマンドを使い、新たにファイルやディレクトリ をバージョン管理下におく場合には **svn add** コ マンドを使うようにしてください。

svn add foo　通常ファイル、ディレクトリ、シンボリックリンクのどれかである `foo` をリポジトリに追加する予告をします。 次のコミットで `foo` は正式に親ディレクトリの 子供になります。`foo` がディレクトリの場合は `foo` にあるすべてのファイルは追加予告の対象に なります。`foo` だけを追加予告したい場合は `--non-recursive` (`-N`) スイッチを指定してください。

svn delete foo　通常ファイル、ディレクトリ、シンボリックリンクのどれかである `foo` をリポジトリから削除する予告をします。 `foo` が通常ファイルまたはシンボリックリンクの場合は作業コピーから直ちに 削除されます。ディレクトリの場合は削除されませんが、Subversion はそれを削除予告の状態に設定します。変更をコミットすると `foo` は作業コピーとリポジトリから削除されます。 [*1]

svn copy foo bar　新しいアイテム `bar` を `foo` の複製として作ります。`bar` は自動的に 追加予告されます。`bar` が次のコミットでリポジトリ に追加される時点で、コピーの履歴が記録されます (それが `foo` のコピーである、という履歴)。 **svn copy** は中間ディレクトリを作成しません。

svn move foo bar　このコマンドは **svn copy foo bar; svn delete foo** を実行することとまったく同じです。

[*1] もちろんリポジトリから完全に削除されてしまうわけではありません — 単に、リポジトリの HEAD から削除されるだけです。削除したリビジョンより 前のリビジョンを指定してチェックアウトすれば (あるいは作業コピーを 更新すれば) 削除前の状態に戻ることができます。

つまり、 bar は foo のコピーとして 追加予告され、foo は削除予告されます。 **svn move** は中間ディレクトリを作成しません。

作業コピーなしでリポジトリを変更すること

以前この章で、変更をリポジトリに反映させるためにはどんな変更も コミットする必要があるといいました。これは完全に正しいとはいえません — リポジトリに対して、ツリーの変更を直接コミットするようないくつかの コマンドもあります。 これは、サブコマンドが、作業コピーパスではなく、直接 URL を操作する場合にだけ起こります。特に **svn mkdir**, **svn copy**, **svn move**, **svn delete** の特殊な利用は、URL を直接操作します。

URL の操作がそのような方法で振る舞うのは、作業コピーに対する操作 コマンドは、作業コピーを、リポジトリにコミットする前に変更点を セットしておくある種の「中間領域」 として使うため です。 URL に働くコマンドはこの余裕がないので、直接 URL に操作するときには 上で述べたアクションはどれも直接のコミットを引き起こすことになります。

3.6.3 自分の変更点の調査

変更が完了したら、リポジトリにコミットする必要がありますが、普通 そうする前に、正確には自分が何を変更したのかを見ておくのは良い考え です。コミットの前に変更点を確認することで、より正確なログメッセージ を付けることができます。また、不十分な修正をしただけであることを 発見するかも知れませんし、コミットする前にその変更を破棄したりする 機会にもなります。さらに、公開する前に変更点を再検討したり詳しく調査 する機会にもなります。 **svn status**, **svn diff**, **svn revert** を使って正確にはどんな変更をしたかを 見ることができます。最初の二つのコマンドで、作業コピー中のどのファイル を変更したかを調べ、三番目のコマンドでそのうちのいくつか (あるいは全部) の変更を取り消すかも知れません。

Subversion はこの作業をやるために効率よく作られていて、多くの操作に ついてはリポジトリと通信することなしに実行できます。特に、 作業コピーには、.svn という隠れたディレクトリが あり、ここに作業コピーの「元なるリビジョン」 のコピーが あります。これをうまく使って Subversion は、あなたの作業ファイルのどれが 変更されたかをすばやく知ることができますし、リポジトリと通信すること なしに、変更を取り消すことすらできます。

3.6.3.1 svn status

多分、どの Subversion コマンドよりも **svn status** コマンドはよく利用されるはずです。CVS ユーザに告ぐ: 更新するのは、ちょっと待った!

作業コピーにどのような変更を行ったかを確認するために、多分、 **cvs update** を使っていることでしょう。**svn status** は、作業コピーに対して行われた 変更について、すべての必要な情報を提供してくれます — しかも、リポジトリに アクセスしませんし、他の人の行った変更が取り込まれる可能性もありません。

Subversion では、**update** は以下の処理を行うだけです — 最後の更新後にリポジトリにコミットされたすべての変更を、 作業コピーに反映すること、です。ローカルコピーに対して行った変更を 確認するために **update** を使う癖を直さなくてはいけません。

自分の作業コピー最上位階層で引数なしに **svn status** を実行すると、自分がツリーにしたすべての修正が検出できます。 以下の例は **svn status** が返すことのできる 異なる状態コードです。 (以下で、# の後に書いてあるテキストは **svn status** からのものではないのに注意してください。)

```
  L     some_dir            # svn left a lock in the .svn area of some_dir
M       bar.c               # the content in bar.c has local modifications
```

```
 M       baz.c               # baz.c has property but no content modifications
X        3rd_party           # dir is part of an externals definition
?        foo.o               # svn doesn't manage foo.o
!        some_dir            # svn manages this, but it's missing or incomplete
~        qux                 # versioned as file/dir/link, but type has changed
I        .screenrc           # svn doesn't manage this, and is set to ignore it
A  +     moved_dir           # added with history of where it came from
M  +     moved_dir/README    # added with history and has local modifications
D        stuff/fish.c        # file is scheduled for deletion
A        stuff/loot/bloo.h   # file is scheduled for addition
C        stuff/loot/lump.c   # file has textual conflicts from an update
 C       stuff/loot/glub.c   # file has property conflicts from an update
R        xyz.c               # file is scheduled for replacement
    S    stuff/squawk        # file or dir has been switched to a branch
     K   dog.jpg             # file is locked locally; lock-token present
     O   cat.jpg             # file is locked in the repository by other user
     B   bird.jpg            # file is locked locally, but lock has been broken
     T   fish.jpg            # file is locked locally, but lock has been stolen
```

この出力形式の中で、**svn status** は 五つの文字を表示していて、その後にいくつかの空白が続き、 ファイルまたはディレクトリ名称がそのあとに続いています。 最初のコラム (左から一文字目の部分) は、ファイルまたはディレクトリの 状態をあらわしています。ここで表示されているコードは:

`A item` 通常ファイル、ディレクトリ、シンボリックリンクのいずれかである `item` はリポジトリに追加予告されています。

`C item` ファイル `item` は衝突の状態にあります。つまり、自分の作業コピーにあるローカルな変更が 更新時にサーバから受け取った変更部分と重なっています。リポジトリに 自分の変更点をコミットする前にこの衝突を解決しなくてはなりません。

`D item` 通常ファイル、ディレクトリ、シンボリックリンクのいずれかである `item` は リポジトリからの削除予告をされています。

`M item` ファイル `item` の内容は 修正されています。

`R item` ファイル、ディレクトリ、シンボリックリンクのいずれか である `item` はリポジトリ中の `item` を置き換えるように準備されています。 これはまずそのオブジェクトがいったん削除され、次に同じ名前の 別のオブジェクトが追加されます。そしてそれは単一のリビジョンで ひとまとまりに実行されます。

`X item` ディレクトリ `item` はバージョン化 されていませんが Subversion の外部定義に関連付けられています。外部定義についての 詳細は項 7.6 を見てください。

`?  item` 通常ファイル、ディレクトリ、シンボリックリンクのいずれかである `item` は バージョン管理下にはありません。`--quiet` (`-q`) スイッチを **svn status** に渡すか、親ディレクトリに `svn:ignore` 属性を 設定することで疑問符の表示を抑制できます。 無視できるファイルについての詳細は 項 7.3.3.3 を見てください。

`!  item` 通常ファイル、ディレクトリ、シンボリックリンクのいずれかである `item` は バージョン管理下にありますが、それは失われているか、何か不完全な 状態にあります。Subversion 以外のコマンドを使って削除された場合 には、そのアイテムは失われてしまいます。ディレクトリの場合、 チェックアウトか、更新が中断された場合、不完全な状態になることが あります。**svn update** を使えばすぐにリポジトリから ファイルまたはディレクトリをもう一度取り出すことができます。 **svn revert file** を使えば、失われたファイルを 復元することができます。

`item` 通常ファイル、ディレクトリ、シンボリックリンクのいずれかである `item` は あるタイプのオブジェクトとして存在しますが、作業コピーには別の タイプのオブジェクトとして存在しています。 たとえば Subversion はリポジトリ中にファイルを持っているが、 **svn delete** や **svn add** を 使わずに、作業コピー中の対応するファイルを削除し、同じ名前の ディレクトリを作ったような場合です。

`I item` ファイル、ディレクトリ、シンボリックリンクのいずれかである `item` はバージョン管理下にはなく、Subversion は **svn add**, **svn import svn status** の実行時にはこれを 無視します。無視されるファイルについてのより詳しい情報は 項 7.3.3.3 を見てください。 このシンボルは **svn status** に`--no-ignore` オプションを渡したときにだけ表示されることに注意してください。 — そうでなければファイルは無視され、まったく表示されません!

二番目のコラムはファイルまたはディレクトリの属性を示しています (詳しくは項 7.3 参照してください)。 もし `M` が表示されていれば 属性は修正されたことを示しています。そうでなければ空白が 表示されます。

三番目のコラムは空白か、`L` が 表示され、後者の場合は Subversion がそのディレクトリの `.svn` 作業領域をロックしていることを意味して います。**svn commit** が実行されている途中で **svn status** を実行すると `L` が表示されます — 多分ログメッセージを変更している最中かも知れません。 Subversion が実行されていないのなら、Subversion は多分中断され たため、ロックは、**svn cleanup** の実行によって 解除しなくてはなりません。(これについてはこの章の後で触れます)

四番目のコラムは空白か `+` が 表示され、あとの場合はファイルまたはディレクトリは 追加または修正され、それが履歴に追加予告されていることを意味します。 これはファイルやディレクトリに対して **svn move** か **svn copy** をしたときによく起こります。 `A  +` の表示がある場合 そのアイテムは履歴付きの追加予告されていることを意味します。 それはファイルか、コピーされたディレクトリのルートであるかです。 `+` はそのアイテムが、履歴に追加 予告されたサブツリーの一部であることを意味します。つまり、 そのアイテムのどれかの親がコピーされ、コミットを待っています。 `M  +` はアイテムが履歴 に追加予告されたサブツリーの一部であり、 かつ ローカルの修正も受けているという場合です。 コミットするとき、最初に親が履歴付き追加されます (コピーされます) その意味はこのファイルはコピーによって自動的に存在するということ です。次いでローカルの修正はコピーにアップロードされます。

五番目のコラムは空白か、`S` に なります。これはファイルかディレクトリは作業コピーの残り パスから、ブランチに (**svn switch** コマンドで) 切り替わっていることを意味します。

六番目のコラムはロックに関する情報を示しています。 詳細は 項 7.4 で説明します。

svn status にパスを指定すると、そのアイテムに 関する情報のみを表示します:

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

svn status も `--verbose` (`-v`) スイッチを とりますが、その場合作業コピー中のすべてのアイテム に対して、たとえ変更がなくてもステータスを表示するという意味になります:

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

これは **svn status** の「長い表示形式」 の出力です。再処理コラムは同じですが、二番目はアイテムの作業リビジョン になります。三番目と四番目はそれぞれアイテムが最後に変更されたリビジョン と、誰がそれをしたかの表示です。

いままで出てきた **svn status** の実行は いずれもリポジトリと通信をしません。それは単に作業コピー中の `.svn` ディレクトリのメタデータを比較する ことによって、ローカルマシン上だけで動作します。 最後に、`--show-updates`(`-u`) スイッチがありますが、これはリポジトリと通信して、古くなった ファイルなどの情報を追加表示します:

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

二つのアスタリスク (’*’) に注意してください:この状態で **svn update** を実行すると `README` と `trout.c` の変更点を受け取ることになります。 これは非常に役に立つ情報です — コミットする前には更新して `README` に関するサーバ上の変更点を取得 しなくてはなりません。さもなければ、最新でないという理由でコミットは失敗するでしょう (詳しくは後で述べます)。

3.6.3.2 svn diff

自分の変更点を調べる別の方法は、**svn diff** コマンドを使うことです。**svn diff** を引数 なしに実行することで、自分がどんな変更をしたかを 正確に 知ることができます。このときの 出力形式は unified diff 形式です: [*2]

```
$ 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
===================================================================
```

[*2] Subversion は内部 diff エンジンを利用し、デフォルトでは unified diff 形式を生成します。もし別の形式の diff 出力がほしい場合には、 `--diff-cmd` で外部 diff プログラムを指定し、 `--extensions` スイッチを使ってフラグを渡して ください。たとえばファイル `foo.c` のローカルな 変更点を context 出力形式で見たいが、空白の変更は無視したい場合、 **svn diff –diff-cmd /usr/bin/diff –extensions '-bc' foo.c** のように実行することができます。

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

svn diff コマンドは.svn 領域 にある、「修正元リビジョン」 のコピーに対して作業コピー 中のファイルを比較した結果を出力します。 追加予告ファイルはすべて追加されたテキストとして表示され 削除予告されているファイルはすべて削除されたファイルとして表示 されます。

出力は、*unified diff* 形式で表示されます。 つまり、削除された行は先頭に - が付き、追加された 行は先頭に + がつきます。**svn diff** はさらに **patch** に便利なようにファイル名称とオフセット 情報を表示します。このため diff の出力をファイルにリダイレクトすることで 「パッチ」 を生成することができます:

```
$ svn diff > patchfile
```

たとえば、パッチファイルを別の開発者に送り、コミット前に再検討や テストをすることができます。

3.6.3.3 svn revert

上の diff 出力を見て、README に対する修正が間違っていることがわかったとしましょう: 多分 エディタで間違ったファイルに保存してしまったりしたのでしょう。

これは、**svn revert** を使うことの できるとても良い機会です。

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

Subversion はそのファイルを.svn 領域に ある「修正元リビジョン」のコピーを上書きすることに よって、修正以前の状態に戻します。 しかし、**svn revert** はどのような 予告操作も取り消すことができるのに注意してください — たとえば 最終的に新しいファイルを追加することをやめることができます:

```
$ svn status foo
?      foo

$ svn add foo
A         foo

$ svn revert foo
Reverted 'foo'

$ svn status foo
?      foo
```

注意

svn revert *ITEM* は、作業コピーから *ITEM* を削除し、それから **svn update -r BASE** *ITEM* を実行したのとまったく同じ効果があります。 しかし、もしファイルをもとに戻そうとしているのなら、 **svn revert** には一つ重要な違いがあります — それはファイル を元に戻すにあたってリポジトリと通信する必要がないのです。

あるいは間違ってバージョン管理からファイルを消して しまったのかも知れません:

```
$ svn status README
       README

$ svn delete README
D          README

$ svn revert README
Reverted 'README'

$ svn status README
       README
```

ママ見て、ネットワークが使えないの!

いままで見てきた三つのコマンド (**svn status**, **svn diff**, **svn revert**) は、ネットワークに対するアクセスなしに実行できます。これで 飛行機旅行中であるとか、通勤電車に乗っているときとか、 ビーチでハックするときのようにネットワークに接続されていない場所 でも修正作業を簡単に続けることができます。

Subversion はこれをやるのに、`.svn` 管理 領域に修正元のリビジョンファイルのプライベートキャッシュを保存します。これで Subversion はネットワークにアクセス せずにファイルに関する報告や修正の取り消しをやる ことができます。このキャッシュ (「text-base」と呼ばれます) は また、ローカル修正をサーバにコミットする際に、修正元バージョン との圧縮された差分 (あるいは「違い」) だけを送れるようにします。このキャッシュを 持っていることは非常に大きな利益になります — 早いネットワーク 接続環境にしたとしても、ファイル全体を転送するよりも修正点だけを 送るほうがずっと早いでしょう。ちっょと考えるとそんな重要なことには思えないかも知れませんが、400MB のファイルに対する一行の変更を コミットしようとして、ファイル全体をサーバに転送しなくては ならないことを考えてみてください。

3.6.4 衝突の解消 (他の人の変更点のマージ)

いままでで、**svn status -u** がどうやって衝突を 予告できたかを知っています。**svn update** を実行して、面白いことが起こったとします:

```
$ svn update
U  INSTALL
```

```
G   README
C   bar.c
Updated to revision 46.
```

U と G のコードは考える ことはありません。この二つはリポジトリからの変更を きれいに吸収することができました。 U でマークされたファイルは ローカルでは何の変更もありませんでしたが、リポジトリからの修正分で更新 (Updated) されました。 G はマージ (merGed) されたことを意味して いますが、これは、ファイルはローカルで変更されていたが、 リポジトリからの変更部分とまったく重ならなかったことを意味 しています。

しかし C は衝突を あらわしています。これはサーバからの変更場所があなた自身の ものと重なっていることを意味していて、あなたは手で どちらかを選択しなくてはなりません。

衝突が起こると、普通はその衝突を知らせて解決することができるように 三つのことが起こります:

- そのファイルがマージ可能なタイプのときには Subversion は更新処理中に C を 表示して、そのファイルが「衝突している」ことを知らせます。 (行番号に基づいた文脈マージ可能なファイルかどうかは svn:mime-type 属性によって決まります。 詳しくは項 7.3.3.2 を見てください。)
- Subversion は衝突マーカ — 衝突を 起こした「両方」の内容を区切る特別なテキスト文字列 のことです — を重なっている場所に置き、衝突内容を見てわかるように します。
- 衝突しているファイルのそれぞれについて、Subversion は最大で三つの バージョン管理対象にはならない特殊なファイルを作業コピーに置きます:

  filename.mine これは作業コピーを更新する前に作業コピー中にあったファイル です — つまり、衝突マーカを含んでいません。このファイルは 自分のやった最後の変更が含まれているだけのものです。(Subversion がこのファイルがマージ可能なものではないとみなした場合には .mine ファイルは作成されませんが、それは 作業ファイルと同一の内容になってしまうだろうからです。)

  filename.rOLDREV これは、作業コピーを更新する前の BASE リビジョンにあったファイル の内容です。つまり、そのファイルは最後にした編集の直前にした チェックアウト時点でのファイルです。

  filename.rNEWREV これは Subversion クライアントプログラムが作業コピーを更新したときにサーバから受け取ったファイルです。これは、リポジトリの HEAD リビジョンに対応しています。

  ここで OLDREV は .svn ディレクトリにあるファイルのリビジョン番号で、NEWREV は HEAD リポジトリのリビジョン番号です。

たとえば Sally がリポジトリにある sandwich.txt に変更を加えるとします。たった今、Harry は自分の作業コピーのそのファイルを変更 してコミットしました。Sally は自分が加えた変更をコミットする前に 作業コピーを更新しますが、そのとき衝突の報告を受けます:

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

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

このとき Subversion は三つの一時ファイルが削除されるまで `sandwich.txt` のコミットを許可 しません。

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

もし衝突があった場合は、三つのうちのどれかを する必要があります:

- 「手で」 衝突テキストをマージします。(ファイル中の衝突マーカを調べ編集することによって)。
- 作業ファイルに、一時ファイルのどれかを上書きします。
- **svn revert** <**filename**>を 実行して、ローカルでしたすべての変更を捨てます。

ひとたび衝突を解消したら、**svn resolved** を実行 して Subversion にそのことを伝えます。これは三つの一時ファイルを 削除して、Subversion はもうそのファイルが衝突の状態にあるとは 考えなくなります。 [*3]

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

3.6.4.1 衝突を手でマージすること

手で衝突をマージするのは最初とても嫌なものですが、 少し練習すればバイクから降りるのと同じくらい簡単に できるようになります。

例をあげます。コミュニケーション不足により、あなたとあなたの同僚 である Sally の両方が `sandwich.txt` というファイルを同時に 編集したとします。Sally は自分の変更をコミットし、それからあなた が作業コピーを更新しようとすると、衝突を受け取ります。それで `sandwich.txt` を編集しなくてはなりません。最初にファイルを見てみます:

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
```

[*3] 一時的なファイルは常に自分で削除することができます が、Subversion がせっかくコマンドを用意しているのに本当にそうしたい のでしょうか? そうは思えませんが。

```
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

小なり記号の文字列、イコールサイン、そして 大なり記号の文字列を衝突マーカと呼びますが、これは実際の 衝突を起こしたデータの一部ではありません。一般的には次のコミットの 前に取り除く必要があります。最初の二つのマーカの間のテキストは衝突領域に あなた自身がした変更です:

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

二番目と三番目の衝突マーカの間のテキストは、 Sally のコミットからのテキストです:

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

通常、衝突マーカと Sally の変更部分を単に削除するわけには いきません — そのようなことをすると Sally は sandwich を 受け取ったときにびっくりしますし、それは彼女が望んでいる ものではないでしょう。あなたは電話をかけるか、オフィスを またいで、Sally に、二人の変更が衝突していることを説明 します。[*4] ひとたびコミットする変更内容について合意がとれたら、 ファイルを編集し衝突マーカを削除します。

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

[*4] そして、あなたが頼めば、彼らは電車で町の外まであなたを つれていってくれるかも知れませんよ。

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

これで、**svn resolved** を実行し 自分の変更をコミットする用意ができました:

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

衝突ファイルを編集中に混乱したら、Subversion があなたのために作った、 作業コピーにある三つの一時ファイルを見てどうするかを考えることができます — その中には更新前にあなたが修正したバージョンのファイルもありま す。この三つのファイルを確認するためにサードパーティー製の対話的な マージツールを使うこともできます。

3.6.4.2 作業ファイルの上にファイルをコピーすること

衝突が起こり、自分のした変更を捨てようとするときには Subversion が作った 一時ファイルのどれかを単に作業コピー上に上書きすることができます:

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls sandwich.*
sandwich.txt  sandwich.txt.mine  sandwich.txt.r2  sandwich.txt.r1
$ cp sandwich.txt.r2 sandwich.txt
$ svn resolved sandwich.txt
```

3.6.4.3 Punting: **svn revert** の利用

衝突が起こり、調査の結果、自分の変更を捨てて編集をやり直す場合は 単に変更を revert することができます:

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

衝突ファイルを元に戻すときは **svn resolved** を 実行する必要はないことに注意してください。

これで自分の変更をコミットする用意ができました。 **svn resolved** はこの章であつかう ほかのほとんどのコマンドとは違って、引数を必要とします。 どのような場合でも十分注意して、ファイル中の衝突を 解消したことが確かな場合だけ **svn resolved** を実行してください — 一時ファイルが削除されてしまうと、 Subversion

はファイルが衝突マーカを含んでいたとしてもコミット します。

3.6.5 変更点のコミット

やっとここまできました。編集は終了し、サーバからの変更を すべてマージしました。これで自分の変更をリポジトリにコミット する準備ができました。

svn commit コマンドは自分の変更点の すべてをリポジトリに送ります。変更をコミットするときには 変更点を説明するログメッセージを 与えてやる必要があります。ログメッセージは自分が作った 新しいリビジョンに付けられます。ログメッセージが簡単な 場合は`--message` (あるいは`-m`) オプションを使ってコマンドライン上で指定することができます:

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

しかし、既にログメッセージを作ってある場合は、 `--file` スイッチでファイル名称を指定することで、Subversion に そのファイルの内容を使うように指示できます:

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

`--message` も`--file` も 指定しなかった場合は、Subversion は自動的にエディタを 起動し、(項 7.2.3.2 の `editor-cmd` セクションを見てください) ログメッセージを作成しようとします。

ティップ

もしコミットメッセージをエディタを起動して書いて いて、そのコミットを中止したいと思った場合には、単に保存せずに そのエディタを抜けてください。既にコミットメッセージを保存して しまった場合であれば、テキストを削除してもう一度保存してください。

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

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

リポジトリは、変更点の内容に意味があるかどうかはまったく 気にしません。Subversion はあなたが見ていないところで、同じファイルに 他の人が修正していないことだけを確認します。もし他の人がそのような 変更を していたら 、コミットはあなたの変更 したファイルのどれかが最新ではないというメッセージを出して失敗します:

```
$ svn commit --message "Add another rule"
Sending        rules.txt
svn: commit failed (details follow):
svn: Out of date: 'rules.txt' in transaction 'g'
```

このような場合は、**svn update** を実行し その結果のマージや衝突を解消し、もう一度コミットしてください。

これで Subversion を使う基本的な作業サイクルを説明しました。 Subversion にはこのほかにもたくさんのリポジトリや作業コピーを管理するための 機能がありますが、この章でいままで説明してきたコマンドだけを使っても、 非常に多くのことができます。

3.7 履歴の確認

以前指摘したように、リポジトリはタイムマシンのようなところが あります。いままでコミットされたすべての変更を記録し、 ファイルやディレクトリ、それに付随したメタデータの以前のバージョン を見ることによって履歴を調べることができます。一つの Subversion コマンドを使って、過去の任意の日付やリビジョン番号時のリポジトリ の状態をチェックアウト (あるいは既にある作業コピーの復元) すること ができます。しかし、過去に戻る のではなく、 単に過去がどうだったかをちょっと覗いて みたい こともよくあります。

リポジトリからの履歴データをあつかうためのコマンドがいくつか あります:

svn log 全般的な情報を表示します: リビジョンに付随した日付、修正者つきの ログメッセージとそれぞれのリビジョンでどのパスが変更されたかを表示します。

svn diff 時間とともにあるファイルがどのように変更されてきたかを 表示します。

svn cat これは特定のリビジョン番号時点でのファイルを抽出し 画面に表示します。

svn list 任意の指定したリビジョンのファイルやディレクトリを一覧 表示します。

3.7.1 svn log

ファイルやディレクトリの履歴に関する情報を見たいときは **svn log** コマンドを使ってください。 **svn log** は、あるファイルやディレクトリを 誰が変更したかの記録を表示し、どのリビジョンでそれが変更され たか、そのリビジョンの時刻と日付、さらにもし存在すれば、 コミットに付随したログメッセージを表示します。

```
$ 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 lines

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

ログメッセージはデフォルトでは 時間と逆の順序で 表示されることに注意して ください。別の順序であるリビジョン範囲を見たい場合や、一つの リビジョンを見たいときには、`--revision` (`-r`) スイッチを渡します:

```
$ svn log --revision 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
```

一つのファイルやディレクトリのログ履歴を見ることもできます。 たとえば:

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

これは作業ファイルが (または URL が) 変更されたリビジョン だけを表示します。

もしファイルやディレクトリについてもっと詳細な 情報がほしいときには、**svn log** は `--verbose` (`-v`) スイッチを とることもできます。Subversion はファイルやディレクトリの 移動やコピーもできるので、ファイルシステム中のパスの 変化を追えることは重要です。冗長モードでは、 **svn log** は出力リビジョンの中に 変更されたパス情報の一覧も含めます:

```
$ 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 は `--quiet` (`-q`) スイッチも指定でき、これはログメッセージの本文を 表示しません。`--verbose` と組み合わせて指定すると 変更したファイルの名前だけを表示します。なぜ、**svn log** の出力が何も ないの?

Subversion を使い始めてすぐのとき、たいていのユーザは 以下のようなことに出くわすでしょう:

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

一見エラーのように見えますが、リビジョンがリポジトリ全体に対する ものであるのに対して **svn log** はリポジトリ中のパスに対して 働くものであるのに注意してください。パスを指定しなければ Subversion は デフォルトの対象として現在の作業ディレクトリを使います。結果として 自分の作業コピーのサブディレクトリで実行し、そのディレクトリにもその 子供のディレクトリ中でも変更がなかったリビジョンに対してのログを見ようと すると、Subversion は空のログを表示するでしょう。そのリビジョンでの変更点 を見たいのならリポジトリの最上位の URL を直接指定して **svn log** を実行しましょう。こんな感じです。**svn log -r 2 http://svn.collab.net/repos/svn**。

3.7.2 svn diff

svn diff は既に見てきました — unified diff 形式でファイルの差分を表示するのでした。 リポジトリにコミットする前に作業コピーにされたローカル 修正点を表示するのに使えます。

実際には **svn diff** には異なる 三種類 の使い方があります:

- ローカルの変更内容の確認
- 作業コピーとリポジトリの比較
- リポジトリとリポジトリの比較

3.7.2.1 ローカルの変更内容の確認

見てきたように、スイッチなしで **svn diff** を 実行すると、作業コピーの内容と、`.svn` 領域にキャッシュされている「修正元リビジョン」 のコピー とを比較します:

```
$ 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
$
```

3.7.2.2 作業コピーとリポジトリの比較

`--revision(-r)` を一つ 指定すると、作業コピーはリポジトリの特定のリビジョンと比較 されます。

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

```
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

3.7.2.3 リポジトリとリポジトリの比較

`--revision(-r)` の引数と してリビジョン番号を二つ、コロンで区切って指定すると 二つのリビジョンが直接比較されます。

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

作業コピーとリポジトリのファイルを比較するためにだけ **svn diff** を利用できるのではなく、URL 引数を与えることで作業コピー を用意しなくてもリポジトリ中のアイテムの間の差を調べることができます。 これは、ローカルマシンに作業コピーがないときに、ファイルの変更点を 知りたいような場合に非常に便利です:

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

3.7.3 svn cat

もし、以前のバージョンのファイルを見たいが、二つのファイル間の違い を見る必要はないような場合には、 **svn cat** が使えます:

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

```
Chew with your mouth open
$
```

直接ファイルに出力することもできます:

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

もしかすると、どうして古いリビジョンに戻すためのファイルの更新に 単に **svn update –revision** を使わないのか、と 思うかも知れません。 **svn cat** を使ったほうが 良い理由がいくつかあります。

まず、外部の diff(多分、GUI かも知れないし、unified diff 形式の出力が 意味を持たないようなファイルなのかも知れません) プログラムによって二つのリビジョンのファイル間の差分を見たいかも知れません。 この場合、古いバージョンのコピーを取得する必要があり、その内容をファイル に出力したものと、作業コピー中のファイルの両方を外部 diff プログラムに 渡さなくてはなりません。

しばしば、他のリビジョンとの間の差分をとるよりも、その古いバージョン のファイル全体を見るほうが簡単なことがあります。

3.7.4 svn list

svn list コマンドはローカルマシンに 実際にファイルをダウンロードすることなしに、リポジトリ にどんなディレクトリがあるかを表示します:

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

もっと詳しい表示がほしいときには `--verbose` (`-v`) フラグを 指定します。出力は以下のようになります:

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

それぞれの項目の意味は、左から順に、ファイルまたはディレクトリが最後に 更新されたリビジョン、修正した人、ファイルであればそのサイズ、日付、そして そのアイテムの名前になります。

3.7.5 履歴機能について、最後に

いままで述べてきたすべてのコマンドに加えて **svn update** と **svn checkout** を、`--revision` 付きで実行することも できます。これは作業コピー全体を「過去のある時点」 に戻します。 [*5]:

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

3.8 その他の役に立つコマンド

この章でいままで述べてきたほど利用されるわけではありませんが、 以下のコマンドがときどき必要になります。

3.8.1 svn cleanup

Subversion が作業コピー (や `.svn` に ある情報) を修正するときには、できるだけ安全にやろうと します。作業コピーの内容を変更する前に Subversion はまず変更手順をログファイル に書きます。次に実際に変更を適用するためにログファイルの中の コマンドを実行していきます。最後に Subversion はログファイルを削除します。 プログラムの構成という意味では、これはジャーナル化ファイルシステムと よく似ています。Subversion の操作が中断されると (プロセスが異常終了 したり、マシンがクラッシュしたり、といった場合) ログファイルは ディスクに残ります。ログファイルを再実行することで Subversion は 以前に開始された操作を完結することができ、作業コピーを正常で 一貫した状態に戻すことができます。

svn cleanup がやるのは、まさにこのことです。 作業コピーを探して、残ったログを実行し、プロセスのロックを 取り除きます。Subversion に作業コピーのどこかが「ロック」 されていると言われたときには、このコマンドを実行してください。 同様に **svn status** はロックされているアイテム の隣に `L` を表示してそのことを示します:

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

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

[*5] おわかりでしょうか? これが Subversion が タイムマシンだと言った意味です。

3.8.2 svn import

svn import コマンドはバージョン管理されていない複数のファイル をリポジトリにコピーし、必要に応じて直ちにディレクトリを作るための簡単 な方法です。

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

Committed revision 1.
```

上の例はディレクトリ `mytree` の内容を リポジトリ中の `some/project` ディレクトリの下に コピーしています:

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

インポートが終わった後で、もとのツリーが作業コピーに変換さ れたわけではないのに注意してください。作業を始めるには、さらにこのツリー のための最初の作業コピーを **svn checkout** する 必要があります。

3.9 まとめ

これで、Subversion クライアントのコマンドの大部分について 説明しました。触れなかったもののうちで重要なのはブランチと マージ (第 4 章参照) 、そして属性です (項 7.3 参照)。 Subversion が持っているたくさんのコマンドの感じをつかむには 第 9 章 をざっと見るのもいいかも知れません — 自分の仕事がどれだけ楽になるか、わかるでしょう。

第 4 章

ブランチとマージ

4.1

ブランチ、タグ、マージはほとんどすべてのバージョン管理システムで 共通の概念です。もしあまりなじみがないのであれば、この章は良い とっかかりになるでしょう。既に詳しいのであれば、これらの概念 を Subversion がどのように実装しているかを知るのに興味深い章である ことがわかるでしょう。

ブランチ化は、バージョン管理の基本にあります。Subversion で自分の データをマージするときには、この機能はときどき必要となる機能です。 この章では、あなたが Subversion の基本コンセプトを既に理解している ことを前提とします (第 2 章)。

4.2 ブランチとは?

あなたの仕事が、何かのハンドブックを扱う企業の一部署で、ドキュメントの 管理をすることだとします。ある日別の部署から同じハンドブックが必要 なのだが、ある部分を「ちょっとだけ」変えたものがほしい、ほんの少しだけ 業務形態に違いがあるから、といわれたとします。

この状況で、あなたはどうしなくてはならないでしょうか? 答えはあたりまえです: ドキュメントのコピーを作って二つのコピーを 別々に管理することにします。それぞれの部署が小さな変更を依頼して くるたび、一方を修正したり、もう一方を修正したりします。

両方のコピーに同じ修正を加えたいこともよくあります。たとえば 最初のコピーにスペルミスがあったとします。もう一方のコピーにも おそらく同じ間違いがあるでしょう。両方のドキュメントはほとんど同じ なのですから。二つはほんの少し違っているだけです。

これは ブランチの基本的な概念です — つまり、一つの開発の流れが、もう一方と独立して存在しているが、もし過去にさかのぼれば、同じ履歴を共有している、という状況です。 ブランチは必ず、何かのコピーから始まり、枝分かれして、 自分自身の歴史を持っていくようになります (図 4.1 を参照してください)。

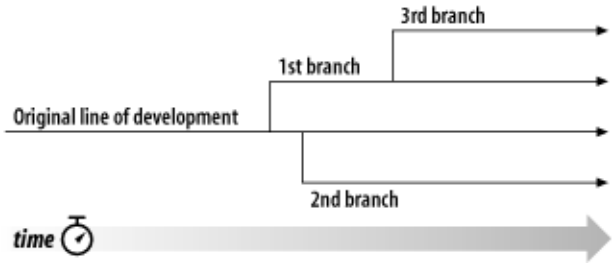

図 4.1 開発のブランチ

Subversion はファイルやディレクトリの平行したブランチを管理するのを 手助けするコマンドがあります。

データをコピーしてブランチを作ったり、 どのように二つのコピーが関係しているかを記憶しておくことができます。 片方のブランチに対する修正をもう一方にも追加する作業を助けることも できます。最後に、作業コピーの一部だけ別のブランチにすることもできる ので、通常の作業で、別の作業のラインを「混ぜあわせる」こともできます。

4.3 ブランチの利用

これまでのところで、それぞれのコミットがどうやってリポジトリに完全に新しい ファイルシステムツリー(「リビジョン」と呼ばれます) を作るかを知っていると思います。 まだ知らないのであれば、戻ってリビジョンに関する項 2.4.2 を 読んでください。

この章では、第 2 章と同じ例を使います。同僚の Sally とあなたが `paint` と `calc` という 二つのプロジェクトのあるリポジトリを共有していたことを思い出して ください。しかし、図 4.2 を見ると、個々の プロジェクトディレクトリは `trunk` と `branches` というサブディレクトリを含んでいること に注意してください。この理由はすぐに明らかになります。

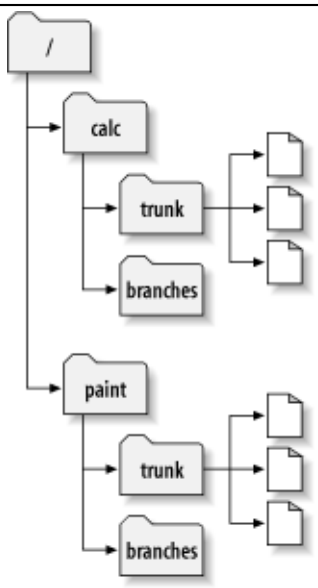

図 4.2 リポジトリレイアウトの開始

以前と同様、あなたと Sally はそれぞれ 「calc」 プロジェクトの作業コピーを持っているとします。特に両者はそれぞれ `/calc/trunk` の作業コピーを持っています。 プロジェクトのすべてのファイルは `/calc` にではなくこのサブディレクトリ中にありますが、それは皆が開発の「主系」 を `/calc/trunk` に置くことに決めたからです。

あなたはプロジェクトの大胆な再編成を任されたとします。 それには長い時間が必要で、プロジェクトの全ファイルに影響 を与えます。問題はあなたは Sally に干渉したくない ということにあります。彼女はまだあちこちにある小さなバグを 潰している最中だからです。彼女はプロジェクトの最終バージョンが (これは`/calc/trunk` にあるのですが) 利用可能だということに依存しています。もし、あなたが自分の 変更をちょっとづつコミットすれば、Sally の作業を確実に中断させて しまうでしょう。

一つのやり方として、閉じこもってしまう方法があります:あなたと Sally は 1,2 週間、情報を共有するのをやめます。つまり、自分の 作業コピー中の全ファイルに対する大手術を始めるのですが、 それが完了するまで、コミットも更新もしないという方法です。 しかしこれにはいろいろな問題があります。まず安全ではありません。 ほとんどの人は、作業コピーにヘンなことが起こらないように、 リポジトリに対してこまめに自分の

作業を保存するのを好みます。 次に、まったく柔軟ではありません。もし、あなたがたが別の マシンで仕事をしているなら、(多分二つの別のマシンに `/calc/trunk` の作業コピーがあるのでしょう) 自分の変更を手であちこちにコピーしなくてはならないか、 一つのマシン上に作業全体をフルコピーするかになります。 同じようにして他の誰との間でも自分の進行中の変更部分を共有する ことは困難です。通常のソフトウェア開発で「一番よいやり方」 はあなたの作業の進行状況を他の人からも参照できるようにすることです。 もしあなたの中間的なコミットを誰も見ることができないとすると あなたは他の人からフィードバックしてもらうことができなくなって しまいます。 最終的に自分の変更作業が完了したとき、その変更を コミットするのは非常に困難であることに気づくでしょう。Sally(と 他のメンバー) はリポジトリに対してたくさんの別の変更を加えており、 それをあなたの作業コピーにマージするのは困難です — 何週間も孤立した作業の後に **svn update** を実行する ような場合には特にそうです。

もっとましなやり方はリポジトリに自分用のブランチ、あるいは 自分用の作業の別ラインを作ることです。これは他の人に干渉 せずに、自分の中途半端な作業をときどき保存できるように しますが、それでも同僚との間で、一部の情報については共有する ことができます。どうやったらこんなことができるかは後で説明 します。

4.3.1 ブランチの作成

ブランチの作成はとても簡単です — **svn copy** コマンドでリポジトリ中のプロジェクトをコピーするだけです。 Subversion では一つのファイルをコピーするだけでなく、ディレクトリ 全体をコピーすることができます。今回は、`/calc/trunk` ディレクトリのコピーがほしいでしょう。新しいコピーはどこに置けば 良いのでしょう? 好きな場所に置けます — あとはプロジェクトのポリシーに よります。チームのポリシーは、リポジトリの`/calc/branches` 領域にブランチを作ることで、ブランチ名は 「my-calc-branch」 としましょう。この場合、`/calc/trunk` のコピーとして、 `/calc/branches/my-calc-branch` という新しいディレクトリ を作る必要があります。

コピーを作るには、二つの方法があります。面倒な方法を最初に 説明して、概念をはっきりさせます。最初にプロジェクトのルートディレクトリ である `/calc` を作業コピーにチェックアウトします:

```
$ 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.
```

あとは、 **svn copy** コマンドに作業コピーパスを 二つ渡すだけでコピーを作れます:

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

この場合、**svn copy** コマンドは再帰的に `trunk` 作業ディレクトリの内容を 新しい作業ディレクトリ `branches/my-calc-branch` にコピーします。**svn status** コマンドで 確認できますが、これで新しいディレクトリはリポジトリへの追加として 予告されます。ただ、A の後に、「+」サインが表示されるのに注意して ください。これは、追加予告が、新規のものではなく、何かの コピー であることを示しています。 変更をコミットすると、Subversion は、ネットワーク越しに 作業コピーデータの全体を再送信するのではなく、 `/calc/trunk` をコピーすることで リポジトリに `/calc/branches/my-calc-branch` を作ります:

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

さて、ブランチを作るもっと簡単な方法は、先に説明すべきでした が: **svn copy** は 引数に直接 URL を二つとることが できるということです。

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

Committed revision 341.
```

この二つの方法には何の違いもありません。 両方とも新しいリビジョン 341 のディレクトリを作り、新しいディレクトリは`/calc/trunk` のコピーに なります。図 4.3 にこれを示しました。 ただし二番目の方法は同時に コミットも発行します。 [*1] 二番目のほうが楽です。リポジトリの 大きなコピーをチェックアウトしなくていいからです。 実際、この方法では、作業コピーそのものを用意する必要 すらありません。

簡易コピー

Subversion のリポジトリは特殊な設計になっています。ディレクトリ をコピーするとき、リポジトリが不要に大きくなるかと心配する 必要はありません。 — Subversion は実際には全然データを コピーしません。そのかわり、既に存在している ツリーを指し示すような新しいディレクトリを作ります。 Unix ユーザなら、これはハードリンクの概念と同じです。 そんなわけで、このコピーは、「ものぐさ方式」と呼ばれます。 つまりコピーされたディレクトリ中の一つのファイルの変更を コミットしたとき、そのファイルだけが変更されます — 残りの ファイルは、依然として最初のディレクトリのもともとのファイル へのリンクのままです。

これが、Subversion ユーザが、「簡易コピー」という言葉をよく聞く 理由です。ディレクトリがどれほど大きいかには無関係なのです — コピーには非常にわずかな、一定の時間がかかるだけです。 これが Subversion でのコミットのやり方の基本です: それぞれの リビジョンは前のリビジョンの「簡易コピー」で、その中のいくつか のアイテムだけが、実際にコピーされます。(もっと知りたい人は Subversion のウェブサイトに行き、Subversion 設計ドキュメント中の 「bubble up」方式を読んでください)

もちろん、このようなデータのコピーと共有の内部的な仕組みは ユーザからは見えず、単にツリーのコピーが見えるだけです。ここ での要点はコピー処理は時間的に空間的にも軽いということです。 好きなだけブランチを作ってください。

[*1] Subversion はリポジトリ間コピーをサポートしていません。**svn copy** や **svn move** で URL を指定する場合、同じリポジトリ内でのみコピーすることができます。

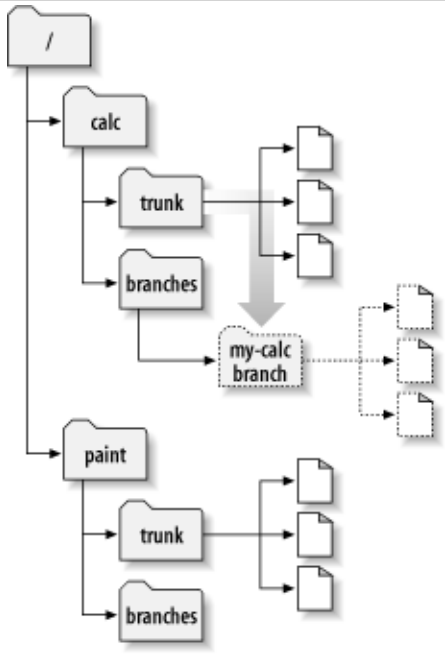

図 4.3 新しいコピーのあるリポジトリ

4.3.2 自分用のブランチでの作業

これでプロジェクトにブランチを作ることができたので それを使った新しい作業コピーをチェックアウトできます:

```
$ 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.
```

この作業コピーについては何も特別なことはありません。単に 別のディレクトリにあるリポジトリのコピーだというだけです。 ただし、あなたが変更をコミットして、その後に Sally が更新しても その変更を見ることはありません。彼女の作業コピーは、 `/calc/trunk` からのものだからです。 (この章の項 4.6 を読んでください: **svn switch** コマンドはブランチの作業コピー を作る別の方法です。)

一週間が経過する間に、以下のコミットが起こったとしましょう:

- `/calc/branches/my-calc-branch/button.c`, に変更を加え、リビジョン 342 を作った。
- `/calc/branches/my-calc-branch/integer.c`, に変更を加え、リビジョン 343 を作った。
- Sally は `/calc/trunk/integer.c` に 修正を加え、リビジョン 344 を作った。

これで、図 4.4 に示すように `integer.c` に二つの独立した開発ラインができました:

`integer.c` のコピーに起きた変更履歴を 見ると面白いことがわかります:

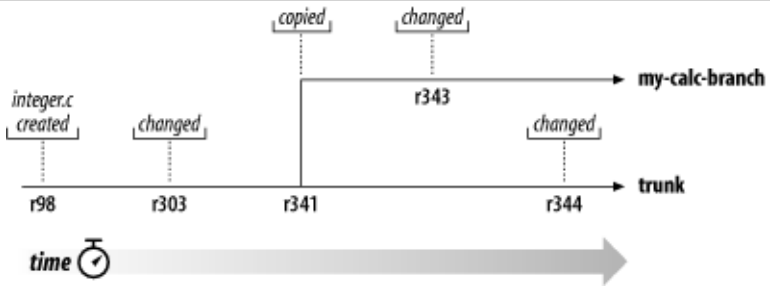

図 4.4 あるファイルの履歴のブランチ化

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

$ svn log --verbose 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.

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

Subversion はブランチにある integer.c の履歴を 時間を逆向きにたどり、これにはコピーされた地点も含まれることに注意 してください。それはブランチの生成を履歴上の一つのできごととして 表示しますが、それは integer.c も /calc/trunk/全体がコピーされたときに暗黙に コピーされたものだからです。今度は Sally が自分のファイルコピー上 で同じコマンドを実行した結果を見てみましょう:

```
$ pwd
/home/sally/calc

$ svn log --verbose 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 は自分のリビジョン 344 の変更を見ることができますが、あなたが リビジョン 343 にやった変更は見ることができません。Subversion では、 この二つのコミットはリポジトリの別の場所にある別のファイルに対して起こります。しかし、Subversion は、二つのファイルが共通の履歴を 持っていることを示してもいます。リビジョン 341 で起きたブランチコピーの前は両者は同じファイルを使っていました。 Sally とあなたがどちらもリビジョン 303 と 98 での変更を見ることができるのは そのためです。

4.3.3 ブランチの背後にある鍵となる考え方

この節での重要事項は二つです。

1. 他のたくさんのバージョン管理システムとは違って Subversion の ブランチはリポジトリ中の普通のファイルシステム のディレクトリ として存在します。特別な仕組みが あるわけではありません。これらのディレクトリは単にある特別な 履歴情報も保持しているというだけのことです。
2. Subversion は内部的にはブランチという概念を持ちません — それはただのコピーです。ディレクトリをコピーした とき、結果としてできたディレクトリが「ブランチ」であるのは、 あなたが そのような意味で見ることに したからです。そのディレクトリを別の意味合いにとらえたり 取り扱ったりすることもできますが、いずれにせよ Subversion に とっては、コピーによって作成された普通のディレクトリの 一つにすぎません。

4.4 ブランチをまたいで変更をコピーすること

さて、あなたと Sally はプロジェクト上の平行したブランチで作業しています。 あなたは自分のプライベートなブランチで作業していて、Sally は *trunk*、あるいは、開発の主系の上で作業していると します。

たくさんの貢献者がいるようなプロジェクトでは、ほとんどの人たちは trunk のコピーを持っているのが普通です。trunk を壊してしまうかも知れない ような長い期間をかけての変更を加える必要がある場合は常に、標準的な手続き としてはまずプライベートなブランチを作り、すべての作業が完了するまで変更 点をそのブランチにコミットします。

そのようなやり方の利点としては、二人の作業はお互いに干渉しないところです。 欠点は二人の作業内容はすぐにひどく 違っていって しまうことです。「引きこもり」戦略の問題の一つは自分のブランチの作業が 完了するときに起こることを思い出してください。恐ろしくたくさんの衝突 なしに、あなたの変更を trunk にマージするのはほとんど不可能でしょう。

そのかわりに、作業中に、あなたと Sally は変更を共有し続けるのが良い でしょう。どのような変更が共有する価値があるのかはあなたが決める ことです。Subversion を使うとブランチ間の選択的な「コピー」ができます。 そしてブランチ上での作業が完全に終ったら、ブランチ上にした変更点の 全体を trunk に書き戻すことができます。

4.4.1 特定の変更点のコピー

前の節で、あなたと Sally は別ブランチ上で `integer.c` に変更を加えたと言いました。もしリビジョン 344 の Sally のログメッセージ を見れば、何かのスペルミスを直したことがわかるかも知れません。 この場合間違いなく、同じファイルのあなたのコピーもやはり同じスペルミスが あるはずです。このファイルに対する今後のあなたの修正はスペルミスのある 場所に影響を与えるかも知れず、自分のブランチをいつかマージするときに は衝突が起こってしまいます。そうなるくらいなら、あまりひどいことになる 前に、Sally の修正をいま受け取ったほうが良いでしょう。

svn merge コマンドを使うときがやってきました。 このコマンドは、 **svn diff** に非常に近い 親戚だということがわかります。(このコマンドは第 3 章で説明しました)。 両方ともリポジトリ中の二つのオブジェクトを比較して、その差を 調べることができます。たとえば **svn diff** に Sally がリビジョン 344 でやった変更点を正確に表示することができます:

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

Index: integer.c
===================================================================
```

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

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

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

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

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

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

svn merge コマンドもほとんど同じです。差分を 画面に表示するかわりに、それはローカルな 修正分として直接あなたの作業コピーに適用 します:

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

$ svn status
M  integer.c
```

svn merge の出力は、あなた用の `integer.c` のコピーがパッチされた 結果です。これで Sally の変更が含まれるようになりました — それは trunk からあなたのプライベートなブランチの作業コピーに 「コピー」され、ローカルな修正の一部となりました。この修正を再検討し、 正しく動作することを確認するのはあなたの仕事です。

別のシナリオとして、そんなにうまくはいかず、 `integer.c` が衝突の状態になることもあります。 標準的な方法を使って衝突を解消するか (第 3 章を見てください)、結局マージが悪いアイディアだったと思ったときには、あきらめて **svn revert** でローカルの変更を取り消すことも できます。

しかし、マージされた変更を確認して、**svn commit** をかけるのが普通です。これで、変更は自分のリポジトリブランチに マージされました。バージョン管理の言い方では、このようなブランチ間の 修正点のコピーを、普通 *porting* による変更と いいます。

ローカルな修正をコミットするときには、あるブランチから別のブランチ に対して特定の変更を移したことを示すようなログメッセージになって いることを確認してください。たとえば:

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

次の節で見るように、これは参考にすべき 「最善の方法」 です。非常に重要です。どうしてパッチを使わないの?

そう思うかも知れません。特にあなたが Unix ユーザならそうでしょう。 なんでわざわざ **svn merge** みたいなものを使うのか? どうして単に OS についている **patch** コマンドを 使って同じことをしないのか? たとえば:

```
$ svn diff -r 343:344 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
```

このような特別な場合なら、おっしゃる通り。何の違いも ありません。しかし **svn merge** は **patch** ではできない特殊な機能があります。 **patch** で使えるファイル形式は非常に 限定されています。それは単にファイ

ル内容をわずかに 変更することができるだけです。複数のファイルやディレクトリの 追加、削除、名称変更のようなツリーを 変更する仕組みを持っていません。 Sally の変更が新しいディレクトリを追加するようなものだった 場合、 **svn diff** はそのことに全然注意を むけないでしょう。 **svn diff** は限定された パッチ形式の出力をするだけで簡単には表現できないことが あります。 [*2] しかし **svn merge** コマンドは作業コピーに直接 働くことでツリー構造と属性の変更点を表現することができます。

注意: **svn diff** と **svn merge** は とてもよく似たコンセプトを持っていますが、いろいろな場合で 別の構文になります。関連した第 9 章をよく読むか、**svn help** を使ってください。たとえば **svn merge** は作業コピー パスを引数とします。つまりツリーの変更を適用する場所の指定が必要 になります。この指定がなければ、よく利用される以下の操作のどちらか を実行しようとしているとみなされます:

1. 現在の作業ディレクトリ中に、ディレクトリの変更点をマージ しようとしている。
2. 現在の作業ディレクトリ中にある同じ名前のファイルに対して、 ある特定のファイルに起きた修正をマージしようとしている。

ディレクトリをマージしようとしている場合で、目的のパスを指定しなかった 場合、**svn merge** は、上にあげた第一の場合であると みなし、現在のディレクトリ中のファイルに対して適用しようとします。 もし、ファイルをマージしようとしている場合で、そのファイル (または 同じ名前のファイル) が作業コピーディレクトリに存在している場合、 **svn merge** は第二の場合であるとみなし、同じ名前の ローカルファイルに対して変更を適用しようとします。

上記以外の場所に適用したい場合には そのことを明示的に指定する必要があります。たとえば作業コピーの親ディレクトリ にいて、変更を受け取るための対象ディレクトリを指定する必要がある場合なら:

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

4.4.2 マージの基本的な考え方

ここまでのところで **svn merge** の例を見てきましたが、さらにいくつかの例を あげます。マージが本当のところどのように機能するかについて何か混乱 した気になるのは何もあなただけではありません。多くのユーザは (特に バージョン管理システムになじみのない人にとっては) まず最初にコマンド の構文に戸惑い、さらにどのようにして、またいつその機能をつかえば良い かということにも戸惑います。しかし怖がることは何もありません。このコマンドは 実際にはあなたが思っているよりずっと単純なものです。**svn merge** がどのように動作するかを正確に知るためのとても簡単な方法があります。

混乱の一番の原因はこのコマンドの名前です。「マージ (merge)」という言葉は、何か二つのブランチが統合されたり、データ同士が、何か神秘的な方法で混ぜ合わされてしまったりするような表現 です。しかし、そんなことがおこるわけではありません。多分このコマンドに 対するもっとふさわしい名前は **svn diff-and-apply**(差分 をとってから、それを適用する) かも知れません。実際、起こることは本当に それだけなのですから: つまり、二つのリポジトリのツリーが比較され、その 差分が、作業コピーに適用されるのです。

このコマンドは三つの引数をとります:

1. 最初の状態を示すリポジトリ・ツリー (比較時の左側 などとよく言われます),

[*2] 将来的には Subversion プロジェクトはツリー構造と属性の変更点を表現する ような拡張したパッチ形式を使う (あるいは開発する) 計画があります。

2. 最終的な状態を示すリポジトリ・ツリー (often called the 比較時の右側 などとよく言われます),
3. 上記二つの間の差分をローカルな変更として受け入れる作業コピー (マージの ターゲットなどとよく言われます).

この三つの引数が指定されると二つのツリーが比較され、結果の 差分がターゲットの作業コピーに対して、ローカルな修正点の形で反映されま す。この結果はあなた自身が手作業でファイルを編集したり、**svn add** や **svn delete** コマンドをいろいろと実行 したのとなんら変わるところはありません。結果の修正内容が満足のいくもの であれば、それをコミットすることができます。気に入らなければ、単に **svn revert** を実行しさえすればすべての変更は元に戻り ます。

svn merge の構文は必要な三つの引数をある程度 柔軟に指定できるようになっています。以下がその例です:

```
$ 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
```

最初の構文は三つのすべての引数を明示的に指定するもので、ツリーについては それぞれ *URL@REV* の形で指定し、ターゲットの作業コピー はその名前で示します。二番目の構文は、同じ URL 上にある異なるリビジョンを 比較する場合の略記法です。最後の構文は作業コピーを省略した場合の例です; デフォルトではカレントディレクトリが指定される決まりです。

4.4.3 マージの一番うまいやり方

4.4.3.1 手でマージする方法

変更のマージは非常に単純なことに思えますが実際には厄介な ものです。問題は、もし一つのブランチを別のブランチに対して 変更点を繰り返しマージすると、間違って同じ変更を 二度やってしまうかも知れないということです。 こういうことが起こっても、問題が起こらないこともあります。 ファイルをパッチするとき、Subversion はファイルが既に変更されている 場合にはそれに気がついて、何もしません。しかし、既に存在している 変更が何らかの方法で修正されていた場合、衝突が起こります。

理想的には、バージョン管理システムはブランチに対して変更点の重複 した適用を回避すべきです。ブランチが既に受け取った変更点を自動的に 記憶し、その一覧を表示できるようにすべきです。そしてバージョン管理システム は自動マージを支援するために可能な限りこの情報を利用すべきです。

残念ながら Subversion はそのようなシステムではありません。CVS と同様 Subversion はまだマージ操作に関するどのような情報も記録しません。 ローカルな修正をコミットしても、リポジトリはそれが **svn merge** を実行したものによるのか、あるいは単に手でファイルを修正した ものによるのか区別できません。

これはユーザにとって何を意味するのでしょうか? それは Subversion にこの 機能がいつか実装されるまではマージの情報を自分で記録しておく必要が あるということです。一番良い場所はコミットログメッセージ中でしょう。 以前の例で説明したように、あなたのブランチにマージした特定のリビジョン 番号 (あるいはリビジョン番号の範囲) をログメッセージ中で示しておくこと をお勧めします。あとで **svn log** を実行してあなた

の ブランチがどの変更点を既に含んでいるかを知ることができます。これで **svn merge** コマンドを繰り返し実行する際に以前に 取り込んだ変更点を再び取り込むことがないように注意することが できます。

次の節ではこの技法の例を実際にお見せします。

4.4.3.2 マージ内容の確認

マージは作業コピーを変更するだけなので、それほど危険な操作では ありません。マージに失敗しても、単に **svn revert** を実行すれば元に戻せるのでもう一度 やり直すことができます。

しかし作業コピーには既にローカルな修正が加えられていることもあります。 マージによって適用された修正は既に加えていた修正と混じってしまう のでこの場合には **svn revert** は使えません。 この二つの修正の組を分離することは不可能です。

このような場合には、実際にマージする前に、マージしたとしたらどうなるか を調べておくべきです。このための一つの簡単な方法としては **svn merge** に渡そうとしているのと同じ引数で **svn diff** を実行する方法があります。それは既にマージの最初の例で見たものです。 もう一つの方法は、マージコマンドに対して `--dry-run` オプションを渡す方法です:

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

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

`--dry-run` オプションは、実際には作業コピーに対してローカルな 修正を適用しません。実際のマージで表示されるであろう 状態コードを表示するだけです。これは **svn diff** ではあまりにも詳細 な内容が表示されてしまうような場合に、潜在的なマージの概要を確認するための 「高度な」方法です。Subversion とチェンジセット

「チェンジセット」の定義は人によって少しずつ違うか、 少なくともバージョン管理システムが「チェンジセットの機能」 を持っているというときに期待するものは異なっています。ここでは チェンジセットは一意の名前がついた変更点の集まりであるとしておきましょう。 この変更はファイル内容の編集、ツリー構造の変形、メタデータの修正などが 含まれています。もっと普通の言い方をすると、チェンジセットとは参照できるような名前がついたパッチのことです。

Subversion では大域的なリビジョン番号 N がリポジトリ中のツリーの名前に なります: それはリポジトリの N 番目のコミット後の様子です。またそれは 暗黙のチェンジセットの名前でもあります: もしツリー N と N-1 を比較 すればコミットされたパッチを正確に求めることができます。この理由により、 「リビジョン N」をツリーと考えることもできますが、また チェンジセットとみなすこともできます。バグを管理するバグ追跡システム を使っているなら、バグを修正した特定のパッチに言及するのにリビジョン 番号を使うことができます — たとえば「この問題はリビジョン 9238 で修正されました」といった具合です。ついで別の人は **svn log -r9238** を実行してそのバグを修正したチェンジセット についてのログを見ることができ、さらに **svn diff -r9237:9238** を実行して、そのパッチ自身を表示することもできます。Subversion の `merge` コマンドもまたリビジョン番号を使います。あるブランチから別の ブランチに特定のチェンジセットをマージしたい場合にはマージの引数で それを与えます: **svn merge -r9237:9238** はチェンジセット #9238 をあなたの作業コピーにマージするでしょう。

4.4.3.3 マージの衝突

svn update コマンドと同様 **svn merge** は変更を作業コピーに対して行うので 衝突を起こすこともあります。しかし **svn merge** によっておきた衝突については様子が 違うこともあり、以下ではこの違いについて説明します。

まず、作業コピーにはローカルな修正が加えられていないとします。 特定のリビジョンに対して **svn update** を実行すると サーバから送られてきた変更点は作業コピーに対して常に 「きれいに」適用されます。サーバは二つのツリーを比較することで差分を 生成します: 作業コピーの仮想的なスナップショットと、適用しようとして いるリビジョンとの間の差分です。前者は作業コピーと全く同じものなので この差分が作業コピーをきれいに後者に変換することは保証されています。

しかし **svn merge** の場合はそのような保証はなく、 結果はもっと混沌としたものになる可能性もあります: ユーザは全く 任意の二つのツリーの比較をするようサーバに 指示することもでき、作業コピーとは全く無関係なものであるかも知れないのです!。 これは人間の側の操作ミスが起こる潜在的な可能性が大きいことを 意味します。場合によってはユーザは間違った二つのツリーを比較し、 きれいに適用できないような差分を作ってしまうかも知れません。 **svn merge** はできる限りこの差分を適用しようと しますが、ある部分は不可能かも知れません。ちょうど Unix の **patch** コマンドが「適用できなかったハンク」について文句を言ってくることがあるのと同じように **svn merge** は「処理を飛ばしたファイル」 について文句を言うかも知れません:

```
$ 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

$
```

この例では比較対象となる二つのブランチのスナップショットの両方に `baz.c` が存在していたため、生成された差分もその ファイルの内容を変更しようとしますが、作業コピー中には対応するファイル が存在しなかったような場合だと考えられます。いずれにせよ 「スキップ」のメッセージはユーザが間違った二つのツリーを 比較してしまったことを意味することがほとんどです。ユーザ側のエラー を示す典型的な状況です。こうなった場合でも (**svn revert –recursive** を使って)、マージによって実行されたすべての変更 点を再帰的に元に戻し、バージョン化されていないファイルやディレクトリ が残っている場合にはそれらも削除し、正しい引数で **svn merge** を再実行するのは難しいことではありません。

前の例では `glorb.h` に衝突が起きたことにも注意 してください。今回の場合作業コピーに対してローカルな修正がされていない ことはすでに述べました: ではなぜ衝突が起きるのでしょうか? この場合 でもやはりユーザは **svn merge** で古い差分を作ってから 作業コピーに適用することができるので、ローカルな修正がなかったと しても、その差分が作業コピーに対してきれいに適用できないような変更を 含んでしまうことはありうるのです。

その他 **svn update** と **svn merge** の 小さな違いとしては衝突がおきたときにできるテキストファイルの名前です。 項 3.6.4 で見たように、update の場合には `filename.mine`, `filename.rOLDREV`, `filename.rNEWREV` という名前のファイルができます。 これにたいして **svn merge** の場合には `filename.working`,

`filename.left`, `filename.right` という名前になります。 この場合「left」 と 「right」 は、それぞれの ファイルが比較した二つのツリーのどちら側に由来するものかを示しています。 いずれにせよファイル名称の違いは、衝突が update コマンドの結果である のか merge コマンドの結果であるかを区別する助けになるでしょう。

4.4.3.4 系統 (Ancestry) を考慮することと無視すること

Subversion 開発者と会話するとき系統 (ancestry) という言葉を非常によく耳にするでしょう。 この言葉はリポジトリ中の二つのオブジェクト間の関係を記述するた めに用いられるものです:もし両者が互いに関係している場合、ある オブジェクトはもう一方の祖先 (ancestor) といわれます。

例えば、リビジョン 100 をコミットし、それが `foo.c` というファイルへの変更を含んでいると します。すると `foo.c@99` は `foo.c@100` の「祖先」ということ になります。一方リビジョン 101 で `foo.c` を 削除するコミットがあり、リビジョン 102 で同じ名前の新しいファイ ルを追加したとしましょう。この場合 `foo.c@99` と `foo.c@102` は関係しているように見えます (なぜなら同じファイル名なのですか ら) が、実際にはリポジトリ中ではまったく別のオブジェクトです。 両者は履歴、あるいは「系統」を共有していないからで す。

ここでこんな話をするのは、**svn diff** と **svn merge** の間の重要な違いを指摘したいからです。前者 は系統を無視しますが、後者は系統を非常に慎重に考慮します。例えば **svn diff** でリビジョン 99 と 102 の `foo.c` を比較した場合、行単位の差分を見ることになる でしょう; `diff` コマンドは二つのファイル名を無条件に比較するからです。 しかし **svn merge** を使っていまと同じ二つのオブジェクトを比較す るとそれらが無関係であることを検知し古いファイルをいったん削除し、それ から新しいファイルを追加しようとするでしょう; 出力は追加のあとに削除した ことを示すものとなるでしょう:

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

ほとんどのマージはお互いに系統上関係したツリーを比較する ので、**svn merge** はデフォルトで上記のような動 作になります。しかし、二つの無関係なツリーを比較するために `merge` コマンドを使いたいと思うこともあるかも知れません。 たとえばあるソフトウェアプロジェクトの、異なる 二つのベンダーリリースを表すようなソースコードツリーをインポー トするかも知れません (項 7.7 参照)。 この二つのツリーを **svn merge** で比較すると最初 のツリー全体がいったん削除され、次いで後のツリー全体が追加され たように見えるでしょう!

このような場合、**svn merge** は単にファイ ル名ベースの比較のみを実行し、ファイルやディレクトリの系統上の 関係を無視したいと考えるでしょう。こんなときはマージコマンドに `--ignore-ancestry` オプションをつければ ちょうど **svn diff** と同じように振舞うようにな ります。(逆に **svn diff** コマンドに `--notice-ancestry` オプションをつけると **svn diff** コマンドは `merge` コマンドと同じよう に振舞うことになります。)。

4.5 典型的な利用方法

ブランチの作り方と **svn merge** にはいくつもの異なったやり方があり、この節では あなたが出くわしそうな一番よくあるパターンについて説明します。

4.5.1 ブランチ全体を別の場所にマージすること

いま、考えてきた例を完結させるため、少し時間が経過したとします。何日か 経過し、たくさんの変更が trunk にもあなたのプライベートなブランチにも 起こったとまします。そしてあなたはプライベートなブランチ上での作業を 終えたとしましょう; 機能追加、またはバグフィックスが完了し、他の 人がその部分を使えるようにするために、あなたのブランチ上の変更点の すべてを trunk にマージしたいとします。

さて、このような状況では、どのようにして **svn merge** を使えば良いのでしょうか? このコマンドは二つのツリーを比較し、その差分を 作業コピーに適用するものであったことを思い出してください。変更点 を受け取るためには、あなたは trunk の作業コピーを手に入れる必要があります。 ここではあなたは (完全に更新された) もともとの作業コピーをまだ持っているか、 `/calc/trunk` の新しい作業コピーをチェックアウトしたもの と仮定します。

しかし、どのツリーとどのツリーを比較すれば良いのでしょうか? ちょっと考えると、 その答えは明らかに思えます: 単に trunk の最新のツリーと、あなたのブランチの最新の ツリーです。しかし、気をつけてください — この仮定は間違いです。そしてこの間違いに、たいていの初心者はやられてしまいます! **svn merge** は **svn diff** のように働くので 最後のトランクとブランチのツリーの比較は単にあなたが自分の ツリーに対して行った変更点のみを示すものではない のがわかります。 そのような比較は、非常にたくさんの変更を表示するでしょう: それは、あなたのブランチに対する追加点だけを表示するのではなく、 あなたのブランチでは決して起こらなかった、trunk 上の変更点の取り消しも表示してしまうことでしょう。

あなたのブランチ上に起きた変更のみをあらわすには、あなたのブランチの 初期状態と、最終的な状態を比較する必要があります。 **svn log** コマンドをあなたのブランチ上で使えば、 そのブランチはリビジョン 341 で作られたことがわかります。そして、ブランチ の最終的な状態は、単に、`HEAD` リビジョンを指定すればわかります。 これはブランチディレクトリのリビジョン 341 と `HEAD` を比較しその違いを トランクの作業コピーに適用したいと考えていることを意味します。

ティップ

ブランチが作成されたリビジョンを見つけるうまい方法は (ブランチの「ベース」リビジョンのことですが)**svn log** で`--stop-on-copy` オプションを利用 することです。log サブコマンドは通常ブランチに対するすべての変更を表示 し、それはブランチが作成されたコピーよりも前にさかのぼります。このた め通常トランクの履歴も表示されてしまいます。 `--stop-on-copy` は、**svn log** がターゲットのコピーあるいは名称変更の個所を見つけると直ちにログの出力を中止します。

それで現在の例で言うと、

```
$ svn log --verbose --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)

$
```

期待したとおり、このコマンドによって表示される最後のリビ ジョンはコピーによって `my-calc-branch` が作成された リビジョンになります。

結局、最終的なマージ処理は以下のようになります:

```
$ 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.
```

ここでもトランクにマージされた変更範囲についてコミットログメッセージは 非常に具体的に触れていることに注意してください。このことを常に憶えて おいてください。後になって必要になる非常に重要な情報だからです。

たとえば、独自の機能拡張やバグフィックスなどのために、もう一週間自分の ブランチ上で作業を続けることにしたとしましょう。リポジトリの HEAD リビジョンはいま 480 となり、あなたは自分のプライベートなブランチから トランクに対するマージの用意ができています。しかし項 4.4.3 で議論したように既に以前マージした 変更を再びマージしたくはありません; 最後にマージしてからブランチ上に 「新しく起きた」変更だけをマージしたいのです。問題はどうやって 新しい部分を見つけるかです。

最初のステップはトランク上で **svn log** を 実行し最後にブランチからマージしたときのログメッセージを見ます:

```
$ 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.
------------------------------------------------------------------------
…
```

ああ、なるほど。341 と 405 の間のリビジョンに起きたすべてのブランチ上での 変更はリビジョン 406 として既にトランクにマージされているので、それ以降にブランチ上で起きた 変更のみをマージすれば良いことがわかります — つまり、リビジョン 406 から HEAD までです。

```
$ cd calc/trunk
$ svn update
At revision 480.

# 現在の HEAD が 480 であることがわかったので、以下のようにマージすれ
# ばよいことになります

$ 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.
```

これでトランクはブランチに起きた変更の第二波全体を含むことになりました。 この時点でブランチを削除する (これについては後で議論します) ことも、 ブランチ上で引き続き作業し、以降のマージについて上記の手続きを繰り返す ことも できます。

4.5.2 変更の取り消し

svn merge のほかのよくある使い方としては、既にコミット した変更をもとに戻したい場合です。`/calc/trunk` の 作業コピー上で作業中に、`integer.c` を修正したリビジョン 303 は完全に間違いであったことを発見したとしましょう。それはコミットすべき ではありませんでした。作業コピーの変更を「取り消す」のに **svn merge** を使い、その後リポジトリに対してローカルな変更を コミットすることができます。やらなくてはならないことは反対向きの 差分を指定することだけです:

```
$ svn merge -r 303:302 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.
```

リポジトリリビジョンについてのもう一つの考え方は、それを 特定の変更のあつまりと考えることです (いくつかのバージョン管理 システムでは、これを、*changesets* と 呼んでいます)。`-r` スイッチを使って **svn merge** を呼び出すことで、あるチェンジ セットを適用するか、もしくはある範囲のチェンジセット全部を作業

コピーに 適用することができます。私たちの場合だと **svn merge** を使ってチェンジセット#303 を作業コピーに反対向きに 適用します。

このような変更の取り消しは、普通の **svn merge** の操作にすぎないので、作業コピーが望む状態になったかどうかは **svn status** と **svn diff** を 使うことができ、その後 **svn commit** でリポジトリに 最終的なバージョンを送ることができるのだ、ということを押さえておいて ください。コミット後はこの特別なチェンジセットはもはや HEAD リビジョンには反映されません。

こう思うかも知れません: とすると、それは「取り消し」じゃない じゃないか。変更はまだリビジョン 303 に存在しているのでは、と。 もし誰かが `calc` プロジェクトのリビジョン 303 と 349 の間のバージョンをチェックアウトしたとしたら、間違った 変更を受け取るのではないか、違うか、と。

おっしゃる通り。私たちが、変更の「取り消し」について語るとき、 本当は HEAD から取り除くことを言っています。もともとの変更は リポジトリの履歴に依然として残っています。ほとんどの状況では これで十分です。とにかくほとんどの人たちはプロジェクトの HEAD を追いかける ことだけに興味があるからです。しかし、コミットに関するすべての 情報を削除したいという例外的な状況もあるでしょう。(多分、誰かが 極秘のドキュメントをコミットしてしまった、など) これはそんなに やさしいことではありません。Subversion は意図的に決して情報が 失われないように設計されているからです。履歴からのリビジョンの 削除は、連鎖的な影響を与え、すべての後続リビジョンと、多分 すべての作業コピーに混乱を起こします。 [*3]

4.5.3 削除されたアイテムの復活

バージョン管理システムの偉大なところは情報が決して失われないという ところです。ファイルやディレクトリを削除した場合でもそれは HEAD リビジョン から消えただけであり、以前のリビジョン中には依然として存在し続けます。 新規ユーザからの一番よくある質問の一つは: 「どうやって古いファイルや ディレクトリを戻せば良いのですか?」 というものです。

最初のステップはあなたが復活させようとしているものは正確には何であるかをはっきりさせることです。うまいたとえがあります: リポジトリ中のそれぞれのオブジェクトは 一種の二次元座標系の中に存在していると考えることができます。第一の軸 は特定のリビジョンツリーで第二の軸はそのツリー中のパスです。するとファイルあるいはディレクトリのそれぞれのバージョンは特定の座標の組で 定義することができます。

Subversion は CVS のような `Attic` ディレクトリを持ちません [*4] ので復活させたいと思う正確な座標ペアを見つける のに **svn log** を使わなくてはなりません。うまいやり方 としては削除されたアイテムがあったディレクトリで **svn log –verbose** を実行することです。`--verbose` オプションはそれぞれの リビジョン中でのすべての変更アイテムのリストを表示します; 必要なことは ファイルやディレクトリをどのリビジョンで削除したかを調べることです。 これはビジュアルにやることもできますし、ログ出力を解析する別のツールを 使うこともできます (**grep** コマンドを通じて、あるいは エディタでのインクリメンタル検索機能を使う形かも知れません)。

```
$ cd parent-dir
$ svn log --verbose
…
------------------------------------------------------------------------
```

[*3] しかしながら、Subversion プロジェクトはいつの日か **svnadmin obliterate** コマンドを 実装する計画があります。これは情報の完全な消去を実行するコマンド です。それまでは回避策として項 5.4.1.3 の方法を利用してください。

[*4] CVS はツリーのバージョン管理ができないので削除されたファイルを 記憶しておくためにリポジトリ用のディレクトリ中に `Attic` 領域を 作ります。

```
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.
…
```

例では削除してしまったファイル `real.c` を探している とします。親ディレクトリのログを見ることでこのファイルはリビジョン 808 で削除されたことを突き止めました。それでこのファイルが存在していた 最後のバージョンはそのリビジョンの直前であることになります。結論: リビジョン 807 から `/calc/trunk/real.c` のパスを 復活させれば良いことになります。

これが面倒な部分です — つまりファイルを見つける作業です。 これで復元したいものが何であるか突き止めました。後は二つの方法が あります。

最初のやり方はリビジョン 808 を 「逆向きに」 適用するために **svn merge** を利用することです。(変更の取り消し の仕方については既に議論しました。 項 4.5.2 を参照してください。) これはローカルな変更として `real.c` をもう一度追加する効果があります。ファイルは追加予告され、コミット後には `HEAD` 上に再び存在するようになります。

しかしこの例は多分最善の方法ではないでしょう。リビジョン 808 の逆向き の適用は `real.c` の追加予告だけではなく、ログメッセージ が示すように、今回必要としない `integer.c` への変更点 も取り消してしまいます。確かにリビジョン 808 を逆向きにマージした後 `integer.c` のローカル変更を **svn revert** することもできますが、この技法はファイルが多くなるとうまくスケールしません。 リビジョン 808 で 90 個のファイルが変更されていたとしたらどうなりますか?

もっと洗練された二番目の方法は **svn merge** は利用せず、 そのかわりに **svn copy** コマンドを使います。正確な リビジョンとパスの 「座標の組」 を指定してリポジトリから自分の作業コピーに 単にコピーするだけです:

```
$ svn copy --revision 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.
```

ステータス表示中のプラス記号はそのアイテムは単に追加予告されただけではなく「履歴と共に」 追加予告されたことを示しています。Subversion はどこからそれが コピーされたかを記憶しています。今後このファイル上に **svn log** を実行するとファイルの復活についてと、リビジョン 807 以前のすべての履歴をたどること

ができます。言いかえるとこの新しい `real.c` は本当に新しいわけではありません; それは 削除されたもとのファイルの直接の子孫になっています。

私たちの例はファイルの復活でしたが、同じ技法が削除されたディレクトリ の復活についても利用できることに注意してください。

4.5.4 ブランチの作り方

バージョン管理システムはソフトウェア開発で一番よく使われるので、ここで 何かの開発チームによって利用される典型的なブランチ化/マージのパターン をちょっと見てみましょう。Subversion をソフトウェア開発に使うのでなけ ればこの節は読み飛ばしてもかまいません。ソフトウェア開発にバージョン 管理システムを使うのが初めてなのであれば、よく読んでください。 ここでのパターンは経験を積んだ多くの開発者によって最良の方法だと考えられて いるからです。このようなやり方は Subversion に限った話ではありません; どのようなバージョン管理システムにでも応用できる考え方です。また同時に 他のシステムのユーザに対しては Subversion ではどんな言葉を使ってこの標 準的なやり方を表現するかを理解する手がかりになるでしょう。

4.5.4.1 リリースブランチ

ほとんどのソフトウェアは典型的な作業サイクルがあります: コーディング、 テスト、リリース、この繰り返しです。このようなやり方には二つの問題が あります。まず開発者は新しい機能を追加し続けなくてはならない一方で 品質保証チームはそのソフトウェアの安定版だと考えられるバージョンを テストするのに時間をついやさなくてはなりません。テスト途中だからといっ て新しい機能追加を中断することはできません。次に開発チームはほとんど の場合、すでにリリースされた古いバージョンのソフトウェアを保守しなくては なりません; もし最新のコードにバグが見つかった場合、すでにリリースしている バージョンにも同じバグが潜んでいる可能性は高く、利用者は次のリリースを 待たずにこのバグを修正して欲しいと望んでいることでしょう。

バージョン管理システムの出番です。典型的なやり方は以下のようなものです:

- 開発者は新規開発部分をトランクにコミットします。 日々の変更点は`/trunk` にコミットされます: 新しい機 バグ修正、その他もろもろです。
- トランクの内容は「リリース」ブランチにコピーされます。 あるチームが、そのソフトウェアがリリースできる状態になったと考えた 時点で (つまり、1.0 のリリースのような場合)、`/trunk` は`/branches/1.0` のような名前でコピーされる ことになります。
- これと並行して、他のチームが作業を続けます。 あるチームがリリースブランチの内容を徹底的なテストを開始する一方で 他のチームは新規開発分 (つまり、バージョン 2.0 に向けた作業) を `/trunk` 上で継続して行います。どちらかの場所で バグが見つかれば、必要に応じてその修正がお互いの間を行き来します。 しかしこの作業もやがては終わります。このブランチはリリース直前の 最終的なテストに向けて「凍結」されます。
- ブランチはタグづけされ、リリースされます。 テストが完了したら`/branches/1.0` は `/tags/1.0.0` にコピーされ、これが参照用のスナップ ショットになります。このタグの内容はパッケージ化され、利用者に対して リリースされます。
- ブランチはその後も保守されます。 バージョン 2.0 に向けた作業が`/trunk` 上で進む一方、 バグ修正個所については`/trunk` から `/branches/1.0` に引き続き反映されます。 十分なバグ修正が反映されたら、管理者は 1.0.1 をリリースする決断をする かも知れません: `/branches/1.0` は `/tags/1.0.1` にコピーされ、このタグはパッケージ 化されてからリリースされます。

このような作業の流れを繰り返すことでソフトウェアは安定していきます: 2.0 の開発が完了したら新しい 2.0 のリリースブランチが作られ、テスト され、タグがつけられ、最終的にリリースされることになります。 何年

かしてリポジトリは「保守対象」の状態になったいくつかの リリースブランチと最終的にリリースされたバージョンを示すタグの集まり になるでしょう。

4.5.4.2 (特定機能の) 開発用ブランチ

開発用ブランチ *(feature branch)* はこの章での 例として中心的な役割を果たしてきたようなタイプのブランチで、その ブランチ上であなたが作業をするのと同時に並行して Sally は `/trunk` 上で作業を継続することができるような ものでした。それは一時的なブランチで、安定している `/trunk` に影響を与えることなく複雑な変更をする ためのものです。リリース用ブランチ (これはずっと保守しつづけなければ ならないかも知れません) とは違って、開発用ブランチは作成されたあと ある程度の期間利用され、変更部分がトランクに反映された後で完全に 削除されてしまいます。利用されるのは、ある決まった期間の中だけです。

プロジェクトの考え方によって、開発用ブランチをいつ作るのが適切である かにはかなりの幅があります。プロジェクトによっては開発用ブランチを 全く使いません: `/trunk` に対するコミットは 全員に許されています。このやり方の長所はその単純さです — 誰も ブランチ化やマージについて理解する必要がありません。欠点はこの方法 だとトランクのソースコードが不安定になったりまったく利用できなく なったりしやすいことです。逆に別のプロジェクトではブランチを極端な 形で使います: どんな変更もトランクに対して直接 コミットすることは認められていません。まったくささいな変更に対しても 短い生存期間をもつブランチを作り、それを注意深く検討し、 トランクに反映させます。それから、そのブランチを削除します。この方法は トランクを常に非常に安定して利用できる状態に置くことができますが それには無視できない処理効率の低下が伴います。

ほとんどのプロジェクトではこの中間のやり方をとります。普通は `/trunk` は常にコンパイル可能な状態であり、 一度フィックスしたバグが元に戻っていないことを保証するためのテスト もクリアした状態にあることを要求します。ある変更をするのに プログラムを不安定にするようなコミットを何度も必要とする場合にだけ開発ブランチが作られます。基本的な方針としては次のようなことを 考えてみることです: もし開発者が孤立した状態で何日も作業した後で 一度に変更点全体をコミットしたとしたら (`/trunk` が 不安定にならないようにするためにそうするのでしょうが)、その変更 内容が正しいかどうかを検討するには大きすぎませんか? もし答えが「イエス」なら、その変更は開発用ブランチでやるべきで しょう。開発者はブランチに対して変更点を少しずつコミットするので 他の人たちはそれぞれの部分について簡単に内容を検証することができ ます。

最後に、開発用ブランチでの作業が進むにつれて、どうやってそれを トランクの内容に「同期」させるのがよいかについて 考えてみます。すでに注意したようにブランチ上で数週間あるいは数ヶ月 ものあいだ作業しつづけるのには大きなリスクが伴います; トランクへの 変更はその間次々と発生し、ついには二つの開発ラインはあまりにもかけ離れて しまい、ブランチの変更内容をトランクにマージによって戻すのは全く 非現実的な話になってしまうかも知れないのです。

この状況を避けるためにはトランクの内容を定期的にブランチにマージする ことです。次のようなルールを決めておきましょう: 一週間に一度、 先週トランク上におきた変更をブランチにマージすること。これは注意して 実行する必要があります; マージは手作業で実行し、繰り返してマージする のを避ける必要があります (これについては 項 4.4.3.1 で説明しました)。ログメッセージ を書く時には注意して、どの範囲のリビジョンが既にマージされているか を正確に控えておきましょう (これは項 4.5.1 でやってみせました)。大変な作業に思えるかも知れませんが、実際には 非常に簡単なことです。

あるところまで作業が進んだら、開発用ブランチの内容をトランクに 「同期」させるためのマージの準備が整います。これには まず、最新のトランクの変更部分をブランチに取り込む最後のマージ 処理を実行することで始めます。この処理の後では、ブランチ上の最後のリビ ジョンとトランク上の最後のリビジョンは、ブランチでの変更部分を のぞけば、完全に同じ状態になります。このような特定の状況下では ブランチとトランクの内容を比較することによってマージすることが できるはずです:

```
$ 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
…
```

トランクの HEAD リビジョンとブランチの HEAD リビジョンを比較することで、ブランチにだけ 加えた修正点を含む差分を作ることができます; 両方の開発ラインとも トランクに起きた修正についてはすでに取り込んでいるからです。

このような作業パターンは、自分のブランチに対して 毎週トランクを同期させる処理は、作業コピーに対して **svn update** を実行するのとよく似ていて、最後のマージ処理に ついては作業コピーから **svn commit** を実行するのに よく似ていると考えることができます。結局、作業コピーと、ちょっと作った プライベートなブランチと、他に何が違うと言うのでしょう? 作業コピー とは、一度に一つの変更しか保存できないような単なるブランチにすぎ ません。

4.6 作業コピーの切り替え

svn switch コマンドは存在している作業コピーを 別のブランチに変換します。このコマンドはブランチで作業するのに 常に必要というわけではありませんが、ユーザに対して便利なショートカット を用意します。前の例で、プライベートなブランチを作ったあと、その新しい リポジトリディレクトリの作業コピーをチェックアウトしました。そうする かわりに、単に/calc/trunk の作業コピーを新しい ブランチの場所のコピーに変更することができます:

```
$ 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
```

ブランチに「スイッチ」したあとでは、作業コピーの内容はそのディレクトリ を新しくチェックアウトした場合とまったく同じものになります。そして 普通このコマンドを使うほうがより効率的です。というのは、たいてい ブランチはほんの少し内容が違うだけです。サーバはそのブランチディレクトリ を反映させるために作業コピーにしなくてはならない最小限の変更だけを 送信すれば済むのです。

svn switch は`--revision` (`-r`) オプションをとることもできるので、常に作業コピー をブランチの「最新状態」にに移す必要があるわけではありません。

もちろん、ほとんどのプロジェクトは `calc` よりは もっと複雑で、複数のサブディレクトリを含んでいます。Subversion ユーザは ブランチを利用するときにはよく、特定のやり方をします。:

1. プロジェクトの「幹 (trunk)」全体を新しいブランチディレクトリ にコピーする。
2. 幹 (trunk) の作業コピーの一部のみを ブランチにミラーする。

言い換えると、ユーザが特定のサブディレクトリ上でだけブランチの作業が起きる ことを知っている場合には **svn switch** を使ってブランチに そのサブディレクトリのみを移動します。(あるいは、たった一つの作業ファイル だけをブランチに switch することさえあります!) その方法では、作業コピー のほとんどすべての更新を普通の「幹 (trunk)」から従来どおり受け取ることが できますが、切り替えた部分だけは変更されることなく残ります (もしブランチ に対して誰かが変更点をコミットしさえしなければ)。この機能は 「混合作業コピー」という概念にまったく新しい次元を付け加える ことになります — 作業コピーは作業リビジョンの混合を含むことができる だけではなく、リポジトリ位置の混合も含むことができます。

もし作業コピーが異なるリポジトリ位置からのスイッチされたサブツリー をいくつか含むなら、それは普通に機能し続けます。更新すると、 それぞれのサブツリーのパッチを適切に受け取るでしょう。コミットするとローカル修正は一つの不可分の変更をリポジトリに適用するでしょう。

リポジトリ位置の混合を作業コピーに反映させることはできますが、 このようなリポジトリ位置はすべて同じ リポジトリ の中になくてはなりません。Subversion のリポジトリはまだお互いに通信 することはできません。これは Subversion1.0 以降で計画されている機能です。 [*5] スイッチと更新

svn switch と **svn update** の 出力が同じなのに気がつきましたか? `switch` コマンドは実際には update コマンドのスーパーセットになっています。

svn update を実行するとき、それはリポジトリに対して 二つのツリーを比較するように要求します。リポジトリはその比較を実行 し、クライアントに差分の内容を送信します。 **svn switch** と **svn update** の 唯一の違いは `update` コマンドは常に二つの同じパスを比較するという ことです。

つまり、もし作業コピーが `/calc/trunk` のコピーなら **svn update** は自動的に`/calc/trunk` の作業コピーを `HEAD` リビジョンの`/calc/trunk` と比較 します。もし作業コピーをブランチに移すと、 **svn switch** は`/calc/trunk` の作業コピーを何か `HEAD` リビジョンの他の ブランチディレクトリと比較します。

言い換えると、更新は時間の方向に作業コピーを動かします。 switch は作業コピーを時間と 空間の 両方の方向に動かします。

svn switch は本質的には **svn update** の変種なので、同じ動作を共有します。作業コピー中のどのようなローカルの 変更もリポジトリから新しいデータが届くときに保存されます。これで あらゆる利口な小技がきくようになります。

たとえば`/calc/trunk` の作業コピーがありそれにいくつか 変更を加えたとします。それから突然、本当

[*5] しかし、サーバ上の URL が変更されたが、既存の作業コピー を捨てたくない場合には、`--relocate` スイッチ付きで **svn switch** を使うことはできます。 より詳しい情報と例については第 9 章の **svn switch** の章を見てください。

はブランチにやる変更だったことに 気づきます。問題ありません。作業コピーを **svn switch** でブランチにスイッチしても、ローカルの変更はそのまま残ります。で、 それをブランチに対してテストし、コミットすることができます。

4.7 タグ

別のバージョン管理の概念に、タグがあります。 タグはある時点でのプロジェクトの「スナップショット」です。Subversion ではこのアイディアは既にさまざまな場所にあるように見えます。それぞれ のリポジトリリビジョンはまさにそれです — つまり、それはコミット直後の ファイルシステムのスナップショットです。

しかし、人はしばしばタグに対して人間になじみのある名前を付けたいと 思うものです。たとえば、「release-1.0」のような。また、ファイルシステム のもっと小さなサブディレクトリのスナップショットがほしいこともあります。 結局、あるソフトの一部の release-1.0 がリビジョン 4822 の特定のサブディレクトリ であることを思い出すのは簡単ではありません。

4.7.1 簡単なタグの作成

もう一度、**svn copy** の助けを借ります。 もし `HEAD` リビジョンの`/calc/trunk` のスナップ ショットを作りたいときには、そのコピーをとればいいのでした:

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

Committed revision 351.
```

この例では`/calc/tags` ディレクトリが既に 存在しているものとしています。(もしそうでないなら、

名前

svn mkdir を見てください)。 コピー完了後、新しい `release-1.0` ディレクトリは、あなたがコピーした時点の `HEAD` リビジョンにおいてプロジェクトがどう見えていたかをスナップ ショットとして永遠に残すものです。 もちろん、どのリビジョンをコピーするかについてもっと正確でありたいと 思うかも知れません。他の人があなたが見ていないときにプロジェクトに 対して変更点をコミットしていたかも知れませんから。もしあなたが `/calc/trunk` のリビジョン 350 が自分のほしいスナップ ショットだと知っていれば **svn copy** コマンドに `-r 350` を指定することができます。

でもちょっと待ってください: このタグ作成の手続きはブランチを作る ために使ってきた手続きと同じじゃないの? 実はその通りです。 Subversion ではタグとブランチには違いはありません。両方とも コピーで作られた普通のディレクトリです。ちょうどブランチのように コピーされたディレクトリが「タグ」であるといわれるのは、単に 人間がそうやって扱うことに決めたから、ただ それだけです。そのディレクトリに誰もコミットしない限り、それは 永遠にスナップショットとして残ります。もし誰かがそれにコミット し始めると、それはブランチになります。

もしリポジトリを管理しているなら、タグを管理するには二通りの 方法があります。最初のアプローチは、「ユーザ任せ」です。 プロジェクトポリシーとして、あなたのタグを置く場所を決め、 すべてのユーザにその

ディレクトリをコピーするときにはどうやって扱うか を知らせます。(つまり、みんながそこにコミットしないように約束します) 二番目のやり方はもっとガチガチです。Subversion が提供するアクセス 制御スクリプトのどれかを使って、タグ領域には新しいコピーを 作ることだけができて、それ以外の操作を禁止します。 (第 6 章を参照してください。) ガチガチ方式は、普通は不要です。もしユーザが間違ってタグディレクトリ に自分の変更をコミットしてしまったら、前の章で説明した方法で その変更を取り消せばいいのですから。結局、Subversion はバージョン 管理システムなのです。

4.7.2 複雑なタグの作成

ときどき、一つのリビジョンの一つのディレクトリよりも もっと複雑な「スナップショット」がほしいことがあります。

たとえば、プロジェクトが私たちの `calc` よりも もっと大きいとします。たくさんのサブディレクトリともっとたくさん のファイルがあるとします。仕事の過程で、特定の機能とバグ修正を含んだ 作業コピーが必要になったと判断します。特定のリビジョンの 以前のファイルとディレクトリを選んで (**svn update -r** liberally を使って), これを作ることもできますし、 特定のブランチにファイルとディレクトリをスイッチすることによっても できます。 (**svn switch** を使う) これをやると、あなたと作業コピーは別々のリビジョンからなる別々の リポジトリ位置のつぎはぎになります。しかしテスト後、自分がまさに必要と している組み合わせであることがわかりました。

さあスナップショットをとります。一つの URL を作業していない別の 場所にコピーします。この場合、やりたいことは特定の作業コピー状態 で、それをリポジトリに格納したいのです。幸運なことに **svn copy** は実際には四種類の異なる使い方が あります。(第 9 章を読んでください) その中には作業コピーツリー をリポジトリにコピーする、というのもあります:

```
$ ls
my-working-copy/

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

Committed revision 352.
```

これでリポジトリに新しいディレクトリができました。 `/calc/tags/mytag` です。これはあなたの作業コピー の正確なスナップショットです — 混合リビジョン、URL そして すべてです。

別のユーザはこの機能の面白い使い方を見つけました。 ときどき、自分の作業コピーにローカルな修正をした ブランチがあるが、それを他のメンバーに見せたいというような 状況です。**svn diff** を使ってパッチファイル (それはツリーの変更、シンボリックリンクの変更、あるいは属性の変更を取得できません) を送るかわりに、**svn copy** を使って、作業コピーをリポジトリの プライベートな領域に「アップロード」します。他のメンバーは 作業コピーを新しくチェックアウトするか、**svn merge** を使って変更点のみを受け取ることができます。

4.8 ブランチの管理

ここまでで、Subversion は非常に柔軟なシステムであることがおわかり いただけたかと思います。ディレクトリのコピーという同じ基本的な 仕組みの上にブランチもタグも実装しており、ブランチもタグも普通のファイルシステムの空間の中にあるので、多くの人々は Subversion の 仕組みにびっくりします。それは柔軟

すぎるくらい です。この節では時間経過と共にどのようにデータを配置し管理するのが 良いかについて、少し説明します。

4.8.1 リポジトリのレイアウト

リポジトリの編成にはある程度、標準化された、おすすめの方法が あります。ほとんどの人々は `trunk` ディレクトリ に開発の「主系」、ブランチのコピーがある `branches` ディレクトリ、そしてタグのコピーがある `tags` ディレクトリ を入れます。リポジトリがただ一つのプロジェクトを含む場合には しばしば、この三つのディレクトリをリポジトリ最上位に作ります:

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

リポジトリが複数プロジェクトを含む場合は、プロジェクトごとに レイアウトをインデックス化します (「プロジェクトルート」についての詳細は 項 5.5.1 を読んでください):

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

もちろん、この標準的なレイアウトを無視してもかまいません。 あなたと、チームが最も作業しやすいように、このレイアウトは どのように変化させてもかまいません。どれを選んでもそれは 永久に固定されたものではありません。いつでもリポジトリを 再編成することができます。ブランチとタグは普通のディレクトリにすぎないので **svn move** コマンドを使えば 好きなように移動、名称変更ができます。あるレイアウトから別の レイアウトへの切り替えは単にサーバ側での何回かの移動の話になります。 もしリポジトリ中の編成に何か気に入らないところがあるなら、ディレクトリ に関連した小技を使ってください。

しかし、ディレクトリの移動は簡単ではありますが、ユーザのこともよく 考える必要があります。この変更は既にある各自の作業コピーの場所 を再配置します。もしユーザが特定のリポジトリのディレクトリの作業 コピーを持っている場合、あなたの **svn move** 操作 は最新リビジョンのパスを削除してしまうかも知れません。ユーザが 次に **svn update** を実行すると、作業コピーは すでに存在しないパスを示しているとされ、新しい場所に移動する ために **svn switch** の実行を強要されてしまう でしょう。

4.8.2 データの寿命

Subversion モデルの別の良い機能としては他のバージョン 管理されたアイテムと同様、ブランチとタグは有限の寿命しか持たない ようにもできることです。たとえば `calc` プロジェクト の個人的なブランチ上ですべての作業が完了したとします。すべての 変更を、`/calc/trunk` にマージしたあとでは その個人的なブランチのディレクトリがまったく不要になります:

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

Committed revision 375.
```

これでブランチはなくなってしまいました。もちろん本当に削除された わけではありません: ディレクトリは単に HEAD リビジョンからなくなった だけで、誰もわずらわせることはなくなりました。 前のバージョンを調べるために **svn checkout**、 **svn switch**、あるいは **svn list** を使えば依然として古いブランチを見ることができます。

削除したディレクトリを閲覧するだけでは不十分な場合は、いつでも 戻すこともできます。データの復活は Subversion ではお手のものです。 HEAD に戻したい削除ディレクトリ (またはファイル) がある場合は、単に **svn copy -r** を使って古いリビジョンから コピーしてください:

```
$ 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.
```

私たちの例では個人的なブランチは比較的短い生存時間を持ちます。バグを 直したり新しい機能を追加するのに利用したからです。作業が終われば、 ブランチの寿命もそこで終わりです。しかし、開発内容によっては非常に長い時間にわたって二つの「主要な」ブランチが並行して生きつづけることも あります。たとえば、安定版の calc プロジェクトを 公にリリースするときがやってきて、そのソフトのバグをとるのには 何ヶ月かかかるとします。リリースバージョンに新しい機能を追加させたくは ありませんが、すべてのメンバーに開発を中止するように言いたくもありま せん。そこでかわりに、あなたはそれほど大きな修正は発生しない 「安定版」のブランチを作ります:

```
$ 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.
```

これで開発者は最先端の機能 (あるいは実験的な機能) を /calc/trunk に追加し続けることが でき、バグフィックスだけを/calc/branches/stable-1.0 にコミットするようなポリシーで進めることができます。 つまり、メンバーが trunk 上で作業し続けるときに、バグフィックスに ついては安定版ブランチ上に持っていくことができます。安定版ブランチ が出荷されたあとでも、そのブランチを長い時間かけて保守し続けるでしょう — つまり、顧客に対してそのリリースをサポートし続ける 限り。

4.9 まとめ

この章では、いろいろな基本に触れました。タグとブランチの概念を議論し、 Subversion が **svn copy** でディレクトリをコピーする ことによってこれらの概念を実装していることを説明しました。 **svn merge** であるブランチから別のブランチに変更点 をコピーしたり、間違った変更を戻したりする方法をお見せしました。**svn switch** を使って、混合状態の作業コピーを 作ってみせました。そして、どのようにしてリポジトリ内のブランチの 編成と寿命を管理するかについて話しました。

Subversion について、このことだけは憶えておいてください: ブランチやタグを作る処理はとても軽いのです。好きなだけ使ってください!

第 5 章

リポジトリの管理

5.1

Subversion のリポジトリは複数のプロジェクトのためのバージョン管理 されたデータを格納する中心的な場所です。こんなわけで、リポジトリは 管理する人間にとってはたまらない魅力のある場所になるかも知れません。 リポジトリは一般的にはそれほど複雑な管理が必要なものではありません が、正しく設定し、潜在的な問題を避け、実際に起こる問題を安全に解決する ためにはどうすれば良いかを理解することは重要です。

この章では、Subversion のリポジトリをどうやって作成し設定するかについて 議論します。リポジトリ管理についても述べますが、 これには **svnlook** と **svnadmin** (この二つは Subversion が提供するツールです) の利用も含まれています。 よくある質問と間違いをとりあげ、リポジトリ中でどのようにデータを 配置するのか良いかについてアドバイスします。

もし、Subversion のリポジトリのバージョン管理下にあるデータに ユーザとしてアクセスするだけなら、(つまり、Subversion のクライアント としてだけ利用するなら) この章は読み飛ばすことができます。しかし、Subversion のリポジトリ管理者か、そうなろうと思っている人は [*1] この章に特別の注意を払ってください。

5.2 リポジトリの基礎

リポジトリ管理についての広範囲な話題に飛び込む前に、リポジト リとはいったい何であるかをもう少し突っ込んで定義しておきましょう。それ はどんな風に見えるのでしょうか? いったいどんなコなんでしょう? 飲み物の好みは? ホット? アイス? 砂糖はいくつ? レモンは? 管理者としては、論理的な見え方 — つまり、リポジトリ内でデータがどのように表現されているか — から物理的な細部に到るまで — つまり Subversion 以外のツールから リポジトリはどう見え、どう振舞うか — の両方について理解している ことが期待されます。以下の節は非常に高レベルの基本的な概念のいくつか について説明します。

5.2.1 トランザクションとリビジョンの理解

概念的に言うと、Subversion のリポジトリはディレクトリツリーの 並びです。それぞれのツリーはある時刻で、リポジトリ中に 管理されたファイルやディレクトリがどのように見えるか、という ことについてのスナップショットです。 このクライアントの操作によって作られるスナップショットを リビジョンといいます。

それぞれのリビジョンはトランザクションツリーとして生まれます。 コミットすると、クライアントは、自分のローカルな変更 (と、クライアント のコミット処理の最初にリポジトリに起きる附加的な変更) を反映 した Subversion のトランザクションを作り、次のスナップショットとして このツリーを格納するようにリポジ

[*1] こう書くと、なんだかとても高尚なことのように思えますが、みんなのデータがある作業コピーの背後で起きている神秘の領域に 興味を持つ人なら、誰でも、という意味です。

トリに命令します。 コミットが成功すれば、トランザクションは新しいリビジョンツリーが できたことを知らせ、新しいリビジョン番号を割り当てます。 コミットが何かの理由で失敗すれば、トランザクションは消されて、クライアント は失敗した旨の通知を受けます。

更新処理も同様に動作します。クライアントは作業コピー の状態を反映した一時的なトランザクションツリーを作ります。リポジトリは そのトランザクションツリーを要求されたリビジョンのツリー (普通は 最新の、あるいは「一番若い」ツリー) と比較し、作業コピーを リビジョンツリーの形に変形するにはどのような変更が必要であるかに ついての情報を戻します。更新が完了した後、その一時的なトランザクションは 削除されます。

トランザクションツリーの利用がリポジトリのバージョン管理された ファイルシステムに普遍的な変更を起こす唯一の方法です。しかし、 トランザクションの生存時間が完全に任意であることを理解するのは 重要です。更新の場合トランザクションはすぐに消滅する一時的なツリー です。コミットの場合は、トランザクションは普遍的なリビジョンに 変わります。(あるいはコミットが失敗したときは削除されますが) エラーやバグがあると、トランザクションはリポジトリの周辺に 取り残されてしまうかも知れません (しかしこれは領域を食うだけで、 何かに悪い影響を与えたりはしませんが)

理論的には、いつの日か、統合された作業環境をサポートする アプリケーションはトランザクションの生存期間をもっと柔軟に 管理することができるようになるかも知れません。 クライアントがリポジトリに対する修正内容の記述を終えたあとでも、 リビジョンになる候補のトランザクションが 静止した状態にとどまるようなシステムを考えることもできます。 これはそれぞれの新しいコミットを別の人、たとえば管理者やエンジニアの QA チームによって再検討することを可能にし、そのトランザクション を本当のリビジョンにしたり、取り下げたりすることができるようになるでしょう。

5.2.2 バージョン化されない属性

Subversion リポジトリでのトランザクションとリビジョンは 付随した属性を持つことができます。そのような属性は 一般的なキー・値のマッピングで、関連したツリーについての 情報を格納するのに一般的に利用されます。属性の名前と値はリポジトリの ファイルシステム中に、残りのツリーデータと一緒に格納されます。

リビジョンとトランザクションの属性はファイルやディレクトリ にそれほど強く結びついていないツリーの情報を記憶して おくのに便利です — 作業コピーによって管理できないような 情報です。たとえば新しいコミットトランザクションが リポジトリに作られると Subversion はそのトランザクションに `svn:date` という名前の属性を追加します — トランザクションが作られた時刻を示すタイムスタンプです。 コミットが完了し、トランザクションが普遍的なリビジョンとなる 時点で、ツリーにはリビジョン作成者のユーザ名称 (`svn:author`) とリビジョンに付けられたログメッセージ (`svn:log`) の 属性が追加されます。

リビジョンとトランザクションの属性は バージョン化されない属性です — 修正されると、それ以前の値は永久に失われてしまいます。 同様にリビジョンツリー自身は不変ですが、ツリーに付けられた 属性はそうではありません。いつでもリビジョン属性を追加、削除、修正 することができます。新しいリビジョンをコミットしたあとで、 間違った情報だったり、ログメッセージにスペルミスがあったり したことがわかったときには、単に `svn:log` 属性の値を正しいログメッセージで置き換えてやるだけです。

5.2.3 リポジトリの保存形式

Subversion 1.1 からは、Subversion リポジトリに二つの保存形 式が選べます。一つはすべてのデータを Berkeley DB データベースに保存 する方法です; もう一つは、独自の形式で構成した通常のフラットファイルの 形にデータを保存する方法です。Subversion 開発者はリポジトリを、 「(バージョン化された) ファイルシ

ステム」 という名前でよく言い表すので、 この習慣に合うように後者のリポジトリを *FSFS* [*2] と呼びますが — それは最初から OS がもっているファイルシステムを使っ てバージョン化されたファイルシステムを作る方法です。

リポジトリを作成する時には、管理者は Berkeley DB を使うか、FSFS を使うかを決めなくてはなりません。両方とも、利点と欠点がありますが、そ れについては後で少し触れます。どちらか一方がもう一方よりも「公式 のもの」であるということはありませんし、リポジトリへのアクセスは これらの実装の詳細とは分離されています。プログラムはどうやって保存して いるデータにアクセスするかを知ることはありません; リポジトリ API 全体 を通じて、抽象化されたリビジョンとトランザクションツリーが見えるだけで す。

表 5.1 に Berkeley DB と FSFS リポジトリの比較表があります。 詳細についてはは次の節を見てください。

表 5.1 Repository 保存形式の比較

機能	Berkeley DB	FSFS
リポジトリの壊れやすさ	非常に壊れやすい; リポジトリが壊れたりパーミッショ ンの問題が起こった場合にはデータベースは「中途半 端な」状態になり、ジャーナル復帰処理が必要になります。	それほどでもない
リードオンリーでマウントできるか	いいえ	はい
プラットフォームに独立した保存形式か	いいえ	はい
ネットワークファイルシステムでも使えるか	いいえ	はい
リポジトリサイズ	わずかに大きい	わずかに小さい
スケール性: リビジョンツリーの数が増えるとどうな るか	データベースなので問題なし	OS のファイルシステムが古い場合、一つのディ レクトリ中に数千のエントリがあるとうまく動かなくなるこ とがある。
スケール性: たくさんのファイルのあるディレクトリ	遅い	速い
スピード: 最新コードのチェックアウト	速い	遅い
スピード: 大きなコミット	遅いが、処理の負荷はコミット全体に分散する	速いが、最終処理はクライアントのタイムアウトにつ ながるかも知れない
グループパーミッション制御	umask の問題に敏感; ひとつのユーザによってアクセ スされるのが一番よい。	umask の問題を回避できる
コードは枯れているか	2001 年から使われている	2004 年から使われている

[*2] Jack Repenning が何も文句を言わないのなら「fuzz-fuzz」 と発音することになっています。

5.2.3.1 Berkeley DB

Subversion の最初の 設計段階で、開発者はさまざまな理由で Berkeley DB を利用することに決めました。その理由にはそのオープンソースライセンス、トランザクションのサポート、 信頼性、パフォーマンス、API の公開、スレッドの安全性、カーソルのサポート などが含まれていました。

Berkeley DB は本当のトランザクション機能をサポートしています — おそらく 上であげた理由の中で最も強力な機能です。Subversion リポジトリにアクセスする 複数のプロセスはそれぞれ他のデータを間違って破壊することを心配する必要はありません。 トランザクションシステムによって提供されている分離機能はどんな操作においても Subversion リポジトリのコードにデータベースを静的に見せることができるように するものです — 他のプロセスによってときどき変更を受けているように見えるのを 防ぐものです — そしてそのような静的な見え方に基づいて、何を実行するか を決めることができるのです。もしその決定が他のプロセスがやったことと衝突 した場合、操作全体は、それがまったく実行されなかったかのようにロールバック され、Subversion はもう一度、新しく更新された (そしてやはりまた静的に見える ような状態での) データベースに対してその処理を再実行することができます。

Berkeley DB のほかのすばらしい機能はホットバックアップ — 「オフライン」にせずにデータベース環境をバックアップできる 能力です。リポジトリのバックアップ方法については項 5.4.6 で議論しますが、オフラインに せずにリポジトリの完全なコピーをとることができる利点は明白でしょう。

Berkeley DB はまた非常に信頼性の高いデータベースシステムです。Subversion は Berkeley DB のログ機能を利用しますが、これはまず最初にこれからやろうと する操作内容をいったんディスク上のログファイルに書き込み、それからその 修正を実際に行うものです。これは何かまずいことが起きた場合にデータベースシステム が直前のチェックポイント — ログファイル中の最後の 問題のない地点 — をバックアップすることと、データが利用可能な 状態に復元されるまでトランザクションを再実行することを保証するものです。 Berkeley DB ログファイルについての詳細は項 5.4.3 を 見てください。

しかしどんなバラにもトゲがあるわけであり、Berkeley DB についてわかっている 制約を記しておく必要があります。まず Berkeley DB 環境は可搬性がありません。 Unix システムで作った Subversion リポジトリを Windows システムに単にコピー して動作することを期待してはいけません。ほとんどの Berkeley DB データベース 形式はプラットフォーム独立ですが、環境中にはそうではない部分もあります。 次に Subversion では Berkeley DB を Windows 95/98 システム上で利用できません — ウィンドウズマシン上でリポジトリを管理しなくてはならない のであれば Windows 2000 か Windows XP 上に構築してください。また Berkeley DB リポジトリをネットワーク共有上には決して置かないでください。Berkeley DB は 仕様の一部に合致するようなネットワーク共有上での正しい動作を保証していますが、 現在実際に利用されているネットワーク共有方式で、そのすべての仕様を満たすよう なものは知られていません。

最後に、Berkeley DB は Subversion に直接リンクされたライブラリな ので典型的なリレーショナルデータベースよりも割り込みに関して敏感です。 例えばほとんどの SQL システムでは、テーブルに対するアクセス全体を取り持つサー バプロセスがあります。何かの理由でデータベースにアクセスするプログラム に異常があった場合でもデータベースデーモンは接続が中断したことを検知し て問題のある中間的な状態をきれいにします。またデータベースデーモンはテー ブルにアクセスする唯一のプロセスなのでアプリケーションはパーミッション の衝突に関して心配する必要はありません。このような性質は Berkeley DB にはありません。Subversion (と Subversion ライブラリを使うプログラム) はデータベーステーブルに直接アクセスしますが、これはプログラムで異常が あると、データベースが中間的な矛盾のある状態、アクセスできない状態のま ま残ってしまうことを意味します。このようなことがおこると、管理者は Berkeley DB に問い合わせてチェックポイントを回復する必要がありますが、 これは少し面倒な作業です。リポジトリが 「中途半端な」状態に なるのはプログラムの異常のほかにも、データベースファイルに対して与えら れたオーナーやパーミッションに関係

することもあります。このように、 Berkeley DB は非常に速くスケール性にも富んでいますが、一つのサーバプロ セスを一つのユーザで実行する — たとえば Apache's **httpd** や **svnserve** (第 6 章 を見てください。) — のが最善の利用 方法であり、`file:///` や `svn+ssh://` のような URL を使ってたくさん の異なるユーザがアクセスするのは避けたほうがよいでしょう。複数ユーザか ら直接 Berkeley DB をアクセスする場合には、かならず項 6.6 を読むようにしてください。

5.2.3.2 FSFS

2004 年の半ばから、第二のリポジトリ保存形式が使えるようになりまし た: これはデータベースをまったく利用しないものです。FSFS リポジトリは リビジョンツリーを単一のファイルに保存し、すべてのリポジトリリビジョン は単一のサブディレクトリの下の複数ファイルになります。トランザクション は分離されたサブディレクトリに作られます。トランザクションが完了すると 単一のトランザクションファイルが作られ、それがリビジョンディレクトリに 移動されます。このためコミットの不分割性が保証されます。そして、リビジョンファイルは永続的なものであり、それ以上変更されないのでリポジトリは Berkeley DB リポジトリのように「ホット」バックアップするこ とができます。

リビジョンファイルの形式は、そのリビジョンのディレクトリ構造、ファ イル内容、そして他のリビジョンツリーのファイルに対する差分を表現したも のです。Berkeley DB データベースとは違いこの保存形式は異なるオペレーティ ングシステム間でもそのまま利用することができ、CPU のアーキテクチャには 依存しません。ジャーナリングのしくみや共有メモリーを使っていないので、 リポジトリはネットワークファイルシステムごしに安全にアクセスすることが でき、リードオンリーな環境を作ることもできます。またデータベース保存形式特 有のオーバーヘッドがないので、リポジトリの大きさは比較的小さくなります。

FSFS はパフォーマンスの特性にも独自の性質があります。非常にたく さんのファイルのあるディレクトリをコミットすると、FSFS は O(N) アルゴ リズムを使ってエントリーを追加しますが、Berkeley DB は O(N^2) のアルゴ リズムを使ってディレクトリ全体を書き換えます。いっぽう FSFS は以前の バージョンと、今回の最新バージョンのファイルの差分を書き込みます。 これは最新ツリーをチェックアウトする場合 Berkeley DB の HEAD リビジョンに保管されている完全な内容にアクセスするよりも少し遅くなる ことを意味しています。FSFS はコミットの最終処理でも相対的に長い遅延が 起こりますが、これは極端な場合には応答を待つ クライアントプログラムをタイムアウトさせてしまうかも知れません。

しかし一番大きな違いは FSFS では何かおかしなことが起こったときで も、「中途半端な」状態にはならないところです。Berkeley DB データベースを使ったプロセスがパーミッションの問題や突然異常終了したよ うな場合だと、データベースは管理者が復帰処理をしない限り利用できない状態に とどまります。もし同じことが FSFS リポジトリに起きても、リポジトリはまっ たく影響を受けません。せいぜいトランザクションデータが見えない場所に取 り残されてしまうだけです。

FSFS に対する唯一の問題は Berkeley DB に比較してそれほど枯れてい ないところです。 Berkeley DB ほどの耐久性テストはされていないので、スピー ドとスケール性についてここで述べた多くの内容は: 妥当な推測に基づくもの です。理屈の上では FSFS は新しい管理者がとりかかる時の敷居を下げて、問 題の影響を受けにくくするはずです。実際にどうかは、いずれ時が答えてくれ るでしょう。

5.3 リポジトリの作成と設定

Subversion リポジトリの作成は非常に簡単な作業です。 Subversion 付属の **svnadmin** ユーティリティー にそれをやるサブコマンドがあります。新しいリポジトリを作るには 単に:

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

これで/path/to/reposディレクトリに 新しいリポジトリが作成されます。この新しいリポジトリはリビジョン 0 で誕生しますが、これは最上位のルート (/) ファイル システムディレクトリに中身が空の状態で存在しているだけです。 初期状態でリビジョン 0 はリビジョン属性を一つ持っていて、 svn:date は、リポジトリが作られた時刻が設定 されています。

Subversion 1.2 では、リポジトリはデフォルトで FSFS のバッ クエンドになります。(項 5.2.3 を見てください)。バックエンドは--fs-typeの引数で 明示的に指定することができます:

```
$ svnadmin create --fs-type fsfs /path/to/repos
$ svnadmin create --fs-type bdb /path/to/other/repos
```

> 警告
>
> 
>
> ネットワーク上で共有された Berkeley DB リポジトリを作らないで ください — NFS, AFS あるいは Windows の SMB のようなリモートファイルシステム 上にリポジトリを置くことはできません。Berkeley DB は利用する ファイルシステムが POSIX のロックの方式に厳密に従っていること、そしてさらに 重要なことは、ファイルをプロセスメモリに直接マップできること、を要求します。 ネットワークファイルシステムでこの性質を持っているものはほとんどありません。 ネットワーク上で共有された場所の上で Berkeley DB を利用した結果については予測できません — すぐに正体不明のエラーが起きるかも知れませんし、自分のデータベースがわずかに 壊れてしまったことに気づくのに何ヶ月もかかるかも知れません。
>
> もしリポジトリに対して複数のコンピュータがアクセスする必要がある なら、ネットワーク共有上に Berkeley DB リポジトリではなく、FSFS リポジ トリを作ってください。あるいはもっと良い方法として、実際の (Apache か **svnserve** のような) サーバプロセ スを設定し、サーバがアクセスできるようなローカルファイルシステム上にリ ポジトリを格納し、リポジトリがネットワークからも利用できるようにしてく ださい。第 6 章でこのやり方の詳細を説明しています。

svnadmin の引数であるパスは単なるファイルシステム パスであって **svn** クライアントプログラムがリポジトリ を参照するときのような URL ではないことに注意してください。 **svnadmin** も **svnlook** も、 サーバ側のユーティリティーだと考えてください — この二つはリポジトリを調べたり状態を変更するため、リポジトリがある マシン上で利用され、ネットワーク越しに実行することはできません。 Subversion 初心者によくある間違いは、二つのプログラムに、URL を 渡してしまうことです。(あるいは、「local」な URL として file: のように指定してしまうことです。)

svnadmin create コマンド実行後には、ディレクトリ にはピカピカの新しい Subversion リポジトリができます。サブディレクトリ には実際には何ができたかをちょっと見てみましょう。

```
$ ls repos
```

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

README.txt ファイルと format ファイル以外は、リポジトリディレクトリは サブディレクトリの集まりです。Subversion の一般的な設計思想と同様 モジュール化に非常に配慮されています。階層化した編成は混沌とした 状態よりも望ましいものです。新しいリポジトリディレクトリについて 簡単に説明しておきます:

conf リポジトリ設定ファイルのあるディレクトリです。

dav Apache と、内部データ管理用 mod_dav_svn の ためのディレクトリです。

db すべてのバージョン化されたデータが可能されています。こ のディレクトリは Berkeley DB 環境 (DB テーブルとその他必要な全体) か、 リビジョンファイルを含む FSFS 環境になります。

format 一つの整数値が書いてあるファイルで、この整数はリポジトリレイアウトの バージョン番号になります。

hooks フックスクリプトテンプレート全体が格納されたディレクトリです (また、インストールされたフックスクリプト自身も)。

locks Subversion リポジトリのロックされたデータのためのディレクトリで リポジトリにアクセスしている人を記録するのに使われます。

README.txt Subversion リポジトリを見る人のための情報が書かれている だけのファイルです。

一般的に、「手で」リポジトリをいじるべきではありません。 **svnadmin** ツールはリポジトリに対するどのような 変更にも十分対応できますし、サードパーティーのツール (たとえば Berkeley DB ツールスイート) でリポジトリの関連した部分を調整する ことができます。いくつかの例外もあるので、それについては後で触れます。

5.3.1 フックスクリプト

hook は、新しいリビジョンの生成やバージョン化されて いない属性の修正といったリポジトリに対するイベントをきっかけに実行される プログラムです。フックのそれぞれは、どんなイベントが起こったか、 何を対象にして操作をしたのか、そのイベントを起こした人のユーザ名 などの情報を扱うことができます。フックの出力や戻り値によって フックプログラムは処理を続けたり終了したり、いくつかの方法で中断したりします。

hooks サブディレクトリには、デフォルトでは さまざまなリポジトリフックのテンプレートがあります。

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

Subversion が実装しているフックごとに一つのテンプレートがあり、 テンプレートスクリプトの内容を見ればどんなトリガーを実行し、どんなデータがそのスクリプトに渡されるかがわかります。 またこれらたくさんのテンプレートはスクリプトを書こうとする人の例 になっていて、他の Subversion 付属のプログラムと協調して、よく出くわす 作業を実行します。 動作するフックをインストールするには、何かの実行ファイルかスクリプト を `repos/hooks` ディレクトリに置くだけで良く、そのフックの名前で実行されます。(**start-commit** とか **post-commit** とかいう感じです。)

Unix では、これは正確にフックの名前を持つスクリプトやプログラムを 置いてやる必要があるという意味です (シェルスクリプトでもいいし、 Python, コンパイルされた C のプログラム、などなどです。) もちろんテンプレートファイルはそういう情報を与えるためだけに あるわけではありません — Unix で一番簡単にフックをインストール するにはテンプレートファイルを `.tmpl` の拡張子を とった新しいファイルにコピーして、内容をカスタマイズし、スクリプトに 実行権限を与えるだけです。Windows では、ファイルが実行できるかどうか は拡張子によって決まるので、ベース名がフックの名前で、拡張子が Windows で実行形式として認識される拡張子のどれかにしてやれば OK です。 たとえば、プログラムなら `.exe` か `.com` ですし、バッチファイルなら `.bat` です。

> ティップ
>
> 
>
> セキュリティー上の理由で、Subversion リポジトリはフックスクリプト を空の環境で実行します — つまり $PATH や %PATH% を含め、環境変数は全く設定されない 状態で実行します。このため多くの管理者は手でフックスクリプトを実行 するとうまくいくのに、Subversion によって実行されたときにはうまくいかな いことに困惑します。環境変数を明示的に設定するか、実行するプログラム を絶対パスで参照していることを確認してください。

Subversion リポジトリには 9 種類のフックが実装されています:

`start-commit` これは、コミットトランザクションが作られる前に実行されます。 典型的にはユーザがコミット権限があるかどうかを決定するのに 使われます。リポジトリはこのプログラムに二つの引数を渡します: リポジトリへのパスと、コミットしようとしているユーザ名です。 もしプログラムがゼロ以外の値を返した場合、コミットは ランザクションが作られる前に中止します。フックが標準エラー出力にデータ を書き込むと、それは適切なデータ形式でクライアントに戻されます。

`pre-commit` これは、トランザクションの完結後、実際のコミットの前に実行 されます。典型的には、コミットの内容や場所 (たとえば あなたのサイトでは、すべてのコミットはバグトラッカーの管理番号 を含むようなブランチに対してしなくてはならないとか、 ログメッセージが空であってはいけないというようなポリシーが あるかも知れません) によってコミットを許可しないようにするために使われます。リポジトリはこのプログラムに二つの引数を渡します: リポジトリのパスと、コミットされるはずのトランザクションの名前 です。もしこのプログラムがゼロ以外の値を返した場合、 コミットは中断され、トランザクションは削除されます。 フックが標準エラー出力にデータ を書き込むと、それは適切なデータ形式でクライアントに戻されます。

Subversion の配布パッケージは、アクセス制御を細かく実装するために **pre-commit** から呼び出すことのできるいくつかの アクセス制御スクリプトを含んでいます (Subversion ソースツリーの `tools/hook-`

scripts ディレクトリにあります)。 他の選択子は Apache の httpd モジュールである **mod_authz_svn** を使うもので、個別のディレクトリに対する読みこみ書き込みのアクセス制御 をすることができます (項 6.5.4.2 を見てください)。 Subversion の今後のバージョンでは、ファイルシステムに直接アクセス制御リス ト (ACL) を実装する計画があります。

`post-commit` これはトランザクションがコミットされ、新しいリビジョンが 作られた後に実行されます。ほとんどの人はこのフックを リポジトリのコミットやバックアップに関する連絡メールを 送るのに使います。リポジトリはこのプログラムに二つの引数を 渡します:リポジトリのパスと、今回作られた新しいリビジョン番号 です。このプログラムの終了コードは無視されます。

Subversion 配布パッケージは **mailer.py** と **commit-email.pl** スクリプトを含んでいます。(Subversion ソースツリーの `tools/hook-scripts/` ディレクトリにあります) それは、今回のコミットに付けられた説明をメールするために使う ことができます。このメールの内容は変更されたパスの一覧、コミットに 付けたログメッセージ、コミットした人、コミットの時刻、そして、 コミットの変更部分の GNU の diff スタイルでの表示です。

Subversion が提供するほかの役に立つツールは **hot-backup.py** スクリプトです。(Subversion ソースツリーの `tools/backup/` ディレクトリにあります)。このスクリプトは Subversion リポジトリの オンラインバックアップをとるので、(今後は BerkeleyDB データベース のバックエンドとしてサポートする予定です) リポジトリのアーカイブ化 や緊急リカバリのためのコミットごとのスナップショットを作るのに 使うことができます。

`pre-revprop-change` Subversion のリビジョン属性はバージョン化されていないので、そのような 属性に対する修正は (たとえば、コミットメッセージ属性である `svn:log`) 以前の属性値を永久に上書きしてしまいます。 データはここで失われてしまうので、Subversion はこのフック (そして この相補的な部分である `post-revprop-change`) を使って、必要に応じてリポジトリ管理者がこのような変更記録を残すことが 出来ます。バージョン化されていない属性データを失うことに対するあらかじめ の警告の意味で、Subversion クライアントはこのフックが自分のリポジトリに 実装されているのではない限りリビジョン属性をリモートに変更することは 決してありません。

このフックはリポジトリにそのような変更が発生する直前に実行されます。 リポジトリはこのフックに四つの引数を与えます: リポジトリのパス、修正される属性があるリビジョン、 変更しようとしている、認証の済んだユーザ名、そして属性の名前自身です。

`post-revprop-change` 以前に指摘したように、このフックは `pre-revprop-change` フックのもう片割れです。実際、神経質な人のことを考えて、このスクリプト は `pre-revprop-change` フックが存在しなければ 実行されません。両方のフックが存在する場合、 `post-revprop-change` フックはリビジョン属性が変更 された直後に実行されます。典型的には、変更された属性の新しい値を メールするのに使います。リポジトリは四つの引数をこのフックに渡します: リポジトリへのパス、属性があるリビジョン番号、変更しようと している認証済みのユーザ名称、そして属性の名前自身です。

Subversion 配布パッケージは **propchange-email.pl** スクリプトを含んでいます。(これは、`tools/hook-scripts/` ディレクトリにあります) これは、リビジョン属性の変更についての詳細を メールするために使われます。Email はリビジョンと変更属性の名前、変更した人、そして新しい属性値です。

`pre-lock` このフックは誰かがファイルをロックしようとしたときには常に実行されます。 これはロックを防ぐのにも利用するこどかできますし、誰が特定のパスに対して ロックできるかという複雑なポリ

シーを正確に設定するのにも使えます。 フックが既にロックがかかっていることに気づいた場合にはユーザはそのロック が外れるのを 「待つ」 かどうかを決めることもできます。リポジトリはフックに三つの引数を渡します: リポジトリへのパス、ロックされているパス、そして ロックしようとしているユーザです。プログラムが 0 ではない値で終了すると ロック処理は異常終了し、標準エラー出力へのメッセージはすべてクライアント 側に転送されます。

`post-lock` このロックはパスがロックされた後に実行されます。ロックされたパスはフックの 標準入力に渡されるほか、フックはまた二つの引数も受け取ります: リポジトリ へのパスと、ロックを実行したユーザです。その後フックは email 通知を送ったり 好きな方法で出来事を記録したりすることが自由にできます。ロックはすでに 実行されてしまっているのでフックの出力は無視されます。

`pre-unlock` このフックは誰かがファイルのロックを取り除こうとした時には常に 実行されます。これを使ってどのユーザがどの特定のパスに対して ロック解除できるかを決めるポリシーを作るために利用できます。 ロック解除に関するポシリーを決めることは非常に重要です。 ユーザ A がファイルをロックした場合、B はそのロックを解除できる でしょうか? ロックが一週間以上も前のものだった場合は? これらの ことはフックによって決定され、強制することができます。リポジトリ は三つの引数をフックに送ります: リポジトリのパス、ロック解除される パス、ロックを解除しようとしているユーザ。プログラムが 0 以外の 終了値を返した場合、ロック解除の処理は異常終了し標準エラーへの 出力はすべてクライアント側に転送されます。

`post-unlock` このフックはパスがロック解除された後で実行されます。ロックが解除 されたパスはフックの標準入力に渡され、その他にも二つの引数がフック に渡されます: リポジトリのパスと、ロックを解除したユーザです。 その後フックは email 通知を送ったり、好きな方法で出来事を記録する ことができます。ロックの解除は既に起こってしまっているのでフックの 出力は無視されます。

> **警告**
>
> 
>
> フックスクリプトによってトランザクションを修正しようとしないで ください。このような例としてよくあるのは、コミットの途中に `svn:eol-style` や `svn:mime-type` のような属性を自動的に設定してしまうことです。いっけん良いアイディア に見えますが、問題を起こします。一番の問題はクライアントはフックスクリ プトでされた変更について知ることができないので、クライアントに対して 最新ではなくなったことを伝える方法がないことです。この矛盾した状況 が予測できないような動作の原因になることがあります。
>
> トランザクションで修正するかわりに `pre-commit` フック中のトランザクションでチェックをし、 正しい要件を満たさない場合にはコミットを拒否するのがずっと良い方法です。

Subversion は Subversion リポジトリにアクセスしているプロセス の所有者としてフックを実行しようとします。ほとんどの場合 リポジトリは Apache HTTP サーバと mod_dav_svn 越しにアクセスされる ので、このユーザは Apache を実行しているユーザと同じになります。 フックは、実行しようとするユーザに対する OS レベルでの実行権限 が必要です。また、これはフックが直接的に、間接的にアクセスする ファイルやディレク

トリ (これには Subversion のリポジトリ自身も 含みますが) もそうでなくてはならないことを意味します。 言い換えるとフックを実行する際に、このようなファイル権限に 関係した問題に注意してください。

5.3.2 Berkeley DB の設定

Bkerlekey DB 環境は一つあるいはそれ以上のデータベース、ログファイル、 領域ファイル、設定ファイルを一つにまとめたものです。 Berkeley DB 環境には、一度にいくつのロックが許されるか、 とか、ジャーナルログファイルの一つの大きさについて、固有のデフォルト 値があります。Subversion のファイルシステムコードはこれに追加して Berkeley DB 設定値のデフォルト値を選んであります。しかし、 ときどき特定のリポジトリが特徴的なデータやアクセスパターンを 持っているために別のオプション設定値を持つのが望ましいことがあります。

Sleepycat 系のプログラム (Berkeley DB の手続き) は異なるデータベース が異なる要求を持つことを理解していて、Berkeley DB 環境のいろいろな 設定値を実行時に上書きするような仕組みがあります。Berkeley はそれぞれの環境ディレクトリ中の `DB_CONFIG` という名前のファイルの存在をチェックして、特別の Berkeley 環境 を使う場合には、その中のオプションを調べます。

あなたのリポジトリ用の Berkeley 設定ファイルは `repos/db/DB_CONFIG` の中の、 `db` 環境ディレクトリにあります。 Subversion 自身はこのファイルをリポジトリの残りの部分を作るときに 作ります。ファイルは初期状態でいくつかのデフォルト値と、Berkeley DB のオンラインドキュメントへの場所があるので、どのオプションが 何をするかについて読んでおくことができます。もちろんどのような Berkeley DB オプションも `DB_CONFIG` に追加 することができます。Subversion はこのファイルの内容を読んだり 解釈したりすることはありませんし、その中にオプション値を設定 したりすることもありませんが、Subversion の残りのコードに 予測できないような影響を与える設定変更は避けてください。 同様に `DB_CONFIG` に対する変更はデータベース 環境を復旧するまで効果を持ちません (**svnadmin recover**)。

5.4 リポジトリの保守

Subversion リポジトリの管理はぞっとするような仕事になること もあります。大部分はデータベースバックエンドのもつシステムから引き継い だ複雑さによります。作業をうまくこなすには、とにかくツールについて深く理解する ことです — そのようなツールがいったい何であり、いつ使えば、またど うやって使えばよいのかを知ることです。この節では Subversion によって提供 されるリポジトリ管理用ツールを紹介し、リポジトリの移行、更新、バックアッ プ、クリーンアップのような作業でどうやって使いこなせばよいかを説明しま す。

5.4.1 管理者用ツールキット

Subversion はリポジトリの作成、調査、修正、修復に便利なユー ティリティーをいくつも提供しています。それぞれについてもっと詳し く見てみましょう。その後 Berkeley DB のディストリビューションに含 まれるユーティリティーのいくつかを簡単にためしてみます。 Berkeley DB は Subversion 自身のツールとしては提供していないリポ ジトリデータベースバックエンドに特化した機能を提供しています。

5.4.1.1 svnlook

svnlook は Subversion が提供するツールで リポジトリ中のいろいろなリビジョンやトランザクションを調査 するのに使われます。このプログラムのどの部分もリポジトリを 変更することはありません — これは単なる「読み出し専用」のツールです。 **svnlook** は典型的には、まさにコミットされようとして いる変更を報告したり (**pre-commit** フック)、コミット直後の報告 (**post-commit** フック) のために リポジトリフックによって使

われます。リポジトリ管理者は診断のために このツールを使うこともできます。

svnlook は単純な構文です:

```
$ 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.
…
```

ほとんどの **svnlook** のサブコマンドは リビジョンかトランザクションツリーのどちらかに対して働き、 ツリー自身の情報か、以前のリポジトリのリビジョンとの違いを 表示します。`--revision` と `--transaction` オプションを使ってどの リビジョンまたはトランザクションについて調査するかを指定する ことができます。リビジョン番号は自然数として表示されますが、 トランザクション名称は英数字の文字列だということに注意してください。 ファイルシステムはコミットされていないトランザクションのみを表示 できることを憶えておいてください (新しいリビジョンを作れなかった トランザクション)。ほとんどのリポジトリにはそのようなトランザクション はありません。トランザクションは普通、コミットされる (そうすると 見えなくなります) か、中断後、削除されるからです。

`--revision` も `--transaction` も指定しないと、 **svnlook** は最新の (あるいは「HEAD」) リビジョンをリポジトリの調査対象とします。それで、以下の二つの コマンドは`/path/to/repos` にあるリポジトリで 19 が最新リビジョンである場合はまったく同じ意味になります:

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

サブコマンドに関する唯一の例外は **svnlook youngest** で、これはオプションをとらず、単に、`HEAD` リビジョンの番号を表示します。

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

svnlook の出力は人間にも マシンにも理解できるように設計されています。 `info` サブコマンドを例にします:

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

```
Greek tree.
```

`info` サブコマンドの 出力は、以下のように定義されています:

1. 作業者、改行
2. 日付、改行
3. ログメッセージの長さ、改行
4. ログメッセージ自身、改行

この出力は人間が読むことができます。日付のタイムスタンプ などは、何かバイナリ表現のようなものではなく、テキスト形式になっています。 しかし、これはまたマシンも解析できる形式のものです — ログメッセージは複数行にわたることができ、長さの制限がないので、 **svnlook** はメッセージ自身の前にその長さを 表示します。これで、このコマンドのスクリプトやほかのラッパープログラムは 賢い判断ができるようになります。たとえば、メッセージにどれだけの メモリを割り当てれば良いか、とか、イベント中で少なくとも何バイト スキップしてもデータストリームの終わりにならないか、などを知ることが できます。

よくある別の **svnlook** の使い方はリビジョンまたは トランザクションツリーの実際の内容を見ることです。**svnlook tree** コマンドは要求されたツリー中の ディレクトリとファイルを表示します。`--show-ids` オプション を指定するとそれらのパスごとのファイルシステムノードリビジョン ID も 表示します (そのようなパスは一般的に言ってユーザよりも開発者に有用なもの でしょう)。

```
$ svnlook tree /path/to/repos --show-ids
/ <0.0.1>
 A/ <2.0.1>
  B/ <4.0.1>
   lambda <5.0.1>
   E/ <6.0.1>
    alpha <7.0.1>
    beta <8.0.1>
   F/ <9.0.1>
  mu <3.0.1>
  C/ <a.0.1>
  D/ <b.0.1>
   gamma <c.0.1>
   G/ <d.0.1>
    pi <e.0.1>
    rho <f.0.1>
    tau <g.0.1>
   H/ <h.0.1>
    chi <i.0.1>
    omega <k.0.1>
    psi <j.0.1>
 iota <1.0.1>
```

ツリー中のディレクトリのファイルの構成が理解できれば **svnlook cat**, **svnlook propget**, そして **svnlook proplist** のようなコマンドを 使ってそれらのファイルやディレクトリについてのより詳細な情報を 取得することができます。

svnlook は他にもいろいろな問い合わせをしたり、 いままで説明した情報の一部を表示したり、指定したリビジョンや トランザクションのどのパスが修正されたかを報告したり、ファイルや ディレクトリに対するテキストや属性の相違点を表示したり、などなど ができます。以下は **svnlook** が理解できる現時点 でのサブコマンドの簡単な説明の一覧と、その出力です:

`author` そのツリーの実行者です

`cat` ツリーの特定のファイルの内容を表示します。

`changed` ツリー中で変更のあったファイルとディレクトリの一覧

`date` ツリーのタイムスタンプです

`diff` 変更されたファイルの unified diff の表示

`dirs-changed` ツリー自身に変更があるか、その子供のファイルに変更が あったディレクトリの一覧表示

`history` バージョン化されたパスの履歴中での、興味深い場所の表示 (どこで変更やコピーが起きたかを示します)。

`info` ツリーの変更者、タイムスタンプ、ログメッセージ文字数、そして ログメッセージの表示

`lock` パスがロックされている場合にロックの属性を表示します。

`log` ツリーのログメッセージの表示

`propget` ツリー中のパスに設定された属性値を表示します。

`proplist` ツリー中のパスに対して設定された属性の名前と値を表示します。

`tree` ツリーの一覧表示をします。オプションでそれぞれのパスに 結びついたファイルシステムノードリビジョンの ID を 表示します。

`uuid` リポジトリの UUID — つまり *U*niversal *U*nique *ID*entifier(普遍的に一意な識別子) を表示します。

`youngest` 最新のリビジョン番号を表示します。

5.4.1.2 svnadmin

svnadmin プログラムはリポジトリ管理者によって 一番よく利用されます。Subversion リポジトリを作成

することのほか このプログラムはリポジトリに対してさまざまな保守操作をします。 **svnadmin** の構文は、**svnlook** のものとよく似ています:

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

Available subcommands:
   create
   deltify
   dump
   help (?, h)
…
```

既に **svnadmin** の `create` サブコマンドを見てきました (項 5.3 参照)。 他のサブコマンドのほとんどをこの章の後のほうで説明します。いまは 利用可能なサブコマンドの全体を軽く見ておきます。

`create` Subversion リポジトリを新規に作成します。

`deltify` リビジョン範囲を指定して実行すると、それらの リビジョンで変更されたパス上で祖先の差分を計算します。リビジョンが指定 されなければこのコマンドは単に `HEAD` リビジョンの差分を計算します。

`dump` 指定範囲のリビジョンのリポジトリの内容をダンプ します。ポータブルダンプ形式で出力します。

`hotcopy` リポジトリのホットコピーをとります。いつでも実行することが でき、他のプロセスがリポジトリを利用しているかどうかにかかわらず、安心して リポジトリのコピーをとることができます。

`list-dblogs` (Berkeley DB リポジトリの場合のみ) リポジトリに関係した Berkeley DB ログファイルのパスを一覧表示します。 このリストはすべてのログファイルを含みます — 現在 Subversion が利用しているもの、もう 利用していないものも含みます。

`list-unused-dblogs` (Berkeley DB リポジトリの場合のみ) リポジトリに関係した Berkeley DB ログファイルで、既に利用していない もののパスを一覧表示します。そのようなログファイルはリポジトリレイアウトから 安全に削除することができますが、リポジトリの壊滅からの復旧には必要となる事態にそなえて アーカイブすることもできます。

`load` データストリームから、リビジョンの集まりをリポジトリにロードします。 データストリームは `dump` サブコマンドで生成されたのと 同じポータブルダンプ形式です:

`lslocks` リポジトリに存在するロックを説明つきで一覧表示します。

`lstxns` 現時点でリポジトリに存在しているコミットされていない Subversion トランザクションの名前を一覧表示します。

`recover` 必要に応じてリポジトリの回復ステップを実行します。普通は リポジトリとの間の通信をきれいに終了できなかったプロセスによって 起きた致命的なエラーの後で実行します。

`rmlocks` 一覧されたパスからロックを無条件に取り除きます。

`rmtxns` リポジトリから Subversion トランザクションをきれいに削除します。 (`lstxns` サブコマンドからの出力をこの プログラムに入力すると便利です)

`setlog` リポジトリ中の指定リビジョンの `svn:log` (コミットログメッセージ) 属性の値を新しい値で置き換えます。

`verify` リポジトリの内容を確認します。これはリポジトリに格納されたバージョン化された データのチェックサム比較、なども含まれます。

5.4.1.3 svndumpfilter

Subversion は非公開のデータベースシステムにすべてのデータを格納 しますが、簡単には手で修正することができないようにするためです。 実際にはそれほど難しいことでもないのですが。そしてデータがリポジトリに いったん格納されてしまうと、Subversion はそのようなデータを削除する ための簡単な機能を提供してはいません。 [*3] しかし、時にはどうしてもリポジトリの履歴を操作したいことがあります。 リポジトリに間違って追加してしまったすべてのファイル (あるいはどんな 理由であれとにかくそこにあるべきではないファイル) を削除したいと思う かも知れません。あるいは一つのリポジトリを共有する複数のプロジェクト があって、それぞれを固有のリポジトリに分割することに決めたのかも 知れません。このような作業のためには、管理者はリポジトリ中のデータ のより柔軟に管理可能で柔軟な表現形式が必要です — それは Subversion のリポジトリダンプフォーマットです。

Subversion のリポジトリダンプフォーマットは時間とともにバージョン化 されたデータに加えた変更点に対する可読な形の表現形式です。ダンプ データを生成するには **svnadmin dump** を使い、 新たらしいリポジトリにそれをロードするには **svnadmin load** を使います (項 5.4.5 参照)。ダンプ形式が可読な形で あることの大きな利点は、注意して扱えばそれを調べたり修正したりできること です。もちろん、欠点としては、もし二年分のリポジトリ内容が一つの巨大な ダンプファイルに保存されているような場合には特定の場所を見つけて修正 するには非常に、非常に長い時間がかかるであろうことです。

管理者が自由にしたい場合に最もよく利用されるツールというわけでは ありませんが、 **svndumpfilter** は非常に特殊な 役に立つ機能を提供しています — パスベースのフィルタとして 実行することによってそのダンプデータをすばやく簡単に修正することが できるのです。保存したいと思うパスのリストか、保存したくないパスの リストを単に与えてこのフィルタにリポジトリのダンプデータをパイプで 入力するだけです。結果は、あなたが (明示的、あるいは暗黙に) 要求した バージョン化されたパスのみを含むような修正済みダンプデータになります。

svndumpfilter の構文は以下のものです:

[*3] ところで、そのような設計は意図したものです。バグ ではありません。

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

Available subcommands:
   exclude
   include
   help (?, h)
```

興味深いサブコマンドは二つだけです。これらのサブコマンドを使って、ストリーム中で明示的に、あるいは暗黙に取得するパスを選ぶことができます。:

`exclude` ダンプデータストリームから特定のパスを排除します。

`include` ダンプデータストリームから、指定したパスだけを 出力するようにします。

このプログラムが実際にどのように動作するか例を見てみましょう。別の場所でリポジトリ中でどのようにレイアウトを選ぶかを決める手順について 議論しました (項 5.5.1) — プロジェクトごとの リポジトリ、あるいはそれらをまとめたものを使って、リポジトリ中で構成を 変更し、などの手法です。しかし、新しいリポジトリが運用されたあとで、よくレイアウトを再編成していくつかの修正をしたいということもあります。一番多いのは一つのリポジトリを共有していた複数のプロジェクトを プロジェクトごとの別々のリポジトリに分離したい、という場合です。

私たちの架空のリポジトリは三つのプロジェクトを含んでいます: `calc`, `calendar`, そして `spreadsheet` です。それらは以下のようなレイアウトになっています:

```
/
   calc/
      trunk/
      branches/
      tags/
   calendar/
      trunk/
      branches/
      tags/
   spreadsheet/
      trunk/
      branches/
      tags/
```

これら三つのプロジェクトごとの固有のリポジトリを手に入れるには、まずリポジトリ全体をダンプします:

```
$ svnadmin dump /path/to/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

次に結果のダンプファイルをフィルタに通しますが、各実行時で ただ一つの最上位ディレクトリを含むように指定することで、 三つの新しいダンプファイルを生成することができます:

```
$ cat repos-dumpfile | svndumpfilter include calc > calc-dumpfile
…
$ cat repos-dumpfile | svndumpfilter include calendar > cal-dumpfile
…
$ cat repos-dumpfile | svndumpfilter include spreadsheet > ss-dumpfile
…
$
```

この時点で、判断しなくてはなりません。上でできた三つのダンプファイルは 正しいリポジトリですが、元のリポジトリ中にあった通りのパス構成で保存されて います。これは `calc` プロジェクト単独のリポジトリを 取得したにもかかわらず、リポジトリはあいかわらず `calc` という名前の最上位ディレクトリ名称を持っていることを意味します。 もし `trunk`, `tags`, そして `branches` ディレクトリそれぞれをリポジトリ のルートディレクトリとしたければダンプファイルを編集して `Node-path` と `Copyfrom-path` ヘッダがもうこれからは先頭に `calc/`というパス部分を持たないようにしなくてはなりません。 同様に `calc` ディレクトリを作ったダンプデータのセクションを 削除したいでしょう。それは何か以下のような感じになっています:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```

警告

もし最上位ディレクトリを削除するためにダンプファイルを手で編集しようと 考えているなら、利用するエディタが改行文字を自動的にマシン固有の形式に 変換してしまわないことを確認してください (たとえば
r
n を
n などに)。 この変換が起きるとダンプファイルの内容はメタデータと一致しなくなり、使 い物にならなくなってしまいます。

この修正後に残ったファイルを使って新しい三つのリポジトリを作成する ことができ、それぞれのダンプファイルを正しいリポジトリにロード することができます:

```
$ svnadmin create calc; svnadmin load calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar; svnadmin load calendar < cal-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : cal.c ... done.
…
$ svnadmin create spreadsheet; svnadmin load spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : ss.c ... done.
…
$
```

svndumpfilter の両方のサブコマンドとも 「空の」リビジョンをどのように扱うかを決めることが できます。パスの変更のみを含んでいるようなリビジョンを除外 すれば、空のリビジョンは興味がないか、不要なものであると 考えることができます。**svndumpfilter** は 以下のコマンドラインオプションを用意しています:

`--drop-empty-revs` 空のリビジョンを生成しません — 単に無視します。

`--renumber-revs` 空のリビジョンが削除された場合に (`--drop-empty-revs` を 利用することによって)、残っているリビジョンのリビジョン番号を変更して リビジョン番号が飛ばないようにします。

`--preserve-revprops` 空のリビジョンが削除されない場合に、それら空のリビジョンに関する リビジョン属性 (ログメッセージ、変更者、日付、カスタム属性、など) を 保存します。そうでなければ、空のリビジョンは元のタイムスタンプと、 このリビジョンは **svndumpfilter** によって空にされた ということ

を示す自動生成されたログメッセージのみを含むことになります。

svndumpfilter は非常に便利で、作業を省力化して くれますが、残念なことにいろいろな問題もあります。まずこのユーティリティー はパスの構文に極端に敏感です。ダンプファイル中のパスが先頭にスラッシュ を含んでいるかどうかに注意してください。`Node-path` と `Copyfrom-path` ヘッダを確認する必要がある かも知れません。

```
…
Node-path: spreadsheet/Makefile
…
```

パスの先頭にスラッシュがある場合、**svndumpfilter include** と **svndumpfilter exclude** に渡す パスの先頭にスラッシュを含める必要があります (そして、逆にスラッシュが ないなら含めてはいけません)。さらにダンプファイルの先頭のス ラッシュが何かの理由で矛盾している場合には [*4] おそらく、すべてをスラッシュ付きにするか、その逆にするような正規化を パスに対して施す必要があります。

また、コピーされたパスは問題を起こすかも知れません。Subversion は リポジトリ中のコピー操作をサポートしていて、ここでは新しいパスは 既に存在するパスからコピーすることによって作成されます。リポジトリの 生存中のどこかでファイルあるいはディレクトリを **svndumpfilter** が排除するようなどこかの場所からコピーし、**svndumpfilter** が含めるような場所にコピーしたかも知れません。ダンプデータに自己一貫性を 保証するため **svndumpfilter** は新しいパス — コピーによって作られた任意のファイルの内容を含むような — を表示 する必要がありますが、それはダンプデータストリームから排除された 存在しないようなソースからのコピーの追加としては表示されません。 しかし Subversion のリポジトリダンプ形式はそれぞれのリビジョンで何が変更 されたかを示すだけなので、コピー元の内容は利用不可能です。もし リポジトリ中でこのようなコピーがある可能性がある場合にはもう一度 **svndumpfilter** に含めるパスと排除するパスを 再考する必要があるかも知れません。

5.4.1.4 Berkeley DB ユーティリティー

Berkeley DB リポジトリを使っている場合は、バージョン化されたファイルシ ステム構造とデータ全体はリポジトリの db サブディレクトリにあるいくつかのデータベーステーブルに保存されています。 このサブディレクトリは通常の Berkeley DB 環境ディレクトリで、どのよう な Berkeley データベースツールとも組み合わせて使うことができます (これ らのツールに関するドキュメントは SleepyCat のウェブサイト <http://www.sleepycat.com/>にあります)。

通常の Subversion の利用ではこれらのツールは不要です。Subversion リポジトリ に必要なほとんどの機能は **svnadmin** を使って実行する ことができます。たとえば **svnadmin list-unused-dblogs** と **svnadmin list-dblogs** は Berkeley の **db_archive** で提供されている機能のサブセットであり、 **svnadmin recover** は **db_recover** ユーティリティーの普通の状況での利用の仕方を反映したコマンドです。

それでもいくつかの Berkeley DB ユーティリティーは知っていると便利です。 **db_dump** と **db_load** プログラムは Berkeley DB データベース のキーと値を表現するカスタム形式ファイルの読み書きを実行します。Berkeley データベースはマシンアーキテクチャをまたいだ互換性 があるので、この形式はアーキテクチャや OS の違いを意識せずに データベースマシン間で転送するのに便利な方法です。また、 **db_stat** ユーティリティーは Berkeley DB 環境の 状態についての有用な情報を表示します。これにはサブシステムの ロックや

[*4] **svnadmin dump** は先頭スラッシュに関して一貫した ポリシーがありますが — 付けないようにするというものです — データをダンプするほかのプログラムはそれほど一貫していません。

データ保存についての詳細統計情報が含まれます。

5.4.2 リポジトリのお掃除

Subversion リポジトリは一般的にいったん設定してしまえばほとんど 注意を払う必要はありません。しかし、管理者による、いくつかの 補助が必要かも知れません。**svnadmin** ユーティリティー には以下のような作業を助けるための機能があります。それは

- コミットログメッセージの修正。
- 死んだトランザクションの削除。
- 「固まってしまった」 リポジトリの復旧。
- リポジトリの内容を別のリポジトリに移すこと。

svnadmin のサブコマンドで一番よく使われるのは 多分 `setlog` です。トランザクションがリポジトリに コミットし、リビジョンを表示したとき、新しいリビジョンに関連した ログメッセージは、そのリビジョン自体のバージョン化されない属性として 格納されます。言い換えると、リポジトリはその属性の最後の値だけを記憶していて、以前のものは捨ててしまいます。

ときどきユーザはログメッセージに間違いを見つけます (スペルミスや 間違った情報など)。もしリポジトリが (`pre-revprop-change` と `post-revprop-change` フックを使って。 項 5.3.1 参照) コミット完了後このログメッセージの変更を受け付けるとすると ユーザは、**svn** プログラムの `propset` コマンドを使ってログメッセージを ネットワーク越しに「修正」 することができます。 (第 9 章参照) しかし、情報が永久に失われてしまうことを防ぐため、Subversion リポジトリはデフォルトではそれをさせません。デフォルトは、 バージョン化されない属性は、管理者のみが変更することが できます。

もしログメッセージを管理者が変更する必要がある場合、 **svnadmin setlog** を使います。このコマンドは リポジトリの指定したリビジョンのログメッセージ (`svn:log` 属性) を、用意したファイルから新しい値を読み出し形で変更します

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

svnadmin setlog コマンドだけでは、リモートクライアント としてバージョン化されていない属性を修正する場合と同じ制約を 受けます — つまり `pre-`と `post-revprop-change` フックはやはり実行され、この仕組みで変更される修正点は反映されて しまいます。しかし管理者はこのような保護機能を **svnadmin setlog** コマンドに`--bypass-hooks` を指定することで回避できます。

> 警告
>
> 
>
> しかしフックを回避すると、属性変更、バージョン化されていない属性変更 を追うためのバックアップシステム、などなどに関係した通知 メール も 回避されてしまうことに注意してください。言い換えると、何を、どのように 修正するかについて、非常に注意して実行してください。

別のよくある **svnadmin** の使い方は終了していない — 多分死んでしまった — Subversion トランザクションに関する リポジトリへの問い合わせです。コミットが失敗したとき、普通 トランザクションはきれいに消去されます。つまりトランザクション はリポジトリから削除され、そのトランザクションに (だけに) 関連したデータも同様に削除されます。 しかし、しばしばトランザクションの掃除が起こらずに失敗することが あります。これにはいくつかの理由が考えられます: 多分クライアントの 操作がユーザによって乱暴に終了されたか、ネットワークの異常などが 処理の途中で起こった場合です。理由にかかわらず、死んだままの トランザクションが残ることはありえます。ディスクをわずかに食うことを のぞけば、このようなトランザクションは全く無害です。それでも 潔癖な管理者はこのようなトランザクションを削除したいと思うかも 知れません。

svnadmin の `lstxns` コマンド を使って、その時点での未完了のトランザクションの名前の一覧表示 することができます。

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

出力結果のそれぞれの項目は **svnlook** (とその`--transaction` オプション) で 使うことができ、誰がトランザクションを作り、それは いつで、どのような変更がトランザクションに起きたか、を 知ることができます。 — 言い換えると、そのトランザクションは削除対象として 安全な候補なのかどうか、ということをです。もしそうなら、 トランザクションの名前を **svnadmin rmtxns** に渡すことができ、そのトランザクションはきれいに削除 されます。`rmtxns` サブコマンドは、 `lstxns` の出力をそのまま入力として とることもできます!

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

このような二つのサブコマンドを使う場合、リポジトリを一時的に クライアントからアクセスできなくする必要があります。 これで誰もあなたがクリーンアップを始める前に正しいトランザクション を開始できなくなります。以下は、リポジトリ内の未解決のトランザクション のそれぞれについての情報をすばやく生成するためのちょっとした スクリプトです:

このスクリプトを **/path/to/txn-info.sh /path/to/repos** のように して実行できます。出力は基本的には **svnlook info** 出力のいろいろな断片をつないだようなものになります。 (項 5.4.1.1 参照), 以下のような 感じです:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
```

例 5.4.1 txn-info.sh (未解決トランザクションの表示)

```
#!/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}" --transaction "${TXN}"
done
```

```
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
$
```

長く放置されているトランザクションは普通は何かに失敗したか、コミットを 中断されたかのどちらかです。トランザクションの日付スタンプは 役に立つ情報を与えてくれます — たとえば 9 ヵ月も前に始まった 操作がいまだに有効である可能性など、いったいどの程度あるのでしょうか?

簡単に言って、トランザクションのクリーンアップの決定は、無分別に やる必要はありません。いろいろな情報源 — Apache のエラーログや アクセスログ、成功した Subversion のコミットログ、などなど — が どうしたら良いかを決める上で役に立ちます。最後に、管理者はしばしば 死んだトランザクションの所有者と思われる人と、(メールなどで) その 死にかかったトランザクションの状態を確認することができます。

5.4.3 ディスク領域の管理

ここ数年で記憶装置のコストは非常に低くなってきた一方で、ディスクの利用方法 は、大量のデータをバージョン管理するために、システム管理者にとっては、やはり 依然として考慮すべきことです。動作中のリポジトリによって消費される追加の 領域はオフラインでバックアップすることが必要な領域でもあり、バックアップ のスケジュール管理を考えると、何倍かになるでしょう。Berkeley DB リポジ トリを使う場合には、データ

保管の仕組みは複雑なデータベースシステムである ので、データのどの部分をオンラインのままに残し、どの部分にバックアップが 必要で、どの部分を安全に削除できるか、ということについて理解しておくことには 意味があります。 この節は Berkeley DB だけに関係しています。FSFS リポジトリには削除したり、 調整が必要な特殊な追加データはありません。

最近まで、Subversion リポジトリに関して最も多くディスクを消費する部分は Berkeley DB が実際にデータベースファイルを修正する前に前もっと書き込むための ログファイルの領域でした。これらのファイルはデータベースのある状態から 別の状態への変化の経緯にそったすべての操作を記録します — データベース ファイルはある特定の時刻にその状態を反映される一方でログファイルはその前後 の状態のすべての本稿を含んでいます。そのようなわけでログファイルは非常に 早いスピードでサイズを増やします。

ありがたいことに、Berkeley DB のリリース 4.2 からデータベース環境は 特に外部操作することなしに未使用のログファイルを削除する能力を持つ ようになりました。Berkeley DB バージョン 4.2 かそれ以降でコンパイルされた **svnadmin** はこの自動的なログファイルの削除が設定 されています。この機能を有効にしたくない場合には単に **svnadmin create** コマンドで`--bdb-log-keep` を渡してください。これを忘れたり、後で変更したい場合には、単に リポジトリの `db` ディレクトリ中にある `DB_CONFIG` を編集して、`set_flags DB_LOG_AUTOREMOVE` ディレクティブをコメントアウトしてから この変更を強制的に有効にするためにそのリポジトリに対して **svnadmin recover** を実行してください。データベースの設定についての詳細は 項 5.3.2 を参照してください。

このような自動ログファイル削除の仕組みを利用しなければ、 リポジトリを利用するにつれてログファイルは蓄積されていきます。 それてこれは実際にデータベースシステムであれば当然付いている機能です — ログファイル以外に何も残っていないような状況でデータベース全体を再構成する ことができるようになっていなくてはならず、そのようなログファイルは データベースの壊滅的な破壊からの復旧で利用できなければならないからです。 しかし普通は Berkeley DB で既に利用されていないログファイルをアーカイブし、 その後ディスクから削除することで領域を広げようとするでしょう。利用していない ログファイルの一覧を見るには **svnadmin list-unused-dblogs** コマンドを使ってください:

```
$ 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!
```

リポジトリのデータサイズをできるだけ小さくするために Subversion は リポジトリに対して差分化 (あるいは「差分記憶」) の処理をします。差分化は別のデータの部分に対する 差分の集まりをひとつのデータの塊として表現するものです。二つのデータ が非常に似ていればこの差分化は差分化されたデータの記憶領域を節約します — もとのデータサイズと同じだけの領域を確保するのにくらべて小さく なります。いわば、「以下の変更点をのぞけば、他の点についてはここに あるデータのままですよ」という表現に必要なだけの領域で済みます。 具体的に言うと、ファイルの新しいバージョンがリポジトリにコミットされる たび、Subversion は前のバージョン (実際には前のバージョンのいくつか) を この新しいバージョンに対する差分として表現します。その結果、大きくなり がちなリポジトリデータ — つまりバージョン化されたファイルの内容 — の大部分を、もとの「完全なテキスト」として 保存するよりはずっと小さなサイズで格納することができます。

注意

差分化の対象となるすべての Subversion リポジトリデータは単一の Berkeley DB データベースファイルに保存されるので保存されているデータのサイズを 小さくしたからといって必ずしもデータベースファイル自身のサイズを減らす ことにはなりません。しかし Berkeley DB はデータベースファイル中の未使用 領域の内部的な記録を保存しておりデータベースファイルのサイズを拡張する 前にそのような領域をまず利用します。そのため差分化は直接に空間の節約につな がりはしなくても今後のデータベースサイズが拡大するスピードを有効に押える ことができます。

5.4.4 リポジトリの復旧

項 5.2.3.1 で触れたように、Berkeley DB リポジトリは正しく閉じられなかった場合には中間的な状態に固まってしまう ことがあります。こうなった場合管理者はデータベースを以前の一貫した状態 にまで戻してやる必要があります。

リポジトリ中のデータを保護するため Berkeley DB はロックのしくみ を利用しています。このしくみはデータベースの特定のが同時に複数のデータベー スアクセスによって修正されないことを保証するためのもので、それぞれのプ ロセスから見たときには、読み込み時にはデータは正しい状態にあるように見 えます。データベース中のどこかを変更する必要ある場合にはまず対象となる データがロックされていないかどうかを確認します。もしロックされていなけ ればそのプロセスはデータをロックし、必要な修正を加え、そのデータに対するロックを外します。他のプロセスはデータベースの内部に引き続きアクセス できるようになる前にロックファイルが削除されるまで待たされます。

Subversion リポジトリを使う上で、致命的なエラー (ディスクがいっぱいに なったり、メモリがなくなったり) や、割り込みによって、データベースに かけたロックを削除する機会をなくしてしまうことがあります。その結果 バックエンドのデータベースは「固まって」しまいます。 こうなったときには、リポジトリへのどのようなアクセスも永久に待たされる ことになってしまいます。(というのは、すべての新しいアクセスはロック が解除されるのを待ちますが、それは決してやってこないからです)

まず、そういうことがリポジトリに起こっても、悲鳴をあげないで ください。Berkeley DB のファイルシステムはデータベーストランザクション とチェックポイント、それに事前ジャーナル書き込みの仕組みをうまく 利用していて、本当に破滅的な出来事以外は [*5] データベース環境を永久に葬り去ることはできないことを保証します。 十分神経質なリポジトリ管理者は何んらかの方法でリポジトリデータの オフラインバックアップをとっているかも知れませんが、バックアップテープを リストアしてくれとシステム管理者を呼ぶのはまだです。

次に、以下の手順を使って、リポジトリの「復旧」 を試してみてください:

1. リポジトリにアクセスしている (あるいはしようとしている) プロセスが 一つもないことを確認してください。ネットワークアクセス可能な リポジトリでは、これは Apache HTTP サーバをシャットダウンする ことも意味します。
2. リポジトリを所有し、管理しているユーザになってください。 これは重要ですが、実行時と同様、復旧時に間違ったユーザで作業する ことによってもリポジトリファイルのパーミッションが変更されて しまうかも知れないからです。これによって実際には「復旧」 したのにアクセス不能のままになってしま

[*5] たとえば: ハードディスク + 強い電磁場 = 破滅。

う可能性があります。

3. **svnadmin recover /path/to/repos** コマンドを 実行してください。以下のような出力が表示されると思います:

```
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 19.
```

このコマンドは完了までに数分かかることもあります。

4. Subversion サーバの再起動

この方法はほとんどのリポジトリロックを解消します。このコマンドは 単に `root` になるのではなく、データベース を所有し、管理しているユーザで実行することに注意してください。 復旧作業は、傷を負ったいろいろなデータベースファイル からの再作成の作業も含みます。(たとえば共有メモリ領域などです) `root` での復旧は、`root` が 所有しているファイルを作成することで、これはリポジトリへの 接続状況が復旧した後でも通常のユーザはこれに対してアクセスする ことができないことを意味します。

もしいま述べた作業が、何かの理由でうまくリポジトリを正常に 戻せない場合、二つのことをすべきです。まず、壊れたリポジトリを どけて、最後のバックアップをリストアします。それから Subversion のユーザリストにメールします。(これは、users@subversion.tigris.org <`mailto:users@subversion.tigris.org`> です) このとき問題点を詳しく 説明してください。データの一貫性は、Subversion 開発者にとって 非常に高いプライオリティです。

5.4.5 リポジトリの移行

Subversion ファイルシステムはさまざまなデータベーステーブルに分散された データを持ちますが、これは一般的には Subversion 開発者だけが知っている (て、興味のある) ことです。しかし、すべての、あるいは一部のデータを 一つの、持ち運びに便利な単純なファイル形式にまとめたいことがあります。 Subversion はそのような仕組みを **svnadmin** サブコマンド の組によって実装しています: `dump` と `load` です。

Subversion リポジトリをダンプしたりロードしたりする一番よくある理由は Subversion 自身の変更にあります。Subversion が完成に近づくにつれ、 バックエンドデータベースのスキーマ変更によっては リポジトリの前のバージョンとの互換性がなくなってしまいます。 ダンプとロードが必要になる他の理由としては、Berkeley DB を他の OS や CPU アーキテクチャに以降する場合、あるいは Berkeley DB と FSFS バックエンド間を切り替えて使う場合です。このために推奨されている作業ステップは比較 的簡単です:

1. 現行 バージョンの **svnadmin** を使ってリポジトリをダンプファイルにダンプしてください。
2. Subversion の新しいバージョンへのアップグレード。
3. 古いリポジトリをどけて、新しい空のリポジトリをそこに作りますが、 これには新しい **svnadmin** を使ってください。
4. もう一度新しい **svnadmin** を 使って、ダンプファイルを、それぞれ作ったばかりのリポジトリに ロードしてください。
5. 古いリポジトリから新しいものに必要なカスタマイズ部分を すべてコピーしてください。これには `DB_CONFIG` ファイルと、フックのスクリプトが含まれます。新しいリリースの Subversion のリリースノートに注意して、最後のアップグレードでフック や設定オプションに変更がないかどうかを見てくだ

さい。

6. もし移行によってリポジトリが別の URL からアクセスされるようになった場 合 (例えば別のコンピュータに移したり、別のスキーマを経由して アクセスしたりするような場合)、おそらくユーザには既存の作業 コピー上で **svn switch –relocate** を実行する ように言わなくてはならないかも知れません。

名前

svn switch を見てください。

svnadmin dump は、リポジトリリビジョンのある 範囲を出力しますが、それは Subversion のカスタムファイルシステム ダンプ形式になっているものです。ダンプ形式は標準出力に表示され、 進行状況などのメッセージは標準エラー出力に表示されます。これで 出力をファイルにリダイレクトすることができ、その一方でステータス の出力については端末ウィンドウ上で見ることができます。たとえば:

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

処理の最後で、指定した範囲のリポジトリリビジョンのデータすべて が保存された一つのファイル (前の例では、`dumpfile`) を手に入れることができます。 **svnadmin dump** は他の「読み出し」プロセス (たとえば **svn checkout** など) がやるのと同じような 方法でリポジトリからリビジョンツリーを読み出すことに 注意してください。そのため、このコマンドはいつでも安全に実行できます。

組になったもう一方のサブコマンドである **svnadmin load** は、標準入力を、Subversion リポジトリのダンプファイルと して解析し、ダンプされたリビジョンを目的のリポジトリに再現します。 それはまた経過情報などを返しますが、こちらは標準出力に 表示します:

```
$ svnadmin load newrepos < dumpfile
<<< Started new txn, based on original revision 1
     * adding path : A ... done.
     * adding path : A/B ... done.
     …
------- Committed new rev 1 (loaded from original rev 1) >>>

<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.
```

```
------- Committed new rev 2 (loaded from original rev 2) >>>

…

<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

load の結果、新しいリビジョンがリポジトリに追加されます — これは 通常の Subversion クライアントからリポジトリに対してコミットをする のと同じ効果があります。またやはりコミットと同様に load 処理中に おきるそれぞれのコミットの前後で実行するフックスクリプトを使うことも できます。 **svnadmin load** に `--use-pre-commit-hook` と `--use-post-commit-hook` オプションを渡すことでロードされるリビジョンごとに Subversion に 対してそれぞれ pre-commit と post-commit のフックスクリプトを実行する ように指示できます。これで例えば、通常のコミット時と同様の妥当性チェック のようなステップをロードされるリビジョンごとに保障するような使い方が できます。もちろんこのようなオプションの利用には注意が必要です — post-commit フックスクリプトで新しいコミットごとにメーリングリストに対して email を送信するようになっていた場合、リビジョンがロードされるたびに リストに数え切れないくらいの email を流したいとは思わないでしょう！フックスクリプトについては 項 5.3.1 により詳しい情報があります。

svnadmin は標準入力と標準出力をリポジトリのダンプ とロード処理に使うので、気の利いた人は、以下のようなやり方を試す こともできます (おそらく、パイプの両側の **svnadmin** は、異なるバージョンであるかも知れません):

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

デフォルトではダンプファイルは非常に大きくなります — リポジトリ 自体よりもずっと大きくなるでしょう。理由はすべてのファイルのすべての バージョンは、ダンプファイル中では完全なテキストとして表現される からです。これはダンプデータをパイプ経由で他のプロセス (圧縮プログラム や、フィルタープログラム、あるいはロードプロセスのようなもの) に送る場合には もっとも早く単純な方法です。しかし長期保存用にダンプファイルを作成する のであれば `--deltas` スイッチを使ってディスク領域を節約 したほうが良いでしょう。このオプションを使うと引き続くリビジョン間の ファイルは圧縮された形のバイナリ差分として出力されます — これはちょ うどリポジトリ中に保存されたリビジョンファイルと同じような形になります。 このオプションを使うと処理は遅くなりますが結果のダンプファイルは 元のリポジトリにかなり近いサイズにまでなります。

前に注意したように **svnadmin dump** はリビジョンの範囲を出力します。 `--revision` オプションを使えば、一つの リビジョンのダンプや、リビジョン範囲のダンプができます。 このオプションを省略すれば、すべての存在するリポジトリ リビジョンがダンプされます。

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

Subversion はそれぞれの新しいリビジョンをダンプするのでその出力には 後で実行されるローダが前のリビジョンを元にしてそのリビジョンを再 作成するのに必要な十分な情報があります。 言い換えると、ダンプファイル中でどのようなリビジョンが指定されても リビジョン中で変更のあったアイテムのみがダンプに現れるということ です。この規則の唯一の例外は、現在の **svnadmin dump** がダンプする最初のリビジョンです。

デフォルトでは、Subversion は前のリビジョンに対する単なる差分として 最初のダンプリビジョンを表現することはありません。この理由の一つは 、ダンプファイルには直前のリビジョンがないからです! 二番目に Subversion はダンプデータがロードされるリポジトリの状態について何も知らない からです。(もしロードが起こるとすれば、ですが。) **svnadmin dump** の個別の実行の出力が自己充足して いるのを保証するため、最初のダンプリビジョンはデフォルトでは すべてのディレクトリ、ファイル、リポジトリにあるそのリビジョンの属性 の完全な表現になっています。

しかし、このデフォルトの振る舞いを変えることもできます。リポジトリを ダンプするときに `--incremental` オプションを追加すると **svnadmin** は最初のダンプリビジョンとリポジトリ中の 直前リビジョンとの差分をとろうとします。残りのすべてのダンプ されるリビジョンにも同じ方法で扱います。 それからダンプ範囲にある残りのリビジョンが出力するのと同じように 最初のリビジョンを — リビジョン中に起こる変更だけを 考慮して出力します。 この利点は大きな一つのダンプファイルのかわりに、ロードに成功するような 小さないくつものダンプファイルを作ることができることです。 こんな感じです :

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

これらのダンプファイルは以下のようなコマンドの流れで 新しいリポジトリ中にロードされます:

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

`--incremental` オプションを使った別の かっこいい方法は、既に存在しているダンプファイルに 新しいダンプリビジョン範囲を追加することです。たとえば `post-commit` フックがあり、それは単に フックをトリガーするような一つのリビジョンのリポジトリ ダンプを追加するものです。あるいは最後にスクリプトを実行した時点より 後にリポジトリに追加されたすべてのリビジョンに対してのダンプファイルを追加する ようなスクリプトを実行するかも知れません。このように利用することで **svnadmin** の `dump` と `load` コマンドは価値のある手段となりますが、これによって、リポジトリの変更を 時間をかけてバックアップして、システム

クラッシュや、他の壊滅的な出来事に そなえるというわけです。

ダンプ形式はまたさまざまな異なるリポジトリの 内容を単一のリポジトリにマージするために利用することもできます。 **svnadmin load** の `--parent-dir` オプションを使って ロードプロセス用の新たな仮想ルートディレクトリを指定することが できます。これは、もし `calc-dumpfile`, `cal-dumpfile`, そして `ss-dumpfile` という三つのリポジトリ のダンプファイルがある場合、最初にそれらすべてを保持するような 新しいリポジトリを作ることができることを意味します:

```
$ svnadmin create /path/to/projects
$
```

それから三つの以前のリポジトリのそれぞれの内容を含んだ 新しいディレクトリをリポジトリ中に作ります:

```
$ svn mkdir -m "Initial project roots" \
      file:///path/to/projects/calc \
      file:///path/to/projects/calendar \
      file:///path/to/projects/spreadsheet
Committed revision 1.
$
```

最後に個々のダンプファイルを新しいリポジトリのそれぞれの 場所にロードします:

```
$ 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
…
$
```

Subversion リポジトリダンプ形式の利用方法について最後にもう一つだけ 触れます — 異なる保存の仕組みやバージョン管理システムから データを変換する方法です。これができる理由はダンプファイル形式は ほとんどの部分が可読であるためです。 [*6] このファイル形式を 使うと、比較的簡単に一般的な変更点のセットを表現することができます — それぞれの変更は新しいリビジョンとして扱われます。 事実、**cvs2svn** ユーティリティー (項 A.12 参照) は、CVS リポジトリの内容を表現するのに ダンプ形式を使うので、その内容を Subversion リポジトリに取り込むことができます。

[*6] Subversion のリポジトリダンプ形式は、RFC 822 形式によく 似ていて、ほとんどのメール で利用されているのと同じ形式です。

5.4.6 リポジトリのバックアップ

現代的なコンピュータが生まれてから技術的には非常に発展してきたものの、 残念なことに、一つのことだけは間違いなく真実です — ときどき、ものごとは まったく台無しになってしまう、ということです。停電、ネットワーク切断、 RAM の破壊、ハードディスクのクラッシュは、魔物以外の何者でもありません。 運命は最も優れた管理者にさえ降りかかるのです。それで、とても重要な トピックに行き着きます — どうやってリポジトリのバックアップを とるか、です。

一般的に、Subversion のリポジトリ管理者にとって、二つのバックアップ方法が あります — 差分バックアップと、フルバックアップです。この章の前の節で どうやって **svnadmin dump –incremental** を使って差分 バックアップをとるかを議論しました (項 5.4.5 参照)。本質的にこのアイディアは最後に バックアップをとってから起きたリポジトリの変更部分だけをバックアップ する方法です。

リポジトリのフルバックアップは文字通りリポジトリディレクトリ全体の 複製を作ることです (これは Berkeley データベース環境も含まれます) さて、一時的にリポジトリに対するすべてのアクセスを禁止しなければ、 単純な再帰的なディレクトリコピーの実行は、不完全なバックアップを作って しまう危険を持っています。というのは誰かが並行してデータベースに書き込んで いるかも知れないからです。

Berkeley DB の場合、Sleepycat のドキュメントは正しいバックアップコピーを 保証するようにデータベースファイルをコピーする場合の順序が書いて あります。同様の順序が FSFS データにもあります。しかしこのよう なプログラムを自分で書く必要はありません。Subversion 開発チー ムがすでにしているからです。**hot-backup.py** スクリプトは Subversion のソース パッケージの `tools/backup/` ディレクトリにあります。リポジトリパスとバックアップ位置を指定すると、**hot-backup.py** — それは単に **svnadmin hotcopy** コマンドのより賢い ラッパープログラムでしかありませんが — は、動作中のリポジトリをハックアップするのに必要なステップを実行します — あなたにリポジトリアクセスを禁止することなしに、です — ついでに、動作中のリポジトリから、死んでいる Berkeley ログファイルを きれいに削除します。

差分バックアップがあるとしても、規則的にこのプログラムを実行したく なるかも知れません。たとえば **hot-backup.py** を プログラムスケジューラに追加しようと考えるかも知れません (Unix であれば **crond** のようなもの)。 あるいは、細かい粒度のバックアップが好きなら、**hot-backup.py** を呼ぶような、post-commit フックスクリプトを書くこともできます。 (項 5.3.1 参照)。これは新しいリビジョンが 作られるたひにリポジトリの新しいバックアップができる方式です。 単に、以下を動作中のリポジトリディレクトリにある `hooks/post-commit` に追加してください:

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

結果のバックアップは、完全に機能する Subversion リポジトリで、現行の リポジトリが何かひどいことになったときには、置き換えて使うことが できるものです。

両方のバックアップ方法にはそれぞれ利点があります。一番簡単なのは フルバックアップで、それは常に現行リポジトリの完全なコピーです。 繰り返しになりますが、何かまずいことが動作中のリポジトリに起きた時には、単純な再帰的なディレクトリコピーでこのバックアップを 復元することができます。残念なことに、もしリポジトリの複数のバックアップ を管理している場合、このようなフルコピーは、実行中のリポジトリと同じくらい、それぞれがディスクを食うということです。

リポジトリダンプ形式を使った差分バックアップはデータベーススキーマ が引き続く Subversion 自身のバージョン間で変更されるときには非常に 役に立ちます。リポジトリの完全なダンプとロードは一般的に リポジトリを新しいスキーマにアップグレードすることが必要です。 そのような作業の半分 (つまり、ダンプの

部分) については既に 済んでいるのでとても便利です。不幸にも、差分バックアップの 作成 — そしてそのリストア — は長い時間がかかりますが、 それは、それぞれのコミットがダンプファイル、またはリポジトリの中で、実際に再実行されるからです。

どちらのバックアップの場合も、リポジトリ管理者は どのようにしてバージョン化されない属性への変更がバックアップに影響を 与えるかに注意する必要があります。このような変更は新しいリビジョンを それ自体で作り出すわけではないので、post-commit フックを呼び出すきっかけ にはならず、pre-revprop-chage や post-revprop-change フックのきっかけ にすらならないでしょう。 [*7] そして、時間の順序に沿わないでリビジョン属性を変更することができる — いつでも、どのリビジョン属性を変更することができます — ので、最後のいくつかのリビジョンの差分バックアップはそれ以前の バックアップの一部として行われたリビジョン属性の修正は取り入れる ことができません。

一般的に言って、本当に人間離れした潔癖さを持った人だけが完全なリポジトリ のバックアップを必要とするのでしょう。つまり、コミットが起こるたびに バックアップをとるわけです。しかし、そのリポジトリが相対的に細かい 粒度 (コミットごとのメールなど) と共に、何か別の冗長性の仕組みを持って いるのであれば、データベースのホットバックアップはリポジトリ管理者が システム全体の日次バックアップの一環として導入したいと考えるものかも知れません。 ほとんどのリポジトリでは、コミット メール をアーカイブするだけで復旧元 データとしての十分な冗長性を持っています。少なくとも最近のいくつかのコミット についてはそうです。しかしデータはとにかくあなたのものです — 必要なだけ保護するのに越したことはありません。

しばしば、リポジトリのバックアップに対する最良の方法は、分散 させることです。フルバックアップと差分バックアップに、コミット メールのアーカイブを追加することができます。たとえば Subversion 開発者は、Subversion ソースコードリポジトリを、新しいリビジョン が作られるたびにバックアップします。そして、すべてのコミットと 属性変更の通知メールをアーカイブしてとっておきます。同様の方法を とってください。ただし、必要な範囲で、便利さと安全性の微妙なバラ ンスをとってください。そして、このようなことを全部やっても、運命 の鉄拳からハードウェアを守ることはできないことに注意してください。 [*8] バックアップは確かにそのような試練の時からあなたを救うはずです。

5.5 プロジェクトの追加

リポジトリが作られて設定されれば、後は使うだけです。もし 既にデータの集まりを持っていて、それをバージョン管理したい 場合は、きっと **svn** クライアントプログラムの `import` サブコマンドを使いたいと思うでしょう。 しかしそうする前に、リポジトリについて長期的な視点で注意深く 考えるべきです。この節では、リポジトリのレイアウトをどのように 計画するか、そしてそのレイアウトの中にどのようにデータを配置する のが良いかについて、少しアドバイスします。

5.5.1 リポジトリレイアウトの選択

Subversion を使うと、あなたは情報を失うことなしにバージョン化されたファイルや ディレクトリをあちこちに移動することができますが、そうすることは、 データが特定の場所にあることを期待している、ときどきリポジトリにアクセス する人たちの作業を中断させてしまうかも知れません。先のこともちょっとは 考えてください。バージョン管理下にデータを置く前に、前もって計画を たててください。リポジトリの内容を、最初にうまく「レイアウト」 しておけば、あとで頭を抱えることがなくなります。

Subversion リポジトリを設定するときに考えておくと良いことがいくつか あります。あなたが、リポジトリ

[*7] たとえば **svnadmin setlog** は、 とにかくフックインターフェースを迂回するような方法で呼び出されるのでした。

[*8] ご存知でしょうか — 彼女のすべての「きまぐれ」をあらわす集合名詞です。

管理者としていくつかのプロジェクトのバージョン 管理システムのサポート責任者になったとしましょう。最初の判断は 複数プロジェクトに対して一つのリポジトリを使うか、プロジェクトごとに リポジトリを用意するか、その両者の折衷案でいくかです。

複数プロジェクトのために一つのリポジトリを使うことにはいくつか 利点があまりす。一番はっきりしているのは、重複した保守作業が不要だと いうことです。一つのリポジトリは、一組のフックスクリプト、一つの定期バックアップ、Subversion のリリースが両立不可能な新しいバージョンに なったときの、一回のダンプとロード、しか必要ありません。 また、プロジェクト間のデータ移動は簡単ですし、履歴バージョン情報を 失うことなしにやることができます。

一つのリポジトリを使うデメリットは、異なるプロジェクトは異なる コミットメーリングリストを持っていたり、異なる認証、許可などが 必要であるかも知れないことです。 また、Subversion はリポジトリグローバルなリビジョン番号を使っている ことに注意してください。人によっては、変更が自分のプロジェクトに 何もないのに、他のプロジェクトが活発に新しいリビジョンを追加する ことによって、最新リビジョン番号がカウントアップされていくのが 好きではないかも知れません。

折衷策をとることもできます。たとえば、お互いに どの程度深く関係しているかによってプロジェクトをグループ化する ことができます。それぞれのリポジトリにいくつかのプロジェクトを持たせる ことで、少ない数のリポジトリを管理することもできます。この方法では データを共有したいプロジェクトは簡単にそうすることができますし、 新しいリビジョンがリポジトリに追加されると、開発者は そのような新しいリビジョンは、自分のプロジェクトか、少なくともそれに 関係しているプロジェクトの誰かがやったものだということがわかります。

リポジトリに関係してどのようにプロジェクトを編成するかを決めたあとは 多分、リポジトリ自身のディレクトリ構成を考えたいと思うでしょう。 Subversion は普通のディレクトリコピーをブランチ化にもタグ付けにも 使うので (第 4 章参照)、Subversion のコミュニティ では、以下のようなディレクトリ構成を推奨しています; プロジェクトルート — プロジェクトに関連 したデータのある「最上位」ディレクトリのこと — ごとにリポジトリの場所を選択します; 次いでそのルートの 下に三つのサブディレクトリを作ります: `trunk` これはプロジェクトの主な開発が行われるディレクトリです; `branches` これは主な開発ラインから分岐したさまざまな名前の付いたブランチを作る ための場所です; `tags` これは作成され、削除される かも知れませんが、決して修正はされないようなブランチを入れるための ディレクトリです。 [*9]

たとえば、リポジトリが以下のようであるとして:

```
/
   calc/
      trunk/
      tags/
      branches/
   calendar/
      trunk/
      tags/
      branches/
   spreadsheet/
      trunk/
      tags/
```

[*9] `trunk`, `tags`, `branches` の三つのファイルの全体を 「TTB ディレクトリ」 と呼ぶことがあります。

```
      branches/
   …
```

それぞれのプロジェクトルートがリポジトリ中のどこにあるかは問題には なりません。もしリポジトリに唯一のプロジェクトがある場合は それぞれのプロジェクトルートを置くための論理的な場所はプロジェクト ごとのリポジトリのルートになります。もし複数のプロジェクトが ある場合は、リポジトリ内部のグループ中にそれを配置したいかも知れません、 おそらく同じサブディレクトリ中の似たような目標や共有するコードと一緒にプロジェクトを置くか、あるいは名前の辞書順にグループ化するか、 などです。配置は以下のようになるでしょう:

```
/
   utils/
      calc/
         trunk/
         tags/
         branches/
      calendar/
         trunk/
         tags/
         branches/
      …
   office/
      spreadsheet/
         trunk/
         tags/
         branches/
      …
```

良いと思われる方法でリポジトリをレイアウトしてください。 Subversion はレイアウトの構成について何も仮定しません — Subversion は、ディレクトリであってディレクトリ以外の何者でも ありません。結局、リポジトリのレイアウトは、それを利用する人々の 必要に応じたふさわしい方法を選んでください。

5.5.2 レイアウトの作成と、初期データのインポート

リポジトリ中でのプロジェクトのレイアウトが決まったら、 そのレイアウトの形にリポジトリを構成して、プロジェクトの初期データを ロードしたいと思うでしょう。これにはいろいろな方法があります。 一つ一つリポジトリレイアウトに従ってディレクトリを 作るのに、**svn mkdir** コマンドを使うことができます (第 9 章参照)。 もっと手っ取り早いのは、**svn import** コマンドを 使うことです。(項 3.8.2 参照) 最初にディスクの一時的な場所にレイアウトを作っておいて、その全体を 一回のコミットでリポジトリにインポートすることができます:

```
$ mkdir tmpdir
```

```
$ cd tmpdir
$ mkdir projectA
$ mkdir projectA/trunk
$ mkdir projectA/branches
$ mkdir projectA/tags
$ mkdir projectB
$ mkdir projectB/trunk
$ mkdir projectB/branches
$ mkdir projectB/tags
…
$ svn import . file:///path/to/repos --message 'Initial repository layout'
Adding         projectA
Adding         projectA/trunk
Adding         projectA/branches
Adding         projectA/tags
Adding         projectB
Adding         projectB/trunk
Adding         projectB/branches
Adding         projectB/tags
…
Committed revision 1.
$ cd ..
$ rm -rf tmpdir
$
```

svn list コマンドでインポート結果を 確認することができます:

```
$ svn list --verbose file:///path/to/repos
      1 harry               May 08 21:48 projectA/
      1 harry               May 08 21:48 projectB/
…
$
```

骨組みとなるレイアウトができて、もし既にインポートしたい データが存在しているならそれをリポジトリにインポートすることができます。 これにも、やはりいろいろな方法をとることができます。 **svn import** を使うかも知れません。新しいリポジトリから 作業コピーをいったんチェックアウトして、作業コピー中でデータを移動したり 編成しなおしてから、**svn add** と **svn commit** コマンドを使うこともできます。しかし、いったんそのような 話を始めると、もう既にリポジトリ管理については議論しません。 もし、まだ **svn** クライアントプログラムに なじみがないのなら、 第 3 章を参照してください。

5.6 まとめ

ここまでで、あなたは、どうやって Subversion リポジトリを作成し、設定するか についての基本的な理解ができたはずです。この作業を助けるさまざまなツール を紹介しました。そして、章全体を通じて、管理者がよくハマりそうなことをあげ、 どうやってそれを避けるかを議論しました。

あとは、リポジトリに、どのようなデータを入れ、それをネットワーク越しに 利用できる形にするかを考えるだけです。次の章ではネットワーク利用について 詳述します。

第 6 章

サーバの設定

6.1

Subversion リポジトリは `file:///`方式でリポジトリのある 同じマシン上で実行されている複数のクライアントから同時にアクセスすること ができます。しかし典型的な Subversion の設定はオフィス全体 — あるいは 全世界にあるコンピュータ上のクライアントからアクセスされる一台のサーバ上 で行います。

この章ではリモートクライアントを使ってホストマシンの外部にさらされる 形の Subversion リポジトリの作り方についての説明です。ここでは現在 Subversion で利用することのできるサーバの仕組みを説明し、その設定方法と 利用方法について説明します。この章を読んだ後であればどのタイプのネットワーク 設定が自分のニーズとって正しいものであるかを決め、どうやれば自分のホスト コンピュータ上でその設定が有効になるかについて理解できるはずです。

6.2 概観

Subversion は抽象的なネットワーク層の設計を含んでいます。これはリポジトリ に対してどのようなタイプのサーバプロセスからもアクセスできるようにプログラムを 作ることができ、クライアントの「リポジトリアクセス」API を使えば、プログラマは それに関連したネットワークプロトコルで通信することのできるプラグインを書くこと ができる、ということを意味します。理論的には Subversion は無数のネットワーク 実装が可能なはずです。ただしこれを書いている現時点では実際には二つの サーバがあるだけです。

Apache は非常に有名なウェブサーバです; **mod_dav_svn** モジュール を使えば Apache はリポジトリにアクセスすることができ、WebDAV/DeltaV プロトコル 経由でクライアントにもリポジトリを利用させることができます。これは HTTP の 拡張の一つです。もう一つの方法は **svnserve** です: これは 非常に小さい、スタンドアロンのサーバプログラムでクライアントとの間で独自の プロトコルを使って通信します。Table 6-1 に二つのサーバの比較をのせまし た。

Subversion はオープンソースプロジェクトの性質上、どのようなタイプのサーバ も「最重要なもの」であるとか、「公式のもの」 である として勧めたりすることはありません。またどのようなネットワーク実装についても 副次的な価値しかないものとして扱うこともありません; それぞれのサーバは それぞれの長所と短所があります。実際、複数の異なるサーバを並行して動作させ、 それぞれの方法でリポジトリにアクセスし、お互いの邪魔をすることがないように 設定できます。(項 6.6 を見てください)。 表 6.1 には、二つの利用可能な Subversion サーバの簡単な説明と比較があります — 管理者は、自分とそのユーザにとって最良の動作をする構成を自由に選ぶ ことができます。

表 6.1 ネットワークサーバの比較

機能	Apache + mod_dav_svn	svnserve
認証オプション	HTTP(S) 基本認証, X.509 認証, LDAP, NTLM, その他 Apache httpd で利用可能な方法	CRAM-MD5 または SSH
ユーザアカウントオプション	固有の 'users' ファイル	固有の 'users' ファイルまたは既存のシステム (SSH) アカウント
認可のオプション	自由な読み書きアクセス、あるいはディレクトリごとの 読み書き制御	自由な読み書きアクセス、あるいはフックスクリプトに よるディレクトリごとの書き込み (読み込みは不可) アクセス制御
暗号化	オプションの SSL を経由することで	オプションで SSH トンネルを利用することで
相互運用性	部分的に他の WebDAV クライアントからも利用可能	相互運用不能
ウェブによる参照	制限された組み込みサポート機能、あるいは ViewCVS のようなサードパーティーのツール経由	ViewCVS のようなサードパーティーツール経由
スピード	やや遅い	やや速い
初期設定	やや複雑	かなり簡単

6.3 ネットワークモデル

この節は実際に利用する具体的なネットワーク実装にかかわらず、 Subversion クライアントがどのようにしてサーバと通信するかの一般的な議論です。 この節を読み終えた後では、クライアントの応答についての設定によって サーバがどんな風に違った形で振る舞うかについて詳しく理解していることでしょう。

6.3.1 要求と応答

Subversion クライアントはほとんどの時間を作業コピーの管理に費やします。 しかしリポジトリからの情報が必要な場合にはネットワーク要求を発行し、 これに対してサーバが適切に応答します。ネットワークプロトコルの詳細は ユーザからは隠されています; クライアントは URL にアクセスしようとし、 URL スキーマの種類によって特定のプロトコルがサーバとの通信に利用され ます (リポジトリの URL)。ユーザは **svn –version** を実行して Subversion クライアントがどの URL スキーマとプロトコル を利用できるかを知ることができます。

サーバプロセスがクライアント要求を受け取ると、普通はクライアントの認証を 要求します。クライアントに対して認証確認を実行し、クライアントは 認証証明を提示することでこれに答えます。いったん 認証が成功すればサーバはクライアントがそもそも要求していた情報を返します。 このシステムは CVS のようなシステムとは異なっていることに注意してください。 CVS などではクライアントは要求を出す前に、あらかじめ認証証明を (「ログインによって」) サーバに送ります。Subversion ではサーバは適当な時点 でクライアントにチャレンジの仕組みによって認証証明を 「要求」 します。 クライアントが自発的にサーバに 「送りつける」 わけではありません。 これはある種の操作をより洗練されたものにします。たとえばもしサーバが 世界中の

誰でもそのリポジトリを読めるように設定すれば、クライアントが **svn checkout** を実行するときに認証確認を実行せずに 済みます。

クライアントネットワーク要求が新しいデータをリポジトリに書き込む場合 (たとえば **svn commit**) 、新しいリビジョンツリーが 作成されます。もしクライアント要求が認証されれば認証されたユーザ名は 新しいリビジョンの `svn:author` 属性の値として 格納されます (項 5.2.2 参照)。もしクライアント が認証されなければ (言い換えるとサーバが認証確認に失敗すれば)、その リビジョンの `svn:author` 属性は空となります。 [*1]

6.3.2 クライアント証明のキャッシュ

多くのサーバは要求ごとに認証を要求するように設定されます。 これはユーザにとっては大きな苦痛となることがあります。常にパスワード を入力しなくてはならないからです。

ありがたいことに、Subversion クライアントはこれに対する処方箋が あります: ディスク上での認証証明をキャッシュするための組み込みシステム があります。デフォルトではコマンドラインクライアントがサーバに対する 認証に成功したときは常にユーザの実行時環境領域にその証明を保存します — この場所は Unix 系システムでは `/.subversion/auth/`、 Windows であれば `%APPDATA%/Subversion/auth/`に なります。(実行時領域については、項 7.2 により詳しい説明があります)。成功した証明はディスクにキャッシュされ ホスト名、ポート、認証方式の組み合わせをキーとして保存されます。

クライアントが認証確認を受けたとき、まずディスクキャッシュにある証明を探し ます; 存在しないかキャッシュされた証明が認証に失敗した場合はクライアントは ユーザに入力を求めるプロンプトを出します。

セキュリティー狂なら思うかも知れません、「パスワードをディスクにキャッシュ するだと? ひどすぎる。絶対やめろ!」 と。まあ落ち着いて。それは見かけほど 危険な状態ではありません。

- `auth/` のキャッシュ領域はパーミッションで保護されている ので (所有者である) ユーザだけがそのデータを読むことができ、誰でもというわけでは ありません。オペレーティングシステムのファイル所有権限はパスワードによって保護 されています。
- Windows 2000 とそれ以降の場合、Subversion クライアントは標準的な Windows の 暗号サービスを利用してディスク上のパスワードを暗号化します。暗号キーは Windows によって管理されユーザ固有のログイン認証に結びついているのでそのユーザだけが キャッシュされたパスワードを復号化できます。(注意: ユーザの Windows アカウント パスワードが変更された場合、キャッシュされているすべてのパスワードは復号化不能 になります。Subversion クライアントはそれが存在していないかのような動作をし、 必要に応じてパスワードの入力をうながします。)
- すべての利便性を犠牲にしてまでセキュリティーを確保したいという、本当にスジガネ入り のセキュリティー狂には、すべての認証キャッシュを完全に無効にすることも可能です。

単一のコマンド中でキャッシュ を無効にする場合は `--no-auth-cache` オプションを渡して ください:

```
$ 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 .
```

[*1] この問題は実際によくある質問で、サーバの設定に間違いがあると起こります。

```
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
…
```

あるいは、証明のキャッシュをずっと無効にし続けたい場合は実行時 `config` ファイルを編集してください (`auth/`ディレクトリの隣にあります)。 単に `store-auth-creds` を `no` に 設定すればディスク上に証明書をキャッシュしなくなります。

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

ときどき特定の証明をディスクキャッシュから削除したくなることがあります。 これには `auth/`領域を調べて適当なキャッシュファイルを 手で削除してください。証明は個別のファイルにキャッシュされています; それぞれ のファイルの中にはキーとその値があります。 `svn:realmstring` キーはファイルが関係している特定のサーバの認証範囲を記録しています:

```
$ 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
```

適切なキャッシュファイルを特定し、それを削除してください。

クライアント認証について最後に一つ: `--username` と `--password` オプションについての説明が少し必要です。たくさんのクライアントサブコマンドはこれらのオプションを 受け付けます; しかしこのようなオプションはサーバに自動的に証明を 送るのではないことに注意してください。既に 説明したようにサーバは必要に応じてクライアントに証明の提示を「要求」 します; クライアントから「自発的に提示する」ことはできないのです。 もしユーザ名とパスワードがオプションとして渡された場合でも、それは サーバが要求したときにのみ提示されるのです。 [*2] 典型的にはこれのようなオプションは以下のような場合に利用されます:

- ユーザは自分のログイン名称とは違うユーザで認証を 受けたいか、
- あるスクリプトがキャッシュされている証明なしに 認証を受けたい。

最後にどのようにして Subversion クライアントが認証確認を受けたときに 振る舞うかをまとめておきます:

1. ユーザがコマンドラインオプション中で`--username` または`--password` を通じて何らかの証明を指定しているか どうかを確認します。指定していないか、オプションによる認証が失敗した 場合には、
2. 実行時 `auth/`領域中でサーバの認証範囲を探して ユーザが既に適切な証明をキャッシュしているかどうかを調べます。 もしそうでないかキャッシュされた証明が認証に失敗した場合はさらに、
3. ユーザに対して証明の入力をうながします。

クライアントが上記のどれかの方法で認証に成功した場合はディスク上にその 証明をキャッシュしようとします (既に述べたように、ユーザがこの動作 を無効にしない限り、そうします)。

6.4 svnserve, 専用サーバ

svnserve プログラムは軽量なサーバで専用の状態プロトコル によって TCP/IP 上でクライアントと通信することができます。クライアントは `svn://`または `svn+ssh://` で始まる URL によって **svnserve** サーバと通信します。この節 では **svnserve** を実行する別の方法を説明しクライアントが どうやってサーバに認証するか、またリポジトリに適切なアクセス制御を設定 するにはどうしたら良いかについて説明します。

6.4.1 サーバの起動

svnserve プログラムの起動にはいくつかの異なる方法が あります。オプションなしで起動した場合は何もせずヘルプメッセージを 表示するだけです。しかし **inetd** 経由で起動するなら `-i`(`--inetd`) オプションを指定すること ができます:

```
$ svnserve -i
( success ( 1 2 ( ANONYMOUS ) ( edit-pipeline ) ) )
```

`--inetd` オプション付きで起動すると **svnserve** は Subversion クライアントとの間 で、専用のプロトコル

[*2] ここでもよくある間違いは認証確認を決して要求しないようにサーバを 間違って設定してしまうというものです。この場合ユーザが `--username` と `--password` オプション をクライアントに渡しているのにそれが利用されないという状況に驚くでしょう。 つまり新しいリビジョンは依然として匿名でコミットされているように見える わけです!

を使い、*stdin* と *stdout* チャンネル経由で通信しようとします。 これは **inetd** を経由して 実行されるプログラムの標準的な振る舞い方です。IANA はポート 3690 を Subversion プロトコルのために予約しているため Unix 風のシステム上 なら`/etc/services` ファイルに (もしまだ追加されていない のなら) 以下の行を追加することができます:

```
svn             3690/tcp    # Subversion
svn             3690/udp    # Subversion
```

そしてもし伝統的な Unix 風の **inetd** デーモンを使って いるのなら `/etc/inetd.conf` に以下のような行を 追加することができます:

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

「svnowner」はリポジトリにアクセスするのに適切な パーミッションをもったユーザであることを確認してください。 これでクライアントがサーバのポート 3690 に接続しにきた時点で **inetd** は **svnserve** プロセスを起動し、処理を任せます。

Windows システムでは **svnserve** をサービスとして 起動するためのサードパーティーのツールがあります。このようなツールの 一覧については Subversion のウェブサイトを見てください。

第二の方法は **svnserve** を単独の「デーモン」プロセスとして 起動する方法です。これには`-d` オプションを使って ください:

```
$ svnserve -d
$               # svnserve is now running, listening on port 3690
```

デーモンモードで **svnserve** を実行するときには `--listen-port=`と `--listen-host=`オプションで待ち受けポートとホスト 名を「指定」することができます。

さらに **svnserve** を起動する第三の方法があり、 それは「トンネルモード」と呼ばれますが、`-t` オプションを付けて起動します。このモードは **RSH** や **SSH** のようなリモートサービスプログラムがユーザを 正しく認証しそのユーザでプライベートな **svnserve** サーバを起動している状況を仮定しています。 **svnserve** プログラムは普通に振る舞い (*stdin* と *stdout* を通じて)、通信データは自動的にクライアントの背後にいる何らかのトンネル にリダイレクトされると仮定しています。**svnserve** が このようなトンネルエージェントによって起動された場合は認証ユーザは リポジトリデータベースファイルに完全な読み書きアクセスを持つことに注意 してください。(サーバとパーミッション: 留意点を参照してください)。 これは本質的には `file:///`URL を使ってリポジトリにアクセス するローカルユーザと同じになります。サーバとパーミッション: 留意点

まず、Subversion リポジトリはデータベースファイルの集まり であることを思い出してください; リポジトリにアクセスするプロセスは すべてリポジトリ全体に対して適切な読み書きのパーミッションを持って いる必要があります。このことに注意していないと、いろいろな問題に 悩むことになります。とくに FSFS ではなく Berkeley DB データベースを 使っている場合はそうです。項 6.6 をよく読んで ください。

次に、**svnserve** を設定するときには Apache の **httpd** あるいは他のどのようなサーバプロセスも サーバプロセスを `root` ユーザで (あるいはパーミッションに制限のない ユーザならどのユーザでも) 起動したくは ない

ということを忘れないでください。あなたが公開しようとするリポジトリ の所有者とパーミッションに応じて、普通は異なる — 多分専用の — ユーザ を使うの賢明でしょう。たとえば多くの管理者は svn という名前の新しいユーザを作り、公開する Subversion リポジトリに対して 排他的な所有と権利を与え、そのユーザでのみサーバプロセスを起動します。

一度 **svnserve** プログラムが実行されるとネットワーク越しに システム上のすべてのリポジトリが利用可能になります。クライアントは リポジトリ URL の絶対パスを指定する必要があります。 たとえば、リポジトリが/usr/local/repositories/project1 にあるならクライアントは svn://host.example.com/usr/local/repositories/project1 によってそこにアクセスするでしょう。セキュリティー を高めるため **svnserve** に -r オプションを渡すこともできますが、これはそのパス以下のリポジトリだけを公開 するように制限します:

```
$ svnserve -d -r /usr/local/repositories
…
```

-r オプションの利用は リモートファイルシステム空間のルートとしてプログラムが扱う場所 を効果的に変更することができます。この場合クライアントはそのルート までの部分を除いたパスを指定することになり、もっと短い (そしてより情報制限された) URL を利用できます:

```
$ svn checkout svn://host.example.com/project1
…
```

6.4.2 組み込みの認証と認可

クライアントが **svnserve** プロセスに接続する とき、以下のことが起こります:

- クライアントは特定のリポジトリを選択します。
- サーバはリポジトリの conf/svnserve.conf ファイルを処理しその中に定義されている認証と認可の方式に強制的に 従います。
- そのときの状況と認可の方式により、以下のどれかになります。
  - クライアントは要求を匿名で行うことができ、どのような認証確認も 要求されないか、
  - クライアントは常に認証許可を求められるか、
  - もし”トンネルモード”で実行されている場合であれば、クライアント は既に外部的に認証されたことを宣言するか、です。

これを書いている時点では、サーバは CRAM-MD5 [*3] 認証確認の方法だけを知っています。本質的にサーバはクライアント に対して少しのデータを送ります。 クライアントは MD5 ハッシュのアルゴリズムを使ってデータとパスワードを一 緒にしたデータについてのフィンガープリントを作成し、これを応答メッセー ジとして送信します。サーバは同じ計算を保存してあるパスワードについて おこない結果が同じであることを確認します。いかなる場合でも ネットワーク上に実際のパスワードが流れることはありません。

[*3] RFC 2195 を参照してください

もちろんクライアントはトンネルエージェント、たとえば **SSH** のようなものを経由して外部的に 認証することもできます。この場合サーバは単に実行している ユーザを確認し、それを認証されたユーザ名であるとして 利用します。より詳しくは 項 6.4.3 を見てください。

もうおわかりだと思いますが、リポジトリの `svnserve.conf` ファイルは認証と認可の方式を制御する中心的な仕組みです。このファイルは他の 設定ファイルと同じ形式をしています。(項 7.2 参照): セクション名は角かっこ ([and]) で示され、コメントはハッシュ文字 (#) で始まり、セクションの それぞれには設定可能な特定の変数が含まれています。(`variable = value`)。 このファイルを見てどのように利用されているか理解してください。

6.4.2.1 ユーザファイルと認証範囲の作成

ここでは `svnserve.conf` の `[general]` セクションに必要な変数のすべてがあります。 ユーザ名とパスワードを含むファイルの定義で始まり、認証範囲を設定 しています:

```
[general]
password-db = userfile
realm = example realm
```

`realm` は自分で定義できる名前です。 それはクライアントに接続先の「認証用の名前空間」の種別を伝えます; Subversion クライアントは認証プロンプトでそれを表示し、ディスク上の キャッシュされた証明のキーとして (サーバのホスト名、ポートと共に) 利用します。(項 6.3.2 参照。) `password-db` 変数はユーザ名称とパスワードのリスト を含む個別のファイルを指す変数で、やはり同じ形式が利用されます。 たとえば:

```
[users]
harry = foopassword
sally = barpassword
```

`password-db` の値はユーザファイルの相対または 絶対パスです。多くの管理者にとって、`svnserve.conf` に従ったリポジトリの `conf/`領域にファイルを 正しく保つのは容易なことです。一方、同じユーザファイルを共有するような 二つ以上のリポジトリがほしいこともあります; そのような場合は ファイルは多分もっと公開された場所に移動すべきでしょう。ユーザファイルを共有 するリポジトリは同じ認証範囲を持つよう設定されていなくてはならず、それは ユーザ全員が本質的にただ一つの認証範囲を定義するためです。 ファイルがある場所であればどこでもファイルの読み書きパーミッション を正しく設定してください。もし **svnserve** を どのユーザが実行しているかわかるのであれば、必要に応じて ユーザファイルに対する読み出しアクセス制限をかけてください。

6.4.2.2 アクセス制御の設定

`svnserve.conf` ファイル中に、さらに二つの変数を 設定できます: それは認証されていない (匿名の) ユーザと、認証された ユーザに何を許すかを決めるものです。その変数 `anon-access` と `auth-access` は `none`、`read`、 あるいは `write` に設定できます。 `none` はどのようなタイプのアクセスも制限します。`read` はそのリポジトリに読み出し許可のみを与え、 `write` はリポジトリに完全な読み書きアクセスを許します。 たとえば:

```
[general]
password-db = userfile
realm = example realm

# anonymous users can only read the repository
anon-access = read

# authenticated users can both read and write
auth-access = write
```

この例としての設定は、実際にはこれらの変数のデフォルト値なので定義 しなくても問題ありません。もしさらに保守的に設定したいのなら、 匿名のアクセスを完全に遮断することもできます:

```
[general]
password-db = userfile
realm = example realm

# anonymous users aren't allowed
anon-access = none

# authenticated users can both read and write
auth-access = write
```

svnserve は単に「無制限の」アクセスコントロールのみ を理解することに注意してください。ユーザは完全な読み書きアクセス、 完全な読み出しアクセス、あるいは、まったくアクセスできない、のいずれか です。リポジトリ中の特定のパスに対する詳細なアクセス制御は 存在しません。多くのプロジェクトとサイトではこのレベルのアクセス制御 は十分すぎるものです。しかしもしディレクトリごとのアクセス制御が必要 なら、Apache を **mod_authz_svn** と一緒に使うか (項 6.5.4.2 を見てください)、書き込み制御を 行う **pre-commit** フックスクリプトを使う必要があります (項 5.3.1 を見てください)。Subversion の ディストリビューション中には **commit-access-control.pl** と、さらに洗練された **svnperms.py** スクリプトがあって、 pre-commit スクリプトの中で利用することができます。

6.4.3 SSH 認証と認可

svnserve の組み込み認証は非常に使いやすいものですが、それは本当のシステム上のアカウントを作る必要がないからです。一方 管理者によっては既に確立された SSH 認証の仕組みを運用しているかも 知れません。そのような場合、プロジェクトユーザのすべてはシステムアカウント を持っており、サーバマシンに対して「SSH による」 アクセスが可能なはず です。

SSH と **svnserve** の組み合わせは簡単なものです。クライアントは 単に `svn+ssh://URL` スキーマを使って接続することができます:

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harry@host.example.com's password:  *****

foo
bar
baz
...
```

この例では、Subversion クライアントはローカルな **ssh** プロセスを起動し `host.example.com` に接続し、ユーザ `harry` として認証し、そのあとプライベートな **svnserve** プロセスをリモートマシン上で、ユーザ `harry` として実行する、というものです。 **svnserve** コマンドはトンネルモード (`-t`) 起動され、そのネットワークプロトコルはトンネルエージェントである **ssh** によって暗号化された接続上で「トンネル」された形で動作します。 **svnserve** はユーザ `harry` で実行されていることを知って いるのでクライアントがコミットしようとすると、その認証済みのユーザ名は新しい リビジョンの変更者として利用されます。

ここで重要なのは Subversion クライアントは **svnserve** デーモンに接続するわけではないということです。このアクセス方法はデーモンは不要で、存在しているかどうかを知る必要も ありません。実際には **ssh** が一時的に起動する **svnserve** プロセスにだけ依存していて、ネットワーク接続が閉じるとそのプロセスも終了します。

`svn+ssh://`の URL を使ってリポジトリにアクセスする 場合、認証を要求するのは **ssh** プログラムであり **svn** クライアントプログラムではないことを思い出して ください。これは自動的なパスワードのキャッシュが起きないことを 意味します (項 6.3.2 を見てください)。 Subversion クライアントはリポジトリに複数の接続を張ることもよくあります がユーザはパスワードキャッシュの仕組みによって通常そのことに気づくことは ありません。しかし `svn+ssh://` URL を使う場合には ユーザは接続ごとに **ssh** が繰り返しパスワードをうながす ことに悩ませられるかも知れません。解決策は Unix 風のシステムなら **ssh-agent**、Windows なら **pageant** のような独立した SSH パスワードキャッシュツールを利用することです。

トンネル上で実行する場合、認可は基本的にはリポジトリデータベースファイルに 対するオペレーティングシステムのパーミッションによって一義的には制御 されます; それはちょうど harry が直接 `file:///` URL で リポジトリにアクセスした場合と同じことになります。 複数のシステムユーザがリポジトリに対して直接アクセスしようとしている場合 そのようなユーザを一つのグループにまとめ、umask を注意して設定する必要が あるでしょう。(項 6.6 をぜひ読んでください)。 しかしトンネルモードを利用する場合でも `auth-access = read` または `auth-access = none` と設定すれば、 `svnserve.conf` ファイルはやはりアクセス遮断のために 利用できます。

SSH トンネルの話はこれで終わりかと思うかも知れませんが、そうではありません。 Subversion では実行時 `config` ファイル中に専用のトンネル モードに関する動作設定をすることができます。(項 7.2 を見てください)。たとえば SSH のかわりに RSH を使いたいとします。`config` ファイルの `[tunnels]` セクションに以下のように指定してください:

```
[tunnels]
rsh = rsh
```

これで新しい変数の名前にマッチする URL スキーマを使って この新しいトンネル定義を利用することができます: `svn+rsh://host/path` となります。 新しい URL スキーマを利用すると Subversion クライアントは実際には裏で **rsh host svnserve -t** コマンドを実行します。もし URL にユーザ名が含まれている場合 (たとえば `svn+rsh://username@host/path`) クライアントはやはりそのコマンドに含めます (**rsh username@host svnserve -t**.) しかし、以下のようにもっと賢いトンネルスキーマを定義することもできます:

```
[tunnels]
joessh = $JOESSH /opt/alternate/ssh -p 29934
```

この例はいろいろなことの参考になります。まずそれはどのようにして Subversion クライアントが非常に特殊なトンネリングのためのプログラムを 特定のオプション付きで起動するかを示しています (この場合それは `/opt/alternate/ssh` にあります)。 この場合 `svn+joessh://` URL にアクセスすると 引数として `-p 29934` の付いた特定の SSH プログラム が起動されるでしょう — もし標準ではないポートにトンネルプログラム を接続したいと考えているならこれは便利です。

次にそれはどのように してトンネルプログラムの名前を上書きする環境変数を定義してやれば 良いかを示しています。 `SVN_SSH` 環境変数を設定するのはデフォルトの SSH トンネル エージェントを上書きする便利な方法です。 しかしもし異なるサーバ上でいくつもの異なる上書きが必要で、それぞれが 異なるポートや異なるオプションを SSH に渡しているような場合には、この例で 示すような仕組みを利用することができます。 もし `JOESSH` 環境変数を設定 してあれば、その値はトンネル変数全体を上書きします — **$JOESSH** は **/opt/alternate/ssh -p 29934** のかわりに実行 されるでしょう。

6.4.4 SSH 設定の技法

クライアントが **ssh** を起動する方法を制御できるだけ ではなく、サーバマシン上の **sshd** の動作の仕方も 制御することができます。この節では **sshd** によって起動される **svnserve** コマンドを正しく制御する 方法を示して、複数のユーザが単一システムアカウントをどのように共有すれば 良いかについて説明します。

6.4.4.1 初期設定

まず **svnserve** を起動するのに使うアカウントの ホームディレクトリを用意します。そのアカウントに SSH の公開鍵/秘密鍵 がインストールされていて、ユーザがその公開鍵でログインできることを 確認してください。パスワード認証は動作しなくなりますが、それは以下の SSH の技法を使うと、すべての処理に SSH `authorized_keys` ファイル を使うためです。

まだ存在していなければ `authorized_keys` ファイルを 作ってください (Unix では普通 `/.ssh/authorized_keys` になります)。このファイルの各行には接続を許す相手先の公開鍵の記述があります。 各行は普通以下のような形をしています:

```
ssh-dsa AAAABtce9euch.... user@example.com
```

最初のフィールドはキーの型で、二番目のフィールドは uuencode された鍵そのもの であり、三番目のフィールドはコメントです。あまり知られていませんが、実は 行全体を `command` フィールドの後におくこともでき

ます。

```
command="program" ssh-dsa AAAABtce9euch.... user@example.com
```

command フィールドが設定されると 通常の **svnserve -t** のかわりに SSH デーモンがその名前のプログラムを実行します。このプログラムが Subversion クライアントの接続先になります。これがサーバ上でのいろいろな技法を 可能にする鍵です。以下の例では、ファイル中で次のように行を省略して説明します:

```
command="program" TYPE KEY COMMENT
```

6.4.4.2 起動コマンドの制御

実行されるサーバ側コマンドを指定することができるので、特定の **svnserve** バイナリを指定したり、追加の引数を 指定して実行することが簡単にできます:

```
command="/path/to/svnserve -t -r /virtual/root" TYPE KEY COMMENT
```

この例では /path/to/svnserve は **svnserve** に対するカスタマイズされたラッパースクリプ トで、umask を設定するようなものかも知れません (項 6.6 を見てください)。それはまた **svnserve** 用の仮想ルートディレクトリをどのように設定 するかも示しています。これはデーモンプロセスとして **svnserve** する場合によく起こることです。たとえばシス テムの特定の部分にアクセス制限する場合や、単に svn+ssh:// URL の絶対パス名を入力する手間を省くため であったりします。

複数のユーザが単一アカウントを共有するようにもできます。それにはまずユー ザごとに独立したシステムアカウントを作るかわりに、メンバーごとに公開鍵 /秘密鍵のペアを生成します。つぎに一行に公開鍵をひとつづつ authorized_users ファイルにおきます。そして --tunnel-user オプションを使うとうまくいきます。

```
command="svnserve -t --tunnel-user=harry" TYPE1 KEY1 harry@example.com
command="svnserve -t --tunnel-user=sally" TYPE2 KEY2 sally@example.com
```

この例では Harry も Sally も公開鍵認証方式によって同じアカウントで接続する ように設定しています。どちらもそれぞれにカスタマイズされたコマンドが実行されます; --tunnel-user オプションは **svnserve -t** が名前つき引数が認証されたユーザであることを認めるように指示しています。 --tunnel-user がなければ、すべてのコミットはひとつの共有された システムアカウントから発行したように見えるようになります。

最後の注意です: 共有アカウントにある公開鍵を経由してユーザにアクセス権を 与えても、他の形の SSH アクセスを禁止したことにはなりません。これは authorized_keys に command の 形の設定をした場合でもそうです。たとえば、ユーザは依然として SSH 経由で シェルを使ったアクセスができますし、あなたのサーバ経由で X11 や、より 一般的なポートフォワードを実行することもできます。ユーザにできるかぎり わずかな権限しか与えないようにするには command の すぐ後にそれぞれの制限オプションを指定する必要があり

ます:

```
command="svnserve -t --tunnel-user=harry",no-port-forwarding,\
        no-agent-forwarding,no-X11-forwarding,no-pty \
        TYPE1 KEY1 harry@example.com
```

6.5 httpd, Apache HTTP サーバ

Apache HTTP Server は 「非常にいろいろなことをしてくれる」 ネットワークサーバ で Subversion の機能も上げることができます。カスタムモジュールを使って **httpd** は Subversion リポジトリを WebDAV/DeltaV プロトコル 経由でクライアントから利用可能にします。WebDAV/deltaV プロトコルは HTTP 1.1 の拡張です (<http://www.webdav.org/> により詳しい情報が あります)。このプロトコルはワールドワイドウェブの核心である、広く利用可能な HTTP プロトコルに対して、書き込み — 特にバージョン化された 書き込み — 機能を付け加えます。結果は標準化された、堅牢なシステム を構成することができ、それは Apache 2.0 の一部としてパッケージ化されて います。また Apache 2.0 はさまざまなオペレーティングシステムとサード パーティー性製品によってサポートされており、それを利用すればネットワーク 管理者は新たなカスタムポートを開く必要がありません。 [*4] Apache-Subversion サーバは **svnserve** よりも多くの 機能を持っていますが、セットアップは少し難しくなります。柔軟性には しばしば複雑さがともなうものです。

以下の議論の多くは Apache の設定ディレクティブへの参照を含んでいます。 いくつかの例はそのようなディレクティブの利用方法になっていますが、 その完全な説明はこの章の範囲外です。Apache チームは非常にすばらしい ドキュメントを管理していて<http://httpd.apache.org> から自由に参照可能です。たとえば設定ディレクティブの一般的な リファレンスは<http://httpd.apache.org/docs-2.0/mod/directives.html> にあります。

また、Apache の設定を変更する場合、しばしば間違いが起こります。 Apache のログシステムにまだなじみがないのであれば、それに注意すると 良いでしょう。`httpd.conf` ファイルには Apache に よって生成されるアクセスログとエラーログのディスク上での場所を指定する ディレクティブがあります。(それぞれ `CustomLog` と `ErrorLog` という名前です)。Subversion の mod_dav_svn も Apache のエラーログインターフェースを利用しています。これらのファイルは 情報取得のために常に閲覧することができ、ほかの方法ではは っきりしない 問題の原因を明らかにするかも知れません。なぜ Apache 2 が必要なのか?

もしあなたがシステム管理者なのであれば、既に Apache ウェブサーバを 実行しており、かなりの経験を持っているかも知れません。このドキュメント を書いている時点で Apache 1.3 は Apache の最も有名なバージョンです。 いくつかの理由で、Apache 2.X 系へのアップグレードは世界的にはゆっくり としか進んでいません: 人によっては変更点、特にウェブサーバにとって 非常に重要な部分についての変更点を怖がります。また Apache 1.3 API で のみ動作するプラグインモジュールに依存していて、2.X 系への移植を待って いるような人もいるでしょう。どんな理由であれ、多くの人々は Subversion の Apache モジュールは Apache 2 API でのみ動作することを最初に知った とき、不安を感じます。

この問題に対する正しい反応は: 心配するな、です。Apache 1.3 と Apache 2 は同時に起動することができます: 単に別の場所にインストールし、Apache 2 を 80 以外のポートで実行する Subversion 用のサーバとすれば良いのです。 クライアントは URL にポート番号を指定することでリポジトリにアクセスできます:

[*4] 彼らはそういう作業を本当に嫌います。

```
$ svn checkout http://host.example.com:7382/repos/project
…
```

6.5.1 必須要件

HTTP 越しにリポジトリにアクセスする場合、基本的には二つのパッケージで利用 可能な四つの部品が必要になります。Apache **httpd** 2.0、 それに付属している **mod_dav** DAV モジュール、Subversion、 そしてそれに付属している **mod_dav_svn** ファイルシステム 提供モジュールです。すべての部品を手に入れてしまえばリポジトリのネットワーク 対応は以下のように簡単です:

- httpd 2.0 を起動し、mod_dav モジュール付きで実行する。
- mod_dav_svn プラグインを mod_dav にインストールする。mod_dav_svn はリポジトリにアクセスするために Subversion のライブラリを利用 します。そして、
- `httpd.conf` ファイルを設定してリポジトリを 公開する。

最初の二つについては **httpd** と Subversion を ソースコードからコンパイルするか、自分のシステム用の既にコンパイル 済みのバイナリパッケージをインストールすることによって取得できます。 どのようにして Apache HTTP サーバと共に Subversion をコンパイルするか、 そしてこの目的のために Apache 自身をどのように設定すれば良いかに ついての最新情報は Subversion ソースコードツリーの最上位にある `INSTALL` ファイルを参照してください。

6.5.2 基本的な Apache の設定

システム上に必要なすべての部品をインストールしたあとは、`httpd.conf` によって Apache の設定をすることだけが残っています。 `LoadModule` ディレクティブを使って mod_dav_svn モジュールを Apache にロードしてください。この ディレクティブはほかの Subversion 関連の設定項目に先立って指定しなくてはなりません。 Apache がデフォルトレイアウトを使ってインストールされているなら、**mod_dav_svn** モジュールは Apache インストールディレクトリの `modules` サブディレクトリ 中になければなりません (たいていの場合、`/usr/local/apache2` の ようなディレクトリになります)。`LoadModule` ディレクティブは単純な構文を持ち、名前の付いたモジュールをディスク上の共有ライブラリの場所に対応付けます:

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

mod_dav が (**httpd** プログラムに直接静的にリンクされるのではなく) 共有オブジェクトとしてコンパイルされた場合、それに対しても同様の `LoadModule` 行が必要になります。 **mod_dav_svn** 行の前に設定することに注意してください:

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

次に、設定ファイルの後の場所のどこかで Subversion リポジトリをどこに 置くかを Apache に伝える必要があります。`Location` ディレクティブは XML 風の記述で、開始タグで始まり、終了タグで終わる間 にさ

まざまなほかの設定ディレクティブを書きます。 Location ディレクティブの目的は Apache に、指定した URL かそのサブディレクトリである特定の処理をするように指示するためにあります。 Subversion の場合、DAV 層で管理するバージョン化された資源のある URL で 処理を単に引き渡すように Apache に指示するだけです。Apache に対して /repos/で始まる部分 (つまり、URL のサーバ名と 場合によって付随するポート番号文字列の後に続く部分) を持ったすべての URL について、/absolute/path/to/repository にあるリポジトリを管理する DAV 提供モジュールに引き渡すように指示することが できます。それには以下のような httpd.conf 構文を使い ます:

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

ローカルディスク上の同じ親ディレクトリにある複数の Subversion リポジトリ を提供する計画がある場合は、別のディレクティブ、 SVNParentPath を使って共通の親ディレクトリを示すこともできます。たとえば http://my.server.com/svn/repos1 とか、 http://my.server.com/svn/repos2 のような URL を経由してアクセスされる/usr/local/svn ディレクトリ 中に複数の Subversion リポジトリを作る場合であれば、以下の例の中にある httpd.conf の設定構文を使うことができます:

```
<Location /svn>
  DAV svn

  # any "/svn/foo" URL will map to a repository /usr/local/svn/foo
  SVNParentPath /usr/local/svn
</Location>
```

この構文を使うと Apache は/svn/で始まるパス部分 を持つすべての URL を Subversion DAV モジュールに渡しますが、すると このモジュールは SVNParentPath によって指定される ディレクトリ中のすべてのアイテムは実際の Subversion リポジトリであると 仮定します。これは SVNPath ディレクティブを利用 するのとは違って新しいネットワーク公開用リポジトリを作るたびに Apache を再起動する必要がないのでとても便利です。

新しい Location を定義する場合は、他の公開された Location と重ならないように注意してください。たとえばメインの DocumentRoot が/www に 設定されている場合、Subversion リポジトリを<Location /www/repos>の中で公開しないでください。 URI /www/repos/foo.c が要求されても Apache は DocumentRoot 中にある repos/foo.c を探せば良いのか、Subversion リポジトリから foo.c を返すために **mod_dav_svn** に取り次げば良いのか 判断できなくなります。サーバ名と COPY 要求

Subversion はサーバ側でのファイルやディレクトリのコピーを するために COPY 要求を利用します。Apache モジュール での信頼性チェックの一環として、コピー元はコピー先と同じマシン上に存在 していなくてはなりません。この要求を満足させるためにはサーバのホスト名 として利用する名前を mod_dav に伝える必要があります。通常これには httpd.conf 中に ServerName ディレクティブを使うことができます。

```
ServerName svn.example.com
```

NameVirtualHost ディレクティブを使って Apache の仮想ホスト機能を利用している場合はサーバを特定するための 追加名称を指定するのに ServerAlias ディレクティブ を使う必要があるかも知れません。やはり詳細については Apache のドキュメントを参照してください。

この時点で、パーミッションがどうなるかというについて十分考慮する ことが必要になります。Apache をある程度の期間にわたって通常利用する Web サーバとしてきた場合、おそらく既にいろいろなコンテンツがある ことでしょう — ウェブページ、スクリプト、などなど。これらの アイテムは既に Apache と協調動作するようなパーミッションの組が設定 されている、あるいはもっと正確には、Apache にそのようなファイルを 扱うことを許可する設定になっています。Subversion サーバとして Apache が利用される場合も、Subversion リポジトリに対して正しい 読み書きのパーミッションを設定する必要があります。(詳しくは サーバとパーミッション: 留意点を見てください)。

既に存在しているウェブページやスクリプトの設定に問題を起こさない ように Subversion の要求を満足させるためのパーミッションを決定しなくて なりません。これは Subversion リポジトリを、Apache が既に あなたに対して提供しているほかのサービスと協調するようなパーミッション に変更するか、あるいは httpd.conf の中で User や Group ディレクティブを 使って Subversion リポジトリを所有しているユーザ・グループで Apache が実行されるべき状態に変更することを意味します。 このための唯一の正しい解法といったものはありませんし、個々の 管理者は正しいやり方をするための異なる理由を持っているはずです。 パーミッションに関連した問題はおそらく Apache を利用した Subversion リポジトリの設定時に一番よく見落とされることであるのに注意してください。

6.5.3 認証オプション

この時点で httpd.conf を以下のような 感じで設定している場合

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

あなたのリポジトリは「匿名で」 世界中からアクセス可能となります。何らかの 認証と認可の仕組みを設定するまで、あなたの作った Subversion リポジトリは Location ディレクティブによって一般的に誰からもアクセスすることができてしまいます。言い換えると、

- 誰でもリポジトリ URL(とその任意のサブディレクトリ) の作業コピーをチェックアウト するために Subversion クライアントを利用することができます,
- 誰でもリポジトリ URL をブラウザで指定することによってリポジトリの最新 リビジョンを閲覧することができます。そして
- 誰でもそのリポジトリにコミットすることができます。

6.5.3.1 基本 HTTP 認証

クライアントを認証する一番簡単な方法は HTTP の基本認証の仕組みを使うことで、それは単純にユーザ名とパスワードを使って、ある人間が自分がその当人である と言っているのを確認します。Apache は **htpasswd** ユーティリティー を用意して、受け入れることのできるユーザ名とパスワードの一覧を管理 しますが、その人たちにだけあなたの Subversion リポジトリにアクセスする権利を 与えることができます。 Sarry と Harry にだけコミット権限を与えてみましょう。 まず彼らをパスワードファイルに追加する必要があります。

```
$ ### First time: use -c to create the file
$ ### Use -m to use MD5 encryption of the password, which is more secure
$ htpasswd -cm /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth-file sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

次に新しいパスワードファイルを何に利用するかというのを Apache に伝える ため、 `Location` ブロック内部で追加の `httpd.conf` ディレクティブが必要になります。 `AuthType` ディレクティブは 利用する認証システムのタイプを指定します。今回は `Basic` 認証システムを指定したいと思います。`AuthName` は 任意の名前で認証ドメインを与えるためのものです。ほとんどのブラウザは ユーザに名前とパスワードを問い合わせるときにこの名前をポップアップダイアログ ボックス中に表示します。最後に `AuthUserFile` ディレクティブは **htpasswd** で作ったパスワードファイルの場所を指定します。

三つのディレクティブを追加した後では、あなたの `<Location>` ブロックは以下のような感じになっていることでしょう:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
</Location>
```

この `<Location>` ブロックはまだ完成しておらず 役に立つことは何もしません。単に Apache に対して、認証が要求されるとき には常に Subversion クライアントからユーザ名とパスワードを取得するように 言うだけです。しかしここで欠けているのは Apache に対してどのような種類のクライアント要求が認証で必要とされるのかを言うための ディレクティブです。これをやるのに最も簡単な方法はすべてのリクエストを保護 す

るこです。`Require valid-user` の追加は Apache に対して すべてのリクエストは認証されたユーザであることを伝えます:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

認可のポリシーを設定する `Require` ディレクティブと、その他の方法についての 詳細については次の節 (項 6.5.4) を読んでください。

一点注意があります: HTTP の基本認証パスワードはほとんど平文のままネットワーク を流れるため、セキュリティー上は非常に弱いものです。もしパスワードの 盗聴が心配なら、SSL 暗号化のような仕組みを使うのが最良でしょう。これで クライアント認証は `http://`のかわりに `https://`を使って認証することになります; 最低限度の 処置として Apache に自己サイン付きサーバ証明書を設定することができます。 [*5] どうすれば良いかについては Apache のドキュメント (と、OpenSSL の ドキュメント) を見てください。

6.5.3.2 SSL 証明書の管理

リポジトリを自社ファイアウォールの外にさらす必要のあるビジネス は認可されていない他人が自分たちのネットワークデータを「盗聴」 しているかも知れないということを意識すべきです。 SSL はこの手の望ましくない意図が重要なデータの流出に帰結する可能性を 小さなものにします。

Subversion クライアントが OpenSSL を使ってコンパイルされた場合、`https://` URL を使って Apache サーバと通信する能力を得ます。Subversion クライアントで 利用される Neon ライブラリはサーバ証明書を検証することができるだけ ではなく、確認要求を受けた場合には自分の証明書を提示する能力も持って います。クライアントとサーバが SSL 証明書を交換しお互いの認証に成功 すれば、その後のすべての通信はセッションキーによって暗号化されます。

どのようにしてクライアントとサーバ証明書を生成するか、またその証明書を 利用するようにどうやって Apache を設定するかについてはこの本の範囲外です。 Apache 自身のドキュメントを含め、さまざまな本でこの方法を説明しています。 ここでは通常の Subversion クライアントでのサーバとクライアント証明書を どのように管理するかについて説明します。

`https://`経由で Apache と通信する場合、Subversion クライアントは二つの異なるタイプの情報を受け取ることができます:

- サーバ証明書
- クライアント証明書の提示要求

クライアントがサーバ証明書を受け取った場合、それが信頼できるものであるか どうかの検証が必要になります: サーバは本当に名乗っているそのサーバなの でしょうか? OpenSSL ライブラリはサーバ証明書にサインし

[*5] 自己サイン付きサーバ証明書は、「中間偽装」攻撃に対しては やはり脆弱ですが、そのような攻撃は、暗号化されていないパスワードを 盗聴するタイプのものに比べてはるかに困難です。

た者、あるいは 認証期間 (CA) を調べることでこれを確認します。もし OpenSSL が自動的に CA を信用することができないか、他の問題が起きた 場合 (たとえば、証明書の有効期間が過ぎていたり、ホスト名が一致していない 場合など)、Subversion コマンドラインクライアントはそのサーバ証明書を とにかく信用するかどうかをユーザに聞いてきます:

```
$ 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?
```

このダイアログはなじみ深いものだと思います; 本質的にはウェブブラウザで 見ることのできるのと同じ質問になっています (ブラウザは Subversion と 同じような HTTP クライアントの一種なんです!)。もし (p)ermanent、常に信用 する、というオプションを選ぶと、サーバはあなたのユーザ名とパスワードを キャッシュしたのとちょうど同じ方法であなたの実行時 `auth/` 領域にそのサーバ証明書をキャッシュします。(項 6.3.2 を参照してください)。キャッシュされてしまえば、Subversion はそれ以降の やり取りについては自動的にこの証明書を信用します。

実行時 `servers` ファイルも Subversion クライアントが自動的に 特定の CA を信頼するように設定することができます。すべてのものについて そうすることもできますし、ホストごとにすることもできます。 単に `ssl-authority-files` を、 PEM で暗号化された CA 証明書をセミコロンで区切ったリストに設定してください:

```
[global]
ssl-authority-files = /path/to/CAcert1.pem;/path/to/CAcert2.pem
```

多くの OpenSSL の設定ではほとんど無制限に信頼する 「default」 CA が、あらかじめ設定されています。Subversion クライアントにそのような 標準的な認証機関を信用させるためには `ssl-trust-default-ca` 変数を `true` に設定してください。

Apache と通信する際、Subversion クライアントはクライアント証明書の 確認要求を受けるかも知れません。Apache はクライアントに対して自分自身を 証明するようにたずねます: あんたは本当にあんたなのか? もしすべてが正し ければ Subversion クライアントは Apache が信用している CA によってサイン されたプライベート証明書を送り返します。クライアント証明書は通常 暗号化された形式でディスク中に保管され、ローカルパスワードによって 保護されています。Subversion がこの確認要求を受けた場合、あなたは 証明書のパスと、それを保護しているパスワードについて聞かれます:

```
$ svn list https://host.example.com/repos/project

Authentication realm: https://host.example.com:443
Client certificate filename: /path/to/my/cert.p12
Passphrase for '/path/to/my/cert.p12':  ********
…
```

クライアント証明書は 「p12」形式のファイル であることに注意してください。クライアント証明書を Subversion で利用する場合、それは標準的な PKCS#12 フォーマットでなければなりません。 ほとんどのウェブブラウザは既にその形式の証明書をインポートしたり エクスポートしたりすることができます。他の方法としては既存の証明書 を OpenSSL のコマンドラインツールによって PKCS#12 形式に変換するという ものです:

ここでも実行時 `servers` ファイルはホスト単位で この確認要求を自動化することを認めています。そのような情報は 実行時変数で指定できます:

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = /path/to/my/cert.p12
ssl-client-cert-password = somepassword
```

いったん `ssl-client-cert-file` と `ssl-client-cert-password` 変数を 設定すれば、Subversion クライアントはユーザに問い合わせることなしに 自動的にクライアント証明書の確認要求に応答することができるようになります。 [*6]

6.5.4 認可のオプション

ここまでのところで、すでに認証についての設定は完了しました が認可はまだです。Apache はクライアントを試し、本当のクライアン トであることを確認することができますが、これらの認証済みクライアントそれぞれにどのようなアクセスを許し、また制限するかについてはまだ説明 していません。この節ではリポジトリに対してアクセス制御するための 二つの方法について説明します。

6.5.4.1 全面的なアクセス制御

アクセス制御の一番簡単な方法は特定のユーザをリポジトリに対して読み出し 専用、あるいは読み書き可能として認可することです。

`<Location>`ブロックに `Require valid-user` ディレクティブを追加することによってすべてのリポジトリ操作にアクセス制限を 設けることができます。前の例を使うと、これは `harry`、 `sally`、あるいはユーザごとの正しいパスワードを入力した 人だけに、Subversion リポジトリに対する任意の操作を許すというものです:

[*6] セキュリティーにもっと神経質な人はクライアント証明書用パスワード を実行時 `servers` ファイルに格納するのを嫌がる でしょう。

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

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # only authenticated users may access the repository
  Require valid-user
</Location>
```

しばしばそのような厳しい設定は不要です。たとえば Subversion 自身の ソースコードリポジトリは <http://svn.collab.net/repos/svn> にありますが、世界中の誰でも読み出しアクセスすることが可能です (それはチェックアウトしたり、ウェブブラウザでリポジトリを閲覧する ような操作です) が、書き込み操作は認証されたユーザにのみ許されています。 この手の制限を付与するには Limit と LimitExcept 設定ディレクティブを使うことができます。 Location ディレクティブのように、この二つの ブロックは開始タグと終了タグがあり、 <Location> ブロック中でネストすることができます。

Limit と LimitExcept ディレクティブ に現れるパラメータは HTTP 要求タイプで、そのブロック全体に影響を 与えます。たとえば、現在サポートされている読み出しのみの操作を 除くすべてのリポジトリアクセスを禁止したい場合、LimitExcept ディレクティブが、GET、PROPFIND、 OPTIONS、そして REPORT 要求タイプ パラメータを渡す形で利用できます。そして既に触れた Require valid-user ディレクティブを、単に<Location>ブロックの中に 置くかわりに、<LimitExcept>ブロックの中に置く形に なります。

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

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # For any operations other than these, require an authenticated user.
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

このようなことは単純な例にすぎません。Apache のアクセス制御と Require ディレクティブについてのさらに詳しい情報 は<http://httpd.apache.org/docs-2.0/misc/tutorials.html> にある Apache ドキュメントチュートリアルの Security セクションを見てください。

6.5.4.2 ディレクトリごとのアクセス制御

第二の Apache httpd モジュールである **mod_authz_svn** を使うと、より詳細なパーミッションの設定が可能です。このモジュールは クライアントからサーバに送信されるさまざまな裸の URL を取得し、 **mod_dav_svn** にそれを解析するように要求し、設定ファイル で定義されたアクセス方式に基づき必要に応じてアクセスを拒否します。

Subversion をソースコードから構築した場合は **mod_authz_svn** は自動的に **mod_dav_svn** のそばにインストールされます。 多くのバイナリ配布でもやはり自動的にインストールします。正しくインストール されているかどうかを確認するには httpd.conf にある、**mod_dav_svn** の LoadModule ディレクティブのすぐ後に来ていることを確認してください:

```
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
```

このモジュールを有効にするには AuthzSVNAccessFile ディレクティブを使うために Location ブロックを 設定する必要があります。このディレクティブはリポジトリにある パスのパーミッションが書かれたファイルを指定します。(すぐあとで このファイルの形式について議論します。)

Apache は柔軟なので三つの一般的なパターンのどれかにブロックを 設定することができます。まず基本的な設定パターンの一つを選びます。 (以下の例は非常に単純です; Apache の認証と認可の設定の詳細に ついては Apache 自身のドキュメントを参照してください。)

最も単純なブロックはすべての人に対して自由にアクセスすることを 許すものです。このやり方では Apache は認証要求を送信することは ないのですべてのユーザは「匿名」として扱われます。

例 6.5.1 匿名アクセスの設定例。

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

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file
</Location>
```

この対極にある設定方法として、すべての人にたいして認証要求するための ブロックを設定することもできます。すべてのクライアントは自身を特定 するための証明を送る必要があります。ブロックは Require valid-user ディレクティブによって無条件に認証を要求し、またその方法を定義します。

三番目の非常に一般的な方法は認証つきアクセスと匿名アクセスの組合せ によるものです。たとえば多くの

例 6.5.2 認証つきアクセスの設定例。

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

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # only authenticated users may access the repository
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

管理者はあるリポジトリのディレクトリを 誰でも読めるようにしたいが、もっと重要な場所については認証された ユーザのみが読めるように (あるいは書き込めるように) したいと考えます。 このような設定ではすべてのユーザはまずは匿名でリポジトリにアクセスし ます。ある時点で本当のユーザ名を要求しなくてはならないアクセスが 発生すると、Apache はクライアントから認証を要求します。このためには `Satisfy Any` ディレクティブと `Require valid-user` ディレクティブの両方を使います。

いったん基本的な `Location` ブロックが設定されて しまえばその中にアクセスファイルを作り、認証の規則を定義すること ができます。 アクセスファイルの形式は **svnserve.conf** や 実行時設定ファイルで利用されるのと同じです。 ハッシュ文字 (#) で始まる行は無視されます。 一番単純な形では、それぞれのセクションはリポジトリ とのその中にあるパスの名前を決め、認証用のユーザ名はセクションごと の中にくるオプション名になります。それぞれのオプションの値は リポジトリパスにアクセスするユーザレベルを記述します; `r` (読み込み専用) か、 `rw`(読み書き可能) のどちらか になります。ユーザがまったく含まれていなければ、アクセスは全面的 に禁止されます。

もっと具体的に言うと; セクション名は `[repos-name:path]` か、`[path]` の形になります。`SVNParentPath` ディレクティブを使っている場合はセクション中でリポジトリ名を指定するのが 重要です。それを省略すると `[/some/dir]` のようなセクション は すべてのリポジトリのパス `/some/dir` にマッチしてしまいます。しかし `SVNPath` ディレクティブ を使っている場合はセクションで唯一のパスを定義するのが良い方法です — 結局そこには唯一のリポジトリしか無いのですから。

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

例 6.5.3 認証つき/匿名の両方でアクセスする場合の設定例。

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

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # try anonymous access first, resort to real
  # authentication if necessary.
  Satisfy Any
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

この最初の例ではユーザ harry は calc リポジトリ中の /branches/calc/bug-142 ディレクトリに対して完全な読み書きアクセスが可能ですが、 sally は読み出し アクセスのみです。それ以外のユーザにはこのディレクトリのアクセスは 禁止されます。

もちろんパーミッションは親ディレクトリから子ディレクトリに継承 されます。これは Sally のために、サブディレクトリ中では異なる アクセス方式を指定することができるという意味です:

```
[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
```

これで Sally はブランチの testing サブディレクトリ では書き込みができますが、ディレクトリのほかの部分では依然として読み出し のみが可能です。一方 Harry はブランチ全体に対して依然として 完全な読み書きアクセスが可能です。

ユーザ名変数を設定しなければ、他の人を継承の規則に従って許可する のを明示的に拒否することもできます:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

[calc:/branches/calc/bug-142/secret]
harry =
```

この例では Harry は `bug-142` のツリーに 対して完全な読み書きアクセスが可能ですが、その中のサブディレクトリ `secret` にはまったくアクセスできません。

留意しておくことは、一番詳しく指定したパスが常に最初にマッ チするということです。**mod_authz_svn** モジュー ルはまず最初にパス自身にマッチするかどうかを調べ、次にその親 ディレクトリ、さらにその親ディレクトリ、と調べていきます。結 果はアクセスファイル中の具体的なパスが有効になると、親ディレ クトリから引き継いでいるパーミッション情報は常に上書きされて しまいます。

デフォルトでは、誰であれリポジトリに対するすべてのアクセスは禁止 されます。これは、もし空のファイルで始めた場合、リポジトリのルート ですべてのユーザに対して少なくとも読み出しパーミッションを与えたい だろうということを意味します。これはアスタリスク変数 (*) を使って、「すべてのユーザ」をあらわすことで可能です。:

```
[/]
* = r
```

これはよくある設定です; セクション名の中にリポジトリ名が存在しないことに 注意してください。これは `SVNPath` を使っていようが `SVNParentPath` を使っていようが、すべてのリポジトリが すべてのユーザによってどこからでも読み込めるようにします。 すべてのユーザがリポジトリに読み込みアクセスできるようになってしまえば 特定のリポジトリの特定のサブディレクトリに特定のユーザが読み書き可能と するため、明示的に `rw` の許可を与えることができます。

アスタリスク変数 (*) も特に注意しておく価値があります: それは匿名ユーザにマッチするような 唯一のパターン です。`Location` ブロックで匿名と認証されたアクセスの 組合せを許すように設定した場合、すべてのユーザは Apache に対して匿名で アクセスするところから話が始まります。 **mod_authz_svn** はアクセスするパスのために 定義された * の値を探します; みつからなければ Apache はクライアントに対して実際に認証要求を出します。

アクセスファイルでもユーザのグループ全体を定義することが てきます。これは Unix の`/etc/group` ファイルと 良く似た形式です:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = harry, sally, joe, frank, sally, jane
```

グループを使ってユーザと同じようにアクセス制御することができ、この場合 グループであることを示す

「アットマーク」(@) を先頭に付けます:

```
[calc:/projects/calc]
@calc-developers = rw

[calc:/projects/paint]
@paint-developers = rw
jane = r
```

グループは他のグループを含むように定義することもできます:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = @calc-developers, @paint-developers
```

... これでほとんどすべてです。

6.5.4.3 パス名にもとづいたチェックの禁止

mod_dav_svn モジュールには、「読み込み禁止」 のしるし がついたデータが間違って外部に漏れないようにいろいろな工夫がしてありま す。これは、 **svn checkout** や **svn update** のようなコマンドからの戻り値となるすべてのパス名とファイルの内容を綿密に チェックする必要があることを意味します。このようなコマンドが認可のポリシーに よって読み込むべきではないファイルパス名に出会うと、通常は完全にそれを 無視します。履歴や名称変更を追うようなコマンドの場合 — 例えばずっと 昔に名称変更されたファイルに対して **svn cat -r OLD foo.c** のようなコマンドを実行するような場合など — 名称変更の履歴は、その ようなファイルの以前の名前に読み込み制約がある場合には、単に異常終了して しまいます。

このようなすべてのパス名に対するチェックは場合によっては非常に効率の悪 いものになり、特に **svn log** コマンドではそうです。リビ ジョンの一覧を取得する場合、サーバはすべてのリビジョンのすべての変更されたパスを見てそれらが読み込み許可されているかどうかを調べます。許可さ れていないパスが見つかるとリビジョンの変更のあったパスの一覧からは除外 され (これは通常`--verbose` オプションで見ることのできる ものです)、ログメッセージ全体が表示されなくなります。言うまでもありま せんが、たくさんのファイルのあるリビジョンでは多くの時間を消費します。 しかしこれはセキュリティーを保つための代償です: **mod_authz_svn** のようなモジュールをまったく設定してい ない場合でも、やはり **mod_dav_svn** モジュールが Apache の **httpd** に対してすべてのパスについての認可チェック をするように要求します。**mod_dav_svn** モジュールは具体 的にどんな認可モジュールがインストールされているかは知らないので、単に Apache に対して、もしそのようなものがあるなら実行するようにと依頼する だけです。

一方、これに関する逃げ道もやはりあって、セキュリティーよりも効率を重視 するようにも設定できます。ディレクトリごとの認可の仕組みをまったく利用 しないのなら (たとえば **mod_authz_svn** やそれに類似 のモジュールを使わないのなら)、このパス名に対するチェックを完全に無効に することもできます。`httpd.conf` ファイルで、 `SVNPathAuthz` ディレクティブを使ってください:

`SVNPathAuthz` ディレクティブはデフォルトでは 「on」 です。「off」 に設定するとすべてのパス名にも

例 6.5.4 Disabling path checks altogether

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

  SVNPathAuthz off
</Location>
```

とづいた認可のチェックが禁止されます; **mod_dav_svn** はすべてのパスについて認可のチェックをしなく なります。

6.5.5 おまけ

ここまで Apache と mod_dav_svn のための認証と認可のオプショ ンの大部分を説明してきました。しかし Apache が用意している、さらに いくつかのすばらしい機能があります。

6.5.5.1 リポジトリ閲覧

Subversion リポジトリで Apache/WebDAV の設定による一番の恩恵は バージョン化されたファイルやディレクトリの最新リビジョンがウェブブラウザ から直接参照可能だということです。Subversion は URL をバージョン化された 資源を特定するために利用するので、そのような HTTP ベースのリポジトリアクセス に利用される URL はウェブブラウザから直接入力することが可能です。ブラウザは その URL に対して `GET` 要求を発行し、その URL がバージョン化 されたディレクトリであるかファイルであるかに応じて mod_dav_svn はディレクトリ の一覧またはファイルの内容を表示します。

URL は見たいと思うリソースのバージョンについての情報は含まれていない ので mod_dav_svn は常に最新のバージョンで答えます。この機能は Subversion URL をドキュメントの参照先として渡すことができ、その URL は常にドキュメントの最新を指すことになる、というすばらしい効果も あります。もちろん他のウェブサイトからのハイパーリンクとして URL を 利用することもできます。

一般的には、(ディレクトリへの URL に比べて) バージョン化されたファイルへ の URL のほうをよく使うことになるでしょう — 結局のところ、関心のある内容 がありそうな場所は、そこなのですから。 しかし、Subversion のディレクトリ一覧を参照する機会はあるかも知れず、 その場合すぐに一覧表示で生成される HTML は非常に基本的なもので、美の追求 (や、何か面白いことをやらかそうということ) を目的としているわけではないの に気づくでしょう。このディレクトリ一覧をカスタマイズするために、 Subversion は XML インデックス機能を用意しています。 `httpd.conf` 中で、リポジトリの `Location` ブロック中で単一の `SVNIndexXSLT` ディレクティブを使うと mod_dav_svn に対して、ディレクトリ一覧表示時に XML 出力を生成し、 好きな XSLT スタイルシートを参照するように設定することができます:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  SVNIndexXSLT "/svnindex.xsl"
```

```
  …
</Location>
```

SVNIndexXSLT ディレクティブとクールな XSLT スタイル シートを使ってディレクトリ一覧をウェブサイトのほかの部分で利用されている 色スキーマや画像に一致させることができます。あるいは、もし望むなら Subversion ソース配布中の tools/xslt/ディレクトリ にあるサンプルスタイルシートを使うこともできます。 SVNIndexXSLT ディレクトリで指定されるパスは実際の URL パスであることに注意してください — ブラウザはそれを利用するためにはスタイルシートが読める場所に なければなりません! 古いリビジョンを表示することはできますか?

普通のウェブブラウザを使って、ですか? それは、ノーです。少なくとも 唯一のツールである **mod_dav_svn** なしには。

あなたのブラウザは普通の HTTP しか理解することができません。つまり 公開された URL を GET コマンドで取得する方法しか理解できず、これは ファイルとディレクトリの最新バージョンを意味するからです。WebDAV/DeltaV の仕様によれば、それぞれのサーバは古いバージョンの リソースを取得るための固有の URL 構文を持ちその構文はクライアント には透過的に見えるというものです。古いバージョンのファイルを見つける にはクライアントは特定の手続きによって適切な URL を「発見」しなくてはなりません; その手続きはいくつかの WebDAV PROPFIND 要求を発行することとと、DeltaV の概念を含んでいます。これは ウェブブラウザで簡単にできるようなことではありません。

それで、最初の問に答えとしては、ファイルやディレクトリの古いバージョン を見るための明らかな方法の一つは、**svn list** と、 **svn cat** コマンドに --revision 引数を渡すことです。しかしウェブブラウザで古いリビジョンを参照 するには、サードパーティー製のソフトウェアが必要になります。良い例は ViewCVS(<http://viewcvs.tigris.org/>) です。ViewCVS は最初ウェブサーバで CVS リポジトリを表示するために 書かれましたが最新の開発バージョンでは (この文書を書いている時点では) Subversion リポジトリも理解することができます。

6.5.5.2 その他の機能

堅牢なウェブサーバとして Apache で既に提供されている機能のいくつかは Subversion においても機能とセキュリティーの向上につながります。 Subversion は Neon という SSL(安全なソケット層。既に述べました) や圧縮 (**gzip** や **PKZIP** と同じような アルゴリズムを使ってファイルより小さなデータの塊に「縮める」 こと) のような仕組みをサポートした一般的な HTTP/WebDAV ライブラリを使って Apache と通信します。やらなくてはならないことは 単に Subversion と Apache で必要な機能をコンパイルし、そのような機能を使える ように正しくプログラムを設定することだけです。

圧縮の仕組みは、実際のサイズを減らすためのネットワーク転送データの圧縮と 解凍処理でクライアントとサーバに少し負荷をかけます。ネットワークの帯域 が細い場合この圧縮はサーバとクライアント間の転送スピードを非常に大きく することができます。極端な場合このデータ転送量の現象は、操作がタイムアウト してしまうか、完了するかの違いになることさえあります。

それほどは面白くありませんが、やはり有用なことは Apache と Subversion に関係したその他の機能で、たとえば特定のポートを指定する機能 (デフォルト HTTP ポートである 80 のかわりに) や、Subversion リポジトリが アクセスする仮想ドメイン名の機能や、プロキシ経由でリポジトリにアクセス する能力などがあります。これらはすべて Neon でサポートされているので、 Subversion は自由にその機能を利用することができます。

最後に **mod_dav_svn** は、ほぼ完全な WebDAV/DeltaV の方言を利用することができるのでサードパーティー製の DAV クライアント を経由してリポジトリにアクセスすることもできます。ほとんどのモダンな オペレーティングシステム (Win32, OS X そして Linux) では標準ネットワーク を「共有」することによって DAV サー

バをマウントする組み込みの機能があります。 これは複雑な話題です; 詳細は付録 B を読んでください。

6.6 複数リポジトリアクセス方法のサポート

いろいろな異なる方法でどのようにしてリポジトリにアクセスするかを見て きました。しかし、複数の方法で同時にあるリポジトリにアクセスすることは 可能 — あるいは安全に — なのでしょうか? 答えはイエスです。少し ばかり慎重になる必要はありますが。

ある時刻において、以下のようなプロセスがあなたのリポジトリに対して 読み出しアクセス、あるいは書き込みアクセス要求を出しているかも知れません :

- Subversion クライアントを使って (自分自身の名のもとに)`file:///`URL を 指定することで直接リポジトリにアクセスする通常のユーザによって:
- リポジトリにアクセスするプライベートに SSH-起動された **svnserve** プロセスに接続する通常のシステムユーザによって:
- デーモンとして、または **inetd** によって起動された **svnserve** プロセスが、特定の固定されたユーザとして実行されることによって:
- Apache **httpd** プロセスが、特定の固定されたユーザとして実行 されることによって

管理者が直面する一番よくある問題はリポジトリの所有権とパーミッションです。 上のリストにあるすべてのプロセス (あるいはユーザ) は Berkeley DB ファイルに対する 読み書きの権限を持っていますか? あなたが Unix 風のオペレーティングシステム を使っているとして、一番素直な解決方法は、すべての潜在的なリポジトリユーザ を新しい `svn` グループに入れてしまい、リポジトリをその グループによって完全に所有されている形にしてしまうことです。しかし それだけでは十分ではありません。プロセスはデータベースファイルに、排他的な umask で書き込むかも知れません — それは他のユーザがアクセスするのを 妨害してしまいます。

それでリポジトリユーザの共通グループ設定の後の次のステップはすべてのリポジトリ アクセスプロセスで、正しい umask を使うことです。リポジトリに直接アクセスする ユーザの場合は **svn** プログラムをくるんでしまうスクリプトを 作り、その先頭で **umask 002** を設定してから実際の **svn** クライアントプログラムを実行すれば良いでしょう。同様のラッパースクリプトを **svnserve** についても書き、Apache の起動スクリプト `apachectl` の先頭にも **umask 002** コマンドを 追加しましょう。たとえば:

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

また別のよくある問題が Unix 風システムではよく起こります。 リポジトリが利用されると BerkeleyDB は必要におうじて新しい ログファイルを作り動作を記録します。リポジトリ自体は 完全に **svn** グループによって所有されていても このようにして新規に作られたファイルが同じグループによって 所有される必要はありません。これが理由でユーザにとっては また別のパーミッションの問題がおこります。うまい回避策としてはリポジトリのある `db` ディレクトリにたいして グループ SUID ビットを立てることです。これによって すべての新たに作成されるログファイルは親ディレクトリと 同じ所有グループになります。

いったんこの問題を乗り越えてしまえば、あなたのリポジトリは すべての必要なプロセスからアクセスすることができるようになって いるはずです。少し面倒で複雑ですが、複数のユーザが書き込みアクセス を共有することで起こる問題は、しばしばきれいに解くことができない 古典的な問題です。

ありがたいことに、ほとんどのリポジトリ管理者はそのような複雑な 設定をする必要はないでしょう。 同じマシン上にあるリポジトリにアクセスしたいと思うユーザは `file://`アクセス URL を利用することに制限されて いるわけではありません — `http://`や `svn://`URL 中にサーバ名として `localhost` を指定する形で Apache HTTP サーバや **svnserve** を使うこともできるのです。そして複数のサーバプロセスを Subversion リポジトリのために管理することは頭痛の種を増やすだけのことです。 自分のニーズに本当に合ったサーバ構成を選択し、そこにしがみついて 離れない、これがおすすめの方法です! svn+ssh:// サーバのチェックリスト

パーミッションの問題を回避してリポジトリを共有するように既存の SSH アカウントのある複数のユーザを設定するのはとても技巧的な話しになるかも しれません。Unix 風システム上で (管理者として) やらなくてはならない 作業に混乱した時にそなえてこの節で議論した内容をもう 1 度まとめて チェックリストにしておきます:

- すべての SSH ユーザはリポジトリに対して読み書きの権限を持っていなくては なりません。すべての SSH ユーザを一つのグループにまとめてください。 リポジトリ全体をそのグループの所有とし、グループパーミッションに 読み書き許可を与えてください。
- リポジトリにアクセスするためには、ユーザは正しい umask を使う必要があり ます。 **svnserve(**これは `/usr/bin/svnserve`、あるいは `$PATH` のどこかにあります) が、**umask 002** 設定後本当の **svnserve** バイナリを実行するプログラムで あることを確認してください。**svnlook** と **svnadmin** を使う場合にも同じ方法をとってください。 正しい umask で実行するか、上で説明したようにラッパースクリプトでくるんでください。

第 7 章

より進んだ話題

7.1

もし、この本を章ごとに、最初から最後まで読んでいるのなら、 もうあなたは、ほとんどのバージョン管理操作を実行するために Subversion クライアントを使うための十分な知識を持っているはずです。 どのようにして、Subversion リポジトリから作業コピーをチェックアウト するかを理解しているはずです。**svn commit** や **svn update** を使った変更点の送受信になじんでいる はずです。そして多分、ほとんど無意識に **svn status** を実行してしまうような反射神経さえ身についているかも知れません。 どんな意図や目的に対しても、典型的な環境で Subversion を使う用意が できているはずです。

しかし、Subversion の機能セットは、「普通のバージョン管理操作」 で終わるわけではありません。

この章ではいくつかの Subversion の機能で、それほど頻繁には利用されない ようなものをとりあげます。その中で、Subversion の属性 (あるいは「メタデータ」) のサポートについて議論し、どのようにして Subversion のデフォルトの振る舞いを実行時設定領域の調整によって変更 することができるかを見ます。また、どのように外部定義を使って、 複数のリポジトリからデータを引っぱってくるように Subversion に命令 するかを説明します。そして、Subversion のパッケージの一部である、 追加のクライアント側、サーバ側のツールのいくつかの詳細にも触れます。

この章を読む前に、Subversion で基本的なファイルとディレクトリに関する バージョン管理の能力についてなじんでいるべきです。もしまだそれについて は読んでいないか、復習が必要なら、第 2 章 と 第 3 章 を読むことをお勧めします。 一度基本をマスターしてからこの章を消化すれば、あなたはもう Subversion のパワーユーザです。

7.2 実行時設定領域

Subversion はたくさんのオプションの振る舞いを用意していて、それはユーザに よって制御することができます。そのようなオプションの多くはユーザがすべての Subversion 操作に適用したいと思うようなことです。それで、このようなオプション を指定するためにユーザにコマンドライン引数を思い出させるように強いるよりも また、実行しようとするすべての操作に対してそれらを使うよりも、Subversion は 定義ファイルを使います。それは Subversion の定義領域に分離されているものです。

Subversion の設定領域 は二層に分かれたオプション名と 値の階層です。普通、これは定義ファイル (最初の層) を含む特別なディレクトリに要約してあり、それは標準的な INI 形式の テキストファイルにすぎません。(そこには「sections」 があり、 それが第二層になります) これらのファイルは好きなテキストエディタを 使って簡単に編集することができます。(emacs とか vi とか) そして、クライアント によって読み出される命令を含んでいて、ユーザが好むさまざまなオプションの 振る舞いをどうするかを決定します。

7.2.1 設定領域のレイアウト

svn コマンドラインクライアントが最初に実行されると、それはユーザごとの構成領域を作ります。Unix 風のシステムなら、この領域は ユーザのホームディレクトリに、`.subversion` という名前の ディレクトリとして用意されます。Win32 システムでは、Subversion は `Subversion` という名前のフォルダを作ります。普通には ユーザプロファイルディレクトリ (これは通常は隠れたディレクトリになりますが) の `Application Data` 領域 の内部になります。しかし、このプラットフォームでは、完全な場所はシステム ごとに違っていて、本当の場所は Windows レジストリ [*1] に設定されています。 ユーザごとの設定領域は、Unix での名前である `.subversion` を使って参照することにします。

ユーザごとの設定領域に加えて、Subversion はシステム全体の設定領域も 理解することができます。これはシステム管理者にあるマシン上での すべてのユーザに対するデフォルトを設定する力を与えます。システム全体 の設定領域は必須のポリシーがあるわけではありません — ユーザ ごとの設定領域は、システム全体の領域を上書きし、**svn** プログラムに与えるコマンドライン引数は振る舞いを決める最後の場所に なります。Unix 風のプラットフォームでは、システム全体の設定領域は `/etc/subversion` ディレクトリにあると期待されて います。Windows マシンの場合は共通アプリケーション データ 領域の内部にある `Subversion` ディレクトリを見に行きます (このディレクトリも Windows レジストリによって指定されます)。 ユーザごとの場合と違って、**svn** プログラムは システム全体の設定領域を作ろうとはしません。

`.subversion` ディレクトリは現在のところ 三つのファイルを含んでいます — 二つの設定ファイル (`config` と `servers` です)、それに `README.txt` ファイルで、これは INI 形式を 説明するものです。それらの生成時には、ファイルは Subversion がサポートするそれぞれのオプションのデフォルト値が入っており、ほとんどがコメントアウトされていて、さらに、どのようにキーに対する 値が Subversion の振る舞いに影響するかということについて、テキストの 説明付きでグループ化されています。何かの振る舞いを変えるためには 関連する設定ファイルをテキストエディタで開き、必要なオプション値で 修正することだけが必要です。もし設定をデフォルトに戻したい場合は、いつでも単にその設定ディレクトリを 削除し、何か無害な **svn** コマンド、たとえば **svn –version** のようなものを実行することができます。 新しい設定用ディレクトリがデフォルト値を含む形で生成されます。

ユーザごとの設定領域は認証データのキャッシュも含みます。`auth` ディレクトリは Subversion でサポートされているさまざまな認証方法で利用される キャッシュ情報の要素を含むサブディレクトリの集まりを保持します。 このディレクトリはユーザ自身だけがその内容を読むことができるような形に 作成されます。

7.2.2 設定と、Windows のレジストリ

普通の INI ベースの設定領域に加えて、Windows プラットフォーム上で実行されて いる Subversion クライアントは Windows のレジストリも設定データを格納する場所 として利用することができます。オプション名とその値は INI ファイル中と同じ です。「file/section」 の階層関係も保存されます。わずかに異なる 方法によりますが — この方法では、ファイルとセクションは単にレジストリ キーのツリーの階層にしかすぎません。

Subversion はシステム全体の設定値を `HKEY_LOCAL_MACHINE`
`Software`
`Tigris.org`
`Subversion` キー の元で検索します。たとえば `global-ignores` オプション、 これは `config` ファイルの `miscellany` セクションにありますが、`HKEY_LOCAL_MACHINE`

[*1] `APPDATA` 環境変数は `Application Data` 領域を 指しているので、常にそのフォルダを `%APPDATA%` `Subversion` のように参照することができます。

`Software`
`Tigris.org`
`Subversion`
`Config`
`Miscellany`
`global-ignores` に見つけることができます。 ユーザごとの設定値は `HKEY_CURRENT_USER`
`Software`
`Tigris.org`
`Subversion`. の下に格納されるはずです。

レジストリベースの設定オプションは、ファイルベースの残りの部分を 検索する前に 検索されます。それで、 このようなオプションは、設定ファイル中で見つかった値によって上書き されます。言い換えると、設定のプライオリティは Windows システムの 場合、以下の順序となることが保証されています:

1. コマンドラインオプション
2. ユーザごとの INI ファイル
3. ユーザごとのレジストリ値
4. システム全体の INI ファイル
5. システム全体のレジストリ値

また、Windows レジストリは「コメントアウト」 のような概念をサポートしていません。しかし、Subversion は、 キーの名前がハッシュ文字 (#) で始まるような すべてのオプションを無視します。これで実際には Subversion のオプション を、レジストリから完全にキーを消さずにコメントアウトすることができます。 明らかに、そのオプションの設定作業は簡単にしています。

svn コマンドラインクライアントは Windows の レジストリに書き込むことは決してありませんし、そこにデフォルトの設定値を作ろうともしません。必要なキーは **REGEDIT** プログラムで作ることができます。他の方法 としては、`.reg` ファイルを作り、エクスプローラ シェルからそのファイルをダブルクリックすると、そのデータが レジストリにマージされます。

この例は、`.reg` の内容を示した例ですが、その 中には、よく利用される設定オプションの大部分とそのデフォルト値が あります。システムの設定 (たとえばネットワークプロキシに関するオプション) と、ユーザごとの設定 (利用するエディタ、パスワード、など) の両方が あることに注意してください。さらにすべてのオプションは、コメントアウト されていることにも注意してください。オプション名の先頭のハッシュ文字 (#) を取り除くだけで、望んでいる値に設定することが できます。

7.2.3 設定オプション

この章では、特定の実行時オプションについて議論します。現在 Subversion がサポートしているものについてです。

7.2.3.1 servers

`servers` ファイルは Subversion の設定オプションで、ネットワーク層に関係したものを含んでいます。二つのセクション名がその ファイルにはあります — `groups` と `global` です。`groups` セクションは、要するにクロスリファレンスの テーブルです。このセクションのキーは、ファイル中にある別のセクションの名前です。その値はグロブ — ワイルドカードを含んで いるかも知れないテキストトークンです — で、Subversion の要求が送信 されるマシンのホスト名称と比較されます。

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

[beanie-babies]
…

[collabnet]
…
```

Subversion がネットワーク越しに利用される場合、groups セクションにあるグループ名に合うサーバ名称とマッチするものを探します。 もしマッチした場合は Subversion は次に、その名前がグループ名称とマッチした servers ファイル中のセクションを探します。そして そのセクションから実際のネットワーク設定オプションを読み出します。

global セクションは groups セクション のどのグロブにも当てはまらなかったすべてのサーバに対する設定があります。このセクションで使えるオプションは、ファイルの別の サーバセクションで利用できるものとまったく同じです。(ただし、 もちろん、groups セクションは例外です) 以下のような感じです:

`http-proxy-host` これは、プロキシコンピュータのホスト名称で、HTTP ベースの Subversion はそこを通じて通信しなくてはなりません。デフォルトは空で、それは Subversion はプロキシを通して HTTP 要求せず、直接、目的のマシンと 通信しようとすることを意味しています。

`http-proxy-port` これは、利用するプロキシホストのポート番号を指定します。 デフォルトは空です。

`http-proxy-username` これは、プロキシマシンで必要なユーザ名を指定します。 デフォルトは空です。

`http-proxy-password` これは、プロキシマシンで必要なパスワードを指定します。 デフォルトは空です。

`http-timeout` これはサーバ応答を待つ時間の最大値を秒単位で指定します。 もし、Subversion の操作がタイムアウトしてしまうような低速の ネットワーク接続に関係した問題を抱えている場合、この オプションの値を増やしてみてください。デフォルト値は `0` で、この場合、HTTP ライブラリである Neon にデフォルトのタイムアウト値を使うように指示します。

`http-compression` これは、DAV が有効なサーバで、Subversion がネットワーク要求データを 圧縮するかどうかを指定します。デフォルト値は `yes` (ただし、圧縮はネットワーク層のコンパイル時に有効になっていなくては なりませんが、) です。`no` に設定すると圧縮は 無効になりますが、これはネットワーク転送のデバッグ時などに使います。

`neon-debug-mask` これは、整数値のマスクで、HTTP ライブラリ Neon がどのようなタイプの デバッグ出力を生成するかを指定するものです。デフォルトは `0` で、すべてデバッグ出力を無効にします。

Subversion が Neon をどのように使うかについての詳細は 第 8 章を見てください。

`ssl-authority-file` これは HTTPS 経由でリポジトリにアクセスするときに Subversion クライアントによって受け入れられる認証機関 (あるいは CA) の 証明書を含むファイルパスのリストをセミコロンで区切ったものになります。

`ssl-trust-default-ca` Subversion が自動的に OpenSSL に付いているデフォルトの CA を信用する ようにしたい場合にはこの変数を `yes` にしてください。

`ssl-client-cert-file` ホスト (あるいは何台かのホスト) が SSL クライアント証明書を要求する場合普通は証明書のあるパスを入力するようにユーザにうながします。そのパスを この変数に設定すると Subversion はユーザの入力なしにクライアント証明書 を自動的に探すことができるようになります。証明書をディスク上に保存する ための標準的な場所はありません; Subversion は指定したどのようなパスから でもそれを取得することができます。

`ssl-client-cert-password` SSL クライアント証明書ファイルがパスフレーズで暗号化されている場合 Subversion はその証明書を利用するたびにパスフレーズの入力を求めます。 これが嫌なら (そして `servers` ファイル中に 自分のパスワードを保存するのが嫌でなければ)、この変数に証明書の パスフレーズを設定することができます。これでパスフレーズを聞かれる ことはなくなります。

7.2.3.2 config

`config` ファイルは、Subversion 実行時 オプションのうち、現在利用できる残りのもので、ネットワークに関連するもの以外が含まれています。現時点ではいくつかのオプション が利用できるだけですが、今後の追加を考えて、別のセクションとして グループ化してあります。

`auth` セクションは Subversion のリポジトリに対する 認証と許可に関係した設定があります。それは:

`store-passwords` これは Subversion にサーバ認証チャレンジに対してユーザが 入力するパスワードをキャッシュするかどうかを指示します。 デフォルトは `yes` です。`no` に 設定するとディスク上でのパスワードのキャッシュを無効にします。 このオプションは **svn** コマンドインスタンスのどれか で `--no-auth-cache` を使うと上書きすることができます。 (あるいはこの引数をサポートしているコマンドであればどれでも)。 詳細は項 6.3.2 を参照してください。

`store-auth-creds` この設定は `store-passwords` と同じですが、ディスク上 にキャッシュする すべての認証情報を有効にしたり 無効にするところが違います。このような情報にはユーザ名、パスワードサーバ証明書、その他のキャッシュ可能な認証情報すべてが含まれます。

`helpers` セクションは Subversion がどの 外部アプリケーションをいくつかの処理で使うかを制御します。このセクションで有効なものは:

`editor-cmd` これは Subversion がコミット時のログメッセージを作るのにどのプログラムを 使うかを指定します。たとえば、**svn commit** が、 `--message` (`-m`) も `--file` (`-F`) オプション もなしで実行されたような場合です。このプログラムはまた **svn propedit** コマンドでも使います — 一時的なファイルに

ユーザが編集したいと思う現在の属性値が書き込まれますが、これは エディタの起動によって実行されます。(項 7.3 参照)。このオプションはデフォルトは空です。もしこのオプションが設定 されていないと Subversion は環境変数 SVN_EDITOR, VISUAL, と EDITOR (この順序で) を調べます。

`diff-cmd` これは差分表示プログラムの絶対パスを指定します。このプログラムは Subversion が「diff」の出力を生成するのに利用 されるものです (**svn diff** コマンド実行時 などです)。デフォルトで Subversion は内部的な差分ライブラリを 利用します — このオプションによって外部プログラムを使って 処理するようになります。そのようなプログラムの使い方の詳細は 項 7.9 を見てください。

`diff3-cmd` これはスリーウェイ差分プログラムの絶対パスを指定します。Subversion はこのプログラムをリポジトリから受け取った、ユーザがした変更点を マージするのに使います。デフォルトで Subversion は内部的な差分 ライブラリを利用します — このオプションを設定すると、外部プログラムを 使って処理を実行するようになります。そのようなプログラムの使い方の 詳細は 項 7.9 を見てください。

`diff3-has-program-arg` このフラグは `diff3-cmd` オプションが `--diff-program` パラメータを受け付ける場合には `true` を指定すべきです。

`tunnels` セクションでは **svnserve** と `svn://`クライアント接続を使った新しいトンネルスキーマを定義する ことができます。詳細は項 6.4.3 を参照してください。

`miscellany` セクションは他の場所に置けない すべてのものの置き場所です。[*2] このセクションには:

`global-ignores` **svn status** コマンドを実行すると Subversion は バージョン化されないファイルとディレクトリをバージョン化されて いるものと一緒に一覧表示します。このときバージョン化されていない ことを? 文字で表現します。 (項 3.6.3.1 参照)。 ときどき、あまり興味のないバージョン化されないアイテムが表示されるのを 見るのを面倒に思うことがあります。 — たとえば、プログラムのコンパイルによってできるオブジェクトファイル など — `global-ignores` オプションは空白で区切られたグロブの リストで、バージョン化されていないのであれば Subversion に表示して欲しく ないものの名前の指定になります。デフォルトは `*.o *.lo *.la #*# .*.rej *.rej .* * .#* .DS_Store` です。

svn status のほか、 **svn add**、**svn import** コマンドもリストにマッチするファイルを無視します。 どのコマンドでも `--no-ignore` フラグを使うとこのオプションをその実行に限って上書きできます。 無視するアイテムについてのもっと細かい制御については 項 7.3.3.3 を見てください。

`enable-auto-props` これは Subversion に新規追加またはインポートしたファイルの属性を 自動的に設定するように指示します。デフォルト値は `no` なので有効にするには `yes` にしてください。 このファイルの `auto-props` セクションはどの属性が どのファイル上に設定されるかを指定します。

`log-encoding` この変数はログメッセージをコミットするキャラクタセットのデフォルト エンコーディングを設定します。これは `--encoding` オプションを無条件に有効にするものです (項 9.2.1 参照)。Subversion リポジトリはログメッセージを UTF8 で保存し、あなたの ログメッセージはあなたのオペレーティングシステムの独自のロカールを使って 書き込まれることを仮定しています。他のエンコーディングで書いたメッセージ をコミットしたい場合には別のエンコーディングを指定すべきです。

[*2] 残り物でディナーはいかが?

`use-commit-times` 通常作業コピーのファイルはどんな操作をしたかにかかわらず最後に触った 時刻を反映したタイムスタンプを持ちます。これはエディタで編集したか、 他の **svn** サブコマンドを使ったかにはよりません。 これは普通はソフトウェア開発者にとって便利なものですが、それは ビルドシステムはよくどのファイルが再コンパイルを必要としているか を決めるのにタイムスタンプを見るためです。しかし別の状況ではリポジトリで変更された最終時刻を反映しているような タイムスタンプが作業ファイルに振られているほうが便利であることも あります。 **svn export** コマンドは常に、このような 抽出したツリー上の 「最終コミット時間」 を振りますが、この設定変数を `yes` にすると、**svn checkout**, **svn update**, **svn switch**, そして **svn revert** についても同様に、それらの操作が 触れたファイルに最終コミット時間を振るようになります。

`auto-props` セクションは Subversion クライアントが追加 またはインポートしたファイルの属性を自動的に設定する能力を制御します。 そこには `PATTERN = PROPNAME=PROPVALUE` の形をした 任意数のキー・属性値の組を置くことができますが、ここで `PATTERN` はファイル名称にマッチするような正規表現で、行の残りの部分は対応する属性と その値です。ファイルに複数の要素がマッチすれば、そのファイルに複数の属性が 設定されることになります; しかし設定ファイル中に列挙された auto-props が その順序で適用されるかどうかの保証はないので一つの規則で別の規則を 「上書き」することはできません。`config` ファイル中に、auto-props を使ったさまざまな例を見つけることができる でしょう。最後に、auto-props を有効にするには `enable-auto-props` を `yes` にするのを忘れずに。

7.3 属性

既に、Subversion がどのようにしてリポジトリ中にあるファルイやディレクトリの いろいろなバージョンを格納し、抽出するかを詳しく見てきました。 すべての章は Subversion というツールによって提供されているこの一番基本的な機能に ささげられてきました。そして、もしバージョン管理のサポートがそれで終わりだとしても、Subversion はバージョン管理の観点からは完全なものであったろうと思います。 しかし話にはまだ先があります。

ディレクトリとファイルのバージョン管理に加えて、Subversion は バージョン化されたファイル、ディレクトリに付随したバージョン化された メタデータの追加、修正、削除のためのインターフェースを用意しています。 このようなメタデータを属性と呼びます。属性は 作業コピー中のアイテムごとに、名前と名前に結びついた任意の値の組から なる二つの列を持つテーブルとして考えることができます。 一般的に、名前が人間が読むことのできるテキストでなくてはならないことを 除けば、名前と属性値は自由に選ぶことができます。そして属性に関する 一番重要なことは、属性もまた、ファイルの内容と同様にバージョン管理できる ということです。テキストの変更点をコミットするのと同じくらい簡単に 属性の変更を、修正したりコミットしたり、取り消したりすることができます。 そして、作業コピーを更新するときに、他の人がした属性変更についても受け取ることができます。Subversion での別の属性

属性は Subversion のほかの場所にも出てきます。ファイルやディレクトリが それに結びついた任意の属性名と属性値を持つのと同じように、ある リビジョンは、それ自体として、任意の属性を持つことができます。同じ制約が当てはまります — 属性名は人間に読めるテキストで、 属性値はバイナリ値を含む任意値で — ただし、リビジョンの属性は バージョン化されません。バージョン化されない属性については 項 5.2.2 を見てください。

この節では、属性をサポートするユーティリティーについて説明します — — Subversion のユーザと、Subversion そのものに対しての説明になります。 属性に関連した **svn** サブコマンドを理解し、属性の変更が通常の Subversion のワークフローにどう影響するかを学びます。Subversion の 属性はあなたのバージョン管

理の経験を広げるものであることが、きっと わかるでしょう。

7.3.1 なぜ属性なんてものが?

属性は作業コピーにとても役に立つ情報を追加することができます。 実際、Subversion 自身も特殊な情報を記録するのに属性を使っていて、 それはある特定の処理が必要になっていることを示すようなときに 使っています。同様に、ユーザは自分自身の目的のためにも属性を使う ことができます。もちろん属性でできることはすべて、バージョン化 したファイルでもできるのですが、まずは以下のような Subversion 属性の 使い方の例を見てください。

あなたは、たくさんのデジタル写真を見せるためのウェブサイトを設計していて、 タイトルと日付を付けて表示したいとします。ここで、写真の内容は常に 変化するので、このサイトの管理をできる限り自動化したいと思っています。 それぞれの写真は非常に大きいので、このような場合の常套手段として あなたはサイトをおとずれた人に小さなサムネイルの画像を用意したいとします。 これを普通のファイルでやることもできます。つまり、ディレクトリに `image123.jpg` と `image123-thumbnail.jpg` の両方を置けば良いのです。あるいは両方のファイル名称を一緒にして、 別ディレクトリにおいてもいいですね。 `thumbnails/image123.jpg` のような感じです。 タイトルと日付についても同様の方法をとることができ、これもまた、もとの 画像ファイルとは別のものになります。すぐ、ファイルのツリーはごちゃごちゃ になり、新しい写真がサイトに追加されるたびに、サイトのデータは何倍にも 膨れ上がります。

Subversion のファイル属性を使った同じ設定を考えてみましょう。 ある画像ファイル `image123.jpg` と、そのファイル の属性として設定する `caption`、 `datestamp`, そして `thumbnail` があるところを想像してください。こうすれは、作業コピーのディレクトリはもっと管理しやすく なります — 実際これで画像ファイル以外の何もないように 見えます。しかし、あなたの自動スクリプトはもっと多くのことを 知っています。それは **svn** (あるいはさらに、 Subversion 言語連携を使うこともできます — 項 8.3.3 参照) を使って拡張情報を追加しますが、それはあなたのサイトが、 インデックスファイルを読んだり、 複雑なファイルパス操作の仕組みをいじることなしに、表示する必要がある ものです。

Subversion の属性をどう使うかはあなたしだいです。既に指摘した ように、Subversion は自分自身が使う属性を持っていて、この章のあとで 少し説明します。しかし、まずは、**svn** プログラム を使って、どのように属性を操作するかを考えましょう。

7.3.2 属性の操作

svn コマンドにはファイルとディレクトリの属性 を追加したり修正したりするいくつかの方法があります。短い可読な属性を 新規に追加する一番簡単な方法は属性の名前と値を **propset** サブコマンドで指定することです。

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

しかし、属性値に対して Subversion が持つ柔軟性については既にさんざん 言ってきました。もし、複数行テキスト、またはバイナリ値を属性値に したいと考えているなら、コマンドラインからその値を入力したくはないと 思います。それで **propset** サブコマンドは `--file(-F)` オプションを使って、新しい属性値が入ったファイルの名前を指定することもできます。

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

属性で利用できる名前にはいくつかの制限があります。属性の名前は 文字、コロン (:) あるいはアンダースコア (_) で始まり、その後では数字、ハイフン (-)、ピリオド (.) も使えます。 [*3]

propset コマンドに加えて、**svn** プログラムは **propedit** コマンドも用意しています。 このコマンドは、設定されたエディタを使って (項 7.2.3.2 参照) 属性を追加したり修正したりします。 このコマンドを実行すると **svn** は現在の属性値を書き込んだ 一時ファイルを作ってエディタを起動します。(新しい属性を追加する場合は これは空になります)。それから、自分が望むような値になるまで新しい属性値 をエディタを使って修正し、一時ファイルを保存してからエディタを抜けます。 Subversion は属性の値が変更されたことを確認すると、それを新しい属性値 として受け入れます。もしエディタを変更することなく抜ければ、属性値の 変更は起こりません。

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

他の **svn** コマンドと同様に、属性に関するこれらの コマンドも複数パスに対して一度に実行することができます。これは 一つのコマンドで複数のファイル上の属性を修正することを可能にします。 たとえば、以下のようなことができます:

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

このような属性の追加や編集は、保管されている属性値を簡単に取得 できないなら、あまり便利ではありません。それで **svn** プログラムはファイルやディレクトリに保管された 属性の名前と値を表示するためのサブコマンドを二つ用意しています。 **svn proplist** はパス上に存在する属性の名前の一覧を 表示します。ノード上の属性名がわかってしまえば、個別に **svn propget** を呼び出してその属性値を要求することが できます。このコマンドは与えられた (一つ以上の) パスと属性名 から、その属性値を標準出力に表示します。

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
  copyright
```

[*3] XML に詳しいのであれば、これは XML "名称" の ASCII サブセットな構文 に近いものです。

```
  license
$ svn propget copyright calc/button.c
(c) 2003 Red-Bean Software
```

proplist コマンドの変種として、すべての属性の 名前と値の両方をリストするものがあります。これには単に、 `--verbose(-v)` オプションを指定すれば OK です。

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

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

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

最後の属性関連サブコマンドは **propdel** です。Subversion は空の値を持つ属性を 格納することを許すので、**propedit** や **propset** を使うだけでは、属性を削除することが できません。たとえばこのコマンドは期待される結果にはなりません :

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

属性の削除には **propdel** コマンドを使う必要が あります。構文は他の属性関連コマンドとよく似ています:

```
$ svn propdel license calc/button.c
property 'license' deleted from ''.
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
```

```
  copyright : (c) 2003 Red-Bean Software
$
```

これで、属性関連の **svn** サブコマンドのすべてに ついて説明したので、日常的な Subversion ワークフローに、属性の変更 がどのような影響を与えるかを見てみましょう。前に指摘したように ファイルとディレクトリの属性は、普通のファイルの内容と同様、 バージョン化されます。結果として、Subversion は他の人がした修正点を 自分自身の上にマージすることができます。 — もちろん通常の マージと同様、うまくいくかも知れませんし、衝突するかも知れません。リビジョン属性のマージ

バージョン化されないリビジョン属性を憶えているでしょうか? **svn** プログラムでこのような属性も修正 できます。単に`--revprop` 引数を追加し、修正したい 属性のあるリビジョンを指定してください。この場合リビジョンは グローバルなので修正しようとするリビジョン属性のリポジトリから チェックアウトした作業コピーにいる限り、パスを指定する必要はあり ません。 [*4]

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

このようなバージョン化されない属性の修正権限は、リポジトリ管理者によって 明示的に与えなくてはなりません (項 5.3.1 参照)。この属性はバージョン化されない ので注意して編集なければ情報を失う危険があります。リポジトリ管理者は この危険から守るための仕組みを作ることはできますが、デフォルトでは バージョン化されない属性の変更は不可になっています。

そしてファイルの中身の場合と同じように、属性の変更はローカルな 修正にしかすぎず、**svn commit** でリポジトリに コミットして初めて修正が確定します。変更はやはり簡単に取り消す こともできます — **svn revert** コマンドは ファイルやディレクトリを編集前の状態に戻し、その内容、属性、 などすべてについてもそうです。さらに、 **svn status** や **svn diff** コマンドを使って、ファイルやディレクトリ属性の状態について 有用な情報を受け取ることができます。

```
$ svn status calc/button.c
 M     calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
___________________________________________________________________
Name: copyright
   + (c) 2003 Red-Bean Software

$
```

status サブコマンドが `M` を最初のコラムではなく、二番目のコラムに表示するのに注意です。 これは、`calc/button.c` の属性を修正したが、 ファイルの内容は変更していないことを示しています。 属性も内容

[*4] コミットログ中の、スペルミス、文法間違い、「つまらないミス」 は多分`--revprop` オプション利用で一番よく起こるものです。

も変更すれば、M は、最初のコラム にも二番目のコラムにも表示されます。 (項 3.6.3.1 参照)。属性の衝突

ファイルの内容と同様、ローカルの属性変更は別の人のコミットに よる修正と衝突するかも知れません。作業コピーを更新して、自分 自身の修正を壊してしまうようなバージョン化されたリソースの 属性の変更を受け取った場合、Subversion はそのリソースが衝突の 状態にあることを報告します。

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

Subversion はまた、衝突したリソースのある同じディレクトリに .prej の拡張子を持ったファイル を作ります。この中に、衝突の詳細が書かれています。このファイル の内容を確認して、衝突をどのように解消するかを決めてください。 衝突が解消されるまで、**svn status** の出力の 二番目のコラムに C が表示されて、ローカル修正 をコミットしようとしても失敗することを示します。

```
$ 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'.
$
```

属性の衝突を解消するには、衝突した属性が正しい値を含んでいるか どうかを確認してから、**svn resolved** コマンドを 使って Subversion に手で問題を解消したことを報告してください。

Subversion が現在の属性の差異を表示する標準的でない方法に 気づかれたかも知れません。 **svn diff** を実行して、出力をパッチファイルを 作るためにリダイレクトすることができます。**patch** プログラムは属性にたいするパッチを無視します — 一般的に それは理解できないゴミをすべて無視します。これは不幸にも **svn diff** で生成されたパッチを完全に適用するには、 属性の修正については手で適用しなくてはならないということを意味します。

見たように、属性の修正は典型的な Subversion のワークフローにはあまり 大きな影響を与えません。作業コピーを更新し、ファイルとディレクトリの 状態をチェックし、自分のした変更点について報告し、そのような修正点を リポジトリにコミットするという一般的なパターンは属性の存在や非存在 とは完全に無関係です。**svn** プログラムにはいくつかの 追加のサブコマンドがあり、実際に属性変更することができます。 しかし、それは、属性関連コマンドの目に見える唯一の非対象性です。

7.3.3 特殊な属性

Subversion は属性について特別のポリシーを持っていません — どのような 目的にも使うことができます。Subversion は、svn: という プレフィックスの付いた属性名を使うのを禁じているだけです。これが、Subversion 自身が使う属性の名前空間です。実際、Subversion は、ファイルや ディレクトリに特殊な効果をおよぼすようなある種の属性を定義しています。 この節ではこの神秘をときあかし、どうやってこれら特殊な属

性が、あなたの Subversion ライフをちょっとだけ楽にするかについて説明します。

7.3.3.1 svn:executable

`svn:executable` 属性は半自動的なやり方で バージョン管理されているファイルのファイルシステム上の実行 権限を制御するのに使われます。この属性は属性値を何も定義 しません — 単に属性名が存在していれば、Subversion によって実行ビット が保存されます。この属性を削除すると、実行ビットの全制御は オペレーティングシステムに戻されます。

たくさんのオペレーティングシステム上で、コマンドとしてファイルを 実行できるかどうかは実行ビットの存在によって支配されています。 このビットは普通、デフォルトでは無効となっていて、必要に応じてユーザ が明示的に有効にしてやる必要があります。作業コピー中では、新しい ファイルが常に作られ、その一方で、更新処理を通じて存在している ファイルの新しいバージョンを受け取ります。これは、あるファイルの 実行ビットを有効にしてから作業コピーを更新した場合、もし更新処理の 一貫としてそのファイルが変更されたときにその実行ビットは無効になって しまう可能性があるということです。そこで Subversion は `svn:executable` 属性を、実行ビットを 有効にし続けるために用意しています。

この属性は FAT32 や NTFS のように実行権限ビットの概念を持たないファイルシステム 上では何の効果もありません。[*5] また、それは定義された値を持ちませんが、Subversion はこの属性が設定される と、強制的にその値を`*`とします。最後に、この属性は ファイルに対してのみ有効で、ディレクトリに対しては意味を持ちません。

7.3.3.2 svn:mime-type

`svn:mime-type` 属性は、Subversion ではいろいろな 目的に使われます。ファイル自身の Multipurpose Internet Mail Extensions (MIME) 上の分類の記憶場所で あると同時に、この属性の値は Subversion 自身のいくつかの動作モードを 決定します。

たとえば、ファイルの `svn:mime-type` 属性が非テキスト MIME タイプである場合 (例外はあるにせよ、一般的には、`text/` 以外で始まるような場合)、Subversion はファイル内容はバイナリであると仮定 します — つまり、可読ではない —。この利点の一つは、Subversion が、作業コピー更新時に、サーバから受け取る変更点を、文脈に依存し行単位に マージする機能を提供することです。しかし、バイナリデータと信じられて いるファイルについては「行」のような概念はまったくありません。 それで、このようなファイルについては、Subversion は更新時に文脈マージを 実行しようとはしません。そのかわり、バイナリの作業コピーファイルを修正し、それが更新される場合はいつでも、あなたのファイルは `.orig` 拡張子を付けた形に名称変更され、それから Subversion は更新で受け取る変更を含むが、あなた自身のローカルな 修正は含んでいない新しい作業コピーファイルを、もとの名前で保存します。 この振る舞いは、文脈マージできないファイルに文脈マージを 実行しようとする間違った意図からユーザを守るためです。

また、もし `svn:mime-type` 属性が設定されていると、 Subversion の Apache モジュールは GET 要求に応答するとき、HTTP ヘッダの `Content-type:`にこの値を使います。これは ブラウザを使ってリポジトリを調べるときに、そのファイルをどうやって 表示すれば良いかの重要な手がかりになります。

7.3.3.3 svn:ignore

`svn:ignore` 属性はある種の Subversion 操作が無視する ファイルパターンのリストを含んでいます。多分もっともよく利用される 特殊属性で、`global-ignores` 実行時設定オプションと ともに利用されます。(項 7.2.3.2 参照)。 それを使って、バージョン化されていないファイルとディレクトリを **svn status**、**svn add**、

[*5] Windows のファイルシステムはファイル拡張を使ってそれが実行ファイルである ことを示します。(`.EXE`, `.BAT`, `.COM` のような拡張子です)

そして **svn import** コマンドの対象から除外します。

`svn:ignore` 属性の背後にある理由は簡単に説明できます。 Subversion は、作業コピーディレクトリにあるすべてのファイルとサブディレクトリ がバージョン管理下にあるとは仮定しません。リソースは **svn add** や **svn import** コマンドを使って明示的に Subversion 管理下に 置く必要があります。結果としてしばしば作業コピー中の多くのリソースが バージョン管理下にないことがあります。

svn status コマンドは出力の一部として作業コピーに あるバージョン化されていないファイルやサブディレトクリを、 `global-ignores` オプション (あるいはその組み込みの デフォルト値によって) によって、まだフィルタされていないものについてのみ表示します。 このように振る舞うのは、ユーザが、あるリソースをバージョン管理下に 追加するのを忘れたときに、そのことがわかるようにするためです。

しかし Subversion は無視すべきすべてのリソースの名前を推測できる わけではありません。さらに、非常によく、特定のリポジトリの、 すべての 作業コピー中で無視したいものが あったりします。そのリポジトリのすべてのユーザに、それぞれの実行時設定 領域に特定のリソースパターンを追加するように強いるのは、負担になる だけではなく、ユーザがチェックアウトした別の作業コピーの設定によって 壊れてしまう危険があります。

これを解決するには、あるディレクトリに現れるかも知れないリソースを 区別して無視できるようなパターンを、ディレクトリ自体に保存することです。 バージョン化されないリソースのよくある例で、基本的にはディレクトリごとに ユニークだが、現れることがあるのは、プログラムのコンパイルからの出力 などがあります。あるいは — この本自身を例にとれば — HTML, PDF, PostScript ファイルなどで、これらはある DocBook XML 入力ファイルを、もっと 読みやすい出力形式に変換した結果生成されるものです。CVS ユーザにとっての無視パターン

Subversion の `svn:ignore` 属性は CVS の `.cvsignore` ファイルに構文も機能も とてもよく似ています。実際、CVS 作業コピーを Subversion に移行する場合、`.cvsignore` の 無視パターンを直接 **svn propset** コマンド の入力ファイルとすることができます:

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

しかし、CVS と Subversion がパターンを無視する方法にはいくつかの 違いがあります。二つのシステムは無視パターンを別のタイミングで 利用し、無視パターンが適用される対象にわずかな相違点があります。 さらに、Subversion は無視パターンが何もない状態に戻すための！ パターンは使うことができません。

このような意味で、`svn:ignore` 属性が解決法に なります。その値はファイルパターンの複数行のあつまりで、一行に 一つのパターンを書きます。属性は、パターンを適用したいと 思うディレクトリに設定されます。 [*6] たとえば、**svn status** からの以下の出力が あったとします:

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

[*6] パターンはそのディレクトリのみに制限されます — サブディレクトリに 再帰的に伝わることはありません。

```
?      calc/debug_log.1
?      calc/debug_log.2.gz
?      calc/debug_log.3.gz
```

この例では、button.c に対する何かの属性の 変更をしましたが、作業コピー中にはいくつかのバージョン管理して いないファイルもあります:ソースコードから コンパイルした calculator プログラム、data.c という名前のソースコード、そして、 デバッグ出力のログファイルです。これで、ビルドシステムは常に calculator を生成することを知っています。 [*7] そして、テストプログラムは常にこのようなデバッグログファイルを 残すことも知っています。このような事実はあなたのだけではなく、 どの作業コピーにとっても正しいことです。そして **svn status** を実行するたびにこれらのファイルを見ることに興味があるのではないことも 知っています。それで、**svn propedit svn:ignore calc** を使っていくつかの無視パターンを calc ディレクトリに 追加します。たとえば svn:ignore 属性の新しい 値として、以下を追加するかも知れません:

```
calculator
debug_log*
```

この属性を追加すると、calc ディレクトリ上に ローカルな属性変更を手に入れることができます。 しかし、**svn status** 出力について 何が変わったかに注意してください:

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

これで、見たくないファイルが出力から全部消えました。 もちろんこのようなファイルはまだ作業コピーにあります。 Subversion はそれが存在していて、バージョン管理下にないことについて は何も言いません。これで、表示からつまらないファイルを全部取り除く 一方、もっと注意する必要のあるアイテムについてはそのままにします — たとえば、バージョン管理下に追加するのを忘れたソースコードファイル などは、依然として表示されます。

無視するファイルを見たい場合は、Subversion に --no-ignore オプションを渡すことができます:

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

[*7] それがビルドシステムの核心では?

無視されるパターンのリストはまた **svn add** や **svn import** でも利用されます。これらの操作は Subversion にあるファイルやディレクトリの集まりを管理させることも 含まれます。ユーザがバージョン管理したいと思うファイルをツリー中 から明示的に選択させるかわりに Subversion は無視パターン規則を使って どのファイルバージョン管理システムから除外されるべきであるかを決定 します。この処理は再帰的なファイル追加処理やインポート処理の一環と して行なわれます。

7.3.3.4 svn:keywords

Subversion はキーワードをファイル自身の内容と して置き換える機能があります — キーワードとは、バージョン化された ファイルについての役に立つ小さくて動的な情報です — 。 キーワードは一般的にファイルが最後に修正された時刻についての情報を あらわしています。この情報はファイルが変更されるたびに変わり、さらに 重要なことにはファイルが変更された直後 に 変わるので、それはデータを完全に最新の状態に保つことは、バージョン管理 システム以外のどのような手段にとっても厄介なものです。編集した人間に まかせれば、その情報は必然的に古くなります。

たとえば、修正された最後の日付を表示したいと思っているドキュメントが あるとします。あなたは、そのドキュメントのすべての著者に、変更点を コミットする直前に、最後に変更された時刻を示す、ドキュメントの一部を ちょっとだけ変える作業を強いる必要がありますが、遅かれ早かれ誰かこれを忘れる人が 出てくるでしょう。 そうするかわりに、単に Subversion に対して `LastChangedDate` キーワードに対してキーワード置換を実行するように指示しましょう。 あなたはドキュメント中の *keyword anchor* を置くこと でキーワードが挿入された、ファイル中の任意の場所を制御することができます。 このアンカー文字列は、単に`$`*`KeywordName`*`$`のように書式化された 文字列です。

すべてのキーワードは大文字小文字の区別がありファイル中での 目印になります: キーワードが展開されるように大文字小文字を正しく 使う必要があります。 `svn:keywords` 属性の値 についても大文字小文字の区別を考慮すべきです — ある種の キーワードは大文字小文字を区別せずに解釈されますがこの仕様は 過去のものです。

Subversion は、置換可能なキーワードのリストを定義しています。 そのリストは、以下の五つのキーワードで、そのいくつかについては 別名を使うこともできます:

`Date` このキーワードはファイルがリポジトリ中で修正された最後の時刻 をあらわし、`$Date: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $`のようなものです。これは `LastChangedDate` と指定することもできます。

`Revision` このキーワードは、ファイルがリポジトリで変更された最後のリビジョン をあらわし、`$Revision: 144 $`のようなものです。これは `LastChangedRevision` または `Rev` と省略することもできます。

`Author` このキーワードはリポジトリ中のこのファイルを最後に変更したユーザ をあらわし、`$Author: harry $`のようなものです。これは `LastChangedBy` と省略することもできます。

`HeadURL` このキーワードはリポジトリ中のファイルの最後のバージョン に対する完全な URL をあらわし、`$HeadURL: http://svn.collab.net/repos/trunk/README $` のようなものです。これは `URL` と省略する こともできます。

`Id` このキーワードは、他のキーワードの圧縮された組み合わせです。 その置き換えは、 `$Id: calc.c 148 2002-07-28 21:30:43Z sally $`のようなもので、ファイル `calc.c` が最後に変更されたのは リビジョン 148 で、時間は July 28, 2002 の夜、変更した人は、 `sally` であることを意味しています。

キーワードアンカーテキストをファイルに付け加えただけでは何も 起きません。明示的に要求しなければ Subversion は 決してテキスト置換をやろうとはしません。ようは、 キーワードのそのものの使い方についてのドキュメントを [*8] 書いているときに、そのすばらしい例自身が Subversion に よって置換されてほしくはないでしょう。

Subversion が特定のファイルの上でキーワードを置換するかどうかを設定 するために、属性関連のサブコマンドに戻ります。 `svn:keywords` 属性は、バージョンファイルに設定 された場合は、そのファイルのどのキーワードが置換されるかの制御を します。その値は、空白で区切られたキーワード名称か別名のリストで、前に書いたテーブルの中にあるもののどれかになります。

たとえば、`weather.txt` という名前の バージョン管理されているファイルがあり、以下のようだとします:

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

`svn:keywords` 属性がファイルに設定されていなければ Subversion は何も特別なことはしません。さて、`LastChangedDate` キーワードの置換を有効にして みましょう。

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

これで、`weather.txt` のローカル属性を変更しま した。そのファイルの内容には何の変化もないでしょう (属性を設定 する前に変更していなければ)。ファイルはキーワードアンカー `Rev` キーワードを含んでいたとします。私たちはこの キーワードをまだ属性値として設定していません。Subversion はファイル に存在しないキーワードを置換する要求を無視しますし、 `svn:keywords` 属性値に存在しないキーワードを 置換することもありません。キーワードと、偽の差分

キーワード置換による目に見える形の結果として、この機能を使って バージョン管理されているファイルは、少なくとも一箇所、内容が違って いると考えるかも知れません。この箇所とは、キーワードアンカーが 置換された場所です。しかし、実際にはこういうことは起こりません。 ローカルな修正点を **svn diff** で調べるときと、 **svn commit** でローカル修正を転送する前では、 Subversion は以前に置換されたどんなキーワードも「もう一度置換 することはありません」。結果として、リポジトリに格納された ファイルのバージョンはユーザがファイルにした実際の修正だけが含まれて います。

[*8] … あるいは、その本の一節を…

この属性の変更をコミットした直後、Subversion は作業ファイルを、新しい 置換テキストで更新します。キーワードアンカー `$LastChangedDate$`を見るかわりに、置換結果を 見ることになるでしょう。この結果はキーワードの名前を含み、ドル記号 文字 (`$`) でくくられています。そして述べたように、 `Rev` は設定していないので、置換されませんでした。

`svn:keywords` 属性を「Date Author」 に設定してもキーワードの目印は`$LastChangedDate$` の別名を使うのでやはりうまく展開されます。

```
Here is the latest report from the front lines.
$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

もし誰か別の人が `weather.txt` に変更点をコミット すれば、ファイルのコピーは前と同じ置換されたキーワード値を表示し続ける でしょう — 作業コピーを更新するまでは。そのとき、 `weather.txt` ファイルのキーワードはそのファイルを コミットした一番最後の状態を反映する情報で置換されるでしょう。

7.3.3.5 svn:eol-style

バージョンファイルの `svn:mime-type` 属性で 指定するのでなければ、Subversion はファイルは可読なデータが含まれている と仮定します。一般的に、Subversion はそのファイルに対する文脈差分を 報告することができるかどうかを決めるためにだけ利用します。そうで なければ、Subversion にとって、バイトはただのバイトでしかありません。

これは、デフォルトでは Subversion はあなたのファイルが利用している 行端 *(EOL)* マーカ の種類に注意を向けないことを 意味します。不幸にも、異なるオペレーティングシステムはファイルのそれ ぞれの行末をあらわすのに別のトークンを使います。たとえば、普通 Windows プラットフォームのソフトによって使われる行末トークンはアスキー制御文字 の組になります — キャリッジリターン (`CR`) と ラインフィード (`LF`) です。しかし Unix では単に `LF` 文字を使って行末を表現します。

これらのオペレーティングシステムの上のさまざまなツールのすべてが 自分が実行されているオペレーティングシステムの もともとの行末スタイル *ending style* とは違った形式の行末を含んでいるようなファイルを理解することができる わけではありません。よくある結果としては、Unix のプログラムは Windows のファイルにある `CR` 文字を通常の文字 (普通、`^M` のように表示します) として扱い、Windows のプログラムは Unix ファイルのすべての行を一つの巨大な行として連結 してしまいますが、これは行末を示すキャリッジリターン - ラインフィード 文字 (あるいは `CRLF`) の組み合わせが見つかれらないため です。

この、別の EOL マーカに関する敏感さは、異なるオペレーティング システム間でファイルを共有しようとする人をいらいらさせます。 たとえば、ソースコードファイルと、このファイルを Windows でも Unix でも編集する開発者を想像してみてください。もしすべての開発者 が常に行末を保存するようなツールを使うのであれば問題は起こりません。

しかし、実際には、たくさんのありふれたツールは別の EOL マーカのファイル を正しく読むことができないか、ファイルが保存されるときに、行末を そのオペレーティングシステム固有のものに変換してしまうかします。 もし開発者に最初のことが起こると、彼は外部の変換ユーティリティー (**dos2unix** や、それとペアになった **unix2dos**) を使ってファイル編集の前処理をしなくては なりません。あとの場合には何も特別の用意はいりません。しかし どちらの場合でも、すべての行が、最初のものと違ってしまいます。 変更をコミットする前に、ユーザには二通りの選択があります。 編集する前の行末スタイルと同じスタイルになるように変換

ユーティリティー を使って修正したファイルを保存するか、単にそのファイルをコミットするか です — この場合、行末は新しい EOL マーカがつきます。

このようなシナリオの結果は時間の無駄と、コミットされたファイルに対する 不必要な修正になります。時間の無駄はそれだけで十分な苦痛です。しかし、 コミットがファイルのすべての行を変更するなら、これは、本当に修正された のはどの行なのかを決定する作業を非常に複雑なものにします。バグの修正は いったいどの行でなされたのか? どの行で構文エラーがあったのか?

この問題の解決は、`svn:eol-style` 属性です。 この属性が正しい値に設定されれば、Subversion はそれを使って、 どのような特殊な処理がファイルに必要であり、その処理をすれば ファイルの行末スタイルが、異なるオペレーティングシステムからの コミットによって、ばたばた変化したりしないか、を決定します。 設定できる値は:

`native` これは、ファイルが、Subversion が実行されているオペレーティング システムの本来の EOL マーカを含むようにします。言い換えると もし Windows 上のユーザが作業コピーをチェックアウトして、そこには `svn:eol-style` 属性が `native` に設定されたファイルがある場合、そのファイルは `CRLF` EOL マーカを含むということです。逆に Unix ユーザが作業コピーをチェック アウトし、そこに、その同じファイルがあった場合は、ファイルのコピー には `LF` EOL マーカが含まれることになります。
Subversion は実際にはリポジトリにファイルを格納するときには、 オペレーティングシステムにはよらず、正規化された `LF` EOL マーカを使います。これは基本的にユーザには意識しなくても良いようになっているわけですが。

`CRLF` これは使っているオペレーティングシステムによらず、ファイルの EOL マーカ に `CRLF` の並びを使います。

`LF` これは使っているオペレーティングシステムによらず、ファイルの EOL マーカ に `LF` 文字を使います。

`CR` これは使っているオペレーティングシステムによらず、ファイルの EOL マーカ に `CR` 文字を使います。この行末スタイルはそれほど一般的ではありません。それは古い Macintosh プラットフォームで利用されていました。(その上では Subversion は実行する ことさえできませんが)

7.3.3.6 `svn:externals`

`svn:externals` 属性は一つ以上のチェックアウト された Subversion 作業コピーでバージョン管理されたディレクトリを 作るための命令を含んでいます。このキーワードに関するより詳しい情報 は項 7.6 を見てください。

7.3.3.7 `svn:special`

`svn:special` 属性は `svn:`属性 のなかでユーザが直接設定したり修正することのできない唯一のものです。 Subversion はシンボリックリンクのような「特殊な (special)」 オブジェクトが追加の予告をされるときは常にこの属性を自動的に設定 します。リポジトリは `svn:special` オブジェクトを 通常のファイルのようにして保存します。しかしクライアントがこの属性 をチェックアウトあるいは更新中に見ようとすると、ファイルの内容を見て そのアイテムを特殊なオブジェクトと解釈します。いまこれを書いている時点 の Subversion のバージョンでは バージョン化されたシンボリックリンクだけがこの属性を持ちますが今後の Subversion ではおそらく他の特殊なノードもこの属性を持つことになるで しょう。

注意: Windows クライアントはシンボリックリンクを持たないのでリポジトリ から取得するファイルが、`svn:special` によってシンボリッ クリンクであるとされていてもその属性は無視されます。Windows ではユーザは作業コピー中に通常のバージョン化されたファイルとしてこれを 受け取ることになります。

7.3.3.8 `svn:needs-lock`

この属性はそのファイルが変数前にロックされるべきであることを 示すのに利用されます。属性の値は無関係です: Subversion はこの値を *に一律置き換えます。存在している場合、ファイル はユーザが明示的にファイルをロックしない限り、読み込むことだけが許され ます。ロックが取得できている場合 (これは **svn lock** 実行の結果ですが)、ファイルは読み書き可能になります。ロックが解除 されるとファイルは再び読み込むことだけができる状態になります。

この属性がどのように、いつ、そしてなぜ有用なのかについての詳しい 情報は 項 7.4.4 を見てください。

7.3.4 属性の自動設定

属性は Subversion での強力な機能で、この章や別の章で議論さ れるたくさんの Subversion の機能の重要な部品として振舞います — テキスト形式の diff やマージのサポート、キーワード置換、改行変換、など です。しかし属性機能を完全に使いこなすには、正しいファイルやディレクト リに設定する必要があります。残念なことに、このステップはきまりきった作 業の中で簡単に忘れられてしまいますが、それは属性の設定し忘れは通常 はっきりしたエラーを起こすことがないからです (少なくても、そう、例えば バージョン管理システムにファイルを追加するのと比較すれば)。必要な場所 で属性が設定されるのを助けるため Subversion は単純ではありますが役に立 つ機能を提供しています。

svn add または **svn import** であるファイルをバージョン管理下に置くばあい、Subversion はそのファイルが人間によって読めるものか読めないものかを 非常に基本的な方法で決定します。もし読めないファイルであった場合、 Subversion は自動的にそのファイルの `svn:mime-type` 属性を `application/octet-stream` に設定します (これは一般的な「これはバイトの集まりですよ」という MIME タイプになります)。もちろん Subversion が間違った推測をするか、より 正確な `svn:mime-type` 属性を設定したい場合 — 多分 `image/png` とか、 `application/x-shockwave-flash` とか — 属性を 削除したり編集したりすることは常に可能です。

Subversion はまた自動属性機能を提供しますが、これはファイ ル名のパターンによって適切な属性名と属性値を設定できるようにするもので す。この規則は実行時設定領域に設定します。やはりファイルの追加やインポー トに影響を与え、この操作中で Subversion が決定するデフォルトの MIME タ イプを上書きするだけではなく、Subversion や固有の追加属性を設定するこ ともできます。たとえば、JPEG ファイルを追加する時には常に — `*.jpg` というパターンに当てはまるファイルを追加する時 には常に — Subversion は自動的に `svn:mime-type` 属性を `image/jpeg` に設定する、といった具合です。 あるいはまた`*.cpp` に当てはまるすべてのファイルには `svn:eol-style` 属性を `native` に設定 し、`svn:keywords` 属性を `Id` に設定する、といった具合です。自動属性のサポートはおそらく Subversion 関連 ツール中で最も手軽に扱うことのできる性質です。この設定に関連した詳細は 項 7.2.3.2 を見てください。

7.4 ロック

Subversion の「コピー・修正・マージ」 モデルは プログラムソースコードのように行を基本としたテキストファイルからなる プロジェクト上で共同作業する場合には最適です。しかし ロックが必要な場合で議論したように、時には Subversion の標準的な共同作業モデルのかわりに 「ロック・修正・ロック解除」モデルを使わなくてはならないことも あります。ファイルがバイナリデータから構成されている場合、異なるユーザに よる

二つの修正をマージするのは困難であったり不可能なことがよくあります。 このため Subversion 1.2 とそれ以降ではロック、 あるいは他のバージョン管理システムでは「保護されたチェックアウト (reserved checkouts)」として知られている機能を提供しています。

Subversion のロック機能は主に二つの目標があります:

- リソースに対する直列化されたアクセス。 ユーザがリポジトリ中のファイルを変更するための排他的な権限を取得できる ようにします。Harry が `foo.jpg` を変更する権利を 取得した場合、Sally にはそのファイルの変更点をコミットするのを禁止すべき です。
- 開発者間のコミュニケーション支援. ユーザがマージ不能な変更をしてしまうような時間のロスを防ぎます。Harry が `foo.jpg` の変更権限を取得した場合、Sally はすぐにそれに 気づいて同じファイルに対する作業を避けることができるようにすべきです。

Subversion のロック機能は現時点ではファイルだけに制限されています — ディレクトリツリー全体へのアクセスに対する利用はまだできません。

7.4.1 ロックの作成

Subversion リポジトリではロックとは あるユーザがファイルを修正する排他的な権限を与える小さなメタデータ です。このユーザはロックの所有者と呼ばれます。 ロックごとにユニークな識別子があり、普通これは長い文字列の形を したもので、 ロック・トークンと言われます。 リポジトリは独立したテーブル中にロック情報を管理し、コミット操作の最中に 強制的にロックをかけます。コミットのトランザクションがファイルを修正 または削除しようとした場合 (あるいはファイルの親を削除しようとした場合)、 リポジトリは二つの情報を要求します:

1. ユーザ認証。 コミットを実行しようとするクライアントはロック所有者として認証されなくては なりません。
2. ソフトウェアによる認可。ユーザの作業コピーはコミットと共にロック・トークン を送信しなくてはならず、これによってどのロックを利用中であるかを正しく知る ことができます。

以下の例によって順序よく説明していきます。Harry が JPEG 画像を修正することに 決めたとしましょう。他の人たちがそのファイルに対する修正をコミットしないように 彼はリポジトリ中のファイルを **svn lock** コマンドによって ロックします:

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

$ 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: 2005-03-15 12:43:04 -0600 (Tue, 15 Mar 2005)
Text Last Updated: 2005-06-08 19:23:07 -0500 (Wed, 08 Jun 2005)
Properties Last Updated: 2005-06-08 19:23:07 -0500 (Wed, 08 Jun 2005)
Checksum: 3b110d3b10638f5d1f4fe0f436a5a2a5
Lock Token: opaquelocktoken:0c0f600b-88f9-0310-9e48-355b44d4a58e
Lock Owner: harry
Lock Created: 2005-06-14 17:20:31 -0500 (Tue, 14 Jun 2005)
Lock Comment (1 line):
Editing file for tomorrow's release.
```

前の例には新しい話がたくさん含まれています。まず Harry は **svn lock** に `--message` オプションを 渡しています。**svn commit** と同様、**svn lock** コマンドは (`--message (-m)` または `--file (-F)` オプションによって) コメントをつけてそのファイル をロックした理由を説明することができます。**svn commit** と違うのは **svn lock** は 自分の好きなエディタによる メッセージを常に要求するわけではないところです。ロックのコメントは オプションですが、コミュニケーションを円滑にするためにつけることを お勧めします。

つぎにロックが成功しています。これはそのファイルはまだロックされて いなかったこと、そして Harry がそのファイルの最新バージョンを得たことを 意味しています。もし Harry の作業コピー中のファイルが古いものであれば リポジトリはその要求を拒否し、Harry に対してまず **svn update** を実行してから再びロックコマンドを発行するように要求します。

リポジトリ中にロックを作成した後、作業コピーはロックについての情報 をキャッシュすることにも注意してください — そのうち最も重要なのは ロック・トークンです。ロック・トークンの存在は非常に重要です。作業コピー はそれによって後でロック機能の認可を受けるからです。 **svn status** コマンドはファイル名のとなりに (locKed の省略として)K を表示しロック・トークンが存在 していることを示します。ロック・トークンに関する注意

ロック・トークンは認証トークンではなく、認可 トークンです。ロック・トークンはセキュリティ的に保護されていません。 実際、ロック・トークンは **svn info URL** を実行することに よって誰でも調べることができます。

ロック・トークンは作業コピー中に存在する時にだけ特別な意味を持ちます。 トークンによってその特別な作業コピー中でロックが作成され、どこか他の場所に ある他のクライアントによってではないことが証明されます。ロック所有者として 認証されるだけでは不測の事故を避けるのに十分ではないのです。

例えば: おそらく作成中のチェンジセットの一部として、あなたの職場の コンピュータを使ってあるファイルにロックをかけたとします。 こうしてしまえば、あなたの家にあるほうのコンピュータ上にある作業コピー (あるいは 別の Subversion クライアント) が、あなたがユーザ認証されているからという 理由だけで、間違って同じファイルに変更点をコミットしてしまうことは 不可能です。言い換えるとロック・トークンは Subversion に関係したあるソフトウェア が、他の場所での作業によって台無しにされるのを防ぎます。(今の例では、 もし別の作業コピーからファイルに対する修正を本当に行う必要があるなら ロックを解除し、もう一度そのファイルに対してロックする必要があるで しょう)。

これで Harry は `banana.jpg` をロックし、Sally は そのファイルを修正したり削除したりできなくなりました:

```
$ whoami
sally

$ 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)
```

しかし Harry は banana の色合いをもう少し黄色くしたあと、その変更点を コミットすることができます。理由はロック所有者としての認可を受けている からであり、彼の作業コピーにはそのための正しいロック・トークンがあるためです:

```
$ whoami
harry

$ svn status
M    K banana.jpg

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

$ svn status
$
```

コミット実行後 **svn status** がロック・トークンが もう作業コピーに存在していないことを示しているのに注意してください。 これが **svn commit** の普通の動作です: 作業コピーを (あるいは一覧表を用意していた場合はそのリストを) 調べてコミット トランザクションの一環として検出したすべてのロック・トークンをサーバに送信します。コミットが成功した後で今回関係していたリポジトリ中のすべて のロックは解除されます — そしてこれはコミット対象とは ならなかったファイルにたいしてもそうなります。 この理由はユーザがみだりにロックしないようにすること、そしてあまり長く ロックし続けないようにするためです。たとえば Harry は

おおざっぱに images という名前のディレクトリ中にある 30 個のファイル にロックしたとします。どのファイルを変更したいのかはっきりしていなかった からです。最終的に、彼は 4 個のファイルに対してだけ修正を加えました。 **svn commit images** を実行するときそのプロセスは残りファイルも 含めた 30 個すべてのロックを解除するでしょう。

この動作は **svn commit** に--no-unlock オプション を指定することで上書きできます。これは修正をコミットしたいが、 さらに別の変更する計画があり、ロックを残しておく必要があるような場合に 一番よく使われます。この動作は実行時 config ファイル に no-unlock = yes を設定することによって半永久的に調整することもできます (項 7.2 を見てください。)

もちろんファイルをロックした後、修正を必ずコミットしなくてはならない という義務はありません。ロックは単に **svn unlock** コマンドを利用していつでも解除することもできます:

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

7.4.2 ロック状況の調査

誰かがロックしているせいでコミットに失敗した時には、原因は割と簡単に 調べることができます。一番簡単な方法は **svn status –show-updates** を実行することです:

```
$ whoami
sally

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

この例では Sally は自分の作業コピーの foo.h が古いだけでなくコミットしようと思っている二つの修正したファイルの 片方はリポジトリ中でロックされていることもわかります。 O の記号は 「Other(他の)」 の意味で ロックがファイル上に存在していてそれをしたのは誰か別の人である という意味になります。彼女がコミットしようとしても raisin.jpg 上のロックが邪魔をするでしょう。Sally はさらにロックしたのは誰で、いつ、 どうしてロックしたのかも知りたいとします。今度は **svn info** が答えを教えてくれます:

```
$ 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: 2005-01-25 12:43:04 -0600 (Tue, 25 Jan 2005)
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2005-02-16 13:29:18 -0500 (Wed, 16 Feb 2005)
Lock Comment (1 line):
Need to make a quick tweak to this image.
```

svn info は作業コピー中のオブジェクトの調査にも利用 できますが、リポジトリ中のオブジェクトに対しても調査することができます。 **svn info** の引数で作業コピーのパスを指定した 場合には作業コピーにキャッシュされているすべての情報が表示されます; ロックに関する上記メッセージのすべては作業コピーがロック・トークンを 持っていることを示しています。(ファイルが別のユーザか別の作業コピーに よってロックされている場合、作業コピーパスでの **svn info** はロックに関する情報をまったく表示しません)。 **svn info** の引数が URL なら情報はリポジトリ中のオブジェクトの 最新バージョンに関するものになります; ロックについての表示はそのオブジェクト の現在のロック状況を示しています。

それで今回の具体的な例の場合、Sally は Hally が「ちょっとした 修正」のために 2 月 16 日にそのファイルをロックしたことが わかります。今は 6 月であるので、Sally は多分 Hally は自分がロックしたことを 忘れてしまっているのではないかと考えます。彼女は Harry に電話してロック をはずしてくれるように頼むかも知れません。彼がつかまらなければ彼女は 自分で強制的にロックを解除するか、システム管理者にそうしてもらうように頼む かも知れません。

7.4.3 ロックの解除と横取り (steal)

リポジトリのロックは不可侵のものではありません; それはロックした人によっても あるいはまったく別の人によっても解除することができます。ロック作成者以外 の別の人がロックを取り除いた場合、ロックは解除された と言います。

管理者にとってはロックを解除するのは簡単です。**svnlook** と **svnadmin** プログラムはリポジトリに対して直接 ロック状況を表示したり解除することができます。(これらのツールに関しての より詳しい情報は 項 5.4.1 を見てください。)

```
$ svnadmin lslocks /usr/local/svn/repos
Path: /project2/images/banana.jpg
UUID Token: opaquelocktoken:c32b4d88-e8fb-2310-abb3-153ff1236923
Owner: frank
Created: 2005-06-15 13:29:18 -0500 (Wed, 15 Jun 2005)
Expires:
Comment (1 line):
Still improving the yellow color.

Path: /project/raisin.jpg
UUID Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
```

```
Owner: harry
Created: 2005-02-16 13:29:18 -0500 (Wed, 16 Feb 2005)
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'.
```

さらに興味深いオプションがあって、ネットワーク越しに人のロックを 解除することができます。これには unlock コマンドに対して `--force` を渡すだけです:

```
$ whoami
sally

$ svn status --show-updates
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.
```

サリーが最初に unlock に失敗したのは自分の作業コピー中のファイルに対して 直接 **svn unlock** を実行したのに、そこにはロックが存在 していなかったためです。リポジトリから直接ロックを取り除くには **svn unlock** に URL の引数を渡す必要があります。 URL をロック解除しようという最初の試みには失敗していますが、それは ロック所有者の認可を受けていない (し、ロック・トークンも持っていない) ため です。しかし `--force` オプションを渡すと、認証と認可の 要求は無視され、他の人によって作成されたロックは解除されます。

もちろん単にロックを解除するだけでは十分ではないでしょう。上記の例では Sally は Harry がずっと長いこと忘れていたロックを解除したいだけではなく 自分自身の作業のためにそのファイルを再ロックしたいのが普通でしょう。 **svn unlock –force** のあとで **svn lock** を実行すれ ばうまくいきます。しかし、この二つのコマンドの間に誰か別の人がロックして しまうわずかな可能性があります。もっと簡単な方法はロックを 横取り

する *(steal)* ことであり、これはロック解除と 再取得を不分割な一まとまりの処理として実行します。これには **svn lock** に `--force` オプションを 指定します:

```
$ 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'.
```

ロックを解除しようと横取りしようと、Harry はびっくりするかも知れません。 Harry の作業コピーにはまだ最初にロックを取得したときのロック・トークンがある のにロックそのものはもう存在していないからです。そのロック・トークンは 無効になった *(defunct)* と言います。 そのロック・トークンによって表されているロックは解除されたか (すでに リポジトリに存在していない)、横取りされたか (別のロックに置き換わった) のいずれかです。どちらの場合も Harry はリポジトリに対して **svn status** コマンドを実行することで様子をつかむことが できます:

```
$ whoami
harry

$ svn status
     K raisin.jpg

$ svn status --show-updates
     B         32   raisin.jpg

$ svn update
  B  raisin.jpg

$ svn status

$
```

リポジトリロックが解除された場合、 **svn status –show-updates** はファイルの隣に B (Broken の意味) の記号を表示します。 古いトークンに変わって新たしいロックが存在している場合だと T (sTolen) の記号を表示します。 また **svn update** は無効になったすべての ロック・トークンを表示し、作業コピーから取り除きます。

ロックのポリシー

ロックをどの程度厳密にとらえるかはシステムごとに違った考えかたがあります。 人によってはロックは非常に厳密なものであって、最初に作成した人か管理者に よってのみ解除できなくてはならないと言います。彼らの論点は、もし誰もが ロックを解除できるとすれば混乱が生じ、ロックの機構が目的とする機能が うまく提供できないというものです。別の人たちはロックはまずは、そしてほとんどの 場合はコミュニケーションのための道具であると言います。ユーザが他人のロック を頻繁に解除するようだと、それはそのチームがうまく

協調して作業することに失敗して いることを意味していて、それはソフトウェアが提供する範囲外の問題である と考えます。

Subversion はデフォルトでは「柔軟な」アプローチをとりますが それでも管理者がフックスクリプトを利用することによって厳密なポリシーを 作ることも認めています。ロックがすでに存在しているかどうかによって これら二つのフックプログラムはある特定のユーザがロックを解除したり横取りしたり するのを許すかどうかを決めることができます。 `post-lock` と `post-unlock` のフックも有効で、それはロック処理の後で email を送信するのに利用する ことができます。

リポジトリフックについてのより詳しい理解には 項 5.3.1 を見てください。

7.4.4 ロックのコミュニケーション

svn lock と **svn unlock** がどうやってロックを作ったり解放したり強制解除したり横取りしたりするか を見てきました。これは特定のファイルに対する直列化したコミットをしたい という目標を満足するものです。しかし、作業時間を無駄にしないという、 より大きな問題についてはどうなのでしょうか?

たとえば Harry がある画像ファイルをロックしてから編集し始めたとしましょう。 いっぽう、かなり離れた場所にいる Sally も同じことがしたかったとします。 彼女は **svn status –show-updates** を実行することを知らないので Harry が 既にそのファイルをロックしていることを知ることができません。彼女は そのファイルを何時間かかけて編集し、その自分の修正点をコミットしようとして はじめてそのファイルはロックされているか、彼女のファイルが最新ではない ことに気づきます。どうであれ、彼女の変更は Harry のものとマージすることが できません。二人のうちのどちらかが自分の作業を捨てなければならず 多くの時間が無駄になります。

この問題に対する Subversion での解決策は編集を始める前に ユーザにそのファイルをまずはロックすべきであることを思い起こしてもらうための 仕組みを提供することです。

この仕組みは特殊な属性を `svn:needs-lock` 用意することで 実現しています。この属性がファイルにつくと (この場合の属性値はどのようで あってもかまいません)、ファイルは読み込み専用のパーミッションを持つように なります。ユーザがファイルをロックし、ロック・トークンを取得するとファイルは 読み書き可能となります。ロックが解放されると — これは明示的にロック解放する かコミットを通じて自動的に解放されるかのどちらかですが — ファイルは再び 読み込み専用に戻ります。

こうすることで、画像ファイルにこの属性がついている場合、Sally は 編集のためにファイルを開いた時に何かおかしなことになっていることに すぐに気づくはずです。彼女が使っているアプリケーションはそのファイル が読み込み専用であることを伝えます。これで彼女は編集前にそのファイルを ロックしなくてはならないことを思いだし、こうして既に存在しているロック に気づくことになります:

```
$ /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: 2005-06-08 07:29:18 -0500 (Thu, 08 June 2005)
Lock Comment (1 line):
Making some tweaks.  Locking for the next two hours.
```

「最良の方法」は、ユーザも管理者もお互いにマージすることが できないようなファイルには常に `svn:needs-lock` 属性をつけておくというものです。この技法はロック機能を利用する上での 良い習慣であり、無駄な作業を防ぐことができます。

この属性はロックシステムとは独立して機能するコミュニケーション用の 仕組みであることに注意してください。言いかえるとどのようなファイル も、このプロパティーがあるかどうかにかかわらずロックすることができます。逆にこの属性の存在だけで、コミット時にリポジトリから常に ロックを要求されるということにはなりません。

これで完璧とはいきません。ファイルがこの属性を持っていたとしても 読み込み専用の警告機能が常に動作するとは限りません。アプリケーション の間違った動作によっては、警告を出さずに黙ってそのファイルに対する 編集を許し、保存してしまう結果、読み込み専用ファイルを「乗っ取って」 しまうこともあるでしょう。残念なことにこのような状況に対して Subversion ができることはあまり多くはありません。

7.5 ペグ・リビジョンと操作対象リビジョン

ファイルやディレクトリをコピーしたり移動したり名称を変更したりすること、 また、ファイルを作って、一度消したあとにもう一度同じ名前でファイルを作 ること — このようなことはコンピュータを使う上で常に、また当たり前 のようにやっている操作です。Subversion もまた、あなたにその手の操作を 当たり前にやってほしいと思っています。Subversion のファイル管理は非常 に自由であり、バージョン化されていないファイルを操作するときに期待され る動作とほとんど同じような柔軟性を、バージョン管理されているファイルに 対してサポートしています。しかしこのような柔軟性はリポジトリの一生を通 じて、バージョン管理されている資源はいろいろなパス名をとること、逆にあ る特定のパス名がまったく別のバージョン管理された資源を表す可能性がある ことも意味しています。

Subversion はあるオブジェクトのバージョンの履歴がそのような「所在 地の変更」を含むような場合でも、そのことを非常に正確に気づきます。 例えば先週ファイル名を変えた、あるファイルのすべての履歴を表示させよう としても、Subversion はそのログ全体をうまく表示してみせます — つ まり名称変更が実際に起きたリビジョンと、それに関連した名称変更の前と後 にあるすべてのリビジョンについてという意味です。そんなわけでほとんどの 場合、あなたは、何かを特に意識する必要もありません。しかし場合によって は Subversion は、あいまいさを解消するためにあなたの助けを必要とするこ ともあります。

一番簡単な例はバージョン管理下にあるディレクトリやファイルがいったん削除され、 その後おなじ名前のディレクトリやファイルがあらためて作成されてから バージョン管理項目として追加されたような場合です。とうぜん削除したファイルと あとから追加したファイルはまったく別のものです。両者は単にたまたま同じパス名を 持っているというだけのことです。このファイル名を `/trunk/object` としておきましょう。さてこの場合、`/trunk/object` の履歴に関する問い合わせを Subversion にする場合、どんな意味になるので しょうか? 現にそのパスに存在しているほうのファイルについての問い合わせ でしょうか、あるいはその場所から以前いったん削除したほうのファイルに ついての問い合わせでしょうか? あるいは全履歴中で、とにかくそのパスに あったオブジェクトに対して実行したすべての操作についての意味でしょうか? 確かに Subversion にはあなたが本当に知りたいと思っていることについての ヒントを与えてやることが必要です。

さらにありがたいことに、ファイル移動の操作によってバージョン管理上の履 歴はさらにずっとややこしい

ことになります。たとえば、 concept という名前のディレクトリがあって、その中 にはまだ始まったばかりの、ままごとプロジェクトがあったとします。しかし 最終的に、その考えはしっかりしたものになり、プロジェクトは真面目な利用 ができるような状態となったので、そのプロジェクトに聞いたこともないよう な名前をつけることにしました。[*9] そのソフトの名前 を Frabnaggilywort としましょう。ここから先はプロジェクトの新しい名前 にふさわしいようにディレクトリを concept から frabnaggilywort に変えるのはもっともな話しです。 開発が進み frabnaggilywort はバージョン 1.0 をリリースすることになり、 それは多くの人々によってダウンロードされ、日々利用され、みんな幸せにな りました。

いい話しです。まったく。しかしこれで話しが終わるわけではありません。あ なたは企業家です。すでに次の着想を得ています。あなたは新しいディレクト リ concept を作り、次の開発サイクルが始まります。 実際にはこのサイクルは何年にもわたって何度も繰り返し発生します。いつも concept ディレクトリから始め、時にはそのアイディア を膨らませるためにディレクトリの名称は変更され、時にはそのアイディアを ボツにするためにディレクトリは削除されます。あるいは、さらにややこしい 場合、いったん concept を別の名前に変えた後、何か 理由があって再び concept に名前を戻したりすること もあるでしょう。

この手の話しになったとき、Subversion に対してファイルパス名を再訪問す るように指示するのはシカゴの West Suburbs にいる運転手に east down Roosevelt Road まで行き、そこでメインストリートに左折するように指示す るのと少し似ています。20 分もしないうちに Wheaton, Glen Ellyn, そして Lombard にある「メインストリート」を横切ることになるでしょ う。しかし、これらはすべて別の道です。私たちの運転手には — そし て Subversion に対しても同様に — 正しい場所に行ってもらうために はもう少し詳しい情報が必要になります。

バージョン 1.1 で、Subversion はどのメインストリートに行きたいかをもっ と正確に伝える方法を取り入れました。これはペグ・リビジョン と呼ばれ、Subversion に対して特定の履歴ラインを指定するた めだけの目的で用意されたものです。ある特定の時点では — あるいはも う少し正確にはある特定のリビジョンでは — あるパス名はせいぜい一つ のバージョン管理されたリソースによって利用されるだけなので、パス名とペ グ・リビジョンの組み合わせはある特定の履歴ラインを指定するのに十分な情 報になります。ペグ・リビジョンは Subversion のコマンドラインクライアン トからアットマーク構文によって指定されますが、 これはその構文が 「アットマーク」 (@) と ペグ・リビジョンをパス名の最後につける形になるからです。このパス名はペ グ・リビジョンに存在しているパス名です。

ではこの本の中でいつも出てくる --revision (-r) で指定 されるほうのリビジョンは何と言われるのでしょうか？ こちらは操作対象リビジョン (あるいはリビジョンの範囲を指定する場合 には、操作対象リビジョン範囲) と呼ばれます。いっ たん特定の履歴ラインがパス名とペグ・リビジョンによって指定されると Subversion は操作対象リビジョンに対して要求された操作を実行します。 この話をシカゴの道順のたとえで説明すると、 606 N. Main Street in Wheaton に行きたい場合だと [*10] 「Main Street」がパス名に、「Wheaton」がペグリビ ジョンにあたるものと考えることができます。この二つの情報によって実際に 行ってほしい道順 (メインストリートの北、あるいは南) を特定することができ、 行き先を探すのに、間違った別のメインストリートを右往左往せずにすみます。 そして、操作対象リビジョンにあたる 「606 N.」 によって実際 に行きたい場所を正確に知ることができるというわけ です。”ペグ・リビジョン” のアルゴリズム

コマンドラインクライアントで以下の形のコマンドを指定したとき:

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

[*9] 「名前なんかつけ ちゃだめだ。いったん名前をつけたら、それにとらわれるようになってしまう よ。」 — Mike Wazowski

[*10] 606 N. Main Street, Wheaton, Illinois, は Wheaton 歴史センターのある場所です。 たずねてみてください — 「歴史センター」だって? 確かに….

... 以下のようなアルゴリズムが実行されます:

- リビジョン *PEG-REV* に移り、*item* を見つけます。この形式でリポジトリ中の特定のオブジェクトを指し示すことができます。
- そのオブジェクトの履歴を後ろ向きに (名称変更があった場合も考慮しながら) リビジョン *OPERATIVE-REV* にある祖先まで たどります。
- その祖先に対して指定した処理を実行します。これがどのリビジョンにあろうと、 また、その時点で何という名前で存在していようとも、です。

明示的にペグ・リビジョンを指定しなくても、それが考慮されることに注意して ください。このデフォルトは、作業コピー中のアイテムの場合は BASE リビジョン、 URL 中のアイテムの場合には HEAD リビジョンになります。

ずっと以前に作っておいたリポジトリがあって、リビジョン 1 で最初の `concept` ディレクトリとその中にある `IDEA` という名前のファイルを追加したとしましょう。 このファイルは実際のコンセプトについての説明が書いてあります。 実際のソースコードを追加したり修正したりしてたくさんのリビジョンが 追加されたあと、リビジョン 20 でこのディレクトリを `frabnaggilywort` に名称変更したとしましょう。 リビジョン 27 で新しい着想を得たので、新規に `concept` ディレクトリを作り、またその中に新規に `IDEA` ファイルを置いて、その内容を書いておきます。そして良くできたロマンス 小説よろしくその後 5 年間で 20,000 リビジョンにまで達したと しましょう。

こうして何年もたってから、リビジョン 1 での `IDEA` ファイルがどんな具合であったか知りたくなったとします。しかしこの場合 Subversion は 現在のファイルの過去がリビジョン 1 でどうであったのか、あるいはとにかくそれが何であれ、リビジョン 1 で `concepts/IDEA` という名前で存在していたファイルの 内容が知りたいのかを知る必要があります。この二つの質問の答えは明らかに 別のものになりますが、ペグ・リビジョンがサポートされているので このどちらの質問もすることができます。現在の `IDEA` ファイルが過去のリビジョン 1 でどうであったかを知りたい場合には以下の ようにします:

```
$ svn cat -r 1 concept/IDEA
subversion/libsvn_client/ra.c:775: (apr_err=20014)
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

この例ではもちろん、現在の `IDEA` ファイルは リビジョン 1 では存在しなかったので Subversion はエラーを出します。 上のコマンドはペグ・リビジョンを明示的に指定する、より長い形の形式 の略記法です。長い形式は以下のようになります:

```
$ svn cat -r 1 concept/IDEA@BASE
subversion/libsvn_client/ra.c:775: (apr_err=20014)
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

実際に実行すると、やはり期待したとおりの結果になります。ペグリビジョン は一般的には作業コピーのパスにつけた場合には BASE の 値 (作業コピーに現在存在するリビジョン) が、また URL につけた場合には HEAD の値が、それぞれデフォルト値になります。

今度はもう一方の質問をしてみましょう。つまり — リビジョン 1 の時点で、とにかく concepts/IDEA という名前で 存在していたファイルの内容はどんなものでしたか? これを指定する ために明示的なペグ・リビジョンを使います。

```
$ 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.
```

正しい出力になっているようです。このテキスト中で、 frabbing naggily worts という単語が出てきているところを見ると、現在 Frabnaggilywort と いう名前で呼ばれているソフトウェアについて説明したファイルであることは まず間違いないところでしょう。実際、明示的なペグ・リビジョンと明示的な 操作対象リビジョンの組合せによってこれを確認することができます。 HEAD では Frabnaggilywort プロジェクトが frabnaggilywort ディレクトリにあることはわかってい ます。それで HEAD で frabnaggilywort/IDEA というパス名で特定される履歴 ラインが、リビジョン 1 ではどのようであったかを知りたいのだ、というこ とを指定してみます。

```
$ 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.
```

そして、ペグ・リビジョンも操作対象リビジョンも、時には非常に重要な意味 を持ちます。例えば、frabnaggilywort がリビジョン 20 で HEAD から削除されているが、リビジョン 20 では 存在していたことを知っていて、その時の IDEA ファイルが、リビジョン 4 と リビジョン 10 の間でどのように変化したか を見たいとします。これには、ペグ・リビジョン 20 を、そのリビジョンで Frabnaggilywort の IDEA ファイルを保持していた URL の後につけて指定します。また同時に操作対象リビジョン範囲として 4 と 10 を指定します。

```
$ 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.
```

ありがたいことにほとんどのユーザはこんな複雑な状況に出会うことはありま せん。しかし万一そんなことになった場合には、Subversion がファイル名の あいまいさを解消するにはペグ・リビジョンを追加で指定してやれば良いこと は覚えておいてください。

7.6 外部定義

ときどき、いくつかの別のチェックアウトによって、一つの作業 コピーを作るのが便利なことがあります。たとえば、リポジトリの別々の場所に ある異なるサブディレクトリがほしいとか、リポジトリ自体が別であるとかです。 そのようなことを手で設定することももちろんできます — **svn checkout** を使ってネストした作業コピー構造のような ものを作るわけです。しかし、このレイアウトがリポジトリを使うすべての人に とって重要であれば、他の全員もあなたがやったのと同じチェックアウト操作を する必要があります。

幸運なことに、Subversion は外部定義をサポート しています。外部定義は、ローカルディレクトリをバージョン管理された リソースの URL — や特定のリビジョン — に結びつけるものです。Subversion では、`svn:externals` 属性を使って外部定義をグループにして 宣言します。**svn propset** か、**svn propedit** コマンドでこの属性を作ったり修正したりすることができます (項 7.3.1 を参照してください)。 この属性はバージョン管理されたディレクトリに設定され、 その値は (属性が設定されたバージョン管理されたディレクトリに相対的な) サブディレクトリと、完全に修飾された Subversion リポジトリ URL の絶対パス名を一行とした複数行テーブルです。

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

`svn:externals` が便利なのは、ひとたびバージョン管理 下のディレクトリに設定してしまえば、そのディレクトリのある作業コピーを チェックアウトした人は誰でも外部定義の恩恵にあずかることができる ところです。言い換えると、誰かがそのようなネストした作業コピーの チェックアウトを定義すれば、他の人は誰もそれについて悩まなくて済む ということです — Subversion は、もともとの作業コピーのチェックアウトの上にも外部作業コピーをチェックアウトすることができます。

前の外部定義の例を見てみましょう。誰かが `calc` ディレクトリの作業コピーをチェックアウトすると、Subversion は その外部定義にあるアイテムも続けてチェックアウトします。

```
$ 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
…
```

もし、外部定義を変更する必要がある場合、通常の属性変更 サブコマンドを使ってやることができます。`svn:externals` 属性への変更をコミットするとき、 Subversion は次の **svn update** を実行するときの 変更された外部定義に対してチェックアウトするアイテムを同期します。 同じことが、他の人が作業コピーを更新し、あなたが変更した外部定義 を受け取るときにも起こります。

svn status コマンドも外部定義がチェックアウトされた サブディレクトリごとに X の状態コードを表示する形で 外部定義を認識し、外部アイテムそれ自身の状態を表示するためにそれらの サブディレクトリに再帰的に降りていきます。

> ティップ
>
> 
>
> 外部定義のすべてに明示的なリビジョン番号を使うことを強くお勧めします。 これによって異なる外部定義のスナップショットを引っ張ってくる時にどれを 持ってくれば良いか決めることができ、正しいものを持ってこれるように なります。自分ではまったく制御できないサードパーティーのリポジトリに 対する変更点に対する変更に対して冷静に対処できるという当たり前の利点 のほか、明示的なリビジョン番号はまた、以前のあるリビジョンに作業コピーを 戻す場合に、外部定義もその以前のリビジョンでの内容に戻るわけです が、それはまた、あなたのリポジトリがその以前のリビジョンであった 時に彼らが見たいと思う状態に合う形で外部作業 コピーが更新されることを意味しています。ソフトウェアプロジェクトにおいて これが複雑なソースコードの古いスナップショットを再構築する時の成否の 鍵になります。

しかし現在の Subversion での外部定義のサポートは少し誤解されています。 まず、外部定義はディレクトリだけを指すことができ、ファイルを指すことは できません。次に外部定義は相対パス (`../../skins/myskin` のようなもの) を指すことはでき ません。さらに外部定義のサポートを通じて作られれた作業コピーは最初の作業コピーとは まだ独立したものです (つまり、`svn:externals` 属性が 実際に設定されているか

も知れないバージョン化されたディレクトリからは 独立したものです)。そして Subversion は、この分離されていない作業コピー 上に対してだけ正しく働きます。このため例えば、もし一つ以上のの外部作業 コピーに対して行った変更をコミットしたい場合、その作業コピー上で明示的 に **svn commit** を実行する必要があります — 最初の作 業コピーでのコミットが外部の作業コピーのコミットを連鎖的に発生させる ことはありません。

またその定義自身も URL の絶対パス名を利用するのでそのパスに関係する ディレクトリの移動やコピーは外部のものとしてチェックアウトした ものに影響を与えません (相対的なローカルターゲットディレクトリはもちろん その名称変更されたディレクトリと共に移動しますが)。これはある種の 状況では混乱の元になるかも知れません — あるいはいらいらさせる かもしれません。たとえば、同じ開発ラインの別の部分を指しているような `/trunk` 開発ライン上のディレクトリで外部定義を 使い、それから **svn copy** でそのラインのブランチを どこか別の場所 `/branches/my-branch` に作ったと すると、新しいブランチ上のアイテムに定義された外部定義はまだ `/trunk` 中のバージョン化されたリソースを参照して います。また、もし作業コピーの親を (**svn switch –relocate** を使って) 再設定する必要がある場合、外部定義がそれに付随することは ありません。

最後に、**svn** のサブコマンドが外部定義を認識しない ようにしたいこともあります。そうしないと外部定義処理の結果として作成 された外部作業コピーに対する処理が実行されてしまうような場合です。 これはサブコマンドに `--ignore-externals` オプションを指定すれば解決します。

7.7 ベンダーブランチ

ソフトウェアを開発する場合が典型的な例ですが、バージョン管理で保守しているデータが しばしば誰かのほかのデータに密接に関係しているか、あるいは依存している ことがあります。一般的にプロジェクトで要求されることは、プロジェクトの 安定性を損なうことなく、外部の資源によって提供されるデータをできる限り最新に保つことです。この考え方は、あるグループによって作られた情報が もう 1 つのグループによって作られるものに影響を与える場合、常に成り立ちます。

たとえば、ソフトウェア開発者がサードパーティー製のライブラリを利用する アプリケーションの開発に取り組んでいるとします。Subversion は Apache ポータブル 実行時ライブラリと、ちょうどそのような関係を持っています。 (項 8.3.1 参照)。Subversion のソースコードは すべての可搬性の要求を満たすために、APR ライブラリに依存しています。 Subversion の開発の初期の段階では、プロジェクトは APR の API の変更を非常に 正確に追いかけていました。常に、ライブラリコードの荒波の、「最先端」についていきました。いまでは APR も Subversion も開発が落ち着い てきたので、Subversion はよくテストされ、安定したリリース状態にある バージョンの APR ライブラリ API とのみ同期をとっています。

もしプロジェクトが他の人の情報に依存しているなら、その情報と自分の ものとを同期させるためのいくつかの方法があります。 一番大変な方法ですが、自分のプロジェクトのすべての貢献者に対して 口頭または文書で手続きを伝えることができます。プロジェクトに必要な サードパーティーの情報の特定のバージョンを確実に手に入れることを 伝えます。もしサードパーティーの情報が Subversion リポジトリで管理 されているなら、Subversion の外部定義を使ってその情報の特定のバージョンを 作業コピーディレクトリ中のある場所へ効果的に「結びつける」こと ができるでしょう (項 7.6 参照)。

しかし、ときどき自分のバージョン管理システムでサードパーティーのデータ に加えた独自の変更を管理したいこともあります。ソフトウェア開発の 例に戻って説明すると、プログラマは自分自身の目的のために、サードパーティー のライブラリに変更を加える必要があるかも知れません。このような修正は 新しい機能の追加であったりバグフィックスであったりするかも知れませんが、 それはサードパーティーのライブラリの公式なリリースの一部になるまでに 限り管理すべきものです。あるいは、変更は決してライブラリ保守担当には伝えられず、ソフトウェア開発者の特殊な要求に合うようなライブラリを 作り上げるための独自の修正点としてずっと残り続けるかも知れません。

ここで、面白い状況に直面します。あなたのプロジェクトは、パッチファイルを 適用したりファイルやディ

レクトリーを完全に別のものに置き換えるような、若干 バラバラな方法でサードパーティーのデータへ独自の修正を加えることができました。 しかし、このようなやり方ではすぐに保守する上で頭痛の種になるので、あなたの独自の変更をサードパーティーのデータに適用する仕組みが必要となります。 さらにあなたが追跡するサードパーティーのデータのそれぞれの連続したバージョンに それらの変更を再生する仕組みも必要となります。

この問題に対する解決は、ベンダーブランチを 使うことです。ベンダーブランチはサードパーティーあるいはベンダー よって提供された情報を含んでいる、こちらのバージョン管理 システム中のディレクトリツリーです。それぞれの バージョンのベンダーのデータで、自分のプロジェクトに取り込もうと 考えているもののことを、ベンダードロップと いいます。

ベンダーブランチは二つの鍵となる利点があります。まず、 自分のバージョン管理システムに、現時点でサポートされている ベンダードロップを格納することによって、プロジェクトのメンバーは 正しいバージョンのベンダーデータを使っているかどうかの心配をする 必要がなくなります。彼らはいつもの作業コピーの更新の一環として、 簡単に正しいバージョンを受け取ります。 次に、データは自分たちの Subversion リポジトリに あるので、それに対する独自の修正を決まった場所に格納することができます。 — 自分たちの独自の修正で置き換えるような自動化された (あるいは最悪の 場合、手でやる) 方法を用意する必要がなくなります。

7.7.1 一般的な、ベンダーブランチを管理する方法

ベンダーブランチの管理は一般的にはこんな感じでやります。 最上位ディレクトリを作り (`/vendor` のようなもの) そこにベンダーのブランチを置きます。それから最上位ディレクトリの サブディレトクリにサードパーティーのコードをインポートします。 それからそのサブディレトクリを、適当な場所にある、自分の主系開発の ブランチにコピーします (たとえば`/trunk` など)。 ローカルな変更は常に主系開発ブランチに対して行います。 追いかけているコードの新しいリリースのたびに、それをベンダーブランチに 持っていき、変更点を`/trunk` にマージします。そして、 ローカルの変更と、ベンダーの変更の間の衝突を解消します。

多分、例をあげると、このステップをはっきりさせることができるかも 知れません。あなたの開発チームがサードパーティーの複素数値計算 ライブラリ libcomplex を使った計算プログラムを作っているとします。 まず、ベンダーブランチの初期生成をし、それから最初のベンダードロップ をインポートします。ここではベンダーブランチのディレクトリを `libcomplex` と呼び、私たちのコードドロップは `current` と呼ばれる私たちのベンダーブランチの サブディレクトリの中に置かれます。**svn import** は 必要なすべての中間的な親ディレクトリを作るので、このような複数の ステップを実際には一つのコマンドで実行することができます。

```
$ svn import /path/to/libcomplex-1.0 \
             http://svn.example.com/repos/vendor/libcomplex/current \
             -m 'importing initial 1.0 vendor drop'
…
```

これで、libcomplex のソースコードを`/vendor/libcomplex/current` に持ってくることができました。このバージョンにタグ付けし、 (項 4.7 参照)、主系開発ブランチにコピーしま す。私たちのコピーは既存の `calc` プロジェクトディレクトリ 中の `libcomplex` という新しいディレクトリを作ります。 これが新たに独自の修正を加えるためのベンダーデータのコピーになります。

```
$ 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'
…
```

プロジェクトの主系のブランチをチェックアウトします。これは最初の ベンダードロップのコピーを含んでいます — そして、libcomplex コードの修正に入ります。もう知っているように、これで修正された libcomplex は、私たちの計算プログラムに完全に統合されています。 [*11]

何週間かたって、libcomplex の開発者はライブラリの新しいバージョンを リリースしました — バージョン 1.1 としましょう — それは われわれがほしかったいくつかの機能と関数を含んでいます。この新しいバージョンにアップグレードしたいものですが、既に手元にあるバージョンに 対する修正を失いたくはありません。既に示唆したように、 本質的にわれわれがやらなくてはならないのは、libcomplex1.1 のコピー で libcomplex1.0 を置き換え、前にやった独自の修正を、新しいライブラリ のバージョンにも再び適用することです。しかし実際には私たちはこの問題に 対して別の方法で対処したいのですが、それはバージョン 1.0 と 1.1 の間 に libcomplex におきた変更点を私たちの修正されたコピー上に適用するという ものです。

このアップグレードをやるのに、わたしたちはベンダーブランチのコピー をチェックアウトし、`current` ディレクトリにある ソースコードを、新しい libcomplex 1.1 のソースコードで置き換えます。 文字通り完全に新しいファイルで既存のファイルを上書きしますが、多分 それは既存のファイルやディレクトリの上に libcomplex 1.1 のリリース用 tarball を展開することになるでしょう。ここでの目的は私たちの `current` ディレクトリ中に libcomplex 1.1 のコードの みを含むようにすることであり、同時にすべてのコードをバージョン管理下に あることを保証するということです。あ、もちろんバージョン管理の履歴に 対しての混乱を最小にとどめるような方法でそうしたいのです。

1.0 のコードを 1.1 のコードに置き換えた後では、 **svn status** はローカルな修正を加えたファイルの一覧とバージョン化されていないファイル、あるいは失われたファイルも表示 するでしょう。いままで述べたような手順を実行していたのなら、バージョン 化されていないファイルは libcomplex の 1.1 のリリースで新しく導入され たようなものだけのはずです — そのようなファイルをバージョン管理下 に置くのに **svn add** を実行します。失われたファイルは 1.0 では存在していたが、1.1 では存在しないようなものに対応しているので **svn delete** を実行します。最後に、私たちの `current` 作業コピーが libcomplex 1.1 のコードのみ を含むようになれば、いまの変更をコミットして、つじつまを合わせます。

私たちの `current` ブランチはこれで新しい ベンダードロップを含むようになります。新しいバージョンに (バージョン 1.0 のベンダードロップに対して前にしたのと同じ方法で) タグづけをして、 それから前のバージョンのタグと新しい現在のバージョンとの間の差分を 私たちの開発ブランチにマージします。

```
$ 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
```

[*11] そして、もちろんあなたのことですから、バグも完全になくなっている、と。

```
$ svn commit -m 'merging libcomplex-1.1 into the main branch'
…
```

簡単な場合だと、この新しいバージョンのサードパーティーツールは、 ファイルとディレクトリの観点から見ると、前のバージョンと同じように 見えます。libcomplex のどのソースファイルも削除されたり名称変更された り別の場所に移動されたりはしません — 新しいバージョンは単に前の ものからテキストの内容の修正を受けただけに見えます。理想的な状況では 私たちの修正は新しいライブラリのバージョンに対してきれいに適用され、 複雑なことや、衝突は一切起きません。

しかし、ものごとというものは常に単純であるとは限りません。 実際、ソースファイルはソフトウェアのリリース間であちこち 動くのが普通です。これはわたしたちの修正が新しいバージョンのコード でも正しいということを確認する作業を複雑にしますし、新しいバージョン での修正を手でもう一度やる必要がある状況に、簡単に落ち込んでしまう ことがあります。Subversion が、与えられたソースファイルの（以前の位置を含めての） 履歴について知っていればライブラリの新しいバージョンのマージのステップは とても単純になります。しかし、わたしたちは、Subversion に ソースファイルのレイアウトがベンダードロップ間でどんな風に変わったかを教えてやる責任があります。

7.7.2 svn_load_dirs.pl

いくつかのファイルの削除、追加、移動があったベンダードロップは サードパーティーデータのアップグレードの手順を複雑にします。 それで Subversion はこの手続きを支援するために **svn_load_dirs.pl** スクリプトを用意しています。 このスクリプトは一般的なベンダーブランチの管理手続きで言ったような インポートのステップを自動化し、間違いを最小にすることができます。 サードパーティーデータの新しいバージョンを主系開発ブランチにマージする ためのマージコマンドを使う責任はまだ残っているものの、 **svn_load_dirs.pl** はより早く簡単にこの処理まで 到達する助けになります。

簡単に言って、**svn_load_dirs.pl** は **svn import** の拡張で、いくつかの重要な 特徴を持っています:

- いつでも、このプログラムを実行して、 リポジトリにあるディレクトリを、完全にそれに一致した外部ディレクトリに持って行き、必要なすべての追加、削除を実行し、 さらにオプションで移動処理も行います。
- このプログラムは、Subversion が必要とする中間的なコミット間で 必要な複雑な一連の処理を注意深く実行します — たとえば ファイルやディレクトリの名称変更を二回やる前など。
- それは、オプションで新しいインポートされたディレクトリを タグ付けします。
- それはオプションで、正規表現にマッチするファイルとディレクトリ に任意の属性を追加します。

svn_load_dirs.pl は三つの必須パラメータを とります。最初の引数は作業対象となるベースになる Subversion ディレクトリの URL です。この引数のあとには URL が続きます — 最初の引数に相対的な 形で — ベンダードロップはそこにインポートされます。最後に 三番目の引数はインポートするローカルディレクトリです。前の例を 使うと、典型的な **svn_load_dirs.pl** の実行は こんな感じになります:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

`-t` オプションにタグ名称を指定して、新しいベンダードロップ をタグ付けするように **svn_load_dirs.pl** に指示することが できます。

```
$ svn_load_dirs.pl -t libcomplex-1.1                                  \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                            \
                   /path/to/libcomplex-1.1
…
```

svn_load_dirs.pl を実行するとき、それは 既に存在している「現在の」ベンダードロップの内容を調べて それを指定された新しいベンダードロップの内容と比較します。 簡単な場合、片方のバージョンにあってもう一方にはないようなファイル はなく、スクリプトは新しいインポートを特に問題なく実行します。 しかし、もし、バージョン間でファイルレイアウトに違いがある場合、 **svn_load_dirs.pl** はこの違いをどうやって解決するか たずねてきます。たとえば、libcomplex のバージョン 1.0 で `math.c` だったファイルは libcomplex1.1 では `arithmetic.c` に 名称変更になったことを知っていることをスクリプトに教えてやることが できます。移動によって説明できないような相違点は、通常の 追加と削除として扱われます。

このスクリプトはまたリポジトリに追加される（正規表現にマッチするような）ファイルとディレクトリの属性を設定する ために、別の設定ファイルを受け付けることができます。 この設定ファイルは **svn_load_dirs.pl** で `-p` オプションを使って指定されます。 設定ファイルの各行は空白で区切られた二つまたは四つの値です: 追加されるパスに対してマッチさせる Perl スタイルの正規表現、 制御キーワード (`break` または `cont`)、そして、オプションで属性名と 属性値がきます。

```
\.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
```

追加されるパスが正規表現にマッチしたとき、その行の属性 がマッチしたパスに追加されていきます。ただし制御の指定が `break` の場合は属性の追加はその行で打ち止めになります (これはそれ以上の属性変更はこのパスに 行わないことを意味しています)。もし制御指定が `cont` — `continue` の省略形ですが — の場合はマッチング処理は設定ファイルの次の行に続いていきます。

正規表現中のすべての空白、属性名、属性値はシングルまたはダブルクオート でくくる必要があります。空白を囲むために利用しているわけではない クオート文字はバックスラッシュ文字 (\) を前に付ける ことでエスケープできます。バックスラッシュは設定ファイルを解析するとき だけクオートするので、正規表現中で必要なもの以外のほかの文字には 使わないでください。

7.8 ローカライゼーション

ローカライゼーション とはプログラムが 特定の地域に応じた動作をするように作ることを言います。あるプログラムが 数値や日付を特定の地域に応じて書式化して出力したり、その地域の言語でメッ セージを出力したり (あるいはそのような入力を受け入れたり) できるのであれ ば、そのプログラムはローカライズされて

いる と言 われます。この節では Subversion のローカライゼーションに向けた取り組み について説明します。

7.8.1 ロケールの理解

最近のほとんどのオペレーティングシステムは 「現在のロケール値」 という考え方を採り入れています — つまり、その時点で考慮されている ローカリゼーションの規約が、どの地域や国に対応しているかという値です。 このような規約は — 普通はコンピュータの実行時の設定のしくみを 通じて選択されるものですが — プログラムがデータをユーザに対して 出力する時や、ユーザからの入力を受け付けるときの動作に影響を与えます。

Unix 風のシステムでは **locale** コマンドを実行して ロケール関連の実行時設定オプションの値をチェックすることができます:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
```

出力されているのはロケール関連の環境変数とその現在値です。この例では変 数はすべてデフォルトの `C` ロケールになっています。ユー ザはこれらの変数を特定の国/言語コードの組合せに変更することができます。例えば `LC_TIME` 変数の値を `fr_CA` に すると、プログラムはフランス語を話すカナダ人に対する書式で時刻と日付を 表示するようになります。あるいは `LC_MESSAGES` 変数を `zh_TW` に設定すれば、プログラムは人が読むためのメッセー ジ中国語で表示するようになります。 `LC_ALL` 変数を設定 すると全てのロケール変数の値が同じ値になります。 `LANG` の値はどのロケール変数も設定されていない場合の デフォルト値になります。Unix システムで可能なロケールの一覧は **locale -a** を実行するとわかります。

Windows ではロケールの設定はコントロールパネルの「地域と言語のオ プション」を通じて設定することができます。その画面で利用できるロ ケール値がそれぞれどのようになっているかを確認し、選択することができ、 (かなり特殊なケースだと思いますが) 表示形式の規約をさまざまにカスタマイ ズすることすらできます。

7.8.2 Subversion でのロケール

Subversion クライアントである **svn** は、ふたつの場面で 現在のロケール値を正しく扱います。まず、`LC_MESSAGES` 変数の値を見て全てのメッセージを特定の言語で表示しようとします。たとえ ば:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

この動作は Unix でも Windows でも同じです。しかし、あなたのオペレーティ ングシステムが特定のロケールをサポートしているとしても Subversion クラ イアントがその特定の言語をしゃべることができるとは限りません。ローカラ イズされたメッセージを出力するにはボランティアによる言語ごとの翻訳が必 要になります。翻訳は GNU gettext パッケージを利用しているので、 `.mo` というファイル拡張子をもった翻訳モジュールが 結果として必要になります。たとえば、ドイツ語の翻訳ファイルは `de.mo` になる、といった具合です。このような翻訳ファ イルはあなたのシステムの、ある特定の場所にインストールされます。Unix であれば普通は`/usr/share/locale/` のような場所にな り、Windows であれば Subversion をインストールしたフォルダの中の

`share`
`locale`

フォルダなどに見つかることがよくあ ります。いったんインストールするとモジュールは翻訳を実行するプログラム の名前に変更されます。例えば `de.mo` というファイル は最終的には `/usr/share/locale/de/LC_MESSAGES/subversion.mo` の ような名前でインストールされます。インストールされている `.mo` ファイルを見れば、Subversion クライアントが実 際にはどの言語を話すことができるかがわかります。

ロケールが考慮される二番目の場面は **svn** があなたの入力を解釈する時です。リポジトリはすべてのパス名、ファイル名 そしてログメッセージを UTF-8 でエンコードされた Unicode で保存します。 この意味でリポジトリは 国際化されています — つまり、リポジトリはどのような自然言語の入力も受け入れる用意ができて います。しかしこれは Subversion クライアントは UTF-8 ファイル名とログ メッセージだけをリポジトリに送る責任があることを意味します。このため Subversion クライアントはデータをネイティブのロケールから UTF-8 へと 変換しなくてなりません。

たとえば `caff` 竪`.txt` という名前のファイルを作り、そ のファイルをコミットするときに「Adesso il caff 竪竪 pi 湛 forte」というログメッセージをつけたとします。ファイル名とログメッ セージの両方に 非 ASCII 文字が含まれていますがロケールが `it_IT` に設定されているので Subversion クライアントは そのような文字列がイタリア語であることを理解することができます。そして イタリア語の文字セットを使ってデータ UTF-8 に変換し、それから結果を リポジトリに送信します。

リポジトリはファイル名とログメッセージは UTF-8 であることを要求します がファイルの内容にはまったく考慮 しません。 Subversion はファイルの内容を単なるバイト列の並びとして扱い、クライア ント側もサーバ側もその内容のキャラクタセットやエンコーディングを理解し ようとはしません。キャラクタセット変換エラー

Subversion を利用しているとキャラクタセット変換に関係したエラー に遭遇するかも知れません:

```
svn: Can’t convert string from native encoding to ’UTF-8’:
…
svn: Can’t convert string from ’UTF-8’ to native encoding:
…
```

このようなエラーは Subversio クライアントが UTF-8 文字列をリポジトリ から受け取ったがその文字列中のすべての文字を現在のロケールのエンコーディング を使って表現できるわけではないような場合に典型的に発生します。たとえば 現在のロケールが `en_US` であるのに他の開発者が日本語の ファイル名をコミットした場合、**svn update** 実行中にファイル を受け取っているときにこのようなエラーが発生するかも知れません。

解決方法としては受け手のロケールを取り込もうとする UTF-8 データを表現 することが 可能などれかのロケールに設定するか、 リポジトリ中のファイル名やログメッセージを変更するかです。(それから その開発者にきちんと次のように伝えておくことも忘れずに — このプロジェクト はある共通の言語で開発することに決めたので、すべての開発者はそのロケール を使うように、と。)

7.9 外部差分ツールの利用

`--diff-cmd` と`--diff3-cmd` オプションや 同じような名前の実行時環境パラメータ (項 7.2.3.2 参照) がある ので、Subversion で外部差分ツール (あるいは 「diff」) とマージツールを 使うのは簡単なことだという間違った考えを持ってしまうかも知れません。 Subversion ではそのようなよく知られたほとんどのツールを使うことが できますが、このような設定に必要な努力はそれほど簡単ではないことが よくあります。

Subversion と外部 diff と merge ツールは Subversion の唯一の文脈差分 を出力する能力が GNU の diffutils の連携した呼び出しだけであった ころに由来しています。具体的には **diff** と **diff3** ユーティリティーです。Subversion が必要とする ような動作をさせるには、そのようなユーティリティーをかなりたくさんの オプションと引数で呼び出されました。それらのほとんどはそれぞれの ユーティリティーごとの非常に特殊なオプションでした。ある時点から Subversion は自分自身の内部差分ライブラリを持つようになり、エラー回避の仕組みとして の [*12] `--diff-cmd` と`--diff3-cmd` オプションが Subversion コマンドラインクライアントに追加されていて、 これによってユーザは、新しい内部 diff ライブラリを使うかわりに自分の 好きな GNU diff や diff3 ユーティリティーを使うこともできたのです。 このようなオプションが利用された場合、Subversion は単に内部 diff ライブラリを無視し、そのような外部プログラムを長い引数つきで実行します。 これが、このようなオプションが今でも残っている理由です。

Subversion がシステム中の特定のディレクトリにある外部 GNU diff と diff3 ユーティリティーを利用するための簡単な設定の仕組みは他の diff や merge ツールにも一般的に利用できることに開発チームが気づくまでにはそんなに 長い時間はかかりませんでした。要するに Subversion は実際に実行せよと 言われた外部ツールが GNU diffutils のツールの組み合わせであるかどうか を実際には確認していなかったのです。しかしこれらの外部ツールを利用する ための、そのツールのシステム中の場所だけは設定しなくてはなりません — 必要なオプションと、パラメータの順序、などを指定するのはもちろんのこと ですが。Subversion は、このような GNU ユーティリティー用のオプションの すべてを、実際にそのオプションが理解されるかどうかにかかわりなく 外部 diff ツールに渡します。そしてここがほとんどのユーザにとって 直観的には理解しにくい部分です。

外部 diff と merge ツール (もちろん GNU diff と diff3 以外のものも含みますが) を Subversion で利用するときの鍵は、Subversion からの入力をその差分ツールが 理解できる何らかの形に変換し、そのツールからの出力内容 — それは たとえば GNU ツールが利用しているような書式ということになるのでしょうが — を Subversion 側で理解できる形に変換するようなラッパースクリプトを使うことです。 以下の節ではこのような考え方を具体的に述べます。

[*12] Subversion 開発者はもちろんみんな優秀ですが、猿も木から落ちるといいます。

注意

Subversion の処理の一部として文脈 diff や merge をいつ利用するかの判断は 完全に Subversion 側で決定され、操作対象となるファイルが人間によって 可読な形式であるかどうかは他の場合と同様 `svn:mime-type` 属性によって決められます。これによって、例えば、仮にあなたが宇宙で一番 すぐれた Microsoft Word 用の差分とマージツールを手にしていたとしても、そのバージョン化された Word ドキュメントが人間によって可読ではないことを 示す MIME タイプに設定されていなければ (たとえば `application/msword` のようなもの) 決して起動されることは ないでしょう。MIME タイプの設定についての詳細は 項 7.3.3.2 を見て ください。

7.9.1 外部 diff

Subversion は外部 diff プログラムを GNU diff ユーティリティーにふさわしい 引数で呼び出し、その外部プログラムには単に成功したことを示すエラーコード を返すことだけを期待します。その他のほとんどの diff プログラムでは 六番目と七番目の引数、これは diff の左側と右側にそれぞれ対応したファイルの パスになりますが、それだけが関係してきます。Subversion は Subversion の操作によって修正されたファイルごとに diff プログラムを起動する ため、その外部プログラムが非同期的に実行される (あるいは「バックグラウンド」 で実行される) 場合にはすべてのインスタンスが同時に実行されてしまうかも知れない ことに注意してください。最後に Subversion はそのプログラムが差分を検出した場合 0 を、また検出しなかった場合には 1 をエラーコードとして返すものとして扱います — これ以外のエラーは致命的なエラーとみなします。 [*13]

例 7.9.1 と 例 7.9.2 はそれぞれ Bourne シェルと Windows バッチスクリプト言語での外部 diff ツールの ラッパー用テンプレートです。

7.9.2 外部 diff3

Subversion は外部マージプログラムを GNU diff3 ユーティリティーにふさわしい 引数で呼び出し、この外部プログラムが成功を示すエラーコードで戻り、 完了したマージ処理の結果としてのファイル内容の全体が標準出力ストリームに 出力されることを期待します (これによって Subversion はその内容を 適切なバージョン管理下にあるファイルにリダイレクトすることができます)> その他のほとんどのマージプログラムでは 9 番目、10 番目、そして 11 番目の 引数だけが処理に関係してきます。これらはそれぞれ 「自分側のファイル (mine)」、 「マージ元になる古いファイル (older)」、そして 「相手側のファイル (yours)」の内容になります。Subversion はそのマージプログラムの出力に依存するので、ラッパースクリプトは 出力が Subversion に転送されてしまうまで終了してはいけません。最終的に プログラムが終了した際にマージが成功していれば 0 を、解消できない衝突が 出力中に残っている場合には 1 を返します — それ以外のエラーコードは 致命的なものとみなします。

例 7.9.3 と 例 7.9.4 はそれぞれ Bourne シェルと Windows バッチスクリプト言語用の外部マージツール ラッパーのテンプレートです。

[*13] GNU diff のマニュアルには以下のようにあります:「0 の終了コードは差分がなかったことを意味し、1 は何か差分があったことを、また 2 は処理に異常があったことを示します。」

7.10 Subversion リポジトリの URL

この本全体を通じて、Subversion は Subversion リポジトリのバージョン管 理されている資源を特定するために URL を使います。ほとんどの場所ではこ のような URL は標準的な構文が利用され、サーバ名とポート番号をその URL の一部として指定することができるようになっています:

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

しかし Subversion の URL には、注意しなくてはならないような微妙な記述 の仕方もあります。たとえば `file:` によるアクセス法を 含むような URL の場合、規約により `localhost` という 名前のサーバ名を指定するか、あるいはまったくサーバ名を指定しないかのど ちらかを選択しなくてはなりません:

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

また Windows プラットフォームで `file:` 構文を使うユー ザは同じマシン上にあるが、クライアントのカレントドライブとは別のドライ ブにあるリポジトリにアクセスするためには公式的なものとはされていませんが 「標準的な」構文を使う必要があります。以下の URL 構文の どちらか一方を使うとうまくアクセスできます。ここで `X` はリポジトリのあるドライブです。

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

二番目の構文では URL を引用符でくくることで縦棒の文字がパイプの意味に 解釈されないようにする必要があります。また URL は Windows の標準ではバッ クスラッシュを使うパスの区切りの文字に通常のスラッシュ文字を使うことに も注意してください。

最後に、Subversion クライアントは必要に応じてちょうど Web ブラウザがや るような具合に自動的に URL をエンコードすることにも注意してください。 たとえば、 URL が空白あるいは上位の ASCII 文字が含まれている場合:

```
$ svn checkout "http://host/path with space/project/espa 単 a"
```

… Subversion は安全に表示できる文字に変換して、あなた自身がそのように 入力したかのように動作します:

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

URL が空白を含む場合、引用符の中にあることを確認してください。これでシェ ルは **svn** プログラムに対してその文字列全体が単一の引 数であるものとして扱うことができるようになります。

例 7.2.1 レジストリエントリ (.reg) ファイルの例

```
REGEDIT4

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

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-proxy-host"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-proxy-password"=""
"#http-proxy-exceptions"=""
"#http-timeout"="0"
"#http-compression"="yes"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-trust-default-ca"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""

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

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

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

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

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

例 7.9.1 diffwrap.sh

```
#!/bin/sh

# ここに自分の好きな diff プログラムを設定してください。
DIFF="/usr/local/bin/my-diff-tool"

# Subversion は 6 番目と 7 番目の引数としてパス名が必要です
LEFT=${6}
RIGHT=${7}

# diff コマンドを呼び出します (merge プログラムで意味を持つように
# 以下の行を変更してください。)
$DIFF --left $LEFT --right $RIGHT

# 差分がなければエラーコード 0 を、差分があれば 1 を返します。
# それ以外のエラーコードは致命的とみなします。
```

例 7.9.2 diffwrap.bat

```
@ECHO OFF

REM ここに自分の好きな diff プログラムを設定してください。
SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"

REM Subversion は 6 番目と 7 番目の引数としてパス名が必要です
SET LEFT=%6
SET RIGHT=%7

REM diff コマンドを呼び出します (merge プログラムで意味を持つように
REM 以下の行を変更してください。)
%DIFF% --left %LEFT% --right %RIGHT%

REM 差分がなければエラーコード 0 を、差分があれば 1 を返します。
REM それ以外のエラーコードは致命的とみなします。
```

例 7.9.3 diff3wrap.sh

```
#!/bin/sh

# ここに自分の好きな diff3/merge プログラムを設定してください。
DIFF3="/usr/local/bin/my-merge-tool"

# Subversion は必要となるパスを、9, 10, 11 番目の引数として用意します。
MINE=${9}
OLDER=${10}
YOURS=${11}

# merge コマンドを呼び出します (merge プログラムで意味を持つように
# 以下の行を変更してください。)
$DIFF3 --older $OLDER --mine $MINE --yours $YOURS

# マージ処理実行後、このスクリプトはマージされたファイル内容を標準出力に
# 表示す必要があります。適切だと考える方法でこれを行ってください。
# エラーコード 0 はマージ成功を、1 は解消不能な衝突が結果に残ったことを
# 示します。それ以外のエラーコードは致命的とみなします。
```

例 7.9.4 diff3wrap.bat

```
@ECHO OFF

REM ここに自分の好きな diff3/merge プログラムを設定してください。
SET DIFF3="C:\Program Files\Funky Stuff\My Merge Tool.exe"

REM Subversion は必要となるパスを、9, 10, 11 番目の引数として用意しますが、
REM 一度にアクセスできる引数の数は 9 個までなので、必要な引数を取得するため
REM 取得前に 9 個のパラメータウィンドウを二回シフトしておきます。
SHIFT
SHIFT
SET MINE=%7
SET OLDER=%8
SET YOURS=%9

REM merge コマンドを呼び出します (merge プログラムで意味を持つように
REM 以下の行を変更してください。)
%DIFF3% --older %OLDER% --mine %MINE% --yours %YOURS%

REM マージ処理実行後、このスクリプトはマージされたファイル内容を標準出力に
REM 表示す必要があります。適切だと考える方法でこれを行ってください。
REM エラーコード 0 はマージ成功を、1 は解消不能な衝突が結果に残ったことを
REM 示します。それ以外のエラーコードは致命的とみなします。
```

第 8 章

開発者の情報

8.1

Subversion はオープンソースのソフトウェアプロジェクトで、 Apache スタイルのソフトウェアライセンスを持っています。プロジェクトは カリフォルニアに本拠地があるソフトウェア開発会社 CollabNet, Inc., の 経済的な支援を受けています。このコミュニティは Subversion の開発を めぐって構成されていますが、このプロジェクトに時間を割いてもらったり 注意を向けてもらえるような形で無償援助してくれる人を常に歓迎しています。ボランティアはどんな形の援助もすることができます。 それは、バグを見つけたり、テストしたり、既にあるコードを改良したり、 まったく新しい機能を追加したりといったことを含みます。

この章はソースコードに自分の手を実際に染めることによって Subversion のいままさに起こっている進化を援護しようとする人に向けてのものです。 ソフトウェアのもっと詳細に触れ、Subversion 自身を開発するのに — あるいは、Subversion ライブラリを使った完全に新しいツールを書くために — 必要になる技術的に重要な点について説明します。もし、そんな レベルの話に参加したくないのであれば、この章は飛ばしてもらって結構 です。Subversion のユーザとしての経験には影響を与えませんので。

8.2 階層化されたライブラリ設計

Subversion はモジュール化された設計になっていて、C ライブラリの集まりと して実装されています。ライブラリのそれぞれはよく定義された目的とインター フェースを持っていて、ほとんどのモジュールは三つの主要な層のどれかに属 します。 — リポジトリ層、リポジトリアクセス層 (RA)、そしてクライア ント層です。これらの層について簡単に見ていきますが、最初に表 8.1 にある Subversion ライブラリ一覧を見てくだ さい。一貫した議論とするため、ライブラリは、拡張子を除いた Unix のライ ブラリ名称で参照することにします (たとえば: libsvn_fs, libsvn_wc, mod_dav_svn)。

表 8.1 に「さまざまな」という言 葉が一つだけ出てきているというのは、良い設計である証拠です。Subversion 開発チームはそれぞれの機能が、正しい層とライブラリにあることを確認する のを、重要なことだと考えています。多分、モジュール化した設計の一番大き な利点は開発者の観点から見た複雑さを減らすことができることです。開発者 として、あなたはすぐに、「話の概要」を知ることができ、それ によって比較的簡単にある特定の機能の場所を特定することができるようにな ります。

モジュール化のほかの利点は、与えられたモジュールをコードの別の部分に 影響与えることなしに、同じ API を実装した新しいライブラリで置き換えることが できるということです。ある意味で、これは Subversion 内部で既に起きて いることです。libsvn_ra_dav, libsvn_ra_local, そして libsvn_ra_svn の すべては、同じインターフェースを実装しています。そして、この三つすべて が、リポジトリ層とやり取りします。 — libsvn_ra_dav と libsvn_ra_svn はネットワーク越しにそう しますし、libsvn_ra_local は直接リポジトリに接続します。 libsvn_fs_base と libsvn_fs_fs ライブラリは、さらにまた別の例です。

表 8.1 Subversion ライブラリの一覧

ライブラリ	説明
libsvn_client	クライアントプログラムへの主要なインターフェース
libsvn_delta	ツリーとバイトストリームの差分ルーチン
libsvn_diff	コンテキスト差分とマージルーチン
libsvn_fs	ファイルシステムの共通関数と、モジュールローダー
libsvn_fs_base	Berkeley DB ファイルシステムバックエンド
libsvn_fs_fs	ネイティブファイルシステム (FSFS) バックエンド
libsvn_ra	リポジトリアクセスのための共通ルーチンとモジュールローダ
libsvn_ra_dav	WebDAV リポジトリアクセスモジュール
libsvn_ra_local	ローカルリポジトリアクセスモジュール
libsvn_ra_svn	独自プロトコルによるリポジトリアクセスモジュール
libsvn_repos	リポジトリインターフェース
libsvn_subr	さまざまな役に立つサブルーチン
libsvn_wc	作業コピー管理ライブラリ
mod_authz_svn	WebDAV 経由で Subversion リポジトリにアクセスするための Apache 認証モジュール。
mod_dav_svn	WebDAV 操作を Subversion のものに対応付ける Apache モジュール

クライアント自身もまた Subversion の設計でのモジュール性をはっきり 示しています。Subversion は現在のところコマンドラインクライアント プログラムのみを実装していますが、Subversion のために GUI として振る舞う ようなサードパーティーによって開発されているいくつかのプログラムが あります。このような GUI も、既に実装されているコマンドクライアント と同じ API を利用します。Subversion の libsvn_client ライブラリは Subversion クライアントを設計するのに必要なほとんどの機能のために 利用することができます。(項 8.2.3 参照)。

8.2.1 リポジトリ層

Subversion のリポジトリ層を参照するとき、一般的に、二つのライブラリに ついて語っています — リポジトリライブラリとファイルシステムライブラリ です。これらのライブラリはバージョン管理されたデータのさまざまなリビジョン のための格納と報告の仕組みを提供しています。この層はリポジトリアクセス層 を経由してクライアント層とつながっていて、Subversion 利用者から見ると、 「通信の相手先」にあるものに見えます。

Subversion のファイルシステムは libsvn_fs API によってアクセスされ、 オペレーティングシステムにインストールされているカーネルレベルの 意味でのファイルシステム (Linux の ext2 や、NTFS のような) ではなく、 仮想的なファイルシステムのことです。「ファイル」と 「ディレクトリ」を (自分の好きなシェルを使って操作することができるような) 現実の ファイルとディレクトリとして保存するのではなく、バックエンドの抽象的な保存の仕組みとして二つのうちのどちらかが利用可能です — Berkeley DB データベース環境か、フ

ラットファイルによる 表現です。(この二つのバックエンドについてより深く知りたい場合は 項 5.2.3 を見てください)。 しかし、Subversion の今後のリリースの中で開発コミュニティによって 興味を引いている、別のバックエンドデータベースシステムの利用 可能性があります。たとえば、オープンデータベースコネクティビティー (ODBC) などです。

libsvn_fs が提供するファイルシステム API は他のファイルシステム API でも期待できるような機能を持っています: ファイルやディレクトリの 作成や削除ができて、コピーや、移動ができ、ファイルの内容を 修正することができて、などなどです。あまり一般的ではないような 機能もあります。たとえば、ファイルやディレクトリに付随した メタデータ (「属性」) の追加、変更、削除、などです。さらに Subversion ファイルシステムはバージョン化可能なファイルシステムで、 これは、ディレクトリツリーに変更を加えると、Subversion はその変更 以前に、そのツリーがどのように見えるかを憶えておくということを 意味します。そして、さらにその前の変更前、さらにその前、などです。 このようにして、ファイルシステムに何かを最初に追加したところまでの すべてのバージョンに戻ることができます。

ツリーに加えたすべての変更は Subversion のトランザクションの中で実行 されます。以下は、ファイルシステムを修正するための単純で一般的な 手続きです:

1. Subversion トランザクションの開始
2. 修正の実行 (追加、削除、属性の修正、など)
3. トランザクションのコミット

トランザクションをコミットすると、ファイルシステムの変更は歴史上の 出来事として永久に記録されます。このようなそれぞれのサイクルは ツリーに新しいリビジョンを作り、それぞれのリビジョンは「あることが どのようであったか」という純粋なスナップショットとしていつでもアクセス できるようになります。トランザクションを邪魔するもの

Subversion のトランザクションという概念は、特に、 それと非常に近い意味の libsvn_fs にあるデータベースの実際のコード を見せられると、データベース自身がサポートしているトランザクションと 容易に混乱してしまいます。両方とも、不可分性と、分離性を持っています。 言い換えると、トランザクションはある処理のあつまりを、「全か、無か」 という形で実行する能力を与えます — そのあつまりの中にあるすべての 処理は、完全に成功するか、何も起こらなかったかの ように扱われます — そして、トランザクション中に、そのデータに別の 処理をするプロセスには一切干渉しません。

データベーストランザクションは一般的に、データベース自身のデータの 修正に関係した小さないくつかの操作を含んでいます。 (たとえば、テーブル行の内容を修正することなどです)。 Subversion のトランザクションは、もっと大きな範囲のもので、 それは、ファイルシステムツリーの次のリビジョンとして格納されることを 目的とした、ファイルやディレクトリに対するいくつかの修正をする ような操作を含んでいます。混乱がないようなら、こう考えてください: Subversion は Subversion トランザクションの生成中に、データベーストランザクション を使います (それで、もし Subversion トランザクションの生成が失敗すれば、 データベースは最初からその生成をしなかったように見えます)

ファイルシステム API 利用者にとって幸運なことに、データベースシステム 自身によって用意されているトランザクションサポートは、ほとんどの場合 隠れていて見えません。(普通のモジュール化されたライブラリの設計から 期待されることですが)。ファイルシステム自身の実装を開始する場合に初めて、 そのようなことが見えるように (あるいは興味深く思えるように) なります。

ファイルシステムインターフェースによって用意される機能のほとんどは ファイルシステムパスに対して起こる操作として提供されます。 つまり、ファイルシステムの外部からは、ファイルやディレクトリの個別の リビジョンを記述し、アクセスする、主要な仕組みは、`/foo/bar` のようなパス文字列を使うことを通じて提供され、それはちょうど、あなたが なじみのシェルプログラムを通じてファイルやディレクトリにアクセスする

ような感じになります。適切なパス名を正しい API 関数に渡すことによって、 新しいファイルやディレクトリを追加することができます。同じ仕組みを使って そのファイルなどに関した情報を問い合わせることができます。

ほとんどのファイルシステムとは違い、パスだけを指定するのは、Subversion のファイルやディレトクリを特定するのに十分な情報ではありません。 ディレクトリツリーを二次元のシステムと考えてください。ここで、あるノードの 兄弟は、左から右に移動することを表現していて、サブディレトクリに降りていく のは、下向きの移動であると考えてください。図 8.1 は典型的なツリーの 表現を示しています。

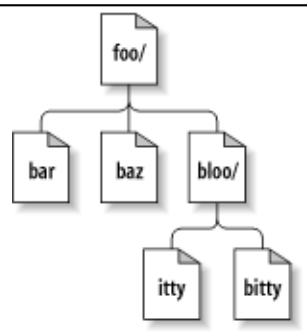

図 8.1 二次元の中のファイルとディレクトリ

もちろん Subversion のファイルシステムは隠れた第三の次元を持っていますが、それはほとんどのファイルシステムが持っていないものです — それは時間の次元です! [*1] ファイルシステムインターフェースで、*path* 引数を 持つほとんどすべての関数はまた *root* 引数も指定し なくてはなりません。この svn_fs_root_t 引数は、リビジョンか、Subversion のトランザクション (それは普通はリビジョンとな るべきものです) のどちらかをあらわし、リビジョン 32 の `/foo/bar` と、リビジョン 98 の同じパスとの間の違いを 理解するのに必要になる三次元コンテキストを用意します。図 8.2 は Subversion ファイルシステムの宇宙に 追加された次元についてのリビジョン履歴をあらわしています。

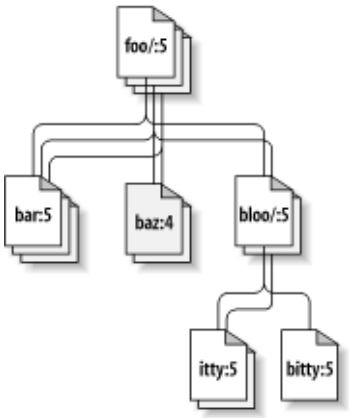

図 8.2 バージョン化した時刻 — 第三の次元!

以前指摘したように、libsvn_fs API は他のファイルシステムと見かけは よくにているが、このすばらしいバージョン管理能力だけは例外です。 それはバージョンファイルシステムに興味のあるすべてのプログラマによって 利用できるように設計されました。偶然の一致ではありませんが、 Subversion 自身もその機能に興味

[*1] わたしたちは、時間は実際には第四の 次元である という印象をずっと持っていた SF ファイルにショックを与えるということを 理解していますし、別の理論をわたしたちが宣言することによって生じる 心理的なトラウマについては謝らなくてはなりませんね。

があります。しかし、ファイルシステム API は基本的なファイルとディレクトリのバージョン管理をサポートしていますが Subversion はさらに多くを要求します — そしてそれは libsvn_repos が 提供するものです。

Subversion リポジトリライブラリ (libsvn_repos) は基本的には ファイルシステム機能のまわりにあるラッパーライブラリです。この ライブラリはリポジトリレイアウトの作成、ファイルシステムの初期化を 正しく実行すること、などに責任を持ちます。Libsvn_repos はまた フックを実装します — 特定の処理が実行されるときにリポジトリの コードによって実行されるスクリプトです。このようなスクリプトは 通知、認証、あるいはリポジトリ管理者が望むようなさまざまな 目的にとって役に立つものです。このタイプの機能と、リポジトリライブラリ によって提供されるほかのユーティリティーはバージョン化ファイルシステム の実装に強く関連しているわけではありません。それが独自のライブラリとして 実装された理由です。

libsvn_repos API を使おうとする開発者には、それがファイルシステムインターフェース に対する完全なラッパではないことがわかるでしょう。つまり、 ファイルシステム操作の一般的なサイクルにある主要なイベントに ついてだけリポジトリインターフェースによってラップされます。 その中のいくつかは、Subversion トランザクションの生成やコミット、 リビジョン属性の修正などです。このような特定のイベントはそれに関連したフックがあるため、リポジトリ層によってラップされます。 将来的には他のイベントもリポジトリ API によってラップされるかも 知れません。しかし、残りのファイルシステムの作用のすべては libsvn_fs API 経由で直接実行されます。

たとえば、ディレクトリが追加されるファイルシステムの新しい リビジョンを作るための、リポジトリとファイルシステムインターフェース について、使い方を説明したコードがあります。 この例で (そして、この本全体を通じてすべてのほかの例でも)、 `SVN_ERR()` マクロは単にラップした関数からの 失敗した場合のエラーコードのチェックです。そしてそのようなもの があればそのエラーを返します。

このコードで、リポジトリとファイルシステムインターフェースの両方に 対する呼び出しがあります。`svn_fs_commit_txn()` を 使ってトランザクションを簡単にコミットできます。しかし、 ファイルシステム API はリポジトリライブラリのフックの仕組みについては 何も知りません。もし Subversion リポジトリにトランザクションをコミット するたびに自動的にある種の非 Subversion 的な作業を実行させたい場合、(たとえば、開発者メーリングリストにそのトランザクションで起きたすべて 変更を説明するメール を送信する、など)、その関数の libsvn_repos で ラップされたバージョンを使う必要があります — `svn_repos_fs_commit_txn()`。この関数は実際にはもし存在すれば、最初に「pre-commit」フックスクリプトを実行し、 それからトランザクションをコミットし、最後に「post-commit」フックスクリプト を実行します。フックは、実際にはコアのファイルシステムライブラリ自身 に含まれない特別の報告の仕組みを用意します。 (Subversion のリポジトリフックについての詳細は項 5.3.1 を見てください)。

フックの仕組みは、残りのファイルシステムコードから 独立したリポジトリライブラリの抽象化が一つの理由です。 libsvn_repos API はほかにもいくつかの重要なユーティリティーを Subversion に提供しています。これには以下のようなものがあります:

1. Subversion リポジトリと、それに含まれるファイルシステム上での ファイルの生成、オープン、削除、そして回復のステップ
2. 二つのファイルシステムツリー間の比較の表示
3. ファイルシステム中で修正されたファイルがあるすべて (あるいは いくつかの) のリビジョンに結びついたコミットログメッセージへの 問い合わせ
4. ファイルシステムの可読な「ダンプ」の生成、ファイルシステム中にある リビジョンの完全な表現
5. ダンプフォーマットの解析、異なる Subversion リポジトリの中に ダンプされたリビジョンをロードすること

Subversion が進化し続けるにつれて、リポジトリライブラリは 増えつづける機能と設定可能なオプションをサ

ポートを提供する ために，ファイルシステムライブラリとともに大きくなり続ける でしょう。

8.2.2 リポジトリアクセス層

もし Subversion リポジトリ層が、「通信路のもう一方の端点」 であるなら、リポジトリ アクセス層は、その通信路そのもののです。クライアントライブラリとリポジトリ の間でデータを相互変換することが課せられたこの層は libsva_ra モジュールローダ ライブラリ、その RA モジュール自身 (現在のところ、libsvn_ra_dav, libsvn_ra_local, そして libsvn_ra_svn を含みます)、そして一つ以上の RA モジュールに必要となる追加 のライブラリ、たとえば、libsvn_ra_dav が通信するための、mod_dav_svn Apache モジュールを含みます。mod_dva_svn モジュールを利用しないときには、 libsvn_ra_svn のサーバである **svnserve** が通信します。

Subversion は、リポジトリリソースを特定するのに URL を利用するので、 URL スキーマのプロトコル部 (普通は、`file:`, `http:`, `https:`, あるいは `svn:`) はどの RA モジュールが通信を処理するか を決めるために使われます。それぞれのモジュールは、プロトコルのリスト を登録しますが、それはどうやって「話せば」良いかを知っているので、 RA ローダが実行時にどの RA モジュールをその処理のために利用するかを 決定することができます。どの RA モジュールが Subversion コマンドライン クライアントに利用可能かを決定することができ、**svn –version** を実行することで、どのプロトコルはサポートしていないと言ってくるか を知ることができます。:

```
$ svn --version
svn, version 1.2.3 (r15833)
   compiled Sep 13 2005, 22:45:22

Copyright (C) 2000-2005 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
```

8.2.2.1 RA-DAV (HTTP/DAV を使ったリポジトリアクセス)

libsvn_ra_dav ライブラリは、サーバとは別のマシン上で 実行されているクライアントによって利用されるように設計されています。 クライアントは URL を使って特定のサーバを指定することで通信します。 ここでいう URL は、`http:` または `https:` のプロトコル部分を含んでいるようなものです。 どのようにこのモジュールが動作するかを理解するために、最初にリポジトリ アクセス層の特定の設定中にあるほかの

いくつかのキーコンポーネントに 触れる必要があります — それは強力な Apache HTTP サーバと、 Neon HTTP/WebDAV クライアントライブラリです。

Subversion の主なネットワークサーバは Apache HTTP サーバです。Apache は 十分にテストされ、拡張可能なオープンソースのサーバプロセスで、それは まじめな用途に利用することができます。それはネットワークの高負荷に 持ちこたえることができ、たくさんのプラットフォーム上で動作します。 Apache サーバはたくさんの異なる標準認証プロトコルをサポートし、 たくさんの人々によってサポートされたモジュールを利用することで 拡張することができます。それはまたネットワークパイプラインやキャッシング のような最適化をサポートしています。サーバとして Apache を利用することに よって、Subversion はこれらのすべての機能を自由に手に入れることが できます。そして、ほとんどのファイアウォールは HTTP の通信を通すように 設定されているので、システム管理者は、普通はファイアウォール設定 を変更する必要すらなく Subversion を動作させることができます。

Subversion は HTTP と WebDAV(DeltaV 付きで) を使って、Apache サーバと 通信します。これについては、この章の WebDAV の節を呼んでください。 しかし、簡単に言えば、WebDAV と DeltaV は標準的な HTTP1.1 プロトコル の拡張で、それは web 上でファイルの共有とバージョン化を可能にします。 Apache 2.0 は mod_dav が用意されていて、これは HTTP の DAV 拡張を 理解するモジュールです。Subversion 自身は mod_dav_svn をサポートして いますが、これは別の Apache モジュールで、mod_dav と協調して動作し、 (実際にはそのバックエンドとして)Subversion 上での具体的な WebDAV と DeltaV の実装となっています。

HTTP 越しにリポジトリと通信するとき、RA ローダライブラリは libsvn_ra_dav を サーバプロセスモジュールとして選択します。Subversion クライアントは 一般的な RA インターフェースを呼び出し、 libsvn_ra_dav はこのような 呼び出しを (それはまだ大雑把な Subversion の動作を具体化します) を、 HTTP/WebDAV 要求に変換します。Neon ライブラリを使って、libsvn_ra_dav はこのような要求を Apache サーバに送信します。Aapche はこのような要求 を受け取り (Web ブラウザがやるのとまったく同じ一般的な HTTP 要求ですが)、DAV 管理の位置として設定された URL に振り向け (`httpd.conf` ファイル中の `<Location>` 命令を使います)、その要求を固有の mod_dav モジュールに渡します。適切に設定されていれば、mod_dav は Apache 付属の一般的な mod_dav_fs ではなく、Subversion の mod_dav_svn をファイルシステムに関連した要求に対して利用することを 知っています。それで、最後には、このクライアントは mod_dav_svn と 通信しますが、これは直接 Subversion リポジトリ層に結び付いている ものです。

これが実際に起こるやり取りの簡略化した説明です。たとえば、 Subversion リポジトリは Apache の認証命令によって保護されているかも知れません。 これによって、リポジトリと最初に通信しようとする試みが、認証付き Apache に よって失敗に終わるかも知れません。 この時点で、libsvn_ra_dav は Apache から、不十分な認証しか得られなかったのでクライアント層に更新された認証データを 取得するためにコールバックした、という通知を受けます。 もしこのデータが正しく取得できれば、ユーザは、許可された最初の操作 を実行する、libsvn_ra_dav の次のアトミックな要求を探し、すべて がうまくいきます。もし十分な認証情報が与えられなければ要求は最終的に 失敗し、クライアントはユーザにその旨を報告します。

Neon と Apache を使って、Subversion はほかのいろいろな複雑な領域への 自由な機能を得ることもできます。たとえば、もし Neon が OpenSSL ライブラリ を見つけた場合、それは Subversion クライアントに SSL で暗号化された 通信を、Apache サーバとすることを認めます。 (その固有の mod_ssl は「その言語を話します」)。また、Neon 自身と Apache の mod_deflate は「deflate」アルゴリズムを理解できるので (PKZIP と gzip プログラムで利用されているのと同じものですが)、 要求はより小さな圧縮された塊として通信路を流れます。Subversion が今後サポートしたいと思っているほかの複雑な機能 としては、自動的にサーバ側のリダイレクトを処理すること (たとえば、 リポジトリが別の新しい URL に移動したような場合) や、HTTP パイプライン の恩恵にあずかること、などです。

8.2.2.2 RA-SVN (固有のプロトコルによるリポジトリアクセス)

標準的な HTTP/WebDAV プロトコルに加えて, Subversion は固有の プロトコルを使う RA の実装も用意しています。libsvn_ra_svn モジュールは 固有のネットワークソケット接続を実装し、リポジトリのあるスタンドアロンサーバ と通信します — `svnserve` です — クライアントは `svn://` スキーマでリポジトリにアクセスで きます。

この RA 実装は、前の節で触れた Apache の利点のほとんどを欠いています。 しかし、それはある種のシステム管理者を引きつけるかも知れません。 それは非常に簡単に設定し実行できます。`svnserve` プロセスの設定は、ほとんど瞬間的に終わります。またそれは Apache よりも (コード行数という意味で) ずっと小さく、セキュリティーや他の事情による チェックもずっと容易です。さらにいくつかのシステム管理者は既に SSH のセキュリティーインフラを持っていて、Subversion にもそれを使わせたい と思っているかも知れません。ra_svn を使うクライアントは SSH を介して プロトコルを簡単にトンネルすることができます。

8.2.2.3 RA-Local (リポジトリへの直接のアクセス)

Subversion リポジトリとのすべての通信が大きなサーバプロセスとネットワーク層 を必要とするわけではありません。ローカルディスク上のリポジトリにアクセス したいだけのユーザにとっては、`file:` を使うことができ、 libsvn_ra_local が提供する機能を使うことができます。 この RA モジュールは直接リポジトリとファイルシステムライブラリと 結びつくので、ネットワーク通信はまったく必要ありません。

Subversion は `file:` URL の一部として `localhost` か、空である サーバ名称を含むことを要求し、ポート指定はありません。 言い方を変えると、URL は何か、`file://localhost/path/to/repos` か `file:///path/to/repos` のような形のものになります。

さらに、Subversion の `file:`URL は普通 web ブラウザが `file:`URL がやる方法では利用できないことに注意して ください。通常の web ブラウザで `file:`URL を閲覧しようと いうとき、ファイルシステムを直接調べることでその場所にあるファイルの 内容を読み出して表示します。しかし、Subversion のリソースは仮想ファイル システム中にあり、(項 8.2.1 参照) あなたのブラウザは そのファイルシステムをどうやって読めば良いか理解できないでしょう。

8.2.2.4 Your RA Library Here

さらに別のプロトコルを使って Subversion のリポジトリにアクセスしたいと いう人にとってこそ、どうしてリポジトリアクセス層がモジュール化されている かという理由になります。開発者は片方で RA インターフェースを実装する 新しいライブラリを簡単に書くことができ、もう一方でそのリポジトリと 通信することができます。新しいライブラリは既に存在しているネットワーク プロトコルを利用することもできますし、自分で開発したものでも良いのです。 プロセス間通信 (IPC) 呼び出しを使うかも知れませんし — ちょっとおバカ かも知れませんが — メールベースのプロトコルを使うことだってできます。 Subversion は API を提供し、あなたは自分の想像性を提供する、と。

8.2.3 クライアント層

クライアント側から見ると、Subversion の作業コピーは すべての処理が起こる場所です。クライアント側ライブラリによって 実装される機能は、作業コピーの管理というただ一つの目的のために 存在します — ローカルな場所に何らかの形で提供される ファイルと他のサブディレクトリのあるディレクトリが、一つ以上の リポジトリ位置を「反映した」ものとし、リポジトリアクセス層との間の 変更を伝えたりします。

Subversion の作業コピーライブラリ、libsvn_wc は作業コピー中の データの管理に直接の責任を負います。これをやるために、このライブラリ は特別なサブディレクトリの中にそれぞれの作業コピーについての管理情

報 を格納します。このサブディレクトリは.svn という 名前ですが、どの作業コピー中にも存在し、管理に関係した動作を するための状態を記録し、作業スペースを確保するためのさまざまな ファイルやディレクトリを含んでいます。CVS になじみのある人なら、 この.svn サブディレクトリは、その目的としては CVS の作業コピーにある管理ディレクトリ CVS に よく似ていることがわかると思います。.svn 管理領域についての詳細は、この章の 項 8.4 を参照してください

Subversion クライアントライブラリ libsvn_client は広範囲の役目を 負います。その仕事は、作業コピーライブラリの機能と、リポジトリアクセス層 の機能を結びつけることで、一般的なリビジョン制御を実行したいと思う すべてのアプリケーションに最上位の API を提供することです。たとえば svn_client_checkout() 関数は引数として URL を とります。この関数は URL を RA 層に渡し、特定のリポジトリに認証された セッションを開きます。それからそのリポジトリに特定のツリーを指定し、 このツリーを作業コピーライブラリに送りますが、今度はそのライブラリが 作業コピー全体をディスクに書き込みます (.svn ディレクトリを含むすべての情報)。

クライアントライブラリはどのようなアプリケーションからも利用できる ように設計されています。Subversion のソースコードは標準的なコマンド ラインクライアントを含んでいるので、そのクライアントライブラリの 上に好きなだけ GUI クライアントを書くことができます。Subversion の 新しい GUI(あるいは実際には新しいクライアント) は、コマンドライン クライアントを含むダサいラッパである必要はありません。 — それは、libsvn_client API を通じてコマンドラインクライアント が使っているのと同じ機能、データ、コールバックの仕組みに完全にアクセス することができます。直接のバインド — 正確さについての言葉

なぜ GUI プログラムは、直接 libsvn_client にバインドし、コマンドライン プログラムをつつむラッパとして動作しないのでしょう? それは単により 効率的であるからだ、というだけではなく、潜在的な正確さについての 問題もあります。コマンドラインプログラム (Subversion が提供しているような もの) はクライアントライブラリに結びついていますが、C 言語の型を持つフィードバックとデータビットの要求を、人間が読める形の出力に効率的に変換する必要が あります。この手の変換は不正確になりがちです。つまり、プログラムは API から取得した情報のすべてを表示しないかも知れませんし、要約した表現形式 になるように情報をつなぎあわせたりするかも知れません。

そのようなコマンドラインプログラムを別のプログラムでラップすると、 ラップするほうのプログラムは既に解釈された (そして注意したように おそらく不完全な) 情報にアクセスすることができるだけで、それは もう一度、自分に固有の 表現形式に変換しなくてはなりません。それぞれのラッピングの層ごとに、 最初のデータの正確さはどんどん失われていく可能性があり、それはちょうど 自分の好きなオーディオやビデオカセットを繰り返しコピーするときに 起こるような話になってしまいます。

8.3 API の利用

Subversion ライブラリ API を使ったアプリケーションの開発は比較的素直な 形で進みます。すべてのヘッダファイルはソースツリーの subversion/include にあります。これらのヘッダは ソースコードから Subversion を作りインストールすると、そのマシンの システムヘッダの置き場所にコピーされます。このようなヘッダには Subversion ライブラリのユーザによってアクセスできるような機能と型の すべてがあります。

最初に気をつけなくてはならないのは Subversion のデータ型と関数は 固有の名前空間によって分離されていることです。すべてのパブリックな Subversion シンボル名は svn_で始まり、そのシンボルが 定義されているライブラリの短いコードが続き、 (wc とか、 client とか、fs など)、 アンダースコアが一つきて、(_) 、最後にシンボル名の 残りの部分がきます。限定的にパブリックな関数 (ライブラリ中のソースファイル 間では利用されるが、ライブラリの外では利用されず、ライブラリディレクトリ 自身の内部でだけ参照可能なもの) はこの命名規約とは違い、ライブラリコード のあとにアンダースコアが一つくるかわりに、二つきます (__)。あるソースファイルでプライベートな関数は 特殊な接頭辞を持たず、static 宣言されます。もちろん コン

パイラはこのような命名規約を解釈しませんが、ある関数のスコープや データ型を明らかにするときの助けになります。

8.3.1 Apache Portable Runtime ライブラリ

Subversion 自身のデータ型とともに、`apr_`で 始まるデータ型への参照をたくさん見かけることがあります — これは Apache の Portable Runtime (APR) ライブラリです。APR は Apache の可搬なライブラリですが、もともと Apache のサーバコードの OS 依存の部分を OS 非依存の部分から分離するために作られました。 結果は、OS ごとに、わずかに、あるいは大きく異なる操作を実行する ための抽象的な API を提供することになりました。Apache HTTP サーバは 明らかに APR ライブラリの最初の利用者でしたが、Subversion 開発者はすぐに APR を使うことの重要性に気づきました。これは実際に Subversion 自身の中にまったく OS に依存していないコードの部分があることを意味 します。さらに、Subversion クライアントはサーバがコンパイルし実行 する場所であればどこでも実行できることを意味します。現時点では このような OS には、Unix ライクなすべて、Win32, BeOS, OS/2 そして Mac OS X が含まれます。

オペレーティングシステム間で異なるシステムコールの一貫した 実装を提供することに加えて、[*2] APR は Subversion がたくさんの独自のデータ型に直接アクセスすることを 可能にしますが、それには、動的な配列やハッシュテーブルがあります。 Subversion はソースコード中でこれらの型を拡張して利用します。 しかし、多分最も広範囲に利用されている APR データ型は、ほとんど すべての Subversion API プロトタイプに現れますが、apr_pool_t です — APR のメモリプールです。Subversion はプールを内部的にすべてのメモリ確保 が必要な場合に利用します。(ただし、外部ライブラリがその API を通じて 受け渡すデータのメモリ管理にこれと異なる形式を要求しない限りにおいて、 です。) [*3] そして、Subversion API に対するコーディングは同じことを要求される わけではありませんが、必要な場所では API 関数のために pool を用意する ことは要求されます。これは APR もリンクする必要のある Subversion API の ユーザは `apr_initialize()` を呼んで APR サブシステム を初期化する必要があり、それから Subversion API 呼び出しを利用するために pool を用意しなくてはならない、ということになります。詳細は 項 8.6 を見てください。

8.3.2 URL と Path の要求

Subversion 全体の問題としてのリモートバージョン管理操作では、国際化 (i18n) のサポートについていくらか注意しておく必要があります。 結局、「リモート」が、「オフィス以外の場所」を意味するのなら、それは 「全世界から」という意味でもあります。このような状況に対応するために Subversion のパス引数をとる、すべてのパブリックインターフェースは パスが正規化され、UTF-8 でエンコードされているものとします。 これはたとえば、libsvn_client インターフェースを呼び出す、 新しいクライアントバイナリはすべて、Subversion ライブラリにパスを渡す 前に、まずパスをローカルコーディングから UTF-8 に変換しなくてはならず、 Subversion からの結果パスを、Subversion 以外の目的に利用する前には ローカルコーディングに再変換しなくてはならないということです。 幸運なことに、Subversion はこのような変換が必要な任意のプログラムが 利用できるような関数を用意しています (`subversion/include/svn_utf.h` を参照してください)。

また、Subversion API はすべての URL 引数が正しく URI エンコードされている ことを要求します。それで `My File.txt` という名前のファイル URL を、 `file:///home/username/My File.txt` と渡すかわりに、 `file:///home/username/My%20File.txt` と渡さなくてはなりません。 やはり Subversion はアプリケーションが利用できるヘルパー関数を用意して います — `svn_path_uri_encode()` と `svn_path_uri_decode()` を使ってそれぞれ URI のエンコードとデコードができます。

[*2] Subversion は ANSI システムコールとデータ型をできる限り 利用しています。
[*3] Neon と Berkeley DB はそのようなライブラリの例です。

8.3.3 C と C++ 以外の言語の利用

C 言語以外のものと組み合わせて Subversion ライブラリを使うのに興味が あるなら — たとえば Python や Perl のスクリプトなどを使った — Subversion は Simplified Wrapper and Interface Generator (SWIG) という形である程度サポートしています。Subversion 用の SWIG は、 `subversion/bindings/swig` にあり、開発はまだ続いては いますが、利用可能な状態にあります。これを使えば、スクリプト言語固有の データ型を、Subversion の C ライブラリで必要なデータ型に変換するラッパを 使って Subversion API を間接的に呼び出すことができるようになります。

言語連携を通じて SubversionAPI にアクセスするのは明らかに利点があります — 単純さ、です。一般的に、Python や Perl といった言語は C や C++ を使うよりもずっと柔軟でやさしいものです。このような言語が用意している 高レベルデータ型と文脈依存のデータ型のチェックのようなものは、もっと うまくユーザからの情報を処理します。ご存知のように、人間はプログラムの入力で ヘマをやらかすことにかけては達人であり、スクリプト言語はそのような間違っ た情報をより適切に扱える傾向になります。もちろんそのような柔軟性は しばしばパフォーマンスを犠牲にしますが。これが、非常に厳しく最適化された C 言語ベースのインターフェース + ライブラリ群と、強力で柔軟な連携言語 の組み合わせを利用するというやり方が強い説得力をもつ理由です。

Subversion の Python SWIG 連携を使って、最新のリポジトリリビジョンを 再帰的に訪問し、その途中で見つかったさまざまなパスを表示するような サンプルプログラムを見てみましょう。

This same program in C would need to deal 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. The following is a brief example of how that library can be accessed via the SWIG bindings to recreate a scaled-down version of the **svn status** command.

現時点で、これが Subversion の Python 連携であり、それはほとんど完成 されたものです。Java 連携についても少し触れておきます。SWIG インターフェース ファイルが正しく設定されれば、すべての SWIG 対応言語 (現在のところ、 C#, Guile, Java, MzScheme, OCaml, Perl, PHP, Python, Ruby, そして Tcl ですが) の 特定のラッパを生成するのは理論的には簡単なことです。しかし、SWIG が インターフェースするのに必要になる複雑な API に対しては、もう少し追加のコーディング が必要となります。SWIG 自身のより詳しい情報は <http://www.swig.org>にあるプロジェクト のウェブサイトをみてください。

Subversion の言語連携は不幸にも Subversion のコアモジュールほど注目 されていません。しかし Python, Perl, そして Ruby 用の関数連携を作る ためにかなり努力されてきました。ある程度の範囲で、このような拳固に 対して容易された SWIG インターフェースファイルは SWIG でサポートされて いるへ科の言語の連携を生成するために再利用することができます。 (このような言語には C#、Guilde, Java, Mzscheme, OCaml, PHP, Tcl その他があります)。 それでも SWIG を汎用的に利用するのに必要な複雑な API のために ある程度特殊なプログラミングが必要にはなります。SWIG 自身の詳細に ついては `http://www.swig.org/`にある プロジェクトウェブサイトを参照してください。

8.4 作業コピー管理領域の内部

以前指摘したように、Subversion 作業コピーのディレクトリのそれぞれは `.svn` という名前の特別のサブディレクトリを持ち、そこに作業コピーディレクトリに関する管理情報を格納します。 Subversion は`.svn` 中の情報を以下のようなことを 記録するのに利用します:

- どこにあるリポジトリが、作業コピーディレクトリのファイルやサブディレクトリ によって表現されているのか。
- どのリビジョンのファイルやディレクトリが現在の作業コピーに あるのか。
- ファイルやディレクトリに結びついたユーザ定義の属性。
- 作業コピーファイルの修正元 (編集前) コピー。

`.svn` ディレクトリに格納されたデータには ほかにもいろいろありますが、最も重要なアイテムのいくつかだけに ついて説明します。

8.4.1 Entries ファイル

`.svn` ディレクトリにある一番重要な ファイルは `entries` ファイルです。 このファイルは XML ドキュメントでその内容は作業コピーディレクトリ中の バージョン管理下にあるリソースについての管理情報のあつまりです。 リポジトリ URL、修正元リビジョン、ファイルのチェックサム、修正元 テキストと属性のタイムスタンプ、予告と衝突状態に関する情報、最後に コミットしたことに関する情報 (実行者、リビジョン、タイムスタンプ)、 ローカルコピー履歴 — Subversion クライアントが管理しているリソース について興味のある情報はすべてここに記録されています。Subversion と CVS の管理領域の比較

典型的な`.svn` ディレクトリの内部を見ると、 それは CVS の管理ディレクトリで CVS が管理 する情報よりも、少し多いことがわかります。 `entries` ファイルは現在の作業コピーディレクトリ の状態を記述した XML を含んでいて、これは基本的に CVS の `Entries` と `Repository` を一緒にしたものになります。

以下は、実際の entries ファイルの例です:

わかるように、entries ファイルは本質的にはエントリのリストです。 `entry` タグのそれぞれは三つのうちのどれかを 表現しています: 作業コピーディレクトリ自身 (「this directory」エントリと呼ばれ、*`name`* 属性が空の値であるものとして示されています)、その作業コピーディレクトリ にあるファイル (*`kind`* 属性が `"file"` に設定されているものとして示されています)、あるいは 作業コピーのサブディレクトリ (*`kind`* がここでは `"dir"`に設定されます)。エントリがこのファイルに格納される ファイルとサブディレクトリは既にバージョン管理下にあるか (上の例の `zeta` ファイルのように)、この作業コピー ディレクトリの変更が次にコミットされるときにバージョン管理下に 追加することが予告されているか、です。エントリのそれぞれは ユニークな名前を持ち、特定のノード種別を持ちます。

開発者は、Subversion が `entries` ファイルを 読み書きするときに使う特別の規則に注意すべきです。 すべてのエントリは自分のリビジョンと、結びついている URL を 持っていますが、サンプルファイル中のすべての `entry` タグが明示的な *`revision`* や *`url`* 属性を持つわけではありません。 Subversion はエントリが明示的にこの二つの属性を持たないことも 認めていますが、それは、その値が、`"svn:this-dir"` エントリにあるデータと同じか、簡単に計算できる場合です。 [*4] また、サブディレクトリのエントリについては、Subversion は 重要な属性 — 名前、種別、url、リビジョン、そして予告状況 のみを保存するということに注意してください。重複する情報を 減らすために、Subversion は、サブディレクトリに関する完全な情報 を決定

[*4] つまり、エントリの URL は親ディレクトリ URL と エントリ名称をつなげたものと同じとみなすということです。

する方法として、そのサブディレクトリに下りていき、その ディレクトリ自身の.svn/entries ファイルの "svn:this-dir" エントリを読むように指示します。 しかし、サブディレクトリへの参照は、その親の entries ファイルに記録されていて、サブディレクトリ がディスクから削除されてしまったような場合でも基本的なバージョン管理 操作をするのには十分な情報を持っています。

8.4.2 修正元コピーと属性ファイル

前に注意したように、.svn ディレクトリはまた 修正元の「text-base」バージョンのファイルを保存しています。 これは.svn/text-base にあります。修正元 コピーの利点は、いくつかあります。 — ネットワークの通信なしに ローカル修正をチェックして差分を報告したり、ネットワーク通信なしに 修正したり削除したファイルを元に戻したり、サーバへの変更点の送信 サイズを減らしたりできます — しかし、少なくともそれぞれの バージョン管理されたファイルを二つディスク上に保存するコストが 発生します。最近では、これはほとんどのファイルについて無視できる程度の ものです。しかし、バージョン管理されたファイルが大きくなるにつれ この状況はひどいことになっていきます。「text-base」 をオプションにしては、という意見もあります。 しかし皮肉にも、バージョン管理するファイルのサイズが大きくなるに つれて、「text-base」の存在も、それだけ重要になっていきます — ファイルにしたほんの少しの変更をコミットしたいだけなのに、ばかでかい ファイルをネットワーク越しに送ろうなんて、誰が考えるでしょう?

「text-base」ファイルと同じような目的で、属性ファイルと、その修正元 「prop-base」コピーがあります。それぞれ、.svn/props と.svn/prop-base にあります。 ディレクトリも属性を持つことができるので、.svn/dir-props と .svn/dir-prop-base ファイルがあります。 これらの属性ファイルのそれぞれ (「作業中」と「元の」バージョン) は 属性名と属性値を格納するのに、単純な「ディスク上ハッシュ」ファイル形式 を使います。

8.5 WebDAV

WebDAV (「Web ベースの分散編集とバージョン化」の略) は標準的な HTTP プロトコルの 拡張で、基本的には読み出し専用の媒体である web を、読み書き可能な 媒体とするために設計されました。考え方としては、ディレクトリとファイルは — 読み書き可能なオブジェクトとして — Web 上で共有できるという ものです。RFC 2518 と RFC 3253 は、HTTP の WebDAV/DeltaV 拡張について記述 されていて、(ほかの有用な情報とともに)<http://www.webdav.org/>で入手可能です。

いくつかのオペレーティングシステムのファイルブラウザは既に WebDAV を 使ったネットワークディレクトリをマウントすることができます。 Win32 では、Windows Explorer は Web フォルダ (それはまさに、WebDAV が用意したネットワークの場所ですが) と呼んでいるものを、あたかも それが普通の共有フォルダであるかのように参照することができます。 Mac OS X もこの能力があり、Nautilus や Konqueror ブラウザもそうです。 (これらは、GNOME と KDE 上でそれぞれ動きます)。

これらすべてはどのようにして Subversion に適用されているのでしょうか? mod_dav_svn Apache モジュールはそのうちの一つのネットワークプロトコルとして WebDAV と DeltaV で拡張された HTTP を使っています。Subversion は mod_dav_svn を Subversion のバージョン化の考え方と RFC 2518, 3253 との間をつなぐものと して利用します。

WebDAV のもっと徹底的な議論、どのように動作し、Subversion はそれを どのように使うか、については、付録 B を 見てください。他の話題とともに、Subversion がどの程度 一般的な WebDAV の仕様を引き継いでおり、一般的な WebDAV クライアント との相互運用性にどんな影響を与えるかについての議論があります。

8.6 メモリプールを使ったプログラミング

C 言語を使ったことのあるほとんどすべての開発者は、ある時点で メモリ管理のことでうんざりしてため息をつくことがあるでしょう。 利用するために必要な十分なメモリを確保し、その利用状況を記録し、 いらなくなったらメモリを解放する — そういう処理は非常に 複雑です。そしてもちろん、それに失敗すると、プログラムが壊れて しまい、ひどいときにはコンピュータが壊れてしまいます。 幸運にも、Subversion が可搬性のために利用している APR ライブラリ は apr_pool_t 型を用意していて、これは メモリのプールを表現するものです。

メモリプールはプログラムによって利用するために確保されたメモリ ブロックの抽象的な表現です。標準的な `malloc()` 関数 とその亜種を使って OS から直接メモリを取得するかわりに、APR をリンクした プログラムは単にメモリプールを作る要求を出すことで行います。 (これには `apr_pool_create()` 関数を使います) APR は OS から自然なサイズのメモリを確保し、そのメモリはすぐにプログラム で使うことができるようになります。プログラムがプールメモリを必要と するときにはいつでも、`apr_palloc()` のような APR プール API 関数のどれかを使うことができて、それはプールから 汎用的なメモリを確保して返します。プログラムは 要求ビットとプールからのメモリを要求し続けることができて、 APR はその要求を承認し続けることができます。プールはプログラムに あわせて自動的にサイズが大きくなり、最初プールに含まれていたよりも 多くのメモリを要求することができます。これはシステムにメモリが なくなるまで続けることができます。

これでプールの話が終わりなら、特別の注意を払う必要もないのですが。 幸運にも、そうではありません。プールは作られるだけではありません: それは またクリアしたり削除することもできます。これには `apr_pool_clear()` と `apr_pool_destroy()` をそれぞれ利用します。 これは開発者にいくつもの — あるいは何千もの — 領域をプールから 取得して、その後一度の関数呼び出して、そのすべてクリアする柔軟性を 与えます。さらに、プールは階層を持っています。既に作られたどのプール にも「サブプール」を作ることができます。プールがクリアされると、そのプール のすべてのサブプールは削除されます。もしプールを削除すると、そのプール 自身と、サブプールの両方が削除されます。

先に進める前に、開発者は Subversion ソースコード中に、いま言ったような APR プール関数の呼び出しが、それほど多くないことに気づくでしょう。 APR プールは、いくつかの拡張メカニズムを持っていて、それはプールに 固有の「ユーザデータ」を接続する能力や、プールが削除されるときに 呼び出されるクリーンアップ関数を登録する仕組みなどがあります。 Subversion はこのような拡張機能を、それほど自明ではない方法で利用します。 それで Subversion は (そしてそのコードを使う人のほとんどは) ラッパ関数 である `svn_pool_create()`, `svn_pool_clear()`, そして `svn_pool_destroy()` を提供しています。

プールは基本的なメモリ管理にも役に立ちますが、ループや再帰的な状況で のプールの構築は本当にすばらしいものです。 ループはしばしばその繰り返し回数が不定であり、再帰的はその深さが不定 なので、このような領域でのコードのメモリ消費量は予測することができま せん。幸運にも、ネストしたメモリプールを使うと、このような潜在的な 恐ろしい状況を簡単に管理することができます。以下の例は、よくある非常に 複雑な情報でのネストしたプールの基本的な使い方を示しています。 — この状況とは、ディレクトリツリーを再帰的にたどりながら、ツリーの すべての場所である処理を実行する、といったものです。

この例はループと再帰的な状況の両方での 効率的な プールの利用法を説明するものです。それぞれの再帰は関数に渡すプール のサブプールを作ることで始まります。このプールはループの領域で利用 され、それぞれの繰り返しでクリアされます。この結果、メモリの利用は、 大雑把にいって再帰の深さにだけ比例し、最上位ディレクトリの子供としての ファイルとディレクトリの合計数には比例しません。この再帰関数の最初の呼び出しが終了した時点で、渡したプールに保存されたデータは実際には 非常に小さなものになります。この関数が、`alloc()` と `free()` 関数を一つ一つのデータに対して 呼び出さなくてはならないとしたときの複雑さを考えてみてください!

プールはすべてのアプリケーションに理想的なものではないかも知れませんが Subversion では非常に役に立ちます。Subversion 開発者として、プールの利用 に親しくなり、どうやってそれを正しく使うかに精通しなくてはなりません。 メモリ利用に関係したバグとメモリリークは API の種類によらず、 診断し、修正するのは難しいものですが、APR によって用意されたプール の作成は、非常に便利で、時間の節約につながる機能を持っています。

8.7 Subversion への貢献

Subversion プロジェクトについての情報の公式なドキュメントはもちろん、 プロジェクトウェブサイトの <http://subversion.tigris.org/>です。 そこにソースコードにアクセスする方法や、メーリングリストに参加する 方法についての情報があります。Subversion コミュニティはいつでも 新しい参加者を歓迎しています。もしソースコードを変更するという形の 貢献によってこのコミュニティに参加することに興味があるなら、 どんな感じに始めたら良いかのヒントを挙げます。

8.7.1 コミュニティへの参加

コミュニティに参加する最初のステップは最新の情報をいつでも入手できる 方法を見つけることです。これを一番効率的にやるには、開発者の議論のための メーリングリストに参加し (dev@subversion.tigris.org <mailto:dev@subversion.tigris.org>)、 コミットメーリングリストに参加することです (svn@subversion.tigris.org <mailto:svn@subversion.tigris.org>)。 このようなメーリングリストにある程度大雑把についていくだけでも、 重要なデザイン上の議論にアクセスできますし、Subversion ソースコードへの 実際の修正を見ることができますし、これらの変更の詳細なレビューに 立会い、変更を提案することができます。 これらの メールベースの議論の場は Subversion 開発での最重要な コミュニケーション手段です。 他の興味のある Subversion 関連リストについては、 Web サイトのメーリングリストのセクションを見てください。

しかし、何が必要かということをどうやって知れば良いのでしょう? プログラマにとって、開発を手助けしようという大きな意図を持っては いるが、良いとっかかりをつかめないのはよくあることです。結局、 掻きたいと思うかゆい場所がどこかを既に知っていてコミュニティに 参加する人はそれほど多くはありません。しかし、開発者の議論を 追いかけることによって、既に存在しているバグや、飛び交う機能要求 に注意を向けることができて、そのどれかがあなたの興味を引くかも知れま せん。また、未解決の、割り当てが決まっていない作業を探す良い場所と して、Subversion ウェブサイト上の Issue Tracking データベースがあります。 そこで現時点で既に知られているバグと、機能要求の一覧を見ることが できます。もし何か小さなことから始めたいのなら、「bite-sized」 という印の付いた問題を見てください。

8.7.2 ソースコードの取得

コードを編集するには、まずはコードを手に入れる必要があります。これは 公開の Subversion ソースリポジトリから作業コピーをチェックアウトしなくては ならないことを意味します。簡単に聞こえますが、少しだけ技巧的な作業に なります。Subversion のソースコードは、Subversion 自身によってバージョン管理 されているので、何か別の方法で既に動作する Subversion を取得することによって 「最初の手がかりを得る」必要があります。 一番普通の方法は、最新のバイナリパッケージをダウンロードする (あなたの マシンで利用できるものがある場合ですが)、か、最新のソースコードの tarball をダウロードして、自分の Subversion クライアントを作るかです。もしソースから 生成するのであれは、手順についてはソースツリーの最上位にある `INSTALL` ファイルに必ず目を通してください。

動作する Subversion クライアントを手に入れれば、 <http://svn.collab.net/repos/svn/

trunk/> にある Subversion のソースリポジトリの作業コピーをチェックアウトする用意 ができています: [*5]

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subversion
A    subversion/HACKING
A    subversion/INSTALL
A    subversion/README
A    subversion/autogen.sh
A    subversion/build.conf
…
```

上のコマンドは、最先端の、最新バージョンの Subversion のソースコード を、現在の作業ディレクトリに `subversion` という 名前のサブディレクトリを作りチェックアウトします。 明らかに、最後の引数は、個別の環境に応じて調整することができます。 新しい作業コピーディレクトリをどのように呼ぼうと、この操作が完了すれば Subversion のソースコードを取得できています。もちろん、他にもいくつかの 補助的なライブラリが必要になります (apr, apr-util, などなど) — 詳細は作業コピーの最上位にある `INSTALL` ファイルを見てください。

8.7.3 コミュニティのやり方に精通すること

これで Subversion ソースコードの最新版がある作業コピーを手に入れた ので、おそらく作業コピーの最上位ディレクトリにある「Hacker's Guide to Subversion」 を見ながらディレクトリの中をあれこれ調べたいと思うでしょう。これは 作業コピーの `www/hacking.html` ファイルにも、また <http://subversion.tigris.org/hacking.html> にある Subversion のウェブサイトからも取得できます。このページには Subversion に貢献するための一般的な手続きが含まれていて、それにはどのように して、残りのコードと矛盾しない形であなたのソースコードを正しく書くかとか、 提案したい変更点にどのような効率的な変更ログメッセージを付けるか、どのように 変更点をテストすれば良いか、などが含まれます。Subversion のソースリポジトリ に対するコミット権限は獲得しなくてはなりません — 実力本位の政府によって。 [*6] 「Hacker's Guide」には自分の提案しようとしている変更が 技術的に拒否されることなく受け入れられるかどうかを確認するためには 非常に貴重な資料です。

8.7.4 コードの変更とテスト

コードとコミュニティのポリシーを理解すれば、変更にとりかかることが できます。大きな問題に取り組んでいる場合でも、巨大な、根こそぎ 既存のものと取り替えてしまうような修正をするかわりに、小さな、しかし関連の ある変更の集まりを作ろうとするのが最良の方法です。 やろうとしていることに必要なコードの修正ができる限り少なければ、提案しよう としている変更はそれだけ簡単に理解されるでしょう (そして、検討するのも楽で しょう)。修正のセットのそれぞれを施したあとでは、あなたの Subversion ツリーは コンパイラが警告を一つも出さない状態になっているべきです。

[*5] この例でチェックアウトする URL は、`svn` で終わるのでは なく、`trunk` というサブディレクトリになって います。この理由については、Subversion のブランチとタグモデルの議論を 参照してください。

[*6] これは何かのエリート主義のように見えるかも知れませんが、「コミット 権限を獲得する」という概念は効率を考慮してのことです — 安全で役に立つ誰かの変更を検討し適用するため努力にかかる時間と、危険な変更を元に戻すという潜在的な時間との間の兼ね合いです。

Subversion にはかなり徹底した [*7] デグレートをチェックするためのテストスイートがあり、 提案しようとしている変更は、どのようなテストでも失敗しないようになって いることが望まれます。ソースツリーの最上位で **make check** を実行する (Unix の場合) ことで、自分の変更のチェックをすることができます。 あなたの貢献が拒絶される一番早い方法は (適切なログメッセージを付けなかった 場合以外は)、テストが通らない変更を送ることです。

一番良いシナリオは、実際に適切なテストを、テストスイートに追加し、 それであなたの変更点が期待したように動作することです。 実際、ときどき人が貢献しうる最良のことは新しいテストを単に追加 することです。エラーのきっかけになるような今後の修正から 守るような意味を込めて、現在の Subversion で動作している機能のために デグレードのテストを書くことができます。 また、既に知られている失敗を見せるための新しいテストを書くこともで きます。この目的のためには Subversion テストスイートは、あるテストは失敗することが期待されているものだと指定することを認めます。 (`XFAIL` といわれます)、そして Subversion が期待する形で失敗する限り、その テストの結果である `XFAIL` 自体は、成功したとみなされます。最後に、 良いテストスイートを用意すればするだけ、わかりにくいデグレートのバグ を診断するために浪費される時間を減らすことができます。

8.7.5 変更点の提供

ソースコードに対する修正をした後、明瞭でまとまったログメッセージを 作って、そのような変更を説明し、その理由を書いてください。それから メールを開発者用メーリングリストに送り、そこにはログメッセージと **svn diff** の出力 (これは Subversion の最上位作業コピー で実行してください) を含めてください。コミュニティのメンバーが あなたの変更が受け入れられると判断した場合、コミット権限を持った 誰か (Subversion のソースリポジトリに新しいリビジョンを作る許可を 持っている人) が、あなたの変更を公開されたソースコードツリーに 追加します。リポジトリに対して変更を直接コミットする権限は、 利点がある場合にだけ認められます — もし Subversion の理解や、 プログラミングの能力や、「チームスピリット」を示せば、あなたは きっとその権限を得ることができるでしょう。

[*7] 多分、ポップコーンでも食べたくなるかも。ここでの「徹底的な」は、非対話的なマシンで、約 30 分かかるという程度の意味に翻訳してください。

例 8.2.1 リポジトリ層の利用

```
/* Create a new directory at the path NEW_DIRECTORY in the Subversion
   repository located at REPOS_PATH.  Perform all memory allocation in
   POOL.  This function will create a new revision for the addition of
   NEW_DIRECTORY.  */
static svn_error_t *
make_new_directory (const char *repos_path,
                    const char *new_directory,
                    apr_pool_t *pool)
{
  svn_error_t *err;
  svn_repos_t *repos;
  svn_fs_t *fs;
  svn_revnum_t youngest_rev;
  svn_fs_txn_t *txn;
  svn_fs_root_t *txn_root;
  const char *conflict_str;

  /* Open the repository located at REPOS_PATH.  */
  SVN_ERR (svn_repos_open (&repos, repos_path, pool));

  /* Get a pointer to the filesystem object that is stored in
     REPOS.  */
  fs = svn_repos_fs (repos);

  /* Ask the filesystem to tell us the youngest revision that
     currently exists.  */
  SVN_ERR (svn_fs_youngest_rev (&youngest_rev, fs, pool));

  /* Begin a new transaction that is based on YOUNGEST_REV.  We are
     less likely to have our later commit rejected as conflicting if we
     always try to make our changes against a copy of the latest snapshot
     of the filesystem tree.  */
  SVN_ERR (svn_fs_begin_txn (&txn, fs, youngest_rev, pool));

  /* Now that we have started a new Subversion transaction, get a root
     object that represents that transaction.  */
  SVN_ERR (svn_fs_txn_root (&txn_root, txn, pool));

  /* Create our new directory under the transaction root, at the path
     NEW_DIRECTORY.  */
  SVN_ERR (svn_fs_make_dir (txn_root, new_directory, pool));

  /* Commit the transaction, creating a new revision of the filesystem
     which includes our added directory path.  */
  err = svn_repos_fs_commit_txn (&conflict_str, repos,
```

例 8.3.1 Python を使ったリポジトリ層

```
#!/usr/bin/python

"""Crawl a repository, printing versioned object path names."""

import sys
mport os.path
import svn.fs, svn.core, svn.repos

def crawl_filesystem_dir(root, directory, pool):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
    a list of all the paths at or below DIRECTORY.  Use POOL for all
    allocations."""

    # Print the name of this path.
    print directory + "/"

    # Get the directory entries for DIRECTORY.
    entries = svn.fs.svn_fs_dir_entries(root, directory, pool)

    # Use an iteration subpool.
    subpool = svn.core.svn_pool_create(pool)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Clear the iteration subpool.
        svn.core.svn_pool_clear(subpool)

        # 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, subpool):
            crawl_filesystem_dir(root, full_path, subpool)
        else:
            # Else it's a file, so print its path here.
            print full_path

    # Destroy the iteration subpool.
    svn.core.svn_pool_destroy(subpool)
```



```
def crawl_youngest(pool, repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
```

例 8.3.2 A Python Status Crawler

```
#!/usr/bin/env python
"""Crawl a working copy directory, printing status information."""

import sys
import os.path
import getopt
import svn.core, svn.client, svn.wc

def generate_status_code(status):
    """Translate a status value into a single-character status code,
    using the same logic as the Subversion command-line client."""

    if status == svn.wc.svn_wc_status_none:
        return ' '
    if status == svn.wc.svn_wc_status_normal:
        return ' '
    if status == svn.wc.svn_wc_status_added:
        return 'A'
    if status == svn.wc.svn_wc_status_missing:
        return '!'
    if status == svn.wc.svn_wc_status_incomplete:
        return '!'
    if status == svn.wc.svn_wc_status_deleted:
        return 'D'
    if status == svn.wc.svn_wc_status_replaced:
        return 'R'
    if status == svn.wc.svn_wc_status_modified:
        return 'M'
    if status == svn.wc.svn_wc_status_merged:
        return 'G'
    if status == svn.wc.svn_wc_status_conflicted:
        return 'C'
    if status == svn.wc.svn_wc_status_obstructed:
        return '~'
    if status == svn.wc.svn_wc_status_ignored:
        return 'I'
    if status == svn.wc.svn_wc_status_external:
        return 'X'
    if status == svn.wc.svn_wc_status_unversioned:
        return '?'
    return '?'

def do_status(pool, wc_path, verbose):
    # Calculate the length of the input working copy path.
    wc_path_len = len(wc_path)
```

例 8.4.1 典型的な.svn/entries ファイル

```
<?xml version="1.0" encoding="utf-8"?>
<wc-entries
   xmlns="svn:">
<entry
   committed-rev="1"
   name=""
   committed-date="2005-04-04T13:32:28.526873Z"
   url="http://svn.red-bean.com/repos/greek-tree/A/D"
   last-author="jrandom"
   kind="dir"
   uuid="4e820d15-a807-0410-81d5-aa59edf69161"
   revision="1"/>
<entry
   name="lambda"
   copied="true"
   kind="file"
   copyfrom-rev="1"
   schedule="add"
   copyfrom-url="http://svn.red-bean.com/repos/greek-tree/A/B/lambda"/>
<entry
   committed-rev="1"
   name="gamma"
   text-time="2005-12-11T16:32:46.000000Z"
   committed-date="2005-04-04T13:32:28.526873Z"
   checksum="ada10d942b1964d359e048dbacff3460"
   last-author="jrandom"
   kind="file"
   prop-time="2005-12-11T16:32:45.000000Z"/>
<entry
   name="zeta"
   kind="file"
   schedule="add"
   revision="0"/>
<entry
   name="G"
   kind="dir"/>
<entry
   name="H"
   kind="dir"
   schedule="delete"/>
</wc-entries>
```

例 8.6.1 効率的なプールの利用

```
/* Recursively crawl over DIRECTORY, adding the paths of all its file
   children to the FILES array, and doing some task to each path
   encountered.  Use POOL for the all temporary allocations, and store
   the hash paths in the same pool as the hash itself is allocated in.  */
static apr_status_t
crawl_dir (apr_array_header_t *files,
           const char *directory,
           apr_pool_t *pool)
{
  apr_pool_t *hash_pool = files->pool;  /* array pool */
  apr_pool_t *subpool = svn_pool_create (pool);  /* iteration pool */
  apr_dir_t *dir;
  apr_finfo_t finfo;
  apr_status_t apr_err;
  apr_int32_t flags = APR_FINFO_TYPE | APR_FINFO_NAME;

  apr_err = apr_dir_open (&dir, directory, pool);
  if (apr_err)
    return apr_err;

  /* Loop over the directory entries, clearing the subpool at the top of
     each iteration.  */
  for (apr_err = apr_dir_read (&finfo, flags, dir);
       apr_err == APR_SUCCESS;
       apr_err = apr_dir_read (&finfo, flags, dir))
    {
      const char *child_path;

      /* Clear the per-iteration SUBPOOL.  */
      svn_pool_clear (subpool);

      /* Skip entries for "this dir" ('.') and its parent ('..').  */
      if (finfo.filetype == APR_DIR)
        {
          if (finfo.name[0] == '.'
              && (finfo.name[1] == '\0'
                  || (finfo.name[1] == '.' && finfo.name[2] == '\0')))
            continue;
        }

      /* Build CHILD_PATH from DIRECTORY and FINFO.name.  */
      child_path = svn_path_join (directory, finfo.name, subpool);
```



```
      /* Do some task to this encountered path. */
      do_some_task (child_path, subpool);
```

第 9 章

Subversion リファレンス

9.1

この章は Subversion の完全なリファレンスです。 コマンドラインクライアント (**svn**) とそのすべてのサブコマンド、 さらにリポジトリ管理プログラム (**svnadmin** と **svnlook**) と そのすべてのサブコマンドです。

9.2 Subversion コマンドラインクライアント: **svn**

コマンドラインクライアントを使うには **svn** と入力してから、使いたいサブコマンドを入力してください。[*1], どんなスイッチや対象となる ターゲットについてもそうです — サブコマンドとスイッチの並びには特定の順序はありません。たとえば **svn status**: を使う場合に、以下はどれも有効です:

```
$ svn -v status
$ svn status -v
$ svn status -v myfile
```

クライアントコマンドの利用法については、もっとたくさんの 例が 第 3 章にあり、属性の管理については 項 7.3 にあります。

9.2.1 **svn** のスイッチ

Subversion はサブコマンドごとに違ったスイッチを持って いますが、どれもグローバルに働きます — つまりそれぞれの スイッチは一緒に利用されるサブコマンドにかかわらず同じ意味 を持ちます。たとえば `--verbose` (`-v`) は常に、どのサブコマンドと一緒に使っても「冗長な出力」を 意味します。

`--auto-props` `config` ファイルにある `enable-auto-props` ディレクティブを上書きする形で auto-props を有効にします。

`--config-dir` *`DIR`* Subversion にデフォルトの場所 (ユーザのホームディレクトリにある `.subversion`) のかわりに特定のディレクトリから設定情報を読み出すように指示します。

[*1] はい、そうですね。 `--version` スイッチを使うときにはサブコマンドは不要です。少し後でそのことに触れます。

`--diff-cmd` *CMD* ファイル間の差異を表示するために使う外部プログラムを指定します。 **svn diff** が起動されると、デフォルトでは unifid diff 形式 である Subversion の内部 diff エンジンが利用されます。外部 diff プログラム を利用したい場合は、`--diff-cmd` を使ってください。 `--extensions` スイッチで diff プログラムを指定することもで きます。(詳しくはこの章のあとで説明します)。

`--diff3-cmd` *CMD* ファイルをマージするために使う外部プログラムを指定します。

`--dry-run` コマンドの実行をそのままたどりますが、実際の動作はしません — 作業コピーに対しても、リポジトリに対するコマンドにも使えます。

`--editor-cmd` *CMD* ログメッセージや属性値を編集するのに使う外部プログラムを指定します。

`--encoding` *ENC* コミットメッセージが指定した文字コード系でエンコードされていることを Subversion に伝えます。 デフォルトはあなたのオペレーティングシステムのもともとの ロケールで、コミットメッセージが別の文字コード系でエンコードされている場合にはそれを 指定する必要があります。

`--extensions(-x)` *ARGS* Subversion がファイル間の差異を得るときに使う外部 diff コマンドに渡さなくてはならない一つ以上の引数を指定します。複数の引数を渡したい 場合は、それらすべてをクオートでくくらなくてはなりません。(たとえば、 **svn diff –diff-cmd /usr/bin/diff -x ”-b -E”**)。 このスイッチは `--diff-cmd` スイッチも指定した場合 にのみ利用することができます。

`--file (-F)` *FILENAME* 指定したサブコマンドに対し、このスイッチの引数として 渡したファイルの内容が使われます。

`--force` 特定のコマンドまたは操作の実行を強制します。 Subversion が通常なら拒否するようないくつかの操作がありますが、 この強制スイッチを使い、Subversion に「そうしたら どうなるかはわかっているから、とにかく やらせてくれ」と伝えることができます。 このスイッチは、電子工作か何かで、最後に電源を入れる状況に 似ています — 何をやろうとしているのかよく理解していなければ きっと嫌な目に遭うでしょう。

`--force-log` `--messages (-m)` や `--file (-F)` オプションに 渡す疑わしい引数を正しいものとして強制的に受け入れる ように指示します。デフォルトでは、Subversion は正しいサブコマンドで あると解釈するかわりに、そのような引数が間違いであるとして エラーを生成します。 たとえば、バージョン化されたファイルのパスを`--file (-F)` オプションに渡すと、Subversion はそのパスが 操作対象として意図されていたと考えずに、その引数には間違いがあると みなし、ログメッセージのためのバージョン化されていない元ファイルを 用意することに失敗してしまいます。自分の意図を宣言し、この種のエラー を無視するためには、`--force-log` オプションをログメッセージ を受け入れるサブコマンドに対して指定する必要があります。

`--help (-h or -?)` 一つ以上のサブコマンドと一緒に使って、それぞれのサブコマンドの 組み込みヘルプテキストを表示します。このスイッチだけを単独で 使うと、一般的なコマンドクライアントのヘルプテキストを表示します。

`--ignore-ancestry` Subversion に対して 差分を計算するときに、系統情報を無視するよう指示します (単にパス内容のみを利用します)。

`--ignore-externals` Subversion に外部定義と外部定義によって管理される外部作業コピー を無視するように指示します。

`--incremental` (他の出力と) 連結するのに適した形式で 出力を表示します。

`--limit` *`NUM`* 最初の *`NUM`* 個のログメッセージだけを 表示します。

`--message` (`-m`) *`MESSAGE`* コマンドライン上でコミットメッセージを指定します。このスイッチの あとに、たとえば:

```
$ svn commit -m "They don't make Sunday."
```

`--new` *`ARG`* *`ARG`* を新しいターゲットとして利用します。

`--no-auth-cache` 認証情報 (ユーザ名やパスワードなど) を Subversion 管理ディレクトリにキャッシュしないよう 指示します。

`--no-auto-props` `config` ファイル中の `enable-auto-props` ディレクティブを上書きする形で auto-props を無効にします。

`--no-diff-added` Subversion が追加されたファイルの差分を表示するのを防ぎます。 ファイルを追加した場合のデフォルトの動作は **svn diff** については、すでに (空の) ファイルが存在していて、そこに新たな ファイルの内容全体が追加された場合の差分の出力と同じになります。

`--no-diff-deleted` 削除されたファイルの差分を表示しないように Subversion に指示します。 デフォルトではファイルを削除したときの **svn diff** の 出力は、ファイルは削除されずゼロバイトで残っているかのような 形になります。

`--no-ignore` `global-ignores` 設定オプション、あるいは svn:ignore 属 性にマッチしたため通常なら省略されるようなファイルの ステータス一覧を表示します。 項 7.2.3.2 と 項 7.3.3.3 にさらに詳しい情報があります。

`--no-unlock` 自動的にファイルをアンロックしません (デフォルトのコミットの動作は コミットの一環として関連するすべてのファイルをアンロックします)。 より詳しくは項 7.4 を見て ください。

`--non-interactive` 認証が失敗したり十分な許可がないときに、認証要求するのを 抑制します。(たとえば、ユーザ名とパスワード) これは、 Subversion を自動スクリプト中で実行しているようなときで 認証要求させるよりも単に失敗したほうが便利な場合に 役に立ちます。

`--non-recursive` (`-N`) サブディレクトリに対してサブコマンドを再帰的に実行するのを 抑止します。ほとんどのサブコマンドはデフォルトで再帰的に実行 されますが、いくつかのサブコマンド — 普通は、作業コピーの 変更に対する削除や取り消しが起きるようなもの — はそうでは ありません。

`--notice-ancestry` 差分を計算するときに系統情報を考慮します。

`--old` *`ARG`* *`ARG`* を古いターゲットとして利用します。

`--password` *`PASS`* コマンドライン上で認証用パスワードを指定します — これを 指定しなければ、必要な場所で Subversion はパスワード入力を 要求してきます。

`--quiet` (`-q`) 実行中に重要な情報だけを表示するように指示します。

`--recursive` (`-R`) サブコマンドをサブディレクトリに対して再帰的に実行する ようにします。ほとんどのサブコマンドはデフォルトで再帰的な 動作をします。

`--relocate` *`FROM TO [PATH...]`* **svn switch** サブコマンドと一緒に使うことで作業コピー が参照しているリポジトリの場所を変更します。これは、リポジトリの 場所が移動しても、既に存在している作業コピーを続けて利用したい場合 に便利です。例として、**svn switch** を参照してください。

`--revision` (`-r`) *`REV`* 特定の操作で、リビジョン (またはリビジョンの範囲) を設定します。 リビジョンはリビジョンスイッチの引数として、番号、キーワード、 日付 (この場合は中かっこを使って) のどれかで指定することができます。 リビジョンを二つ指定するときにはコロンで区切ります。たとえば:

```
$ 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}
```

詳しくは項 3.4.2 を見てください。

`--revprop` ファイルやディレクトリの属性のかわりに、リビジョンの属性に対して 操作するようにします。このスイッチを使う場合は `--revision(-r)` を使ってリビジョンも 渡す必要があります。 バージョン化しない属性についての詳細は項 5.2.2 を見てください。

`--show-updates` (`-u`) 作業コピーのどのファイルが最新であるかの情報を表示します。 これは実際に自分に作業ファイルを更新しません — もし **svn update** を実行したとしたら、どの ファイルが更新されるかを表示するだけです。

`--stop-on-copy` Subversion のサブコマンドでバージョン化されたリソースを巡回する ようなものについて、コピーを発見したときには履歴情報の収集を止める ようにします。 — ここで言うコピーとは履歴の中でリポジトリの ほかの場所からコピーされたリソースがあるような場所のことを言って います。

`--strict` Subversion が厳密なセマンティクスを使うように します。ここで言う”セマンティクス”の意味ですが、具体的なサブコマンドに則して 説明しなければ漠然としたものになってしまいます。

`--targets` *`FILENAME`* コマンドライン上にすべてのファイルを記入するかわりに、 指定したファイルから操作したいファイルの一覧を取得するよう Subversion に指示します。

`--username` *`NAME`* コマンドライン上での認証に、指定したユーザ名称を使うように 指示します — そうでなければ、必要な場所で、Subversion は その入力をユーザに求めます。

`--verbose (-v)` サブコマンド実行時に、できるたけ多くの情報を表示するように 指示します。Subversion は、追加フィールド、すべてのファイルの 詳細情報、動作内容についての追加情報などを表示するようになります。

`--version` クライアントプログラムのバージョン番号を表示します。この情報は バージョン番号のほか、Subversion リポジトリにクライアントプログラム がアクセスするために利用できるモジュールの一覧も表示します。

`--xml` XML フォーマットで出力します。

9.2.2 **svn** サブコマンド

名前

svn add — 通常ファイル、ディレクトリ、シンボリックリンクを追加します。

名前

svn add — 通常ファイル、ディレクトリ、シンボリックリンクを追加します。

用法

```
svn add PATH...
```

説明

通常ファイル、ディレクトリ、シンボリックリンクを作業コピーに 追加しリポジトリに対する 追加予告をします。次のコミットでリポジトリにアップロードして 追加されます。何かを追加し、コミット前に気が変わった場合は、 **svn revert** で追加を取り消す ことができます。

別名

なし

変更対象

作業コピー

Accesses Repository

なし

スイッチ

```
--targets FILENAME
--non-recursive (-N)
--quiet (-q)
--config-dir DIR
--auto-props
--no-auto-props
--force
```

例

作業コピーにファイルを追加:

```
$ svn add foo.c
A         foo.c
```

ディレクトリを追加するとき **svn add** のデフォルトは 再帰的です:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

内部にあるファイルを追加することなしに、ディレクトリのみ追加する ことができます:

```
$ svn add --non-recursive otherdir
A         otherdir
```

通常、コマンド **svn add *** はすでにバージョン管理下に あるすべてのディレクトリを飛ばします。しかし作業コピーにあるすべての バージョン化されていないオブジェクトを再帰的に追加したいこともあるでしょ う。**svn add** に `--force` オプションを渡すと 既にバージョン化されているディレクトリにも再帰的に降りていきます:

```
$ svn add * --force
A         foo.c
A         somedir/bar.c
A         otherdir/docs/baz.doc
…
```

名前

svn blame — 指定されたファイルまたは URL の変更者とリビジョン情報をインラインで表示 します。

名前

svn blame — 指定されたファイルまたは URL の変更者とリビジョン情報をインラインで表示 します。

用法

```
svn blame TARGET[@REV]...
```

説明

指定されたファイルまたは URL の変更者とリビジョン情報をインラインで表示 します。それぞれのテキスト行には先頭に変更者 (ユーザ名) と、最後にその行に 変更があったリビジョン番号の注釈がつきます。

別名

praise, annotate, ann

変更

なし

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--verbose
```

例

自分のテスト用リポジトリにある `readme.txt` の注釈付きソースを見たい場合には:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry You should read this.
```

名前

svn cat — 指定したファイルまたは URL の内容を表示します。

名前

svn cat — 指定したファイルまたは URL の内容を表示します。

用法

```
svn cat TARGET[@REV]...
```

説明

指定したファイルまたは URL の内容を表示します。 ディレクトリの内容の表示については、**svn list** を見てください。

別名

なし

変更対象

なし

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

チェックアウトすることなしにリポジトリ中の readme.txt を表示したい 場合:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```

ティップ

作業コピーが最新ではない (か、作業コピーに何か修正を加えた 場合) 状態で、作業コピーのあるファイルの HEAD リビジョンを見たい場合、パスを指定すれば **svn cat** は 自動的に HEAD リビジョン にアクセスします:

```
$ cat foo.c
This file is in my local working copy
and has changes that I've made.

$ svn cat foo.c
Latest revision fresh from the repository!
```

名前

svn checkout — リポジトリから作業コピーをチェックアウトします。

名前

svn checkout — リポジトリから作業コピーをチェックアウトします。

用法

```
svn checkout URL[@REV]... [PATH]
```

説明

リポジトリから作業コピーをチェックアウトします。 *PATH* が省略されれば コピー先として URL のベース名が使われます。 複数の URL が指定された場合には、 それぞれが、*PATH* のサブディレクトリに チェックアウトされますが ここでのサブディレクトリの名前は URL のベース名になります。

別名

co

変更対象

作業コピーが新たに作られます。

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--ignore-externals
--config-dir DIR
```

例

mine と呼ばれるディレクトリに作業コピーをチェックアウト:

```
$ svn checkout file:///tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2.
$ ls
mine
```

二つの異なるディレクトリを二つの別々の作業コピーに チェックアウト:

```
$ 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
```

二つの異なるディレクトリを二つの別々の作業コピーに チェックアウトするが、両方とも working-copies と呼ばれる ディレクトリ中に作る:

```
$ 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
```

チェックアウトを中断する (かまたは、ネットワーク接続の不具合などにより チェックアウトが中断されるかした) 場合、もう一度純粋なチェックアウトを 実行しても、不完全な作業コピーを更新することでも再開できます:

```
$ svn checkout file:///tmp/repos/test test
A  test/a
A  test/b
^C
svn: The operation was interrupted
svn: caught SIGINT

$ svn checkout file:///tmp/repos/test test
A  test/c
A  test/d
^C
svn: The operation was interrupted
svn: caught SIGINT

$ cd test
$ svn update
A  test/e
A  test/f
Updated to revision 3.
```

名前

svn cleanup — 作業コピーを再帰的に正常化する。

名前

svn cleanup — 作業コピーを再帰的に正常化する。

用法

```
svn cleanup [PATH...]
```

説明

作業コピーを再帰的に正常化するため、操作が未完了の ロックを削除します。「作業コピーロック」エラーが 発生したときには、このコマンドを実行して、ロックを解除し、 作業コピーを正常に戻してください。

何かの理由で **svn update** の失敗が外部 diff プログラム の実行の問題である場合 (たとえばユーザ入力やネットワークの問題など) には 外部 diff プログラムでのマージ処理を完結させるために `--diff3-cmd` を渡すことができます。また `--config-dir` スイッチで任意の設定ディレクトリの指定もできます。ただしこれらのスイッチが 必要な場面は非常にまれです。

別名

なし

変更対象

作業コピー

リポジトリへのアクセス

発生しない

スイッチ

```
--diff3-cmd CMD
--config-dir DIR
```

例

svn cleanup は何も出力しないので、あまり例はありません。 もし、*PATH* を指定しなければ、「.」をかわりに使います。

```
$ svn cleanup

$ svn cleanup /path/to/working-copy
```

名前

svn commit — 作業コピーの変更点をリポジトリに送ります。

名前

svn commit — 作業コピーの変更点をリポジトリに送ります。

用法

```
svn commit [PATH...]
```

説明

作業コピーの変更点をリポジトリに送ります。 `--file` か、 `--message` のオプションを指定しなければ **svn** はエディタを起動し、ユーザに コミットメッセージを作成させます。 項 7.2.3.2 にある `editor-cmd` の章を見てください。

svn commit は `--no-unlock` を指定しなければ 見つかったロックトークンを送信し、コミットされた すべての *PATHS* 上に あるロックを (再帰的に) 開放します。

> ティップ
>
> 
>
> コミット処理を始め、Subversion がメッセージ作成のためのエディタを起動 した後でもまだ変更をコミットせず中断することができます。コミットを 取り消したければコミットメッセージを保存せずに単にエディタを終了 してください。Subversion はコミットを中断するか、メッセージなしで継続 するか、メッセージを再編集するかを聞いてきます。

別名

ci
ci (「check in」 の短縮形です; 「co」ではありません。 「co」は「checkout」の短縮形です。)

変更対象

作業コピー、リポジトリ

リポジトリへのアクセス

発生する

スイッチ

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--no-unlock
--non-recursive (-N)
--targets FILENAME
--force-log
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

```
--encoding ENC
--config-dir DIR
```

例

コマンドラインで指定したコミットメッセージとカレントディレクトリ (「.」) を暗黙のターゲットとして単純な変更をコミットします:

```
$ svn commit -m "added howto section."
Sending        a
Transmitting file data .
Committed revision 3.
```

ファイル foo.c (コマンドラインで明示的に 指定) の変更点を、ファイル msg の内容をコミットメッセージと してコミット:

```
$ svn commit -F msg foo.c
Sending        foo.c
Transmitting file data .
Committed revision 5.
```

コミットメッセージとして、--file で 指定したファイルがバージョン管理下にある場合、--force-log スイッチを 使う必要がある:

```
$ 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 --file-log file_under_vc.txt foo.c
Sending        foo.c
Transmitting file data .
Committed revision 6.
```

削除予告ファイルをコミット:

```
$ svn commit -m "removed file 'c'."
Deleting       c

Committed revision 7.
```

名前

svn copy — 作業コピーやリポジトリ中の、 ファイルやディレクトリをコピー。

名前

svn copy — 作業コピーやリポジトリ中の、 ファイルやディレクトリをコピー。

用法

```
svn copy SRC DST
```

説明

作業コピーまたはリポジトリ中のファイルをコピーします。 *SRC* と *DST* は、作業コピー (WC) 上のパスでも、URL でもかまいません:

WC -> WC　追加用にファイルをコピーし、追加予告します。 (with history).

WC -> URL　WC のコピーを直接 URL にコミット。

URL -> WC　URL を WC にチェックアウトし、追加予告する。

URL -> URL　完全なサーバ上のみでのコピー。これは普通ブランチやタグに 利用されます。

> 注意
>
> 
>
> ファイルは一つのリポジトリの内部でのみコピー可能です。 Subversion はリポジトリ間コピーをサポートしていません。

別名

cp

変更対象

コピー先が URL である場合はリポジトリ。
コピー先が作業コピーのパスである場合は作業コピー。

リポジトリへのアクセス

コピー元またはコピー先がリポジトリであるか、 ソースリビジョン番号を参照する必要がある場合には発生する。

スイッチ

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

例

作業コピー中のファイルまたはディレクトリをコピー (コピー予告のみ — 次のコミットまでリポジトリには何も 起こらない):

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +   bar.txt
```

リポジトリ中の URL に作業コピー中のファイルまたはディレクトリ をコピー (同時にコミットされるので、コミットメッセージを 指定する必要あり):

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

リポジトリのファイルまたはディレクトリを作業コピーに コピー (コピーの予告のみ — 次のコミットまでリポジトリには 何も起こらない):

> ティップ
>
> 
> これはリポジトリ中の死んだファイルを復活させるおすすめの 方法です

```
$ svn copy file:///tmp/repos/test/far-away near-here
A         near-here
```

そして最後に二つの URL の間でコピーする方法:

```
$ svn copy file:///tmp/repos/test/far-away file:///tmp/repos/test/over-there -m "remote co

Committed revision 9.
```

> ティップ
>
> 
> リポジトリ中のリビジョンに「tag」をつける一番簡単な方法 — そのリビジョン (普通は HEAD) をタグ付けされた ディレクトリに、単に **svn copy** する。

```
$ svn copy file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prerelease -m "

Committed revision 12.
```

そのタグを忘れるのを心配する必要はありません — いつでも古いリビジョンを指定してタグ付けすることが できます:

```
$ svn copy -r 11 file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prereleas

Committed revision 13.
```

名前

svn delete — 作業コピーかリポジトリから ファイルまたはディレクトリを削除します。

名前

svn delete — 作業コピーかリポジトリから ファイルまたはディレクトリを削除します。

用法

```
svn delete PATH...
```

```
svn delete URL...
```

説明

PATH で指定されたファイルまたはディレクトリは次のコミット で削除することを予告します。ファイル (と、まだコミットしていない ディレクトリ) は直ちに作業コピーから削除されます。このコマンドは バージョン化されていないか修正されているものに対しては 動作しません。`--force` スイッチを使えばこの 動作を変更できます。

URL で指定されたファイルまたはディレクトリは 直接コミットを発行する形で削除されます。複数の URL は単一のトランザクションと して不分割にコミットされます。

別名

del, remove, rm

変更対象

ファイルに対して実行した場合は作業コピー。URL に対して実行した場合はリポジトリ

リポジトリへのアクセス

URL に対して操作した場合のみ発生する

スイッチ

```
--force
--force-log
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--targets FILENAME
--username USER
--password PASS
```

```
--no-auth-cache
--non-interactive
--encoding ENC
```

例

svn を使って作業コピーからファイルを 削除すると、単に削除が予告されるだけです。ファイルは次のコミット でリポジトリから削除されます。

```
$ svn delete myfile
D         myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

URL を削除しますが、それは直ちに起こるので ログメッセージを指定する必要があります:

```
$ svn delete -m "Deleting file 'yourfile'" file:///tmp/repos/test/yourfile

Committed revision 15.
```

この例は作業コピーに修正があるファイルを強制削除する方法です:

```
$ svn delete over-there
svn: Attempting restricted operation for modified resource
svn: Use --force to override this restriction
svn: 'over-there' has local modifications

$ svn delete --force over-there
D         over-there
```

名前

svn diff — 二つのパスの間の相違点を表示します。

名前

svn diff — 二つのパスの間の相違点を表示します。

用法

```
svn diff [-r N[:M]] [TARGET[@REV]...]

svn diff [-r N[:M]] --old OLD-TGT[@OLDREV] [--new NEW-TGT[@NEWREV]] [PATH...]

svn diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

説明

ふたつのパスの間の違いを表示します。 **svn diff** には 三つの使い方があります:

svn diff [-r N[:M]] [–old OLD-TGT] [–new NEW-TGT] [PATH...] は *OLD-TGT* と *NEW-TGT* の間の違いを表示します。 もし *PATH* があれば *OLD-TGT* や *NEW-TGT* に相対的なものとして扱われ、出力される 相違点はそのようなパスのみに限定されます。 *OLD-TGT* と *NEW-TGT* は作業コピーパスであるか *URL*[*@REV*] であるかです。 *OLD-TGT* のデフォルトは 現在の作業コピーで、 *NEW-TGT* のデフォルトは *OLD-TGT* になります。 *N* のデフォルトは BASE ですが、もし *OLD-TGT* が URL であれば HEAD になります。 *M* のデフォルトは現在の作業バージョン ですが、もし *NEW-TGT* が URL であれば HEAD になります。 **svn diff -r N** は *OLD-TGT* のリビジョンを *N* に設定します。 **svn diff -r N:M** も *NEW-TGT* のリビジョンを *M* に設定します。

svn diff -r N:M URL は **svn diff -r N:M –old=URL –new=URL** の略記法です。

svn diff [-r N[:M]] URL1[@N] URL2[@M] は **svn diff [-r N[:M]] –old=URL1 –new=URL2** の略記法です。

TARGET が URL なら N と M のリビジョンは `--revision` を経由したものであるか既に述べたように「@」記法で指定されたものになります。

TARGET が作業コピーパスであれば `--revision` スイッチの意味は:

`--revision N:M` サーバは *TARGET@N* と *TARGET@M* を比較します。

`--revision N` クライアントは *TARGET@N* と作業コピーを比較します。

(no `--revision`) クライアントは *TARGET* のベースリビジョンと作業コピーを 比較します。

別の構文を使うとサーバはそれぞれリビジョン *N* と *M* にある *URL1* と *URL2* を比較します。もし N か M が省略されれば *HEAD* の値が使われます。

デフォルトでは **svn diff** はファイルの系統を無視し、 単に比較対象になっている二つのファイルの内容を比較するだけです。 `--notice-ancestry` を使うとリビジョンを比較する際に、 問題になっているパスの系統が考慮されます (つまり、同じ内容を持っては いるが異なる系統を持つ二つのファイルに対して **svn diff** を実行すると、ファイル全体が一度削除され、 もう一度すべてが追加された形の結果を得ることになります)。

別名

di

変更対象

なし

リポジトリへのアクセス

相違点を知るのに必要な場合は発生する。ただし作業コピーの BASE リビジョンを取得する場合は除く。

スイッチ

```
--revision (-r) REV
--old OLD-TARGET
--new NEW-TARGET
--extensions (-x) "ARGS"
--non-recursive (-N)
--diff-cmd CMD
--notice-ancestry
--username USER
--password PASS
--no-auth-cache
--non-interactive
--no-diff-deleted
--config-dir DIR
```

例

BASE リビジョンと作業コピーを比較します。(**svn diff** の一番よくある使い方):

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
```

作業コピーの変更を以前のリビジョンと比較する方法:

```
$ svn diff -r 3900 COMMITTERS
```

```
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

リビジョン 3000 とリビジョン 3500 を「@」構文を使って比較:

```
$ svn diff http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000 http://svn.collab.net
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

リビジョン 3000 とリビジョン 3500 を範囲指定で比較 (この場合は一つの URL のみを渡せばいい):

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

範囲指定によって `trunk` にある全てのファイルの リビジョン 3000 と リビジョン 3500 を比較する:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

範囲指定によって `trunk` にある三つのファイルに ついてのみリビジョン 3000 と リビジョン 3500 を比較する:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk COMMITTERS README
```

作業コピーがある場合、長い URL 指定することなしに 相違点を取得することができます:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

--diff-cmd *CMD* -x を使って 外部 diff プログラムに直接引数を渡します

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
```

名前

svn export — ディレクトリツリーのエクスポート

名前

svn export — ディレクトリツリーのエクスポート

用法

```
svn export [-r REV] URL[@PEGREV] [PATH]
svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

説明

最初の構文では指定された URL のリポジトリからディレクトリツリーを エクスポートします。このさい、*REV* が指定されれば リビジョン *REV* から、そうでなければ HEAD のリビジョンが利用され、結果は *PATH* に出力されます。 *PATH* が省略されれば *URL* の 最後の部分がローカルディレクトリ名称として利用されます。

二番目の構文では *PATH1* で指定されたローカル作業コピーを *PATH2* に出力 します。すべての作業コピーへの変更は保存されますが、バージョン管理下 にないファイルはコピーされません。

別名

なし

変更対象

ローカルディスク

リポジトリへのアクセス

URL からのエクスポートの場合のみ発生する

スイッチ

```
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--non-recursive
--config-dir DIR
--native-eol EOL
--ignore-externals
```

例

作業コピーからのエクスポート (すべてのファイルとディレクトリをいちいち表示しない):

```
$ svn export a-wc my-export
Export complete.
```

リポジトリから直接エクスポート (すべてのファイルとディレクトリを表示):

```
$ svn export file:///tmp/repos my-export
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

オペレーティングシステムごとのリリースパッケージを作る場合、行末に特定 の EOL 文字を使ってエクスポートすることができると便利です。`--native-eol` オプションはこれをしますが、それはファイルが `svn:eol-style = native` 属性を持っているものだけに効果があります。 たとえばすべてが CRLF 改行を持っているファイルからなるツリーをエクスポート する場合 (おそらく Windows の .zip ファイルでの配布のような場合が考え られますが):

```
$ svn export file://tmp/repos my-export --native-eol CRLF
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

名前

svn help — ヘルプ

名前

svn help — ヘルプ

用法

```
svn help [SUBCOMMAND...]
```

説明

Subversion を使う場合の最良のガイドで、それに比べたらこの本などは 及びません!

別名

?, h

変更対象

なし

リポジトリへのアクセス

発生しない

スイッチ

```
--version
--quiet (-q)
```

名前

svn import — バージョン管理されていないファイルやツリーをリポジトリに コミットします。

名前

svn import — バージョン管理されていないファイルやツリーをリポジトリに コミットします。

用法

```
svn import [PATH] URL
```

説明

PATH のコピーを再帰的に *URL* にコミットします。 *PATH* がなければ 「.」 が仮定されます。 必要に応じて親ディレクトリがリポジトリに作られます。

別名

なし

変更対象

リポジトリ

リポジトリへのアクセス

発生する

スイッチ

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
```

```
--config-dir DIR
--auto-props
--no-auto-props
--ignore-externals
```

例

これは、ローカルディレクトリ `myproj` をリポジトリの根元に インポートします:

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/test
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

これはローカルディレクトリ `myproj` をリポジトリの `trunk/misc` にインポートします。ディレクトリ `trunk/misc` はインポートする前に 存在していなくてもかまいません — **svn import** は再帰的にディレクトリを作ります:

```
$ svn import -m "New import" myproj \
    http://svn.red-bean.com/repos/test/trunk/misc/myproj
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 19.
```

データほインポートした後でも、もとのツリーがバージョン 管理下に入ったわけではないのに注意し てください。作業を始めるには、まずそのツリーのための新しい作業コ ピーを **svn checkout** コマンドで作る必要が あります。

名前

svn info — ローカル、あるいはリモートパスにあるアイテムについての情報を表示します。

名前

svn info — ローカル、あるいはリモートパスにあるアイテムについての情報を表示します。

用法

```
svn info [TARGET...]
```

説明

作業コピーのパスと URL の両方についての情報を表示します。これには:

- パス
- 名前
- URL
- リビジョン
- リポジトリのルート
- リポジトリの UUID
- ノードの種類
- 最後に修正した人
- 最後に修正したリビジョン
- 最後に修正した日時
- 最後に更新したテキスト
- 最後に更新した属性
- チェックサム
- ロック・トークン
- ロックの所有者
- ロックの生成日時

別名

なし

変更対象

なし

リポジトリへのアクセス

URL に対する実行時のみ

スイッチ

```
--targets FILENAME
--recursive (-R)
--revision (-r)
--config-dir DIR
```

例

svn info は作業コピー中の ファイルまたはディレクトリに関する有用な情報を表示します。 次はファイルに関する情報を表示します:

```
$ svn info foo.c
Path: foo.c
Name: foo.c
URL: http://svn.red-bean.com/repos/test/foo.c
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 4417
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 20
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)
Checksum: /3L38YwzhT93BWvgpdF6Zw==
```

ディレクトリに対する情報を表示することもできます:

```
$ svn info vendors
Path: vendors
URL: http://svn.red-bean.com/repos/test/vendors
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 19
Node Kind: directory
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 19
Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
```

svn info は URL に対しても処理することができます (この例での readme.doc ファイルはロックされているので、ロック情報もまた 表示されていることに注意してください):

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc
Path: readme.doc
```

```
Name: readme.doc
URL: http://svn.red-bean.com/repos/test/readme.doc
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 1
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 42
Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Text Last Updated: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Checksum: d41d8cd98f00b204e9800998ecf8427e
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
```

名前

svn list — リポジトリ中のディレクトリエントリを一覧表示します。

名前

svn list — リポジトリ中のディレクトリエントリを一覧表示します。

用法

```
svn list [TARGET[@REV]...]
```

説明

それぞれの *TARGET* ファイルと *TARGET* ディレクトリのリポジトリ 中の内容を一覧表示します。 *TARGET* が作業コピーのパスである場合、対応するリポジトリ URL が利用されます。

デフォルトの *TARGET* 値は 「.」 で、現在の作業コピーディレクトリ のリポジトリ URL を意味します。

`--verbose` を使うと以下のフィールドがアイテムごとの 状態を示します:

- 最後のコミットのリビジョン番号
- 最後のコミットをした人
- データサイズ (バイト単位の)
- 最後のコミットの日時

`--xml` オプションをつけると XML 形式で 出力します (`--incremental` を同時に指定し なければヘッダとタグでくくられたドキュメント要素も一緒に出力され ます)。すべての情報が対象になります; `--verbose`

オプションは認められません。

別名

ls

変更対象

なし

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
--verbose (-v)
--recursive (-R)
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

svn list は、作業コピーを ダウンロードすることなしにどんなファイルがリポジトリにあるかを知る ときに役立ちます:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
…
```

追加情報を表示するのに `--verbose` スイッチを渡す こともできます。これだと UNIX の **ls -l** コマンドの出力のような感じになります:

```
$ 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 例/
```

詳しくは項 3.7.4 を見てください。

名前

svn lock — 作業コピーパスまたはリポジトリ中の URL をロックし、他のユーザがそこに 変更点をコミットできないようにします。

名前

svn lock — 作業コピーパスまたはリポジトリ中の URL をロックし、他のユーザがそこに 変更点をコミットできないようにします。

用法

```
svn lock TARGET...
```

説明

TARGET それぞれをロックします。 他のユーザによってすでに *TARGET* のどれかが ロックされていた場合、警告を出して残りの *TARGET* をロックします。他のユーザ、あるいは作業コピーの設定したロックを `--force` オプションで横取りすることもできます。

別名

なし

変更対象

作業コピー、リポジトリ

リポジトリへのアクセス

発生する

スイッチ

```
--targets ARG
--message (-m) ARG
```

```
--file (-F) ARG
--force-log
--encoding ARG
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
--force
```

例

作業コピー中のふたつのファイルをロックする:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

他のユーザによって現在ロックされている作業コピー中の ファイルをロックする:

```
$ svn lock tree.jpg
svn: warning: Path '/tree.jpg is already locked by user 'harry in \
     filesystem '/svn/repos/db'

$ svn lock --force foo
'tree.jpg' locked by user 'sally'.
```

作業コピーに関係しないファイルをロックする:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'sally'.
```

より詳細は 項 7.4 を見てください。

名前

svn log — コミットログメッセージの表示。

名前

svn log — コミットログメッセージの表示。

用法

```
svn log [PATH]
```

```
svn log URL [PATH...]
```

説明

デフォルトのターゲットは現在の作業ディレクトリのパスになります。 引数を指定しなければ **svn log** は自分の作業コピーの現在の作業ディレクトリ 自身とその内部のすべてのファイルとディレクトリに関するログメッセージを 表示します。 一つのパス、一つ以上のリビジョン、あるいはそれらの組み合わせを 指定することで結果内容を指定をできます。ローカルパスのデフォルト リビジョン範囲は、`BASE:1` です。

URL だけを指定すれば、その URL に含まれるすべてのログメッセージが 表示されます。URL の後にパスを付ければ URL 中のそれらのパスに含まれる メッセージだけが表示されます。URL のデフォルトリビジョン範囲は `HEAD:1` です。

`--verbose` を指定すると **svn log** はそれぞれのログメッセージと共に関連したすべてのパスを表示します。`--quiet` を指定すると **svn log** はログメッセージの本体部分を表示しなくなります (これは `--verbose` スイッチと両立します)。

それぞれのログメッセージは、そのリビジョンに影響のあるパスが二度 以上要求されても、一度だけ表示されます。ログはデフォルトではコピーされた 履歴に従います。`--stop-on-copy` はこの振る舞いを無効にしますが、ブランチが 起きた場所を特定する場合には役に立ちます。

別名

なし

変更対象

なし

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
```

```
--quiet (-q)
--verbose (-v)
--targets FILENAME
--stop-on-copy
--incremental
--limit NUM
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

最上位で **svn log** を実行することによって作業コピー中の変更されたすべての パスのログメッセージを見ることができます:

```
$ 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
…
```

作業コピー中の特定のファイルに関するすべてのログメッセージを調べます:

```
$ 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
…
```

作業コピーが手元にない場合、URL を log することができます:

```
$ 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
…
```

同じ URL の下のいくつかの別のパスがほしい場合 `URL [PATH...]` 構文を使うことができます。

```
$ 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
…
```

複数の log コマンドの結果をつなげたい場合、`--incremental` スイッチを使うことができます。 **svn log** は普通メッセージの最初にダッシュの行を表示 し、それぞれの引き続くログメッセージを表示し、最後のログメッセージが それに続きます。 もし **svn log** を二つのリビジョン範囲で実行した場合、次のような出力 になります:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | ...

------------------------------------------------------------------------
r15 | ...

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

しかし、二つの順番になっていないログメッセージをファイルに 出力したい場合、何か次のような感じになるでしょう:

```
$ svn log -r 14 > mylog
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | ...

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | ...

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | ...

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

incremental スイッチを使えば、出力中の重複したダッシュ行 の表示を避けることができます:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | ...

------------------------------------------------------------------------
r19 | ...

------------------------------------------------------------------------
r27 | ...
```

`--incremental` スイッチは、 `--xml` スイッチを使ったときと同じような出力 制御をします。

ティップ

もし **svn log** を特定のパス上の特定のリビジョン指定で実行すると 何も出力されないでしょう

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

それはパスはリビジョンによっては修正されなかったことを意味 しています。リポジトリの最上位で log するか、そのリビジョンで 修正したファイルを知っているなら、明示的にそれを指定することが できます:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

名前

svn merge — 二つのソースの差を作業コピーパスに反映します。

名前

svn merge — 二つのソースの差を作業コピーパスに反映します。

用法

```
svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]
```

```
svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]
```

```
svn merge -r N:M SOURCE[@REV] [WCPATH]
```

説明

最初の形式と二番目の形式ではソースとなるパス (これは最初の形式では URL, 二番目の形式では作業コピーパスとなります) はリビジョン *N* と *M* で指定され、 その二つが比較されます。リビジョンが省略されれば HEAD を指定されたものと みなします。

三番目の形式では *SOURCE* は URL か、作業コピーアイテムであり、その場合、 対応した URL が利用されます。この、リビジョン *N* と *M* の URL が、比較対象と なります。

WCPATH が変更を受け取る作業コピーパスです。もし *WCPATH* が省略されると、 デフォルトとして 「.」 が利用されます。ただし、両方のソースのベース名が同じで、 さらに、その名前のファイルが 「.」 にある場合は別で、この場合は、差分はその ファイルに適用されます。

svn diff とは違い、マージコマンドはマージ操作の実行 時にファイルの系統を考慮します。これはあるブランチでの変更点を別の ブランチにマージする場合に、あるブランチでは名称を変更したが、もう一方 ではそうしなかったような場合に非常に重要になります。

別名

なし

変更対象

作業コピー

リポジトリへのアクセス

URL に対して動作するときのみ発生する

スイッチ

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--force
--dry-run
--diff3-cmd CMD
--ignore-ancestry
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

ブランチを主系にマージします (主系の作業コピーがあり、ブランチがリビジョン 250 で作られたと仮定します):

```
$ 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
```

リビジョン 23 で分岐 (ブランチ化) して、そのブランチの中の主系に変更を マージしたいとします。これにはブランチの作業コピーの中で以下のような操作を します:

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors
U  myproj/thhgttg.txt
…
```

変更を一つのファイルにマージするには:

```
$ cd myproj
$ svn merge -r 30:31 thhgttg.txt
U  thhgttg.txt
```

名前

svn mkdir — バージョン管理下にある新しいディレクトリを作ります。

名前

svn mkdir — バージョン管理下にある新しいディレクトリを作ります。

用法

```
svn mkdir PATH...

svn mkdir URL...
```

説明

PATH または URL を最後の部分とするようなディレクトリを作ります。 作業コピー *PATH* で指定されたディレクトリは作業コピーへの追加として 予告されます。URL によって指定されたディレクトリは作成と同時に コミットされます。複数のディレクトリ URL は不分割にコミットされます。 どちらの場合でも途中のディレクトリはすべて存在していなくてはなりません。

別名

なし

変更対象

作業コピー。URL を指定した場合はリポジトリ

リポジトリへのアクセス

URL を指定した場合は発生する

スイッチ

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

例

作業コピー中にディレクトリを作る:

```
$ svn mkdir newdir
A         newdir
```

リポジトリに作る (コミットが発生するのでログメッセージを 指定する必要がある):

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir

Committed revision 26.
```

名前

svn move — ファイルやディレクトリを移動する。

名前

svn move — ファイルやディレクトリを移動する。

用法

```
svn move SRC DST
```

説明

このコマンドは作業コピーまたはリポジトリにある ファイルやディレクトリを移動します。

> ティップ
>
> 
>
> このコマンドは、**svn copy** の後に **svn delete** を実行するのと同じことです。

> 注意
>
> 
>
> Subversion では作業コピーと URL の間の移動はサポートしていません。 さらに、ファイルの移動は一つのリポジトリの内部でのみ可能です — Subversion はリポジトリ間の移動をサポートしていません。

WC -> WC　移動してから、ファイルやディレクトリを追加予告します。

URL -> URL　サーバ上での名称変更

別名

mv, rename, ren

変更対象

作業コピー。URL を指定した場合はリポジトリ

リポジトリへのアクセス

URL を指定した場合は発生する

スイッチ

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

例

作業コピーのファイルを移動:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

リポジトリのファイルを移動 (コミットが発生するので コミットメッセージを指定する必要がある):

```
$ svn move -m "Move a file" http://svn.red-bean.com/repos/foo.c \
                            http://svn.red-bean.com/repos/bar.c
```

```
Committed revision 27.
```

名前

svn propdel — アイテムから属性を削除します。

名前

svn propdel — アイテムから属性を削除します。

用法

```
svn propdel PROPNAME [PATH...]
```

```
svn propdel PROPNAME --revprop -r REV [URL]
```

説明

これはファイル、ディレクトリ、リビジョンから属性を削除します。 最初の形式は作業コピーのバージョン管理された属性を削除し、 二番目の形式ではリポジトリリビジョン上のバージョン管理されていない属性を削除します。

別名

pdel, pd

変更対象

作業コピー。URL を指定した場合はリポジトリ

リポジトリへのアクセス

URL を指定した場合のみ発生する

スイッチ

```
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--username USER
--password PASS
```

```
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

作業コピーのファイルから属性を削除する

```
$ svn propdel svn:mime-type  some-script
property 'svn:mime-type' deleted from 'some-script'.
```

リビジョン属性を削除する:

```
$ svn propdel --revprop -r 26 release-date
property 'release-date' deleted from repository revision '26'
```

名前

svn propedit — バージョン管理されている一つ以上のアイテムの属性を編集する。

名前

svn propedit — バージョン管理されている一つ以上のアイテムの属性を編集する。

用法

```
svn propedit PROPNAME PATH...
```

```
svn propedit PROPNAME --revprop -r REV [URL]
```

説明

一つ以上の属性を好きなエディタで修正します。 最初の形式は作業コピー中のバージョン管理された属性を編集します。 二番目の形式ではリポジトリリビジョン上のバージョン管理されていない 属性を編集します。

別名

pedit, pe

変更対象

作業コピー。URL を指定した場合はリポジトリ

リポジトリへのアクセス

URL を指定した場合のみ発生する

スイッチ

```
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--editor-cmd EDITOR
--config-dir DIR
```

例

svn propedit は、複数の値を持つ 属性を簡単に変更することができます:

```
$ svn propedit svn:keywords  foo.c
    <svn will launch your favorite editor here, with a buffer open
    containing the current contents of the svn:keywords property.  You
    can add multiple values to a property easily here by entering one
    value per line.>
Set new value for property 'svn:keywords' on 'foo.c'
```

名前

svn propget — 属性の値を表示します。

名前

svn propget — 属性の値を表示します。

用法

```
svn propget PROPNAME [TARGET[@REV]...]
```

```
svn propget PROPNAME --revprop -r REV [URL]
```

説明

ファイル、ディレクトリ、リビジョンの属性値を表示します。 最初の形式は作業コピーにある一つ以上のアイテムのバージョン管理された属性を表示します。 二番目の形式ではあるリポジトリリビジョンのバージョン管理していない属性を 表示します。属性についての詳細は 項 7.3 を見てください。

別名

pget, pg

変更対象

作業コピー。URL を指定した場合はリポジトリ

リポジトリへのアクセス

URL を指定したときのみ発生する

スイッチ

```
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

作業コピー中のアイテムの属性を調べる:

```
$ svn propget svn:keywords foo.c
Author
Date
```

Rev

リビジョン属性についても同様:

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

名前

svn proplist — すべての属性を一覧表示します。

名前

svn proplist — すべての属性を一覧表示します。

用法

```
svn proplist [TARGET[@REV]...]
```

```
svn proplist --revprop -r REV [URL]
```

説明

ファイル、ディレクトリ、リビジョンのすべての属性を一覧表示します。 最初の形式では作業コピー中のバージョン管理された属性を表示しますが、二番目の 形式ではあるリポジトリリビジョンの属性を表示します。

別名

plist, pl

変更対象

作業コピー。URL が指定されている場合はリポジトリ

リポジトリへのアクセス

URL を指定した場合のみ発生する

スイッチ

```
--verbose (-v)
--recursive (-R)
--revision (-r) REV
--quiet (-q)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

作業コピーのアイテムの属性の一覧を見たい場合は proplist コマンドを使うことができます:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

しかし`--verbose`フラグを付けると、それぞれの属性の値も 一緒に表示することができるのでとても便利です:

```
$ svn proplist --verbose foo.c
Properties on 'foo.c':
  svn:mime-type : text/plain
  svn:keywords : Author Date Rev
  owner : sally
```

名前

svn propset — ファイル、ディレクトリ、リビジョンの PROPNAME の値 を PROPVAL に設定する。

名前

svn propset — ファイル、ディレクトリ、リビジョンの PROPNAME の値 を PROPVAL に設定する。

用法

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH [PATH [PATH ... ]]
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [URL]
```

説明

ファイル、ディレクトリ、リビジョンの *PROPNAME* の値を *PROPVAL* に設定します。 最初の例はバージョン管理された作業コピー中の属性値の変更で、二番目は バージョン管理されていないリポジトリ上のリビジョン属性値の作成です。

> ティップ
>
> 
>
> Subversion は、動作に影響を与えるたくさんの「特別な」属性 を持っています。詳しくは 項 7.3.3 を見てください。

別名

pset, ps

変更対象

作業コピー。URL を指定した場合のみリポジトリ

リポジトリへのアクセス

URL を指定した場合のみ発生する

スイッチ

```
--file (-F) FILE
--quiet (-q)
--revision (-r) REV
```

```
--targets FILENAME
--recursive (-R)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force
--config-dir DIR
```

例

ファイルの MIME タイプを設定する:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

UNIX 上で、あるファイルに実行属性を付けたいときには:

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

多分、共同作業者の便宜を考えると、ある属性を設定 するには内部的なポリシーがなくてはなりません:

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

特定のリビジョンのログメッセージを間違ってしまい、それを変更したいとき、 `--revprop` を使って、`svn:log` に新し いメッセージを設定します:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

あるいは、作業コピーを持っていない場合でも、URL を設定することが できます。

```
$ svn propset --revprop -r 26 svn:log "Document nap." http://svn.red-bean.com/repos
property 'svn:log' set on repository revision '25'
```

最後に、属性値をファイルを入力として設定することもできます。 この方法で、属性値にバイナリ値を設定することさえできます:

```
$ svn propset owner-pic -F sally.jpg moo.c
property 'owner-pic' set on 'moo.c'
```

注意

デフォルトでは Subversion リポジトリ中のリビジョン属性は変更 できません。リポジトリ管理者は `pre-revprop-change` という 名前のフックを作ることで明示的にリビジョン属性の 修正を有効にしなくてはなりません。 フックスクリプトについて詳しくは 項 5.3.1 を見てください。

名前

svn resolved — 作業コピーのファイルまたはディレクトリの 「衝突」 状態を取り除きます。

名前

svn resolved — 作業コピーのファイルまたはディレクトリの 「衝突」 状態を取り除きます。

用法

```
svn resolved PATH...
```

説明

作業コピーのファイルまたはディレクトリの 「衝突」 状態を取り除きます。 このコマンドは衝突マーカを意味的に解消するのではなく、単に 衝突に関係した中間ファイルを削除して、*PATH* でもう一度コミットするだけです。つまり Subversion にその衝突は既に「解消された」と伝えます。 衝突の解消についての詳細は 項 3.6.4 を見てください。

別名

なし

変更対象

作業コピー

リポジトリへのアクセス

発生しない

スイッチ

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

例

更新操作中に衝突があった場合、作業コピーは三つの新しいファイルを 作ります:

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

衝突を解消し、`foo.c` のコミットの準備が できた状態にある場合、**svn resolved** は あなたの作業コピーに、注意しなくてはならないすべてのことを 伝えます。

警告

単に衝突ファイルを削除してからコミットすることもできます が **svn resolved** は作業コピー管理領域の記録として、衝突ファイルを削除した ことも付け加えるので、このコマンドを使うのをお勧めします。

名前

svn revert — ローカルファイルへのすべての編集を取り消します。

名前

svn revert — ローカルファイルへのすべての編集を取り消します。

用法

```
svn revert PATH...
```

説明

ファイル、ディレクトリに対する変更をすべて取り消して 衝突の状態を解消します。**svn revert** は 作業コピーのアイテムの内容だけではなく、属性の変更も取り消します。 さらに既にやった予告操作を取り消すのにも使えます。 (たとえば、ファイルにたいする追加または削除の予告 も「取り消され」ます。

別名

なし

変更対象

作業コピー

リポジトリへのアクセス

発生しない

スイッチ

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

例

ファイルに対する変更の取り消し:

```
$ svn revert foo.c
Reverted foo.c
```

ディレクトリ全体を取り消したい場合は、 `--recursive` フラグを使います:

```
$ svn revert --recursive .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

最後に、どの予告操作も取り消すことができます:

```
$ 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
```

注意

svn revert の対象を指定しなければ、それは何もしません — 間違った修正の破棄から作業コピーを守るために、**svn revert** 操作は 少なくとも一つの引数を指定するように求めます。

名前

svn status — 作業コピーにあるファイルやディレクトリの状態を表示しす。

名前

svn status — 作業コピーにあるファイルやディレクトリの状態を表示しす。

用法

```
svn status [PATH...]
```

説明

作業コピーにあるファイルやディレクトリの状態を表示します。 引数がない場合は、ローカルで修正されたアイテムだけが表示されます (リポジトリに 対するアクセスは発生しません `--show-updates` を使うと、作業リビジョンと、 サーバの最新ではない情報も追加されます。 `--verbose` を使うと、すべてのアイテムに対する 完全なリビジョン情報を表示します。

出力の最初の 6 列のコラムはそれぞれ一文字幅で、作業コピーアイテムごとに いろいろな情報を表示します。

最初のコラムは、アイテムが追加、削除、それ以外の変更、のどの状態かを を示します。

' ' 変更はありません。

'A' アイテムは追加予告されています。

'D' アイテムは削除予告されています。

'M' アイテムは修正されました。

'R' アイテムは作業コピー中で置き換えられました。

'C' (属性ではなく) アイテムの内容はリポジトリから 受け取った更新によって衝突した状態にあります。

'X' アイテムは何かの外部定義に関係しています。

'I' アイテムは無視されている属性です (たとえば、 `svn:ignore` のような)

'?' アイテムはバージョン管理下にありません。

'!' アイテムは失われました (これはたとえば、 **svn** を使わずにファイルを削除したり移動した場合に 起こります)。また、これはディレクトリが不完全であることを示しています (チェックアウトや更新が中断された、など)。

' ' アイテムはある種類のオブジェクト (ファイル、ディレクトリ、リンク) としてバージョン管理されていますが、 別の種類のオブジェクトで置き換えられてしまいました。

二番目のコラムはファイルやディレクトリの 属性の状態を示します。

' ' 修正はありません。

'M' このアイテムの属性は修正されました。

'C' このアイテムの属性はリポジトリから受け取った属性更新によって 衝突した状態にあります。

三番目のコラムは作業コピーがロックされている場合にだけ使われます。

' ' アイテムはロックされていません。

'L' アイテムはロックされています。

四番目のコラムはアイテムが追加予告されている場合にのみ 使われます。

' ' コミット待ちの予告はありません。

'+' コミット待ちの予告があります。

五番目のコラムはアイテムが親に対して相対的に切り替えられた ときにだけ使われます。 (項 4.6 を見てください)。

' ' アイテムは親ディレクトリの子供です。

'S' アイテムは切り替わっています。

六番目のコラムにはロック情報が表示されます。

' ' `--show-updates` が指定された場合ファイルはロックされて いません。`--show-updates` が指定されない 場合、これは単にこの作業コピー中でファイルがロックされていないことを示すだけです。

K ファイルはこの作業コピー中でロックされています。

O ファイルは他のユーザあるいは他の作業コピーによってロックされています。 これは `--show-updates` が指定された場合にだけ現れます。

T ファイルはこの作業コピー中でロックされていましたが、それは 「横取りされ」、無効となりました。このファイルは現在 リポジトリ中でロックされています。これは `--show-updates` が指定された時だけ現れます。

B ファイルはこの作業コピーでロックされていましたが、それは 「破壊され」て無効となりました。このファイルはもうロック状態には ありません。`--show-updates` が指定された時だけ現れます。

最新状態に関係した情報が 7 番目のコラムに表示されます (`--show-updates` スイッチを渡した場合)。

' ' 作業コピーのアイテムは最新です。

'*' サーバにはアイテムのもっと新しいバージョンが 存在します。

残りのフィールドは空白で区切られた可変長です。 --show-updates か--verbose を指定した場合は作業リビジョンが次のフィールドになります。

--verbose スイッチを指定すると最後のコミットリビジョン とそれをした人が次に表示されます。

作業コピーのパスは常に最後のフィールドになるので、空白を含むことが できます。

別名

stat, st

変更対象

なし

リポジトリへのアクセス

--show-updates が指定された場合に のみ発生する

スイッチ

```
--show-updates (-u)
--verbose (-v)
--non-recursive (-N)
--quiet (-q)
--no-ignore
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

例

作業コピーにした変更点を調べるための一番簡単な方法:

```
$ svn status wc
 M     wc/bar.c
A  +   wc/qax.c
```

作業コピー中、どのファイルが最新でないかを知りたい場合 は`--show-updates` スイッチを指定してください (これは 作業コピーの内容を決して変更しません)。 最後に自分の作業コピーを更新してからリポジトリの `wc/foo.c` に変更があったときには 次のようになります:

```
$ svn status --show-updates wc
 M           965    wc/bar.c
       *     965    wc/foo.c
A  +         965    wc/qax.c
Status against revision:    981
```

> 注意
>
> 
>
> `--show-updates` は、 最新ではないアイテムの隣にアスタリスクを置くだけ です (つまり、**svn update** を実行したとすればリポジトリから の情報で更新されるであろうアイテムの前にのみ、という意味です)。 `--show-updates` は、アイテムのリポジトリバージョンを反映 した状態一覧を表示するわけではありません。

最後に、status サブコマンドで一番たくさんの情報を得るには:

```
$ 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
```

svn status のもっとたくさんの例は 項 3.6.3.1 にあります。

名前

svn switch — 作業コピーを別の URL に更新します。

名前

svn switch — 作業コピーを別の URL に更新します。

用法

```
svn switch URL [PATH]
```

```
switch --relocate FROM TO [PATH...]
```

説明

このサブコマンドは自分の作業コピーを新しい URL に更新し、複製を作ります。 — そうする必要はありませんが、普通は その URL は元になる作業コピーと共通の祖先を持ちます。 これが Subversion で作業コピーを別のブランチに移動させる方法 です。 項 4.6 により詳しい説明があります。

別名

sw

変更対象

作業コピー

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--diff3-cmd CMD
--relocate
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

例

いま `vendors-with-fix` から分岐した `vendors` というディレクトリの 内部にいて、そのブランチの作業コピーに移りたいときには:

```
$ 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.
```

そして、元に戻りたいときには、最初に作業コピーをチェックアウトした リポジトリの場所を URL として指定するだけです:

```
$ 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.
```

ティップ

作業コピー全体を切り替えたくない場合、その一部だけをブランチに 切り替えることもできます。

管理者はときどき、リポジトリの「格納場所」を変更したい と思うこともあります — 言い換えると、リポジトリの内容に変更はな くても、リポジトリの根元のディレクトリにアクセスするための URL を変え たいと思うことがあります。例えばホスト名が変更されたり、URL スキーマが 変更されたり、リポジトリパス URL の先頭部分のどこかが変更されるような場合もあるでしょう。 新しい作業コピーをチェックアウトするよりも、**svn switch** を使って作業コピーの中に記録されているすべての URL の 先頭部分を一括して「書き換えて」やるほうが良いでしょう。この置換には `--relocate` オプションを使ってください。ファイルには一切 修正を加えませんし、このコマンドでリポジトリにアクセスすることもありま せん。これは、Perl のスクリプトなどを使って、作業コピーの `.svn/`配下に対して **s/OldRoot/NewRoot/**コマンドを実行するのに似ています。

```
$ 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.
```

警告

`--relocate` オプションの利用には注意してください。 引数を間違って入力すると作業コピー中に意味のない URL 情報ができて しまい、作業コピー全体が利用不能になり、修復にはちょっとしたコツが 必要になってしまいます。また `--relocate` を利用すべきか、すべきでないかをはっきりと完全に理解していることも 重要です。以下が原則です:

- 作業コピーがリポジトリの 中にある新しいディレクトリ に移る場合には、単に **svn switch** を使う。
- 作業コピーは同じリポジトリのディレクトリを指しているが、リポジトリ自身の 場所が移った場合には **svn switch –relocate** を使う。

名前

svn unlock — 作業コピーパスまたは URL をアンロックします

名前

svn unlock — 作業コピーパスまたは URL をアンロックします

用法

```
svn unlock TARGET...
```

説明

TARGET それぞれをアンロックします。 *TARGET* のどれかが他のユーザによってロックされて いたり作業コピー中に正常なロックトークンが存在しない場合は警告を出して 残りの *TARGET* のアンロックを続けます。他のユーザまたは作業コピーに属するロックを破壊するには `--force` を使ってください。

別名

なし

変更対象

作業コピー、リポジトリ

リポジトリへのアクセス

発生する

スイッチ

```
--targets ARG
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
--force
```

例

作業コピー中の二つのファイルをアンロックする:

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

現在別のユーザによってロックされている作業コピー中のファイル をアンロックする:

```
$ svn unlock tree.jpg
svn: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

作業コピーには無関係にファイルをアンロックする:

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

より詳細は 項 7.4 を見てください。

名前

svn update — 作業コピーの更新。

名前

svn update — 作業コピーの更新。

用法

```
svn update [PATH...]
```

説明

svn update は、リポジトリの修正を 作業コピーに反映します。リビジョンを指定しなければ、HEAD リビジョンの最新 の内容が反映されます。そうでなければ、`--revision` スイッチ で指定されたリビジョンに作業コピーを同期します。同期処理の一部として、 **svn update** は作業コピーに存在する未完了のロックを 取り除きます。

更新されるアイテムごとに、どのような動作を起こしたかを示す文字で始まる 行が表示されます。この文字は以下のような意味です:

A 追加

D 削除

U 更新

C 衝突

G マージ

最初のコラムの文字は実際のファイルの更新を示しますが、 ファイルの属性の更新状況は二番目のコラムで示します。

別名

up

変更対象

作業コピー

リポジトリへのアクセス

発生する

スイッチ

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
--diff3-cmd CMD
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

例

最後の更新後に起きたリポジトリの修正を取り込みます:

```
$ svn update
A  newdir/toggle.c
A  newdir/disclose.c
A  newdir/launch.c
D  newdir/README
Updated to revision 32.
```

もっと古いリビジョンで作業コピーを更新することもできます。 (Subversion は CVS のような「張り付き」ファイル の概念を持ちません。 付録 A を見てください):

```
$ 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.
```

ティップ

一つのファイルの古いバージョンを調べたいときには **svn cat** を使いたくなるかも知れません。

9.3 svnadmin

svnadmin は Subversion リポジトリを監視したり修復したりするための管理ツールです。 詳しくは項 5.4.1.2 を見てください。

svnadmin は直接リポジトリに対するアクセスに より動作するので (そして、リポジトリが存在するマシン上でのみ利用することが できるので)、URL ではなく、パス名によってリポジトリを参照します。

9.3.1 svnadmin スイッチ

`--bdb-log-keep` (Berkeley DB 固有) データベースログファイルの 自動ログ削除機能を無効にします。

`--bdb-txn-nosync` (Berkeley DB 固有) データベーストランザクション コミット時に fsync をしません。

`--bypass-hooks` リポジトリフックシステムを迂回します。

`--clean-logs` 利用していない Berkeley DB ログを削除します。

`--force-uuid` デフォルトではリポジトリに対して既に存在しているリビジョンを ロードする場合、**svnadmin** はダンプストリーム にある `UUID` を無視します。このスイッチは リポジトリの `UUID` をストリームからの `UUID` に設定します。

`--ignore-uuid` デフォルトでは空のリポジトリをロードする場合 **svnadmin** はダンプストリームからの `UUID` を利用します。 このスイッチはダンプストリームからの UUID を無視します。

`--incremental` リビジョンの全体をダンプするのではなく、以前のリビジョンに対する差分として ダンプします。

`--parent-dir` *`DIR`* ダンプファイルをロードするときルートパスとして`/` のかわりに *`DIR`* を使います。

`--revision (-r) ARG` 操作対象となる特定のリビジョンを指定します。

`--quiet` 通常の進行状況を表示しません — エラーのみ表示します。

`--use-post-commit-hook` ダンプファイルをロードする時、新規にロードされたそれぞれのリビジョンの完了 処理の後でリポジトリの post-commit フックを実行します。

`--use-pre-commit-hook` ダンプファイルをロードする時、新規にロードされたそれぞれのリビジョンの完了 処理の前にリポジトリの pre-commit フックを実行します。フック処理に失敗した 場合、コミットは中断されてロード処理は終了します。

9.3.2 **svnadmin** サブコマンド

名前

svnadmin create — 新規の空のリポジトリを作ります。

名前

svnadmin create — 新規の空のリポジトリを作ります。

用法

```
svnadmin create REPOS_PATH
```

説明

指定したパスに新規に空のリポジトリを作ります。 パスが存在しなければ自動的に作られます。 [*2] Subversion 1.2 では **svnadmin** はデフォルトで `fsfs` ファイルシステムのバックグラウンドで 新しいリポジトリを作成します。

スイッチ

```
--bdb-txn-nosync
--bdb-log-keep
--config-dir DIR
--fs-type TYPE
```

[*2] **svnadmin** はローカル パスに対してのみ 動作し、*URL* には働かないことに注意 してください。

例

新しくリポジトリを作るには単に以下のようにします:

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

Subversion 1.0 では Berkeley DB リポジトリが常に作られます。 Subversion 1.1 では Berkeley DB リポジトリがデフォルトですが `--fs-type` オプションで FSFS リポジトリを作る こともできます:

```
$ svnadmin create /usr/local/svn/repos --fs-type fsfs
```

名前

svnadmin deltify — あるリビジョン範囲にある変更パスを差分化します

名前

svnadmin deltify — あるリビジョン範囲にある変更パスを差分化します

用法

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

説明

svnadmin deltify は歴史的な理由で 1.0.x のみに存在します。このコマンドは時代遅れであり、もう不 要です。

これは Subversion がリポジトリ中の圧縮方法についてシス テム管理者に対してより大きな制御権限を与えるためのものでした。しかし 非常にわずかな利益のために多くの複雑さを導入する ことになることがわかったため、この「機能」は廃止されました。

スイッチ

```
--revision (-r)
--quiet
```

名前

svnadmin dump — ファイルシステムの内容を標準出力にダンプします。

名前

svnadmin dump — ファイルシステムの内容を標準出力にダンプします。

用法

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

説明

ファイルシステムの内容を「dumpfile」可搬可能形式でダンプし、進行状況 を標準出力に表示します。 リビジョン *LOWER* から、 *UPPER* までをダンプします。リビジョンが指定 されなければすべてのリビジョンツリーをダンプします。 *LOWER* だけが 指定された場合は一つのリビジョンツリーのみダンプします。 実際の使い方については項 5.4.5 を見てください。

デフォルトでは、Subversion ダンプファイルの内容は 以下のようになります。まず要求されたリビジョン範囲の最初のリビジョン の内容が出力されます。ここにはそのリビジョンに含まれるすべてのファイルと ディレクトリが、すべて一度にリポジトリに追加されたかのような形式に なります。その後に、(要求されたリビジョン範囲内の) 引き続くすべての リビジョンの内容が続きます。ここにはそれぞれのリビジョンで修正された ファイルとディレクトリの情報だけが含まれます。修正されたものがファイルの 場合には、そのファイルの完全な内容と、属性が出力されます。 ディレクトリの場合には、そのすべての属性が出力されます。

ダンプファイル生成する際に有用な二つのオプションがあります。 一つ目は `--incremental` オプションで、これを指定すると ダンプファイル出力中の最初のリビジョンが、そのリビジョンを示す新しいツリー 全体ではなく、そのリビジョンで修正のあったファイルとディレクトリのみに なります。それ以降のリビジョンについてもまったく同様です。これは ダンプ元のリポジトリに存在しているファイルとディレクトリをすでに含んで いる別のリポジトリにロードするためのダンプファイルを生成する場合に便利です。

もう一つの有用なオプションは `--deltas` です。 このスイッチは、ファイルの内容と属性についての完全なテキスト表現を出力 するかわりに、それぞれのリビジョンの直前のリビジョンとの差分だけを 出力するよう **svnadmin dump** に指示します。これは **svnadmin dump** が生成するダンプファイルのサイズを (場合によっては劇的に) 減らします。欠点としてはこのオプション — 差分化 ダンプファイルの指示 — は生成にあたって CPU により大きな負荷が かかること、**svndumpfilter** によって処理できなくなって しまうこと、そして、サードパーティー製の **gzip** や **bzip2** を使う場合には、差分化しないものに対するほど圧縮されない傾向があること、 があります。

スイッチ

```
--revision (-r)
--incremental
```

```
--quiet
--deltas
```

例

リポジトリ全体のダンプ:

```
$ 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
…
```

リポジトリの一つのトランザクションの 差分ダンプ:

```
$ svnadmin dump /usr/local/svn/repos -r 21 --incremental
* Dumped revision 21.
SVN-fs-dump-format-version: 1
Revision-number: 21
Prop-content-length: 101
Content-length: 101
…
```

名前

svnadmin help

名前

svnadmin help

用法

```
svnadmin help [SUBCOMMAND...]
```

説明

このサブコマンドはネットワーク接続がうまくいかなくなって この本のコピーが読めなくなったりしてどこかに迷い込んでしまった ようなときに役立ちます。

別名

?, h

名前

svnadmin hotcopy — リポジトリのホットコピーを作ります。

名前

svnadmin hotcopy — リポジトリのホットコピーを作ります。

用法

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

説明

このサブコマンドはすべてのフック、設定ファイル、そしてもちろんデータベース ファイルを含む、リポジトリの完全な「ホット」バックアップを とります。 `--clean-logs` スイッチを渡すと **svnadmin** はリポジトリのホットコピー作成後、もとのリポジトリから利用していない Berkeley DB ログ を削除します。このコマンドは常に利用可能で、他のプロセスがリポジトリを利 用しているかどうかにかかわらずリポジトリのコピーを安全に作ることができ ます。

スイッチ

```
--clean-logs
```

名前

svnadmin list-dblogs — 指定した Subversion リポジトリにどのような Berkeley DB ログファイルが あるか問い合わせます。(bdb バックエンドを利用しているリポジトリの場合だけです)。

名前

svnadmin list-dblogs — 指定した Subversion リポジトリにどのような Berkeley DB ログファイルが あるか問い合わせます。(bdb バックエンドを利用しているリポジトリの場合だけです)。

用法

```
svnadmin list-dblogs REPOS_PATH
```

説明

Berkeley DB は障害から復旧できるようにリポジトリの すべての変更点のログをとります。 DB_LOG_AUTOREMOVE を有効にしない限りログファイルは蓄積されていきますが、そのほとんどは それ以上利用されることはなく、ディスク領域を確保するために削除可能な ものです。詳細は項 5.4.3 を見てください。

名前

svnadmin list-unused-dblogs — Berkeley DB にどのログファイルが安全に削除可能かを問い合わせます。 (bdb バックエンドを使っているリポジトリの場合だけです)。

名前

svnadmin list-unused-dblogs — Berkeley DB にどのログファイルが安全に削除可能かを問い合わせます。(bdb バックエンドを使っているリポジトリの場合だけです)。

用法

```
svnadmin list-unused-dblogs REPOS_PATH
```

説明

Berkeley DB は障害から復旧できるようにリポジトリの すべての変更点のログをとります。 DB_LOG_AUTOREMOVE を有効にしない限りログファイルは蓄積されていきますが、そのほとんどは それ以上利用されることはなく、ディスク領域を確保するために削除可能な ものです。詳細は項 5.4.3 を見てください。

例

リポジトリからすべての未使用ログを削除するには:

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

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## disk space reclaimed!
```

名前

svnadmin load — 標準入力から「ダンプファイル」形式 のデータを読み出します。

名前

svnadmin load — 標準入力から「ダンプファイル」形式 のデータを読み出します。

用法

```
svnadmin load REPOS_PATH
```

説明

標準入力から「ダンプ形式」 のデータを読み出し、リポジトリのファイルシステムに新しい リビジョンをコミットします。進行状況は標準出力に表示されます。

スイッチ

```
--quiet (-q)
--ignore-uuid
--force-uuid
--use-pre-commit-hook
--use-post-commit-hook
--parent-dir
```

例

これはバックアップファイルからリポジトリをロードする最初の 部分です (もちろんあらかじめ **svnadmin dump** でダンプして おいたものです):

```
$ 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.
…
```

あるいはサブディレクトリにロードしたい場合は:

```
$ svnadmin load --parent-dir new/subdir/for/project /usr/local/svn/restored < repos-ba
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

名前

svnadmin lslocks — すべてのロック状況を表示します。

名前

svnadmin lslocks — すべてのロック状況を表示します。

用法

```
svnadmin lslocks REPOS_PATH
```

説明

リポジトリ中のすべてのロックに関する情報を表示します。

スイッチ

なし

例

/svn/repos にあるリポジトリ中でロックされている 唯一のファイルを一覧表示しています。

```
$ svnadmin lslocks /svn/repos
Path: /tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

名前

svnadmin lstxns — コミットされていないすべてのトランザクションの名前の表示。

名前

svnadmin lstxns — コミットされていないすべてのトランザクションの名前の表示。

用法

```
svnadmin lstxns REPOS_PATH
```

説明

コミットされていないすべてのトランザクションの名前を表示します。 コミットされていないトランザクションはどのように作られ、どのように 扱うべきかについては項 5.4.2 を見て ください。

例

リポジトリ中の未解決のトランザクション一覧の表示。

```
$ svnadmin lstxns /usr/local/svn/repos/
1w
1x
```

名前

svnadmin recover — リポジトリデータベースの一貫した状態への復帰。 (bdb バックエンドを使っているリポジトリの場合だけです)。 さらに repos/conf/passwd が存在しなければデフォルトの パスワードファイルも作ります。

名前

svnadmin recover — リポジトリデータベースの一貫した状態への復帰。 (bdb バックエンドを使っているリポジトリの場合だけです)。 さらに repos/conf/passwd が存在しなければデフォルトの パスワードファイルも作ります。

用法

```
svnadmin recover REPOS_PATH
```

説明

リポジトリは修復される必要があるというエラーメッセージを受け取った ときにはこのコマンドを実行してください。

スイッチ

```
--wait
```

例

ハングしてしまったリポジトリの修復:

```
$ 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.
```

データベースの修復にはリポジトリを排他的にロックする必要があります。他 のプロセスがリポジトリにアクセスしている場合、**svnadmin recover** はエラーになります:

```
$ 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?

$
```

しかし `--wait` オプションを指定すると **svnadmin recover** は他のプロセスの接続が切れるまで待ちつづけます:

```
$ svnadmin recover /usr/local/svn/repos --wait
Waiting on repository lock; perhaps another process has it open?

### time goes by...

Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

名前

svnadmin rmlocks — リポジトリにある一つ以上のロックを無条件に取り除きます。

名前

svnadmin rmlocks — リポジトリにある一つ以上のロックを無条件に取り除きます。

用法

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

説明

LOCKED_PATH ごとにロックを取り除きます。

スイッチ

なし

例

リポジトリ/svn/repos にある tree.jpg と house.jpg に 設定されているロックを削除しています。

```
$ svnadmin rmlocks /svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

名前

svnadmin rmtxns — リポジトリからトランザクションを削除します。

名前

svnadmin rmtxns — リポジトリからトランザクションを削除します。

用法

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

説明

未解決のトランザクションをリポジトリから削除します。 これは、項 5.4.2 で詳しく触れられています。

スイッチ

```
--quiet (-q)
```

例

名前の付いたトランザクションの削除:

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

幸運なことに、**lstxns** の出力は **rmtxns** の入力と同じくらいうまく動きます:

```
$ svnadmin rmtxns /usr/local/svn/repos/  ‘svnadmin lstxns /usr/local/svn/repos/‘
```

これはリポジトリからすべてのコミットされていないトランザクション を削除します。

名前

svnadmin setlog — リビジョンにログメッセージを設定します。

名前

svnadmin setlog — リビジョンにログメッセージを設定します。

用法

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

説明

FILE の内容をリビジョン REVISION のログメッセージとして設定します。

これは、**svn propset –revprop** を使ってリビジョン上に `svn:log` 属性を設定するのと似ていますが、`--bypass-hooks` を使ってコミット前後のフックの実行を避ける ことができます。これは pre-revprop-change フック中でリビジョン属性の修正 が有効ではない場合に便利です。

警告

リビジョン属性はバージョン管理下にはないのでこのコマンドは以前の ログメッセージを完全に上書きしてしまいます。

スイッチ

```
--revision (-r) ARG
--bypass-hooks
```

例

ファイル `msg` の内容をリビジョン 19 のログメッセージに設定します:

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

名前

svnadmin verify — リポジトリに保管されているデータを検証します。

名前

svnadmin verify — リポジトリに保管されているデータを検証します。

用法

```
svnadmin verify REPOS_PATH
```

説明

リポジトリの完全性を検証したい場合にこのコマンドを実行してください。 これは基本的には内部的にすべてのリビジョンをダンプしては出力を捨てる ことによって繰り返し実行されます。

例

固まってしまったリポジトリの検証:

```
$ svnadmin verify /usr/local/svn/repos/
* Verified revision 1729.
```

9.4 svnlook

svnlook は Subversion リポジトリの 別の部分を調べるためのコマンドラインツールです。このコマンドはリポジトリには何の変更も加えません — 単に「調べる」ため に利用されます。**svnlook** はリポジトリフックで利用されるのが 典型的ですが、リポジトリ管理者は診断の目的にも利用できることに気づくかも 知れません。

svnlook は直接リポジトリにアクセスする形で実行される ので (そして、それが理由でリポジトリの存在するマシン上でのみ利用することが できるのですが)、URL ではなく、パス名称によってリポジトリを参照します。

リビジョンやトランザクションが指定されなければ、 **svnlook** はリポジトリの最新リビジョンを使います。

9.4.1 svnlook スイッチ

svnlook 中のスイッチは **svn** や **svnadmin** などと一緒で グローバルに働きますが、ほとんどのスイッチは一

つのサブコマンドに しか効果がありません。それは **svnlook** の機能の 有効範囲が (意図的に) 限られているからです。

`--no-diff-deleted` **svnlook** が削除されたファイルの差異を表示しないようにします。 トランザクション/リビジョン中でファイルが削除されたときのデフォルト 動作は、そのファイルが空のファイルとして残っているときと同じに なります。

`--revision (-r)` 調べたい特定のリビジョン番号を指定します。

`--revprop` ファイルまたはディレクトリを特定した Subversion の属性のかわりに リビジョン属性に対して処理をします。このスイッチを使う場合、 `--revision(-r)` スイッチも一緒に 指定してやる必要があります。バージョン化されない属性についての 詳細は 項 5.2.2 を見て ください。

`--transaction (-t)` 調べたい特定のトランザクション番号を指定します。

`--show-ids` ファイルシステムツリー中の、それぞれのパスごとのファイルシステム ノードリビジョン番号を表示します。

9.4.2 svnlook

名前

svnlook author — 処理した人の表示。

名前

svnlook author — 処理した人の表示。

用法

```
svnlook author REPOS_PATH
```

説明

リポジトリのリビジョンやトランザクションを実行した 人を表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

svnlook author は便利ですが、 それほど面白いコマンドではありません:

```
$ svnlook author -r 40 /usr/local/svn/repos
sally
```

名前

svnlook cat — ファイルの内容を表示します

名前

svnlook cat — ファイルの内容を表示します

用法

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

説明

ファイルの内容を表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

以下は`/trunk/README` にあるトランザクション `ax8` 中のファイルの内容を表示しています:

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README

               Subversion, a version control system.
               ====================================
```

```
$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

     I. A FEW POINTERS
    II. DOCUMENTATION
   III. PARTICIPATING IN THE SUBVERSION COMMUNITY
…
```

名前

svnlook changed — 変更されたパスを表示します。

名前

svnlook changed — 変更されたパスを表示します。

用法

```
svnlook changed REPOS_PATH
```

説明

「svn update-style」 の第一ステータス文字と同じように 特定のリビジョンやトランザクションで変更されたパスを表示します:

'A' アイテムはリポジトリに追加されました

'D' アイテムはリポジトリから削除されました

'U' ファイル内容が変化しました

'_U' アイテムの属性が変化しました

'UU' ファイルの内容と属性が変化しました

ディレクトリパスの最後には '/' がつくのでファイルと ディレクトリは区別することができます。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

テストリポジトリのリビジョン 99 で修正したすべてのファイルの 一覧を表示します:

```
$ svnlook changed -r 39 /usr/local/svn/repos
A   trunk/vendors/deli/
A   trunk/vendors/deli/chips.txt
A   trunk/vendors/deli/sandwich.txt
A   trunk/vendors/deli/pickle.txt
U   trunk/vendors/baker/bagel.txt
_U  trunk/vendors/baker/croissant.txt
UU  trunk/vendors/baker/pretzel.txt
D   trunk/vendors/baker/baguette.txt
```

名前

svnlook date — 日付を表示します。

名前

svnlook date — 日付を表示します。

用法

```
svnlook date REPOS_PATH
```

説明

リポジトリ中の特定リビジョンやトランザクションの日付を表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

これは、テストリポジトリのリビジョン 40 の日付を表示します:

```
$ svnlook date -r 40 /tmp/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

名前

svnlook diff — 変更されたファイル、ディレクトリの差分を表示します。

名前

svnlook diff — 変更されたファイル、ディレクトリの差分を表示します。

用法

```
svnlook diff REPOS_PATH
```

説明

リポジトリ中で変更されたファイル、属性の差分を GNU 形式で表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
--no-diff-added
--no-diff-deleted
```

例

これは、新しく追加されたファイル (空のファイル)、削除されたファイル、コピーされた ファイルを表示します:

```
$ 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
===================================================================================
```

ファイルが非テキスト的な `svn:mime-type` 属性を持っている場合、差分は明示的には表 示されません。

名前

svnlook dirs-changed — 変更のあったディレクトリを表示します。

名前

svnlook dirs-changed — 変更のあったディレクトリを表示します。

用法

```
svnlook dirs-changed REPOS_PATH
```

説明

(属性を編集によって) それ自身に変更があったか、 その子供のファイルに変更があったディレクトリを表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

私たちのサンプルリポジトリ中のリビジョン 40 で修正された ディレクトリを表示します:

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos
trunk/vendors/deli/
```

名前

svnlook help

名前

svnlook help

用法

```
Also svnlook -h and svnlook -?.
```

説明

svnlook のヘルプを表示します。 **svn help** のヘルプと同じで、何かがわからなくなったときには常に 役に立ちます。

別名

?, h

名前

svnlook history — リポジトリ中のパスの履歴に関する情報を表示します (あるいはパスが指定されなかった場合にはルートディレクトリになります)。

名前

svnlook history — リポジトリ中のパスの履歴に関する情報を表示します (あるいはパスが指定されなかった場合にはルートディレクトリになります)。

用法

```
svnlook history REPOS_PATH [PATH_IN_REPOS]
```

説明

リポジトリ中のパスの履歴に関する情報を表示します (あるいはパスが指定されなかった場合にはルートディレクトリになります)。

スイッチ

```
--revision (-r)
--show-ids
```

例

以下は例として作ったリポジトリのリビジョン 20 で見た パス `/tags/1.0` の履歴表示です。

```
$ svnlook history -r 20 /usr/local/svn/repos /tags/1.0 --show-ids
REVISION   PATH <ID>
--------   ---------
      19   /tags/1.0 <1.2.12>
      17   /branches/1.0-rc2 <1.1.10>
      16   /branches/1.0-rc2 <1.1.x>
      14   /trunk <1.0.q>
      13   /trunk <1.0.o>
      11   /trunk <1.0.k>
       9   /trunk <1.0.g>
       8   /trunk <1.0.e>
```

```
7   /trunk <1.0.b>
6   /trunk <1.0.9>
5   /trunk <1.0.7>
4   /trunk <1.0.6>
2   /trunk <1.0.3>
1   /trunk <1.0.2>
```

名前

svnlook info — 作業者、日付、ログメッセージの大きさ、ログメッセージを表示します。

名前

svnlook info — 作業者、日付、ログメッセージの大きさ、ログメッセージを表示します。

用法

```
svnlook info REPOS_PATH
```

説明

作業者、日付、ログメッセージの大きさ、ログメッセージを表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

これはサンプルリポジトリのリビジョン 40 に対する info の出力です。

```
$ svnlook info -r 40 /usr/local/svn/repos
sally
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
15
Rearrange lunch.
```

名前

svnlook lock — リポジトリ中の特定のパスがロックされている場合、それについての 情報を表示します。

名前

svnlook lock — リポジトリ中の特定のパスがロックされている場合、それについての 情報を表示します。

用法

```
svnlook lock REPOS_PATH PATH_IN_REPOS
```

説明

PATH_IN_REPOS 上のロックについて取得できるすべての 情報を表示します。 *PATH_IN_REPOS* がロックされて いなければ何も表示しません。

スイッチ

なし

例

`tree.jpg` ファイル上のロックに関する 情報を表示しています。

```
$ svnlook lock /svn/repos tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

名前

svnlook log — ログメッセージを表示します。

名前

svnlook log — ログメッセージを表示します。

用法

```
svnlook log REPOS_PATH
```

説明

ログメッセージを表示します。

スイッチ

```
--revision (-r)
--transaction (-t)
```

例

これはサンプルリポジトリのリビジョン 40 のログ表示です:

```
$ svnlook log /tmp/repos/
Rearrange lunch.
```

名前

svnlook propget — リポジトリ中のパス上に設定された属性値の生の値を表示します。

名前

svnlook propget — リポジトリ中のパス上に設定された属性値の生の値を表示します。

用法

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

説明

リポジトリ中のパス上に設定された属性値を一覧表示します。

別名

pg, pget

スイッチ

```
--revision (-r)
--transaction (-t)
--revprop
```

例

以下では HEAD リビジョンにある /trunk/sandwich ファイルの「seasonings」属性値を表示しています:

```
$ svnlook pg /usr/local/svn/repos seasonings /trunk/sandwich
mustard
```

名前

svnlook proplist — バージョン化されたファイルとディレクトリの属性の 名前と値を表示します。

名前

svnlook proplist — バージョン化されたファイルとディレクトリの属性の 名前と値を表示します。

用法

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

説明

リポジトリ中のパスの属性を一覧表示します。--verbose を使って属性値も表示できます。

別名

pl, plist

スイッチ

```
--revision (-r)
--transaction (-t)
--verbose (-v)
--revprop
```

例

これは HEAD リビジョンにあるファイル/trunk/README に設定された属性名を表示しています:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README
  original-author
  svn:mime-type
```

これは前の例と同じコマンドですが、今回は属性値もいっしょに表示 しています:

```
$ svnlook --verbose proplist /usr/local/svn/repos /trunk/README
  original-author : fitz
  svn:mime-type : text/plain
```

名前

svnlook tree — ツリーを表示します。

名前

svnlook tree — ツリーを表示します。

用法

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

説明

PATH_IN_REPOS から始まるツリーを表示します。(*PATH_IN_REPOS* の指定がない場合にはルートから始まるツリーを表示します。)。オプションで ノードリビジョン ID を表示させることもできます。

スイッチ

```
--revision (-r)
--transaction (-t)
--show-ids
```

例

これは、(ノード番号付きで) サンプルリポジトリのリビジョン 40 のツリーを 表示したものです:

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids
/ <0.0.2j>
 trunk/ <p.0.2j>
  vendors/ <q.0.2j>
   deli/ <1g.0.2j>
    egg.txt <1i.e.2j>
    soda.txt <1k.0.2j>
    sandwich.txt <1j.0.2j>
```

名前

svnlook uuid — リポジトリの UUID を表示します。

名前

svnlook uuid — リポジトリの UUID を表示します。

用法

```
svnlook uuid REPOS_PATH
```

説明

リポジトリの `UUID` を表示します。UUID はリポジトリ の *u*niversal *u*nique *id*entifier(訳者:生成するたびに常に一意であることが保証されるような性質をもった番号のことです。 この一意性は特定のマシン内に限定されているわけではなく、異なるマシン 間でも一意になるような性質があります) のことです。 Subversion クライアントはあるリポジトリと別のリポジトリを区別する のにこの識別子を使います。

例

```
$ svnlook uuid /usr/local/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

名前

svnlook youngest — 最新のリビジョン番号を表示します。

名前

svnlook youngest — 最新のリビジョン番号を表示します。

用法

```
svnlook youngest REPOS_PATH
```

説明

リポジトリにある最新のリビジョン番号を表示します。

例

これは、サンプルリポジトリの最新のリビジョン番号を表示しています:

```
$ svnlook youngest /tmp/repos/
42
```

9.5 svnserve

svnserve は svn ネットワー クプロトコルによって Subversion リポジトリにアクセスすることを可 能にするものです。独立したサーバプロセスとしても起動できますし、 **inetd**、**xinetd** あるいは **sshd** のような別のプロセスを使って起動すること もできます。

クライアントが URL を送ることでリポジトリを選択すると **svnserve** はリポジトリディレクトリにある `conf/svnserve.conf` という名前のファイルを読んで どのような認証用データベースを使い、またどのような認証方法を使うかの ような設定をリポジトリごとに決めます。 `svnserve.conf` ファイルの詳細は項 6.4 を見てください。

9.5.1 svnserve スイッチ

既に説明してきたコマンドとは違い、 **svnserve** はサブコマンドがありません — **svnserve** はスイッチによって排他的に制御されます。

`--daemon (-d)` **svnserve** がデーモンモードで実行 するようにします。**svnserve** 自身がバックグラウンドで実行され、svn ポート (通常は 3690) 上に TCP/IP 接続を受け付け、接続を 用意します。

`--listen-port=PORT` デーモンモード時に *PORT* で 待ち受けるように指定します。

`--listen-host=HOST` **svnserve** が *HOST* で指定されるインターフェー ス上で待ち受けるように指定します。ホスト名か IP アドレスの いずれかで指定できます。

`--foreground` `-d` と一緒に利用すると **svnserve** がフォアグラウントで待機するよ うにできます。このスイッチは主にデバッグ時に利用されます。

`--inetd (-i)` **svnserve** が標準入力/標準出力のファ イル記述子を利用するように指定します。**inetd** と共に利用する場合に適しています。

`--help (-h)` 利用方法の概略を表示し抜けます。

`--version` バージョン情報と、利用可能なリポジトリバックエンドモジュール一覧を 表示してから終了します。

`--root=ROOT (-r=ROOT)` **svnserve** によって提供されるリポジ トリの仮想的なルートを設定します。クライアントによって指定される URL 中のパス名はこのルートに相対的なものと解釈され、その外にアクセスするこ とを許しません。

`--tunnel (-t)` **svnserve** がトンネルモードで実行す るように指定します。**inetd** と (標準入出力を使って接続 するという意味で) 同様ですが、現在の uid に対応したユーザ名であらかじめ 認証されていると考えます。このフラグは **ssh** のような トンネル用エージェント越しに実行するときクライアントによって指定される ものです。

`--tunnel-user NAME` `--tunnel` スイッチと一緒に使うと svnserve は svnserve プロセスの UID のかわりに *NAME* が認証されたユーザであると見なします。SSH 越しに 一つのシステムアカウントを共有しながら、コミットの主体としては分離して 管理したいようなユーザ間では便利です。

`--threads (-T)` デーモンモードで実行される場合、接続ごとに **svnserve** がプロセスを起動するかわりにスレッドを起動するように指定します。 **svnserve** 自身は やはり起動時にバックグラウンドに常駐します。

`--listen-once (-X)` **svnserve** が svn ポート上の接続を ひとつだけ受け入れ、サービスを提供後抜けるようにします。この オプションは主にデバッグ時に利用されます。

9.6 svnversion

名前

svnversion — 作業コピーのローカルリビジョンについて簡単に表示します。

名前

svnversion — 作業コピーのローカルリビジョンについて簡単に表示します。

用法

```
svnversion [OPTIONS] WC_PATH [TRAIL_URL]
```

説明

svnversion は作業コピーの混合リビジョン状況を表示します。 リビジョン番号またはリビジョン範囲が標準出力に出力されます。

TRAIL_URL を指定した場合は、 *WC_PATH* 自身がスイッチされた場合の URL の終わりの部分として利用されます。(*WC_PATH* 内部でのスイッチの検出は *TRAIL_URL* にはよりません)。

スイッチ

svnserve と同様 **svnversion** に サブコマンドはありません。スイッチがあるだけです。

`--no-newline (-n)` 通常は出力する末尾の改行文字を省略します。

`--committed (-c)` 現在の (つまりローカルで一番新しい) リビジョンのかわりに 最終的な更新があったリビジョンを使います。

`--help (-h)` 簡単なヘルプを表示します。

--version **svnversion** のバージョンを表示してから プログラムを正常終了します。

例

作業コピーが完全に単一リビジョン状態にある場合 (たとえば update コマンドの 直後など)、リビジョンは以下のように表示されます:

```
$ svnversion .
4168
```

TRAIL_URL をつけて自分が思っている場所から作業コピーが スイッチされていないことを示すこともできます:

```
$ svnversion . /repos/svn/trunk
4168
```

混合リビジョンの作業コピーの場合には、リビジョン範囲が 表示されます:

```
$ svnversion .
4123:4168
```

作業コピーに修正がある場合には ”M” を追加して表示します:

```
$ svnversion .
4168M
```

作業コピーがスイッチされた場合には ”S” を追加して表示します:

```
$ svnversion .
4168S
```

それで、たとえば混合リビジョンであり、作業コピーはスイッチされて いて、さらにローカルな修正があった場合には以下のようになります:

```
$ svnversion .
4212:4168MS
```

作業コピーではないディレクトリで実行されると、 **svnversion** はそれがエクスポートされた 作業コピーであると仮定し ”exported” と表示します:

```
$ svnversion .
exported
```

9.7 mod_dav_svn

名前

mod_dav_svn 設定ディレクティブ — Apache の HTTP サーバを通じて Subversion リポジトリを管理する場合の Apache の設定ディレクティブです。

名前

mod_dav_svn 設定ディレクティブ — Apache の HTTP サーバを通じて Subversion リポジトリを管理する場合の Apache の設定ディレクティブです。

説明

この節では Subversion に関係した Apache の設定ディレクティブをそれぞれ 簡単に説明します。(Subversion 利用時の Apache の設定についてのより詳しい説明は 項 6.5 を見てください。)

ディレクティブ

`DAV svn` このディレクティブは Subversion リポジトリのあるすべての `Directory` あるいは `Location` ブロックに 含めなくてはなりません。すべての要求の処理にたいして mod_dav のバックエンドとして Subversion を使うよう httpd に指示します。

`SVNAutoversioning On` このディレクティブは WebDAV クライアントからの書き込み要求を認め、 それが不分割なコミットになるようにします。一般的なログメッセージが 自動生成されリビジョンごとに付与されます。自動バージョン化を有効に する場合、おそらく `ModMimeUsePathInfo On` も 設定したくなるかも知れません。これによって `mod_mime` は 自動的に正しい mime-type を `svn:mime-type` に設定 できるようになります (もちろん `mod_mime` が可能な 範囲で、です)。より詳細は 付録 B を見てください。

`SVNPath` このディレクティブは Subversion リポジトリが使うファイルを置くための ファイルシステム中の場所を指定します。Subversion リポジトリのための 設定ブロック中では、このディレクティブか、`SVNParentPath` のどちらか一方が存在しなてくはなりません。しかし両方存在してはいません。

`SVNSpecialURI` Subversion 用の特殊なリソースのための URI 名 (名前空間) を指定します。 デフォルトは 「`!svn`」 で、ほとんどの 管理者は実際にこのディレクティブを使うことはないでしょう。 リポジトリ中で `!svn` という名前のファイルを 使う特殊な事情がある場合にだけ設定します。サーバ上ですでに利用している 値を変更したりすれば、外部にあるすべての作業コピーが壊れてしまい、 あなたはユーザに捕らえられ、何か恐ろしい目にあうでしょう。

`SVNReposName` `HTTP GET` 要求で利用する Subversion リポジトリの 名前を指定します。この値はすべてのディレクトリ一覧表示のタイトル として利用されます (これは Web ブラウザで Subversion リポジトリを 閲覧するときに使われます)。このディレクティブはオプションです。

`SVNIndexXSLT` ディレクトリインデックスのための XSL 変換の URI を指定します。 このディレクティブはオプションです。

`SVNParentPath` そのディレクトリのサブディレクトリが Subversion のリポジトリである ような親ディレクトリの、ファイルシステム中での位置を指定します。 Subversion リポジトリの設定ブロック中には、このディレクティブか `SVNPath` のどちらかが存在しなくてはなりません。 しかし両方存在してはいません。

`SVNPathAuthz` パス名にもとづいた認可を許可したり禁止したりするのに使います。 詳しくは 項 6.5.4.3 を見てください。

付録 A

CVS ユーザのための Subversion

A.1

この補遺は Subversion になじみのない CVS ユーザへのガイドです。 この章は基本的に、「10 キロくらい離れた」二つのシステム間の違い についての一覧です。それぞれの節で、できるだけ関連した章への 参照を用意しました。

Subversion の目標は現在と未来の CVS ユーザを乗っ取ることですが、 CVS で「問題」となっている振る舞いを改良するために いくつかの新しい機能と、設計の変更が必要でした。これは CVS ユーザとしての、いままでの習慣を断ち切る必要があるかも知れない ことを意味します — それは最初からおかしかったのですから。

A.2 リビジョン番号の意味が変わります

CVS では、リビジョン番号はファイルごとについていました。理由は、CVS は RCS ファイルにデータを格納していたからです。それぞれのファイルは RCS ファイルをリポジトリに持ち、そのリポジトリは大雑把に言って、プロジェクト ツリーの構造と一致するようなレイアウトでした。

Subversion では、リポジトリは一つのファイルシステムのように見えます。 それぞれのコミットはまったく新しいファイルシステムツリーを作ります。 要約して言うと、リポジトリとは、そのようなツリーが一列にたくさん並んだ もののことです。このようなツリーそれぞれは、一つのリビジョン番号でラベル 付けされています。誰かが「リビジョン 54」 と言うとき、彼らは特定 のツリー (そして間接的に、54 番目のコミット後のファイルシステムの見え方) について語っています。

技術的には、「`foo.c` のリビジョン 5」 という言い方は正しくありません。そうではなく、 「リビジョン 5 に出てくる `foo.c`」 と言うべきです。同様に、ファイルの変化についての前提に気をつけて ください。CVS では、リビジョン 5 とリビジョン 6 の `foo.c` は常に異なっていました。Subversion では リビジョン 5 と 6 では、たいていの場合 `foo.c` は変更されていません 。

これについての詳細は項 2.4.2 を見てください。

A.3 ディレクトリのバージョン

Subversion はツリーの構造を追いかけるのであって、それはファイルの内容 だけにはとどまりません。これは CVS を置き換えるために Subversion が書かれた 大きな理由の一つです。

CVS ユーザとしてのあなたに、これが何を意味するかをここで挙げておきます:

- **svn add** と **svn delete** コマンドは ファイルだけではなく、ディレクトリに対しても動作します。 **svn copy** と **svn move** もそうです。 しかし、これらのコマンドは、リポジトリに対して直接の変更を加えることはありません。そのかわりに、作業アイテムは単に、追加または削除 の「予告」 を受けるだけで

す。**svn commit** が実行されるまで、リポジトリはいっさい変更されません。

- ディレクトリはもう、ただの入れ物ではありません。それはファイルと 同じように、リビジョン番号を持っています。(あるいはもっと適切には「リビジョン 5 の中にある、ディレクトリ `foo/`」という言い方が正しいのですが)。

最後の点についてもっと説明します。ディレクトリのバージョン管理は 難しい問題です。それは、混合リビジョンの作業コピーを認めたいので、 このモデルを乱用することに対する制限が必要になります。

理論的な見地からすると、「ディレクトリ `foo` の リビジョン 5」 というのは、ディレクトリエントリと属性の、ある特定のあつまりを 意味します。`foo` のファイルを追加したり削除 し、コミットしたとします。リビジョン 5 の `foo` がまだあるといえば嘘になります。しかし、コミット後に `foo` のリビジョン番号を上げたとすれば、 これもやはり嘘になります。まだ更新してはいないので、 `foo` に、まだ受け取っていないほかの人の変更が 加えられたかも知れないからです。

Subversion はこの問題を、`.svn` の領域に コミットされた追加と削除を静かに記録することで取り扱います。 **svn update** を実行すると、 すべての変更点がリポジトリに反映され ディレクトリの新しいリビジョン番号は正しく設定されます。 そのため、更新後においてだけ、ディレクトリの「完全な」 リビジョンを手にしている、と安全に言うことができます。 ほとんどの場合、作業コピーは「不完全な」 ディレクトリ リビジョンを含んでいます。

同様にして、もしディレクトリ上の属性変更をコミットしようとした ときに問題が起こります。普通、コミットは作業ディレクトリの ローカルなリビジョン番号を上げます。しかし、やはり、それは 嘘です。というのは、更新がかかっていないことにより、ディレクトリ がまだ受け取っていない追加や削除があるかも知れないからです。 それで、ディレクトリが最新の状態になければ、ディレクトリ上の 属性変更をコミットすることはできません。

ディレクトリのバージョン管理の制限についての詳細は、 項 2.4.4 を見てください。

A.4 切断状態での豊富な操作

最近では、ディスク容量は非常に安く豊富になりましたが、ネットワークの 帯域はそうではありません。そのため Subversion の作業コピーはこの貴重な資源に ついて最適化されてきました。

`.svn` 管理ディレクトリは、CVS ディレクトリと同じ目的のために用意されていますが、例外は、 ファイルの「修正元コピー」を読み出し専用で保存してある ことです。これを使っていろいろな作業をオフラインでやることができます:

svn status ローカルに起きたすべての変更点を表示します (項 3.6.3.1 参照)

svn diff 変更点の詳細を表示します (項 3.6.3.2 参照)

svn revert 変更点を取り消します (項 3.6.3.3 参照)

さらに、キャッシュされた修正元ファイルを使うと、Subversion クライアントは、コミット時に差分のみを送信することができるよう になります。これは CVS にはできません。

このリストの最後のサブコマンドは新しいものです。それは単にローカル の修正を削除するだけではなく、追加や削除の予告操作も取り消すことが できます。ファイルの修正取り消しのためのおすすめの方法です。**rm file; svn update** を実行しても動作しますが、 更新の意味をあいまいにしてしまいます。そして、われわれはまだこの 問題と格闘しているのですが…

A.5 状態と更新の区別

Subversion では、**cvs status** と **cvs update** コマンドの間にあるたくさんの混乱 をなくそうとしてきました。

cvs status コマンドは二つの目的があります: まず、作業コピー中のローカルな変更をユーザに示すこと、二番目に どのファイルが最新ではなくなっているかをユーザに示すこと。 不幸にも、CVS の非常に読みにくい状態表示のために、たくさんの CVS ユーザは このコマンドをまったく生かしていません。そのかわりに、変更点を見る のに、**cvs update**、あるいは **cvs -n update** を実行する癖をつけてしまいました。`-n` オプションを忘れ ると、まだ扱う準備ができていないリポジトリ上の変更もマージしてしまいま す。

Subversion では、**svn status** の出力を 人間が読むにも、プログラムで扱うにも簡単になるように改良 しました。また、**svn update** は更新されたファイル についての情報のみを表示し、ローカルの変更は表示 しないように しました。

svn status はローカルに修正されたすべての ファイル を表示します。デフォルトではリポジトリにアクセスしません。このコマンド はいろいろなオプションをとりますが、以下は最もよく利用されるものです:

`-u` 過去の情報を取得してから表示するためにリポジトリに アクセスします。

`-v` バージョン管理下にあるすべての 情報を表示します。

`-N` 非再帰的に実行します (サブディレクトリに降りていきません)。

status コマンドには二つの出力形式があります。 デフォルトの「短い形式」では、ローカルの変更は以下のように 表示されます:

```
$ svn status
M      foo.c
M      bar/baz.c
```

`--show-updates`(`-u`) スイッチを指定すると もっと長い出力形式が利用されます:

```
$ svn status -u
M            1047   foo.c
       *     1045   faces.html
       *            bloo.png
M            1050   bar/baz.c
Status against revision:   1066
```

この場合、二つの新しいコラムが表示されます。二番目のコラムは ファイルやディレクトリが最新でない場合にはアスタリスク ('*') が表示 されます。三番目のコラムはアイテムの作業コピーリビジョン番号です。 上の例では、アスタリスクはもし更新しようとすれば `faces.html` はパッチされ、 `bloo.png` はリポジトリに新規追加されるだろう ことを示しています。(`bloo.png` の前にリビジョン番号が 表示されていないのは、作業コピーにはまだ存在していないからです)

最後に、最もよく表示されるステータスコードの簡単なまとめを載せておきます:

`A` リソースは追加予告されています
`D` リソースは削除予告されています
`M` リソースはローカルに変更されています
`C` リソースは衝突しています (変更箇所がリポジトリと作業コピーとの間でまだ完全にはマージされていません)
`X` リソースはこの作業コピーから見て外部のものです (他のリポジトリから来たものです。を参照してください)
`?` リソースはバージョン管理下にはありません。
`!` リソースは失われたか、不完全です (`Subversion` 以外の別のツールによって削除されました)

Subversion は CVS の `P` と `U` コードを連結し、単に `U` を表示します。 マージや衝突が起こるときには、Subversion はそれに関するすべての 内容を表示するかわりに単に `G` または `C` を表示します。

svn status に関する詳細は項 3.6.3.1 を参照してください。

A.6 ブランチとタグ

Subversion はファイルシステムの空間と「ブランチ」 の空間を 区別しません。ブランチとタグはファイルシステム中の普通のディレクトリ です。これは多分 CVS ユーザが乗り越えなくてはならない一番大きい心理的な 障害です。 これについては第 4 章全体を読んでください。

警告

Subversion はブランチとタグを通常のディレクトリのように扱うので、 いつもプロジェクトの `trunk` (`http://svn.example.com/repos/calc/trunk/`) をチェックアウトし、プロジェクト自身 (`http://svn.example.com/repos/calc/`) をチェックアウトしないように注意してください。プロジェクト自身をチェック アウトすると、作業コピーはすべてのブランチとタグを含むプロジェクト全体 になってしまいます。 [a]

[a] つまり、チェックアウトが終わる前にディスクを食いつぶしてしまうことになりかねません。

A.7 メタデータの属性

Subversion の新しい機能の一つに、ファイルやディレクトリに任意のメタデータ (あるいは 「属性」) を結びつけることができます。属性は任意の 名前/値のペアで作業コピーのファイルやディレクトリに結び付いています。

属性名を設定したり、取得したりするには、 **svn propset** と **svn propget** サブコマンドを使ってください。あるオブジェクト上のすべての属性を一覧表示 するには **svn proplist** を使ってください。

より詳しくは項 7.3 を参照してください。

A.8 衝突の解消

CVS はファイル中の「衝突マーカ」を使って衝突を知らせ、更新時には `C` を表示します。歴史的にはこれは問題を起こしてきました。それは CVS が十分に取り扱わなかったからです。 たくさんのユーザは、端末が警告した後に、`C` に ついて忘れてしまいます (あるいは見もしません)。しばしば衝突マーカが 依然として存在していることを忘れてしまい、衝突マーカを含んだファイルを 間違ってコミットしてしまうことがありました。

Subversion はこの問題を、衝突マーカをもっと良く見える形で設定する ことによって解決しています。Subversion はファイルが衝突状態にある ことを憶えていて、**svn resolved** を実行するまで、 あなたの変更点をコミットすることを許しません。 詳細は 項 3.6.4 を見てください。

A.9 バイナリファイルと変換

一番一般的な意味で、Subversion はバイナリファイルを CVS よりももっと 適切に扱います。CVS は RCS を利用するので、変更されたバイナリファイルの 完全なコピーを、常に格納するしかありませんでした。しかし Subversion はバイナリ差分アルゴリズムを使ってファイル間の相違を 表現します。そのファイルがテキストなのかバイナリなのかによらず そうします。これはすべてのファイルがリポジトリ中に差分の形として (圧縮されて) 格納されるということを意味します。

CVS ユーザは、`-kb` フラグを使ってバイナリファイルを マークする必要がありましたが、それはデータが文字化けすることを 防ぐためです。(キーワード展開や行末変換によってこのようなことが 起こります)。このことはときどき忘れられてしまいます。

Subversion はもっと神経質な方法をとります — まず、キーワード展開や 行末変換は、明示的にそのような指示を出さなければ実行されません (詳細は項 7.3.3.4 と 項 7.3.3.5 を見てください)。 デフォルトでは、Subversion はすべてのファイルデータを文字通り 単なるバイトの並びとして扱い、ファイルは常に、無変換の状態で リポジトリに保存されます。

次に、Subversion はファイルが「テキスト」 であるか 「バイナリ」 であるかの内部的な記録を管理しますが、 この記録は、作業コピー中にしか ありません。 **svn update** 実行中、Subversion は、ローカルに 修正のあったテキストファイルについて文脈マージをやりますが、 バイナリファイルに対してはそうしません。

文脈マージが可能かどうかを決めるのに、Subversion は `svn:mime-type` 属性を調べます。もしファイルが `svn:mime-type` 属性を持たないか、テキストを 示すような内容であれば (たとえば、`text/*`)、Subversion はそれを テキストであると判断します。それ以外の場合、ファイルはバイナリ であるとみなされます。Subversion は **svn import** と **svn add** 実行時に、バイナリ検出アルゴリズムを 実行することでユーザを助けます。これらのコマンドは、良い推測を 行い、(可能なら) バイナリの `svn:mime-type` 属性 を追加されるファイルに設定します。(もし Subversion の推測が間違って いた場合は、ユーザはいつでも手でその属性を削除することができます)

A.10 バージョン管理されたモジュール

CVS とは違い、Subversion の作業コピーはモジュールとしてチェックアウトされた ことを記録しています。これは誰かがモジュールの定義を (たとえば、部品を 追加したり削除することで) 変更してから **svn update** を呼び出した場合、部品を追加したり削除す ることで、作業コピーを正しく更新することを意味します。

Subversion はモジュールをあるディレクトリ属性のあるディレクトリ の集まりとして定義します。項 7.6 を見てください。

A.11 認証

CVS の pserver では読み出し、書き込み操作の前に必ずサーバに「login」 しなくてはなりませんでした — 匿名操作のためにログインする場合でもそうでした。 サーバに Apache **httpd** または **svnserve** を使った Subversion リポジトリの場合には処理の前に 認証確認をする必要はありません — 認証が必要な処理を実行するときになって 初めてサーバはあなたの認証しようとします (それがユーザ名、パスワードの方式 であっても、クライアント証明書であっても、両方であっても、です)。それで もしリポジトリが不特定多数の人に対して読み出しアクセス権限を与えている場合 は読み出し操作のために認証する必要は一切ありません。

CVS と同様、Subversion でも `--no-auth-cache` スイッチを使って 明示的に禁止しない限り、やはりあなたの認証情報をディスク上に保存します。 (自分の `/.subversion/auth/` ディレクトリ配下になります。)

しかしこの動作には例外があり、それは `svn+ssh://` URL スキー マを使って SSH トンネル越しに **svnserve** にアクセ スした場合です。この場合には **ssh** プログラムはト ンネルを使った通信を開始するにあたって無条件に認証を要求します。

A.12 CVS から Subversion へのリポジトリ変換

おそらく CVS ユーザが Subversion に慣れる一番重要な方法は既存の プロジェクトを新しいシステムを使って継続することです。これには 抽出した CVS リポジトリを Subversion リポジトリに単にインポート することでもある程度達成できますが、さらに徹底した方法としては データの最新の状態だけではなく、それ以前の履歴全体を旧システムから 新システムに移すことになります。これは解くのが極端に難しい問題です が、それは不分割の性質を持たない CVS データの変更からチェンジセット を求めること、ブランチについて全く別の考え方を持っている両システム 間でデータ変換すること、その他の複雑な問題があるためです。それでも少なくとも部分的には既存の CVS リポジトリを Subversion リポジトリ に変換するためのツールがあります。

そのようなツールのひとつに cvs2svn (<`http://cvs2svn.tigris.org/`>) が ありますが、これはもともと Subversion 自身の開発メンバによって作られた Python のスクリプトです。他には Chia-liang Kao が作った VCP ツール (<`http://svn.clkao.org/revml/branches/svn-perl/`>) のプラグインである Subversion コンバータや Lev Serebryakov の作った RefineCVS (<`http://lev.serebryakov.spb.ru/refinecvs/`>) があります。これらのツールの完成度はさまざまで CVS リポジトリの履歴の扱い方について全くことなる考え方にもとづいているかも知れません。どのツールを使う場合 でも変換結果について納得がいくまで十分な確認をしてください — ようするに この履歴を作り上げるにはかなり労力が必要になります。

知られている変換ツールへのリンクの最新情報については Subversion ウェブサイト (<`http://subversion.tigris.org/project_links.html`>) のリンクページを見てください。

付録 B

WebDAV と、自動バージョン化

B.1

WebDAV は HTTP の拡張で、ファイル共有のための標準として ますます一般的なものになっていっています。今日のオペレーティング システムは極端に Web を意識していて、多くの OS は WebDAV サーバによって公開された「共有」 をマウントするための仕組みを組み込みで サポートしています。

もし Apache/mod_dav_svn を Subversion ネットワークサーバとして 利用するなら、ある程度 WebDAV サーバも実行しなくてはなりません。 この補遺は、このプロトコルの性質についてのいくつかの背景を与え、Subversion がどのようにそれを利用し、WebDAV を考慮しているほかのソフト とどのようにうまく協調するかを示します。

B.2 WebDAV の基本的な概念

この節は WebDAV の背後にあるアイディアについての、とても簡単で 一般的な概要を示します。それはクライアントとサーバの間の WebDAV の互換性に関する問題を理解するための基礎になります。

B.2.1 単純な WebDAV

RFC 2518 はいくつかの概念と、それにともなう HTTP 1.1 の 拡張メソッドを定義しています。それは web をもっと普遍的な読み書き 可能な仕組みにするものです。基本的なアイディアは WebDAV 互換のウェブ サーバは、一般的なファイルサーバのように振る舞うことができるという ことです。クライアントは WebDAV の「共有」をマウントする ことができ、NFS や、SMB 共有のように動きます。

しかしながら、RFC 2518 は、DAV の文字列中の「V」にもかかわらず、 どんなタイプのバージョン管理のモデルも提供してはいない ということを知っておくのは重要です。基本的な WebDAV クライアントとサーバは ファイルやディレクトリの一つのバージョンのみが存在するのが前提となっていて、 繰り返し上書きすることができます。

基本的な WebDAV で導入された新しい概念とメソッドは:

リソース　WebDAV の世界ではサーバ側にあるすべてのオブジェクト (それは URI によって記述されるものですが) は、 リソースと言われます。

新しい書き込みメソッド　標準的な HTTP `PUT` メソッドに 加えて (それは web リソースを作ったり上書きしたりしますが)、WebDAV は新しい `COPY` と `MOVE` メソッドを定義し、リソースを複製したり移動したりすることができます。

集合　集合 は WebDAV の用語では ひとまとまりのリソースのことを言います。ほとんどの場合、それは ディレクトリのようなものになります。ファイルリソースは PUT メソッドで書き込まれたり作られたりしますが、 集合リソースは新しい MKCOL メソッドで作られます。

属性　これは Subversion に出てるのと同じアイディアです — ファイルと集合に付随したメタデータです。クライアントは新しい PROPFIND メソッドを使ってリソースに付随 した属性を一覧表示したり抽出したりできます。そして、 PROPPATCH メソッドを使って変更できます。 いくつかの属性は完全にユーザによって作られ制御されます (たとえば、「color」と呼ばれる属性)、また他のものは 完全に WebDAV サーバによって作られ制御されます (たとえば、ファイル の最後の修正時刻を含む属性)。最初のものは「死んだ」 属性と呼ばれ、あとのものは「生きた」 属性と呼ばれます。

ロック　WebDAV サーバはクライアントに対するロックの機能を与える ことができます。 — この機能は任意です。ほとんどの WebDAV サーバはこの機能を 提供していますが。もし存在すれば、クライアントは新しい LOCK と UNLOCK メソッドを 使ってリソースへのアクセスを調停することができます。 ほとんどの場合、これらのメソッドは排他的な書き込みロックを作る ために利用されます (項 2.3.2 で議論した ように)、ただしサーバの実装によっては共有書き込みロックも可能です。

アクセス制御　より最近の仕様 (RFC 3744) では WebDAV リソースに対するアクセス制御 リスト (ACL) を定義するためのシステムを規定しています。クライアントや サーバによってはこの機能を実装し始めているものもあります。

B.2.2 DeltaV 拡張

RFC 2518 はバージョン化の概念がないので、 他の機能グループは RFC 3253 にまかされました。それは、WebDAV にバージョン化の機能を追加したものです。この部分は「DeltaV」 と呼ばれます。WebDAV/DeltaV クライアントとサーバはしばしば単に「DeltaV」 クライアントとサーバと呼ばれます。DeltaV は基本的な WebDAV の 存在を含んでいるからです。

DeltaV はまったく新しい単語を導入しましたが、 びっくりしないでください。考え方は非常に直接的です:

リソースごとのバージョン化　CVS や他のバージョン管理システムのように DeltaV はそれぞれのリソースは 無限の数の状態をとりうると仮定しています。クライアントは新しい VERSION-CONTROL メソッドを使ってリソースを バージョン管理下に置くことによって始めます。これには新しい VERSION-CONTROL メソッドを使います。

サーバ側作業コピーモデル　DeltaV サーバによっては仮想的な作業スペースをサーバ上に作る能力が あります。すべての作業はそこで実行されます。クライアントは MKWORKSPACE メソッドを使ってプライベートな領域を 作り、作業スペースに「チェックアウト」することで 特定のリソースを変更したいということを示し、編集した後、もう一度 「チェックイン」します。 HTTP の言葉で言えば、メソッドの流れとしては、 CHECKOUT, PUT, CHECKIN となります。

クライアント側作業コピーモデル　DeltaV サーバによってはクライアントがローカルディスク上にプライベート な作業コピーを持つこともできるという考え方をサポートします。 クライアントがサーバに変更点

をコミットしたい場合、まず MKACTIVITY メソッドによって一時的なサーバトランザクション (アクティビティー と呼ばれます) を作ることで処理を開始します。 それからクライアントは変更したいリソースごとに CHECKOUT を実行し、PUT 要求を送ります。最後にクライアントは リソースに対する CHECKIN を実行するか MERGE 要求を送ってすべてのリソースを一度に チェックインします。

設定　DeltaV では「設定」と呼ばれるリソースの 汎用的な集まりを定義することができますが、かならずしもそれは特定の ディレクトリに対応する必要はありません。設定はファイルの特定のバージョン を指し示すのに作成したり、「ベースライン」のスナップショット を作ったりできます。後者はタグによく似たものです。

拡張性　DeltaV は新しいメソッド REPORT を定義しますが それはクライアントとサーバが独自データ交換を実行するのを許す ものです。DeltaV はクライアントが要求可能な標準化された履歴情報 をいくつも定義してありますが、さらに自由にカスタム情報を定義することも できます。クライアントは REPORT 要求を 独自のデータのある属性ラベルの付いた XML のボディーをともなって 送信します。サーバがこの特定のレポート型を理解できることを 仮定して、それはやはり独自の XML ボディーを応答します。この技術は XML-RPC とよく似ています。

B.3 Subversion と DeltaV

当初の WebDAV 標準は幅広く成功をおさめました。現在利用されている コンピュータのオペレーティングシステムは一般的な WebDAV クライアント を組み込みで持っています (詳しくは後述します)、また良く知られたさまざまな スタンドアロンのアプリケーションも WebDAV を話すことができます — 例をあげれば Microsoft Office, Dreamweave, Photoshop などがあります。 サーバ側では Apache ウェブサーバが 1998 年以降 WebDAV の機能を提供できる ようになり事実上のオープンソース標準と考えられています。他にもさまざまな 商用の WebDAV サーバが利用可能であり、それには Microsoft 自身の IIS も 含まれます。

しかし不幸にも DeltaV の方はそれほど成功をおさめていません。DeltaV の クライアントやサーバの実装をみつけるのはとても困難なことです。 わずかに存在するものは比較的知られていない商用製品ですし、そのため 相互運用性をテストするのも非常に困難です。人によっては、それは単に 仕様が複雑すぎるからだと言いますし、別の人は WebDAV の機能は大衆受け する (最小限度の技術的な知識しか持っていないユーザでもネットワークの ファイル共有は喜んで利用します) のに対して、多くの人にとってバージョン 制御機能は興味がないか、あるいは不要であるのが理由だと言います。 さらに最後の意見として、DeltaV はそれを実装しているオープンソースの サーバ製品がまだ存在しないから人気がないのだと言うものもあります。

Subversion がまだ設計段階にあったとき Apache httpd を主要なネットワーク サーバとして利用するというのは素晴らしいアイディアに思えました。 それはすでに WebDAV サービスを提供するモジュールを持っていたからです。 DeltaV は比較的新しい仕様でした。Subversion のサーバモジュール (mod_dav_svn) は最終的にはオープンソースの DeltaV の標準的な実装に 進化できるのではないかという期待がありました。しかし不幸なことに DeltaV は非常に特殊なバージョンモデルであり Subversion のモデルとは それほど親和性が良いとは言えません。概念的には対応させることが可能だと 言う人もいますし、いやダメだという人もいます。

結論としては

1. Subversion クライアントは完全な DeltaV クライアントを実装している わけではない。
   クライアントは DeltaV では提供することのできないようなサーバからある種の情報 を得る必要があり、そのため Subversion 特有の REPORT に幅広く依存していて、それは mod_dav_svn にしか理解できないような性質のものである。

2. mod_dav_svn は DeltaV サーバの完全な実装ではない。
   DeltaV 仕様の多くの部分は Subversion には無関係であり実装され ずに放置されている。

このような状況にきちんと対応すべきかどうかについては開発者の間でまだ 議論があります。Subversion の設計を DeltaV に合うように変更することは およそ現実的ではないので、おそらくクライアントは一般的な DeltaV サーバ から必要なすべての情報を得ることはできないでしょう。いっぽう mod_dav_svn はすべての DeltaV を実装ためにさらに開発を進めるかも 知れませんが、本当にそうしようと言う強い動機は見当たらないのが現状です — それと協調して動作する DeltaV クライアントがほとんどひとつも存在 していないのですから。

B.4 自動バージョン化

Subversion クライアントはまだ完全な DeltaV クライアントではありませんし Subversion サーバも完全な DeltaV サーバではありませんが、 WebDAV 協調動作できるうれしい機能があります: それは自動バージョン化と 呼ばれるものです。

自動バージョン化は DeltaV 標準ではオプションで実装することのできる機能 として定義されているものです。典型的な DeltaV サーバは単純な WebDAV ク ライアントがバージョン管理下にあるファイルに対して `PUT` 命令を実行してすることを拒否します。バージョン管 理下にあるファイルを変更するにはサーバに対して適切なバージョン要求が必 要になります: それはたとえば `MKACTIVITY`, `CHECKOUT`, `PUT`, `CHECKIN` のような感じの命令です。しかし DeltaV サーバ が自動バージョン化をサポートすると基本的な WebDAV クライアントからの書き込み 要求も受け付けられるようになります。サーバはクライアントが適切なバージョン要求のコマンド列を発行した*かのように*振舞い、内部的にコミットを実行します。 言い換えると DeltaV サーバは通常の WebDAV クライアントと協調動作できる ようになります。

すでに非常にたくさんのオペレーティングシステムが WebDAV クライアント機能 を備えているのでこの機能は非常に広範囲にわたって利用できることに なります: 通常のユーザが Microsoft Windows や Mac OS を実行しているような オフィスを考えてみてください。それぞれのユーザは Subversion リポジトリを「マウント」し、しかもそれは通常のネットワークフォルダのように 見えます。通常やっているような共有フォルダを使うのと同じ感覚で操作できます: ファイルを開き、編集し、そして保存します。一方サーバは自動的にすべてをバージョン化します。管理者の側では (あるは知識のあるヘビーユーザは) 依然として Subversion クライアントを使ってこの履歴を調べたり、古いバージョンのデータ を取得することができるのです。

この筋書きは架空のものではありません: Subversion 1.2 かそれ以降では 実際に動作するのです。mod_dav_svn で自動バージョン化機能を有効にする には `httpd.conf` の Location ブロック中で `SVNAutoversioning` ディレクティブを使ってください。 こんな感じです:

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

SVNAutoversioning が有効な場合には WebDAV クライアントからの要求は 不分割なコミットとなります。一般的なログメッセージが自動生成され それぞれのリビジョンにつけられます。

しかしこの機能を有効にする前に本当にやろうとしていることを理解して ください。WebDAV クライアン

トは たくさんの 書き込み要求を実行しがちであり、これは自動的にコミットされる非常に たくさんのリビジョンが発生することを意味します。たとえば データを保存する場合、たくさんのクライアントがゼロバイトのファイルの PUT を実行し、その後に別の PUT が実際のデータをともなって実行されるでしょう。単一のファイル書き込み が二つの別々のコミットになってしまいます。さらに多くのアプリケーション では数分に一度の自動保存機能がはたらき、さらに多くのコミットが発生 してしまうでしょう。

たとえばメールを送信するような post-commit のフックプログラムがある 場合、email 生成を完全に禁止したいと思うか、リポジトリの特定の部分に 関して禁止したいと思うかも知れません; それは email の流入量が意味のある通知と考えられるかどうかにかかっています。さらに、賢い post-commit フックプログラムは自動バージョン化機能によって発生した トランザクションと通常の **svn commit** によって発生したトランザクションとを区別する ことができます。これには `svn:autoversioned` という 名前のリビジョンプロパティーを見るとうまくいきます。もし存在して いればそのコミットは一般的な WebDAV クライアントによるものです。

B.5 クライアントの協調動作

すべての WebDAV クライアントは三つのどれかに分類されます — スタンドアロン・アプリケーション、ファイルエクスプローラ拡張、そして ファイルシステムの実装です。これらの分類はおおざっぱに言ってユーザに 対して提供できる WebDAV の機能の種類を決めます。 表 B.1 はその分類と、WebDAV が利用可能な ソフトウェアの共通部品の簡単な説明です。これらのソフトウェアが提供する機能の 詳細と一般的な分類についてはその後の節で見ることができます。

B.5.1 スタンドアロン WebDAV アプリケーション

WebDAV アプリケーションは WebDAV サーバと通信可能な WebDAV プロトコル の機能を組み込んだプログラムのことです。このような形での WebDAV を サポートしている最も有名なプログラムのいくつかを紹介します。

B.5.1.1 Microsoft Office, Dreamweaver, Photoshop

Windows 上では Microsoft Office のような、WebDAV のクライアント機能を 統合した有名なアプリケーションがいくつかあります。 [*1] Adobe の Photoshop と Macromedia の Dreamweaver 。 両方とも直接 URL を開いたり保存したりすることができますが ファイルを編集するさい WebDAV の排他制御を頻繁に利用する傾向が あります。

Mac OS X 上にも同じようなたくさんのプログラムが存在しますが、 それらのプログラム上では直接 WebDAV がサポートされているようには見えません。じっさい Mac OS X 上では、**File->Open** ダイアログは パスや URL の入力がまったく許されてはいません。これらのプログラムの Macintosh バージョンでは WebDAV の機能はわざと実装されていない ように見えますが、それは OS X 自身がすでに WebDAV 用のすばらしい 低レベルファイルシステムを提供しているからです。

B.5.1.2 Cadaver, DAV Explorer

Cadaver は生の Unix コマンドラインプログラムで、WebDAV 共有を 閲覧したり変更したりすることのできるものです。Subversion クライアントと同様、 neon HTTP ライブラリが必要になります — しかしびっくりする必要はありません。 neon と cadaver は同じ作者によって書かれています。Cadaver はフリーソフトウェア (GPL ライセンス) で、<http://www.webdav.org/cadaver/>から取得できます。

[*1] ある理由で WebDAV のサポートは Microsoft Access からは 削除されましたが、それ以外の Office スイートには存在します。

cadaver を使うのは、ちょうどコマンドラインの FTP プログラムを使うような感じ なので、基本的な WebDAV 機能をデバッグするのにとても役立ちます。困った時には ファイルをアップロードしたりダウンロードしたりするのに使うことができますし、 プロパティーを調べたり、ファイルのコピー、移動、ロック、アンロックも やってのけます:

```
$ 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 はもう一つのスタンドアロン WebDAV クライアントであり Java で書かれています free Apache-like license のもとで、 <http://www.ics.uci.edu/~webdav/>から取得可能です。 DAV Explorer は cadaver でできるすべてのことができますが、より可搬性に すぐれ、ユーザに優しい GUI アプリケーションです。またこれは WebDAV アクセス制御プロトコル (RFC 3744) をサポートした最初のクライアントの一つでもあります。

もちろん DAV Explorer の ACL サポートはこの場合やくには立ちません。mod_dav_svn がサポートしていないからです。Cadaver と DAV Explorer がどちらも制限された DeltaV コマンドをサポートしていても、あまり役には立たちません。 `MKACTIVITY` 要求が許されていないからです。しかし、とにかく それは関係のないことです; ここで仮定しているのはこれらのクライアントの すべては自動バージョン化されたリポジトリを操作できるということです。

B.5.2 ファイルエクスプローラの WebDAV 拡張

いくつかの有名なファイルエクスプローラ GUI プログラムは WebDAV 拡張を サポートしていて、ユーザに、DAV 共有領域を、単にあたかもローカルコンピュータ 上にある別のディレクトリのように見せたり、その共有領域のアイテムに対する 基本的なツリー編集操作を可能にするものです。 たとえば Windows Explorer では、ひとつの「ネットワークプレース」と して WebDAV サーバを閲覧できます。ユーザはファイルをデスクトップ上で移動 したり、名称変更、コピー、削除などの処理を通常のやりかたで操作できます。 しかしそれはファイルエクスプローラの機能でしかないので、通常のアプリケーション には DAV 共有領域は見えません。すべての DAV 操作は、エクスプローラのインターフェース を通じて実行しなくてはなりません。

B.5.2.1 Microsoft Webfolders

Microsoft は WebDAV 仕様の元来の支援者の一つであり、Windows 98 で最初の クライアントを出荷し始めました。これは「Webfolders」と言う 名前で知られています。このクライアントはまた Windows NT4 と 2000 でも 出荷されました。

もともとの Webfolder クライアントはエクスプローラの拡張であり、これは ファイルシステムを閲覧する際の主要な GUI プログラムでした。これはとても うまく動作しています。Windows 98 では 「マイコンピュータ」 の中に Webfolder がない場合には明示的にインストールする必要がありました。 Windows 2000 では単に新しい 「ネットワークプレース」を追加 し、URL を入力すれば WebDAV 共有が閲覧用にポップアップします。

Windows XP のリリースで、Microsoft は Webfolder の新しい実装を出荷し 始めましたが、これは「WebDAV mini-redirector」という名前で 知られています。新しい実装はファイルシステムレベルのクライアントであり WebDAV 共有をドライブ文字をつけてマウントできるようにしたものです。 残念なことにこの実装はかなりバグがあります。クライアントは通常 http URL (`http://host/repos`) を UNC 共有記法 (

`host`
`repos`) に変換しようとします; また Windows ドメイン認証を使って HTTP 基本認証要求に応答しますが、このとき ユーザ名には `HOST`
`username` が利用されます。 このような協調動作の問題は深刻なもので、多くのユーザに対する不満を 解消するための数えきれないくらいのドキュメントがネット上に存在して います。Apache の WebDAV モジュールを最初に設計した Greg Stein で すら Apache サーバに対して XP Webfolder を利用することを推奨 していません。

最初の 「エクスプローラ機能のみの」 Webfolder の実装は XP でもなくなってはいませんが、埋もれてしまっています。以下のような 方法を使っていまでも探すことはできます:

1. ’ ネットワークプレース’ を選択します。
2. 新しいネットワークプレースを追加します。
3. プロンプトが出たらリポジトリの URL を入力しますが URL には ポート番号を含めます。例えば `http://host/repos` は、かわりに `http://host:80/repos` と指定します。
4. 認証要求に正しく答えます。

この問題の回避策として他にもさまざまなうわさがありますが、Windows XP の すべてのバージョンとパッチレベルでうまく動作するものはなさそうです。 私たちのテストでは今示したアルゴリズムがどのシステムでも常にうまくいく ように思えます。WebDAV コミュニティーの一般的に合意されていることは、 新しい Webfolders の実装のかわりに古いものを使うべきであることと、 Windows XP 上で本当のファイルシステムレベルのクライアントが必要な場合 には WebDrive や NetDrive のようなサードパーティー製のプログラムを使うこと です。

B.5.2.2 Nautilus, Konqueror

Nautilus は GNOME デスクトップの公式なファイルマネージャ/ブラウザ で (<`http://www.gnome.org`>)、Konqueror は KDE デスクトップの ファイルマネージ/ブラウザです (<`http://www.kde.org`>)。どちらの アプリケーションもエクスプローラレベルでの組み込み WebDAV クライアントで、 自動バージョン化されたリポジトリに対してもうまく動作します。

GNOME の Nautilus の場合、ファイルメニューで **Open location** を 選択し、URL を入力します。これで

リポジトリは他のファイルシステムと同じように 見えるはずです。

KDE の Konqueror の場合は、ロケーションバーに URL を入力するさいに `webdav://`スキーマを使う必要があります。`http://` の形の URL を入力すると、Konqueror は通常のウェブブラウザのように動作します。 たぶん mod_dav_svn によって生成される一般的な HTML のディレクトリ一覧が表示される ことでしょう。 `http://host/repos` のかわりに `webdav://host/repos` を入力すると、Konqueror は WebDAV クライアントとなり、ファイルシステムとしてリポジトリを表示するようになります。

B.5.3 WebDAV ファイルシステムの実装

WebDAV ファイルシステムの実装は おそらく WebDAV クライアントの最もすぐれた ものの一つです。それは低レベルの ファイルシステム、典型的にはオペレーティングシステムのカーネル内部 に実装されます。これが意味することは、DAV 共有は他のネットワークファイル システムと同じように、たとえば Unix なら NFS や、Windows ならドライブ文字を つけた SMB 共有のような形でマウントすることができるということです。結果として、この手のクライアントはすべてのプログラムに対して完全に 読み書き透過な WebDAV アクセスを提供できることになります。アプリケーションは 実際には WebDAV 要求が発生していることに気づきもしないでしょう。

B.5.3.1 WebDrive, NetDrive

WebDrive も NetDrive も非常にすばらしい商用製品であり、WebDAV 共有を ドライブ文字に対応することができます。これらの製品を使って接続を失敗 させることはどうしてもできませんでした。これを書いている時点で WebDrive は South River Technologies (<http://www.southrivertech.com>) から購入可能です。NetDrive は Netware に付属しており、それ自体は無料です。「netdrive.exe」 のサイトをウェブで検索すると出てきます。 (奇妙な話にだと思うのはあなただけではありません。以下の Novell のウェブサイトを 見てください: <http://www.novell.com/coolsolutions/qna/999.html>)

B.5.3.2 Mac OS X

Apple 社の OS X オペレーティングシステムには ファイルシステムレベルで統合された WebDAV クライアントがあります。 ファインダーの **Go menu** から **Connect to Server** アイテムを選択します。 この状態で WebDAV URL を入力すれば、他のマウントされたボリュームと 同じようにデスクトップ上のディスクとしてアクセスできるようになります。 [*2]

使っている mod_dav_svn がバージョン 1.2 よりも古い場合 OS X はマウント共有部分を読み書きモードでマウントするのを拒否し、 読み込み専用になります。OS X が読み書きモードでの共有を サポートしていますが、ファイルロックの機能は Subversion 1.2 で初めて サポートされたものだからです。

もう一点だけ: OS X の WebDAV クライアントは HTTP のリダイレクトに 対して必要以上に神経質に動作することがあります。リポジトリをまったく マウントできない場合には、Apache サーバの `httpd.conf` ファイルの、BrowserMatch ディレクティブを有効にする必要があるかも 知れません:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

[*2] Darwin の端末の場合は、`mount -t webdav URL /mountpoint` を実行しても 同じことです。

B.5.3.3 Linux davfs2

Linux davfs2 は Linux カーネル用のファイルシステムモジュールで <http://dav.sourceforge.net/> で開発されています。 一度インストールしてしまえば WebDAV ネットワーク共有は通常の Linux の mount コマンドによってマウントできます:

```
$ mount.davfs http://host/repos /mnt/dav
```

表 B.1 よく利用される WebDAV クライアント

ソフトウェア	分類	説明
Adobe Photoshop	スタンドアロン WebDAV アプリケーション	画像編集ソフトで、WebDAV URL を直接開いたり書き込んだりすることができます。
Cadaver	スタンドアロン WebDAV アプリケーション	コマンドライン WebDAV クライアントで、ファイル転送、ツリー、排他操作をサポートしています。
DAV Explorer	スタンドアロン WebDAV アプリケーション	WebDAV 共有を閲覧するための GUI ツールです。
davfs2	WebDAV のファイルシステム実装	Linux ファイルシステムドライバで、WebDAV 共有をマウントできます。
GNOME Nautilus	ファイルエクスプローラの WebDAV 拡張	WebDAV 共有上のツリー操作を可能にする GUI のファイルエクスプローラです。
KDE Konqueror	ファイルエクスプローラの WebDAV 拡張	WebDAV 共有上のツリー操作を可能にする GUI のファイルエクスプローラです。
Mac OS X	WebDAV のファイルシステム実装	WebDAV 共有をオペレーティングシステムに、ローカルにマウントする機能を組み込みでサポートしています。
Macromedia Dreamweaver	スタンドアロン WebDAV アプリケーション	WebDAV URL に対して直接の読み書き可能な Web ページ作成ツール。
Microsoft Office	スタンドアロン WebDAV アプリケーション	オフィスの生産性を高めるさまざまなコンポーネントからなる製品で、WebDAV URL に対して直接読み書きすることが可能です。
Microsoft Webfolders	ファイルエクスプローラの WebDAV 拡張	WebDAV 共有上のツリー操作を可能にする GUI のファイルエクスプローラ・プラグラムす。
Novell NetDrive	WebDAV ファイルシステムの実装	Windows のドライブ識別文字をマウントされたリモート WebDAV 共有に 割り当てるための、ドライブマッピング・プログラムです。
SRT WebDrive	WebDAV ファイルシステムの実装	機能の一部として、Windows のドライブ識別文字を マウントされたリモート WebDAV 共有に割り当てることも可能な、ファイル転送ソフトウェアです。

付録 C

サードパーティー製ツール

C.1

Subversion のモジュール化された設計 (項 8.2 で議論しました) と可能な言語連係 (項 8.3.3 に記述があります) は他のソフトウェア の拡張やバックエンドとして利用することのできる選択子になります。Subversion の機能を後ろで使うようなサードパーティー製のツール の一覧は、Subversion ウェブサイト (<http://subversion.tigris.org/project_links.html>) のリンクページにあります。

付録 D

Copyright

D.1

Your fair use and other rights are in no way affected by the above.

The above is a summary of the full license below.

==========

Creative Commons Legal Code
Attribution 2.0

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion

picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

d. "Original Author" means the individual or entity who created the Work.

e. "Work" means the copyrightable work of authorship offered under the terms of this License.

f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital

audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other

comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark

"Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

==========